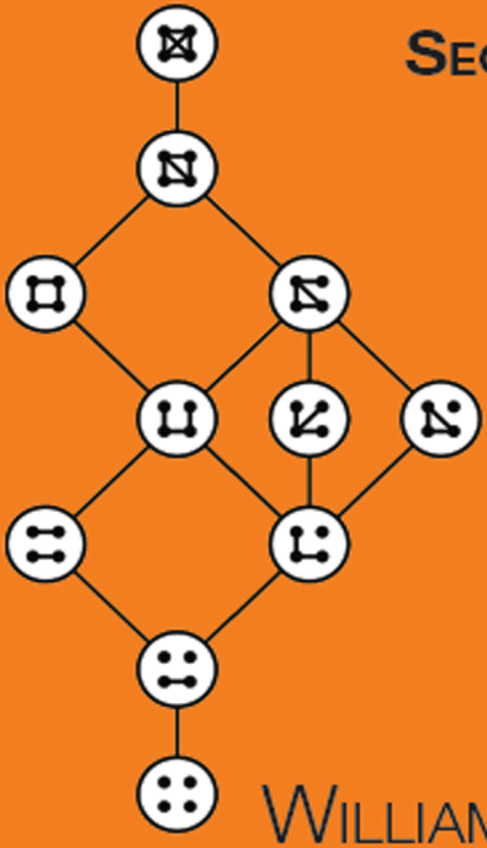
DISCRETE MATHEMATICS AND ITS APPLICATIONS

GRAPHS, ALGORITHMS, AND OPTIMIZATION

SECOND EDITION



WILLIAM L. KOCAY
DONALD L. KREHER

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

GRAPHS, ALGORITHMS, AND OPTIMIZATION

SECOND EDITION

DISCRETE MATHEMATICS AND ITS APPLICATIONS

R. B. J. T. Allenby and Alan Slomson, How to Count: An Introduction to Combinatorics, Third Edition

Craig P. Bauer, Secret History: The Story of Cryptology

Jürgen Bierbrauer, Introduction to Coding Theory, Second Edition

Katalin Bimbó, Combinatory Logic: Pure, Applied and Typed

Katalin Bimbó, Proof Theory: Sequent Calculi and Related Formalisms

Donald Bindner and Martin Erickson, A Student's Guide to the Study, Practice, and Tools of Modern Mathematics

Francine Blanchet-Sadri, Algorithmic Combinatorics on Partial Words

Miklós Bóna, Combinatorics of Permutations, Second Edition

Miklós Bóna, Handbook of Enumerative Combinatorics

Miklós Bóna, Introduction to Enumerative and Analytic Combinatorics, Second Edition

Jason I. Brown, Discrete Structures and Their Interactions

Richard A. Brualdi and Dragoš Cvetković, A Combinatorial Approach to Matrix Theory and Its Applications

Kun-Mao Chao and Bang Ye Wu, Spanning Trees and Optimization Problems

Charalambos A. Charalambides, Enumerative Combinatorics

Gary Chartrand and Ping Zhang, Chromatic Graph Theory

Henri Cohen, Gerhard Frey, et al., Handbook of Elliptic and Hyperelliptic Curve Cryptography

Charles J. Colbourn and Jeffrey H. Dinitz, Handbook of Combinatorial Designs, Second Edition

Abhijit Das, Computational Number Theory

Matthias Dehmer and Frank Emmert-Streib, Quantitative Graph Theory: Mathematical Foundations and Applications

Martin Erickson, Pearls of Discrete Mathematics

Martin Erickson and Anthony Vazzana, Introduction to Number Theory

Titles (continued)

Steven Furino, Ying Miao, and Jianxing Yin, Frames and Resolvable Designs: Uses, Constructions, and Existence

Mark S. Gockenbach, Finite-Dimensional Linear Algebra

Randy Goldberg and Lance Riek, A Practical Handbook of Speech Coders

Jacob E. Goodman and Joseph O'Rourke, Handbook of Discrete and Computational Geometry, Second Edition

Jonathan L. Gross, Combinatorial Methods with Computer Applications

Jonathan L. Gross and Jay Yellen, Graph Theory and Its Applications, Second Edition

Jonathan L. Gross, Jay Yellen, and Ping Zhang Handbook of Graph Theory, Second Edition

David S. Gunderson, Handbook of Mathematical Induction: Theory and Applications

Richard Hammack, Wilfried Imrich, and Sandi Klavžar, Handbook of Product Graphs, Second Edition

Darrel R. Hankerson, Greg A. Harris, and Peter D. Johnson, Introduction to Information Theory and Data Compression, Second Edition

Darel W. Hardy, Fred Richman, and Carol L. Walker, Applied Algebra: Codes, Ciphers, and Discrete Algorithms, Second Edition

Daryl D. Harms, Miroslav Kraetzl, Charles J. Colbourn, and John S. Devitt, Network Reliability: Experiments with a Symbolic Algebra Environment

Silvia Heubach and Toufik Mansour, Combinatorics of Compositions and Words

Leslie Hogben, Handbook of Linear Algebra, Second Edition

Derek F. Holt with Bettina Eick and Eamonn A. O'Brien, Handbook of Computational Group Theory

David M. Jackson and Terry I. Visentin, An Atlas of Smaller Maps in Orientable and Nonorientable Surfaces

Richard E. Klima, Neil P. Sigmon, and Ernest L. Stitzinger, Applications of Abstract Algebra with Maple™ and MATLAB®, Second Edition

Richard E. Klima and Neil P. Sigmon, Cryptology: Classical and Modern with Maplets

Patrick Knupp and Kambiz Salari, Verification of Computer Codes in Computational Science and Engineering

William L. Kocay and Donald L. Kreher, Graphs, Algorithms, and Optimization, Second Edition

Donald L. Kreher and Douglas R. Stinson, Combinatorial Algorithms: Generation Enumeration and Search

Hang T. Lau, A Java Library of Graph Algorithms and Optimization

C. C. Lindner and C. A. Rodger, Design Theory, Second Edition

San Ling, Huaxiong Wang, and Chaoping Xing, Algebraic Curves in Cryptography

Nicholas A. Loehr, Bijective Combinatorics

Toufik Mansour, Combinatorics of Set Partitions

Titles (continued)

Toufik Mansour and Matthias Schork, Commutation Relations, Normal Ordering, and Stirling Numbers

Alasdair McAndrew, Introduction to Cryptography with Open-Source Software

Elliott Mendelson, Introduction to Mathematical Logic, Fifth Edition

Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, Handbook of Applied Cryptography

Stig F. Mjølsnes, A Multidisciplinary Introduction to Information Security

Jason J. Molitierno, Applications of Combinatorial Matrix Theory to Laplacian Matrices of Graphs

Richard A. Mollin, Advanced Number Theory with Applications

Richard A. Mollin, Algebraic Number Theory, Second Edition

Richard A. Mollin, Codes: The Guide to Secrecy from Ancient to Modern Times

Richard A. Mollin, Fundamental Number Theory with Applications, Second Edition

Richard A. Mollin, An Introduction to Cryptography, Second Edition

Richard A. Mollin, Quadratics

Richard A. Mollin, RSA and Public-Key Cryptography

Carlos J. Moreno and Samuel S. Wagstaff, Jr., Sums of Squares of Integers

Gary L. Mullen and Daniel Panario, Handbook of Finite Fields

Goutam Paul and Subhamoy Maitra, RC4 Stream Cipher and Its Variants

Dingyi Pei, Authentication Codes and Combinatorial Designs

Kenneth H. Rosen, Handbook of Discrete and Combinatorial Mathematics

Yongtang Shi, Matthias Dehmer, Xueliang Li, and Ivan Gutman, Graph Polynomials

Douglas R. Shier and K.T. Wallenius, Applied Mathematical Modeling: A Multidisciplinary Approach

Alexander Stanoyevitch, Introduction to Cryptography with Mathematical Foundations and Computer Implementations

Jörn Steuding, Diophantine Analysis

Douglas R. Stinson, Cryptography: Theory and Practice, Third Edition

Roberto Tamassia, Handbook of Graph Drawing and Visualization

Roberto Togneri and Christopher J. deSilva, Fundamentals of Information Theory and Coding Design

W. D. Wallis, Introduction to Combinatorial Designs, Second Edition

W. D. Wallis and J. C. George, Introduction to Combinatorics

Jiacun Wang, Handbook of Finite State Based Models and Applications

Lawrence C. Washington, Elliptic Curves: Number Theory and Cryptography, Second Edition

DISCRETE MATHEMATICS AND ITS APPLICATIONS

GRAPHS, ALGORITHMS, AND OPTIMIZATION

SECOND EDITION

WILLIAM L. KOCAY
University of Manitoba
Winnipeg, Canada

DONALD L. KREHER
Michigan Technological University
Houghton, USA

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20160727

International Standard Book Number-13: 978-1-4822-5116-6 (Hardback)

The authors would like to take this opportunity to express their appreciation and gratitude to the following people who have had a very significant effect on their mathematical development:

Adrian Bondy, Earl Kramer, Spyros Magliveras, Ron Read, and Ralph Stanton.

This book is dedicated to the memory of
William T. Tutte, (1917–2002)
“ the greatest of the graphmen ”

Contents

Preface

Our objective in writing this book is to present the theory of graphs from an algorithmic viewpoint. We present the graph theory in a rigorous, but informal style and cover most of the main areas of graph theory. The ideas of surface topology are presented from an intuitive point of view. We have also included a discussion on linear programming that emphasizes problems in graph theory. The text is suitable for students in computer science or mathematics programs.

Graph theory is a rich source of problems and techniques for programming and data structure development, as well as for the theory of computing, including NP-completeness and polynomial reduction.

This book could be used a textbook for a third or fourth year course on graph algorithms which contains a programming content, or for a more advanced course at the fourth year or graduate level. It could be used in a course in which the programming language is any major programming language (e.g., C, C++, Java). The algorithms are presented in a generic style and are not dependent on any particular programming language.

The text could also be used for a sequence of courses like "Graph Algorithms I" and "Graph Algorithms II". The courses offered would depend on the selection of chapters included. A typical course will begin with Chapters 1, 2, 4, and 5. At this point, a number of options are available.

A possible first course would consist of Chapters 1, 2, 4, 5, 7, 10, 11, 12, 13, and 14, and a first course stressing optimization would consist of Chapters 1, 2, 3, 5, 10, 11, 12, 17, 18, and 19. Experience indicates that the students consider these substantial courses. One or two chapters could be omitted for a lighter course.

We would like to thank the many people who provided encouragement while we wrote this book, pointed out typos and errors, and gave useful suggestions. In particular, we would like to convey our thanks to Ben Li and John van Rees of the University of Manitoba for proofreading some chapters.

William Kocay
Donald L. Kreher
August, 2004

Preface to the second edition

The second edition of Graphs, Algorithms, and Optimization contains three completely new chapters. New material has also been added to previously existing chapters. There is a new chapter on subgraph counting containing identities connecting various kinds of subgraphs in a graph. The graph reconstruction problem is introduced in this chapter. There is a chapter on graphs and symmetries, where the relation of permutation groups to graphs is considered. This chapter contains the basic theory of permutation groups. In particular, groups are used to construct symmetric graphs, and to understand self-complementary graphs and graphs with pseudo-similar vertices. A chapter on graph embeddings on the Klein bottle and double torus has also been added.

Some of the new material added to previously existing chapters is :

- A proof of the Erdös-Gallai theorem has been included;
- Sections on isometries of surfaces have been added, according to their application to graph embeddings in the plane and torus;
- The treatment of the double cover of graphs on the projective plane has been expanded;
- Automorphism groups of graph embeddings have been included;
- The proof of the NP-completeness of 3-coloring a graph has been corrected;
- The algorithm using Vizing's technique for edge-coloring a graph has been improved;
- A section on modules in graphs and digraphs has been added;
- Edmonds' matching algorithm using blossoms now has a section;
- Nowhere-zero flows are introduced;
- The use of coordinate averaging to produce nice drawings of graphs on surfaces is introduced;
- The basics of graph homomorphisms are now presented.

Also all the diagrams in the text have been redrawn and enhanced. We convey our thanks and gratitude to Andrei Gagarin for his help in the proof-reading of this second edition.

William Kocay
Donald L. Kreher
August, 2016

William Kocay obtained his Ph.D. in Combinatorics and Optimization from the University of Waterloo in 1979. He is currently a member of the Computer Science Department, and an adjunct member of the Mathematics Department, at the University of Manitoba, and a member of St. Paul's College, a college affiliated with the University of Manitoba. He has published numerous research papers, mostly in graph theory and algorithms for graphs. He was managing editor of the mathematics journal *Ars Combinatoria* from 1988 to 1997. He is currently on the editorial board of that journal. He has had extensive experience developing software for graph theory and related mathematical structures.

Donald L. Kreher obtained his Ph.D. from the University of Nebraska in 1984. He has held academic positions at Rochester Institute of Technology and the University of Wyoming. He is currently a University Professor of Mathematical Sciences at Michigan Technological University, where he teaches and conducts research in combinatorics and combinatorial algorithms. He has published numerous research papers and is a co-author of the internationally acclaimed text "Combinatorial Algorithms: Generation Enumeration and Search", CRC Press, 1999. He serves on the editorial boards of two journals.

Professor Kreher is the sole recipient of the 1995 Marshall Hall Medal, awarded by the Institute of Combinatorics and its Applications.

1

Graphs and Their Complements

1.1 Introduction

The diagram in Figure 1.1 illustrates a graph. It is called the graph of the cube. The edges of the geometric cube correspond to the line segments connecting the nodes in the graph, and the nodes correspond to the corners of the cube where the edges meet. They are the vertices of the cube.

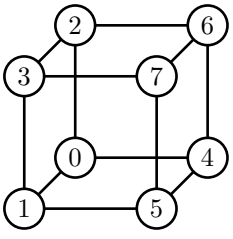


FIGURE 1.1
The graph of a cube

This diagram is drawn so as to resemble a cube, but if we were to rearrange it, as in Figure 1.2, it would still be the graph of the cube, although it would no longer look like a cube. Thus, a graph is a graphical representation of a relation in which edges connect pairs of vertices.

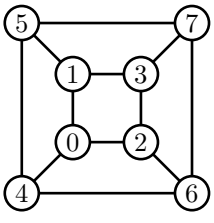


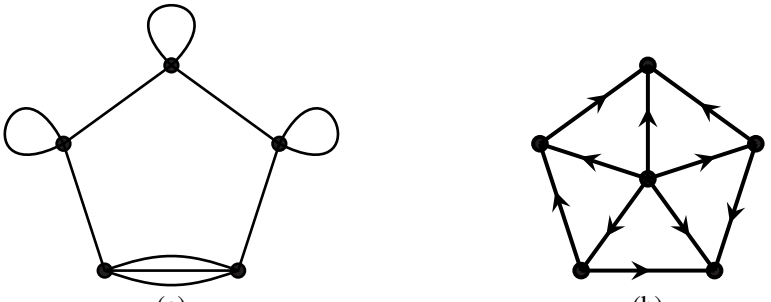
FIGURE 1.2
The graph of the cube

DEFINITION 1.1: A *simple graph* G consists of a *vertex set* $V(G)$ and an *edge set* $E(G)$, where each edge is a pair $\{u, v\}$ of vertices $u, v \in V(G)$.

We denote the set of all pairs of a set V by $\binom{V}{2}$. Then $E(G) \subseteq \binom{V(G)}{2}$. In the example of the cube, $V(G) = \{0, 1, 2, 3, 4, 5, 6, 7\}$, and $E(G) = \{01, 13, 23, 02, 45, 57, 67, 46, 15, 37, 26, 04\}$, where we have used the shorthand notation uv to stand for the pair $\{u, v\}$. If $u, v \in V(G)$, then $u \longrightarrow v$ means that u is joined to v by an edge. We say that u and v are *adjacent*. We use this notation to remind us of the linked list data structure that we will use to store a graph in the computer. Similarly, $u \not\longrightarrow v$ means that u is not joined to v. We can also express these relations by writing $uv \in E(G)$ or $uv \notin E(G)$, respectively. Note that in a simple graph if $u \longrightarrow v$, then $v \longrightarrow u$. If u is adjacent to each of $u_1, u_2, \ldots, u_k$, then we write $u \longrightarrow \{u_1, u_2, \ldots, u_k\}$.

These graphs are called *simple* graphs because each pair u, v of vertices is joined by at most one edge. Sometimes we need to allow several edges to join the same pair of vertices. Such a graph is also called a *multigraph*. An edge can then no longer be defined as a pair of vertices, (or the multiple edges would not be distinct), but to each edge there still corresponds a pair $\{u, v\}$. We can express this formally by saying that a graph G consists of a vertex set $V(G)$, an edge set $E(G)$, and a correspondence $\psi : E(G) \to \binom{V(G)}{2}$. Given an edge $e \in E(G)$, $\psi(e)$ is a pair $\{u, v\}$ which are the *endpoints* of e. Different edges can then have the same endpoints. We shall use simple graphs most of the time, which is why we prefer the simpler definition, but many of the theorems and techniques will apply to multigraphs as well.

This definition can be further extended to graphs with *loops* as well. A loop is an edge in which both endpoints are equal. We can include this in the general definition of a graph by making the mapping $\psi : E(G) \to \binom{V(G)}{2} \cup V(G)$. An edge $e \in E(G)$ for which $\psi(e) = u \in V(G)$ defines a loop. Figure 1.3(a) shows a graph with multiple edges and loops. However, we shall use simple graphs most of the time, so that an edge will be considered to be a pair of vertices.



(a) (b)

FIGURE 1.3
A multigraph (a) and a digraph (b)

A *directed graph* or *digraph* has edges which are *ordered* pairs (u, v) rather than unordered pairs $\{u, v\}$. In this case an edge is also called an *arc*. The direction of an edge is indicated by an arrow in diagrams, as in Figure 1.3(b).

The number of vertices of a graph G is denoted $|G|$. It is called the *order* of G. The number of edges is $\varepsilon(G)$. If G is simple, then obviously $\varepsilon(G) \leq \binom{|G|}{2}$, because $E(G) \subseteq \binom{V(G)}{2}$. We shall often use *node* or *point* as synonyms for vertex.

Many graphs have special names. The *complete graph* K_n is a simple graph with $|K_n| = n$ and $\varepsilon = \binom{n}{2}$. The *empty graph* $\overline{K}_n$ is a graph with $|\overline{K}_n| = n$ and $\varepsilon = 0$. $\overline{K}_n$ is the complement of K_n.

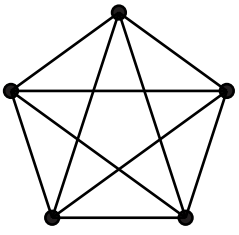


FIGURE 1.4
The complete graph $\boldsymbol{K_5}$

DEFINITION 1.2: Let G be a simple graph. The *complement* of G is $\overline{G}$, where $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \binom{V(G)}{2} \setminus E(G)$.

$E(\overline{G})$ consists of all those pairs uv which are not edges of G. Thus, $uv \in E(\overline{G})$ if and only if $uv \notin E(G)$. Figure 1.5 shows a graph and its complement.

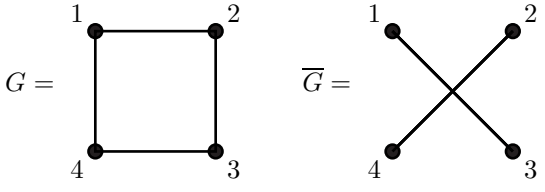


FIGURE 1.5
A graph and its complement

Figure 1.6 shows another graph and its complement. Notice that in this case, when $\overline{G}$ is redrawn, it looks identical to G.

In a certain sense, this G and $\overline{G}$ are the same graph. They are not equal, because $E(G) \neq E(\overline{G})$, but it is clear that they have the same structure. If two graphs have the same structure, then they can only differ in the names of the vertices. Therefore, we can rename the vertices of one to make it exactly equal to the other graph. In the

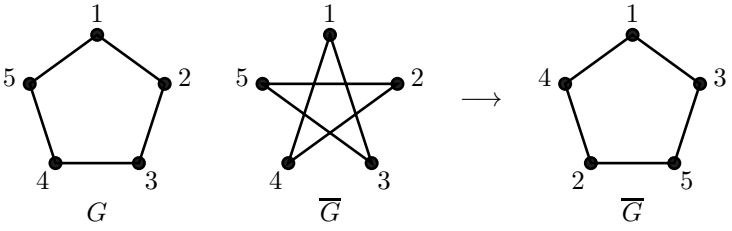


FIGURE 1.6
Another graph and its complement

Figure 1.6 example, we can rename the vertices of G by the mapping θ given by

k:	1	2	3	4	5
$\theta(k)$:	1	3	5	2	4

then $\theta(G)$ would equal $\overline{G}$. This kind of equivalence of graphs is known as *isomorphism*. Observe that a one-to-one mapping θ of the vertices of a graph G can be extended to a mapping of the edges of G by defining $\theta(\{u, v\}) = \{\theta(u), \theta(v)\}$.

DEFINITION 1.3: Let G and H be simple graphs. G and H are *isomorphic* if there is a one-to-one correspondence $\theta : V(G) \to V(H)$ such that $\theta(E(G)) = E(H)$, where $\theta(E(G)) = \{\theta(uv) : uv \in E(G)\}$.

We write $G \cong H$ to denote isomorphism. If $G \cong H$, then $uv \in E(G)$ if and only if $\theta(uv) \in E(H)$. One way to determine whether $G \cong H$ is to try and redraw G so as to make it look identical to H. We can then read off the mapping θ from the diagram. However, this is limited to small graphs. For example, the two graphs G and H shown in Figure 1.7 are isomorphic, because the drawing of G can be transformed into H by first moving vertex 2 to the bottom of the diagram, and then moving vertex 5 to the top. Comparing the two diagrams then gives the mapping

k:	1	2	3	4	5	6
$\theta(k)$:	6	4	2	5	1	3

as an isomorphism.

It is usually more difficult to determine when two graphs G and H are not isomorphic than to find an isomorphism when they are isomorphic. One way is to find a portion of G that cannot be part of H. For example, the graph H of Figure 1.7 is not isomorphic to the graph of the prism, which is illustrated in Figure 1.8, because the prism contains a triangle, whereas H has no triangle. A *subgraph* of a graph G is a graph K such that $V(K) \subseteq V(G)$ and $E(K) \subseteq E(G)$. If $\theta : G \to H$ is a possible isomorphism, then $\theta(K)$ will be a subgraph of H which is isomorphic to K.

A subgraph K is an *induced subgraph* if for every $u, v \in V(K) \subseteq V(G)$, $uv \in E(K)$ if and only if $uv \in E(G)$. That is, we choose a subset $U \subseteq V(G)$ and all

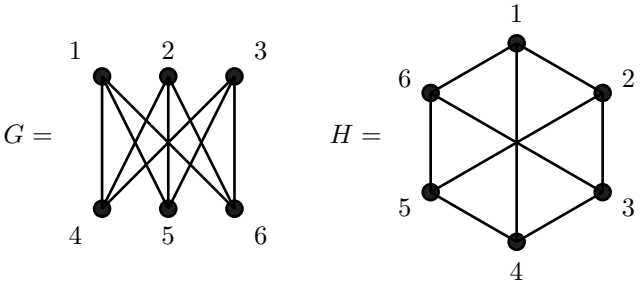


FIGURE 1.7
Two isomorphic graphs

edges uv with both endpoints in U. We can also form an *edge subgraph* or *partial subgraph* by choosing a subset of $E(G)$ as the edges of a subgraph K. Then $V(K)$ will be all vertices which are an endpoint of some edge of K.

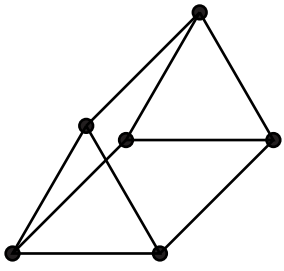


FIGURE 1.8
The graph of the prism

The *degree* of a vertex $u \in V(G)$ is $\text{DEG}(u)$, the number of edges which contain u. If $k = \text{DEG}(u)$ and $u \longrightarrow \{u_1, u_2, \ldots, u_k\}$, then $\theta(u) \longrightarrow \{\theta(u_1), \theta(u_2), \ldots, \theta(u_k)\}$, so that $\text{DEG}(u) = \text{DEG}(\theta(u))$. Therefore a necessary condition for G and H to be isomorphic is that they have the same set of degrees. The examples of Figures 1.7 and 1.8 show that this is not a sufficient condition.

In Figure 1.6, we saw an example of a graph G that is isomorphic to its complement. There are many such graphs.

DEFINITION 1.4: A simple graph G is *self-complementary* if $G \cong \overline{G}$.

Lemma 1.1. *If G is a self-complementary graph, then $|G| \equiv 0$ or $1 \pmod 4$.*

Proof. If $G \cong \overline{G}$, then $\varepsilon(G) = \varepsilon(\overline{G})$. But $E(\overline{G}) = \binom{V(G)}{2} \setminus E(G)$, so that $\varepsilon(\overline{G}) = \binom{|G|}{2} - \varepsilon(G) = \varepsilon(G)$, so $\varepsilon(G) = \frac{1}{2}\binom{|G|}{2} = |G|(|G|-1)/4$. Now $|G|$ and $|G|-1$

TABLE 1.1
Graphs up to 10 vertices

n	No. graphs
1	1
2	2
3	4
4	11
5	34
6	156
7	1,044
8	12,346
9	247,688
10	12,005,188

are consecutive integers, so that one of them is odd. Therefore $|G| \equiv 0 \pmod 4$ or $|G| \equiv 1 \pmod 4$. □

So possible orders for self-complementary graphs are $4, 5, 8, 9, 12, 13, \ldots, 4k$, $4k + 1$, etc.

Exercises

1.1.1 The four graphs on three vertices in Figure 1.9 have 0, 1, 2, and 3 edges, respectively. Every graph on three vertices is isomorphic to one of these four. Thus, there are exactly four different isomorphism types of graphs on three vertices.

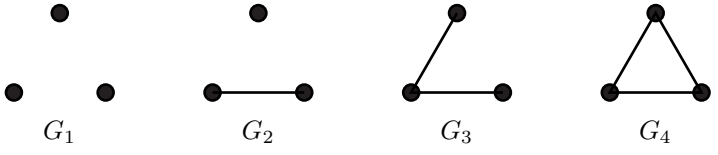


FIGURE 1.9
Four graphs on three vertices

Find all the different isomorphism types of graph on 4 vertices (there are 11 of them). *Hint*: Adding an edge to a graph with $\varepsilon = m$, gives a graph with $\varepsilon = m + 1$. Every graph with $\varepsilon = m + 1$ can be obtained in this way. Table 1.1 shows the number of isomorphism types of graphs up to 10 vertices.

1.1.2 Determine whether the two graphs shown in Figure 1.10 are isomorphic to each other or not. If they are isomorphic, find an explicit isomorphism.

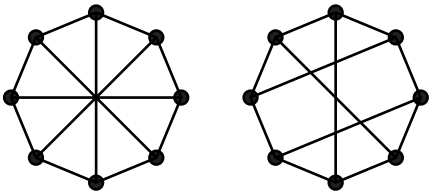


FIGURE 1.10
Two graphs on eight vertices

1.1.3 Determine whether the three graphs shown in Figure 1.11 are isomorphic to each other or not. If they are isomorphic, find explicit isomorphisms.

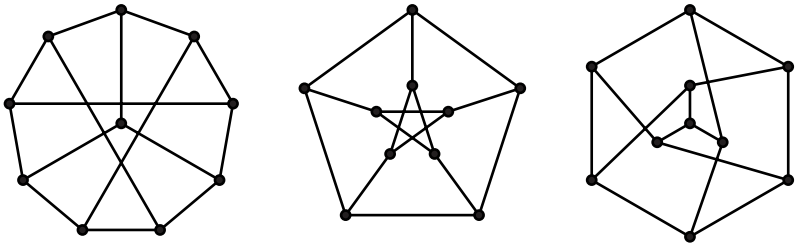


FIGURE 1.11
Three graphs on 10 vertices

1.1.4 Find a self-complementary graph on four vertices.

1.1.5 Figure 1.6 illustrates a self-complementary graph, the pentagon, with five vertices. Find another self-complementary graph on five vertices.

1.1.6 We have seen that the pentagon is a self-complementary graph. Let G be the pentagon shown in Figure 1.6, with $V(G) = \{u_1, u_2, u_3, u_4, u_5\}$. Notice that $\theta = (u_1)(u_2, u_3, u_5, u_4)$ is a permutation which maps G to $\overline{G}$; that is, $\theta(G) = \overline{G}$, and $\theta(\overline{G}) = G$. θ is called a *complementing* permutation. Because $u_2u_3 \in E(G)$, it follows that $\theta(u_2u_3) = u_3u_5 \in E(\overline{G})$. Consequently, $\theta(u_3u_5) = u_5u_4 \in E(G)$ again. Applying θ twice more gives $\theta(u_5u_4) = u_4u_2 \in E(\overline{G})$ and $\theta(u_4u_2) = u_2u_3$, which is where we started. Thus, if we choose any edge u_iu_j and successively apply θ to it, we alternately get edges of G and $\overline{G}$. It follows that the number of edges in the sequence so-obtained must be even. Use the permutation (1,2,3,4)(5,6,7,8) to construct a self-complementary graph on eight vertices.

1.1.7 Can the permutation (1,2,3,4,5)(6,7,8) be used as a complementing permutation? Can (1,2,3,4,5,6)(7,8) be? Prove that the only requirement is that every sequence of edges obtained by successively applying θ be of even length.

1.1.8 If θ is any permutation of $\{1, 2, \ldots, n\}$, then it depends only on the cycle structure of θ whether it can be used as a complementing permutation. Discover what condition this cycle structure must satisfy, and prove it both necessary and sufficient for θ to be a complementing permutation.

1.2 Degree sequences

Theorem 1.2. *For any simple graph G we have*

$$\sum_{u \in V(G)} \text{DEG}(u) = 2\varepsilon(G).$$

Proof. An edge uv has two endpoints. Therefore each edge will be counted twice in the summation, once for u and once for v. □

We use $\delta(G)$ to denote the *minimum degree* of G; that is, $\delta(G) = \text{MIN}\{\text{DEG}(u) \mid u \in V(G)\}$. $\Delta(G)$ denotes the *maximum degree* of G. By Theorem 1.2, the average degree equals $2\varepsilon/|G|$, so that $\delta \leq 2\varepsilon/|G| \leq \Delta$.

Corollary 1.3. *The number of vertices of odd degree is even.*

Proof. Divide $V(G)$ into $V_{odd} = \{u \mid \text{DEG}(u) \text{ is odd }\}$, and $V_{even} = \{u \mid \deg(u) \text{ is even }\}$. Then $2\varepsilon = \sum_{u \in V_{odd}} \text{DEG}(u) + \sum_{u \in V_{even}} \text{DEG}(u)$. Clearly 2ε and $\sum_{u \in V_{even}} \text{DEG}(u)$ are both even. Therefore, so is $\sum_{u \in V_{odd}} \text{DEG}(u)$, which means that $|V_{odd}|$ is even. □

DEFINITION 1.5: A graph G is a *regular* graph if all vertices have the same degree. G is *k-regular* if it is regular, of degree k.

For example, the graph of the cube (Figure 1.1) is 3-regular.

Lemma 1.4. *If G is simple and $|G| \geq 2$, then there are always two vertices of the same degree.*

Proof. In a simple graph, the maximum degree $\Delta \leq |G| - 1$. If all degrees were different, then they would be $0, 1, 2, \ldots, |G| - 1$. But degree 0 and degree $|G| - 1$ are mutually exclusive. Therefore there must be two vertices of the same degree. □

Let $V(G) = \{u_1, u_2, \ldots, u_n\}$. The *degree sequence* of G is

$$\text{DEG}(G) = (\text{DEG}(u_1), \text{DEG}(u_2), \ldots, \text{DEG}(u_n))$$

where the vertices are ordered so that

$$\text{DEG}(u_1) \geq \text{DEG}(u_2) \geq \cdots \geq \text{DEG}(u_n).$$

Sometimes it is useful to construct a graph with a given degree sequence. For example, can there be a simple graph with five vertices whose degrees are $(4, 3, 3, 2, 1)$? Because there are three vertices of odd degree, Corollary 1.3 tells us that there is no such graph. We say that a sequence

$$D = (d_1, d_2, \ldots, d_n),$$

is *graphic* if

$$d_1 \geq d_2 \geq \cdots \geq d_n,$$

and there is a simple graph G with $\mathrm{DEG}(G) = D$. So $(2, 2, 2, 1)$ and $(4, 3, 3, 2, 1)$ are not graphic, whereas $(2, 2, 1, 1)$, $(4, 3, 2, 2, 1)$, and $(2, 2, 2, 2, 2, 2)$ clearly are.

Problem 1.1:	Graphic
Instance:	a sequence $D = (d_1, d_2, \ldots, d_n)$.
Question:	is D graphic?
Find:	a graph G with $\mathrm{DEG}(G) = D$, if D is graphic.

For example, $(7, 6, 5, 4, 3, 3, 2)$ is not graphic; for any graph G with this degree sequence has $\Delta(G) = |G| = 7$, which is not possible in a simple graph. Similarly, $(6, 6, 5, 4, 3, 3, 1)$ is not graphic; here we have $\Delta(G) = 6$, $|G| = 7$ and $\delta(G) = 1$. But because two vertices have degree $|G| - 1 = 6$, it is not possible to have a vertex of degree one in a simple graph with this degree sequence.

When is a sequence graphic? We want a construction which will find a graph G with $\mathrm{DEG}(G) = D$, if the sequence D is graphic.

One way is to join up vertices arbitrarily. This does not always work, because we can get stuck, even if the sequence is graphic. The following algorithm always produces a graph G with $\mathrm{DEG}(G) = D$, if D is graphic.

procedure GRAPHGEN(D)
Create vertices $u_1, u_2, \ldots, u_n$
comment: upon completion, u_i will have degree $D[i]$

graphic $\leftarrow$ **false** "assume not graphic"
$i \leftarrow 1$
while $D[i] > 0$
do
- k←D[i]
- **if** there are at least k vertices with DEG > 0
  - **then**
    - join u_i to the k vertices of largest degree
    - decrease each of these degrees by 1
    - $D[i] \leftarrow 0$
    - **comment:** vertex u_i is now completely joined
  - **else exit** "u_i cannot be joined"
- $i \leftarrow i + 1$

graphic $\leftarrow$ **true**

This uses a reduction. For example, given the sequence

$$D = (3, 3, 3, 3, 3, 3),$$

the first vertex will be joined to the three vertices of largest degree, which will then reduce the sequence to $(*, 3, 3, 2, 2, 2)$, because the vertex marked by an asterisk is now completely joined, and three others have had their degree reduced by 1. At the next stage, the first remaining vertex will be joined to the three vertices of largest degree, giving a new sequence $(*, *, 2, 2, 1, 1)$. Two vertices are now completely joined. At the next step, the first remaining vertex will be joined to two vertices, leaving $(*, *, *, 1, 1, 0)$. The next step joins the two remaining vertices with degree one, leaving a sequence $(*, *, *, *, 0, 0)$ of zeroes, which we know to be graphic.

In general, given the sequence

$$D = (d_1, d_2, \ldots, d_n),$$

where

$$d_1 \geq d_2 \geq \cdots \geq d_n,$$

the vertex of degree d_1 is joined to the d_1 vertices of largest degree. This leaves the numbers

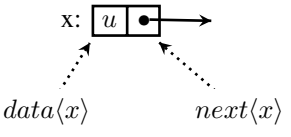
$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n,$$

in some order. If we rearrange them into descending order, we get the reduced sequence D'. Write

$$D' = (d'_2, d'_3 \ldots, d'_n),$$

where the first vertex u_1 has been deleted. We now do the same calculation, using D' in place of D. Eventually, after joining all the vertices according to their degree, we either get a graph G with $\mathrm{Deg}(G) = D$ or else at some stage, it is impossible to join some vertex u_i.

An excellent data structure for representing the graph G for this problem is to have an *adjacency list* for each vertex $v \in V(G)$. The adjacency list for a vertex $v \in V(G)$ is a linked list of the vertices adjacent to v. Thus it is a data structure in which the vertices adjacent to v are arranged in a linear order. A node x in a linked list has two fields: $\mathit{data}\langle x\rangle$, and $\mathit{next}\langle x\rangle$.



Given a node x in the list, $\mathit{data}\langle x\rangle$ is the data associated with x and $\mathit{next}\langle x\rangle$ points to the successor of x in the list or $\mathit{next}\langle x\rangle = \mathrm{NIL}$ if x has no successor. We can insert data u into the list pointed to by L with procedure LISTINSERT(), and the first node on list L can be removed with procedure LISTREMOVEFIRST().

procedure LISTINSERT($pseudocode$)
$L, ux \leftarrow$ NEWNODE()
$data\langle x\rangle \leftarrow u$
$next\langle x\rangle \leftarrow L$
$L \leftarrow x$

procedure LISTREMOVEFIRST(L)
$x \leftarrow L$
$L \leftarrow next\langle x\rangle$
FREENODE(x)

We use an array $AdjList[\cdot]$ of linked lists to store the graph. For each vertex $v \in V(G)$, $AdjList[v]$ points to the head of the adjacency lists for v. This data structure is illustrated in Figure 1.12.

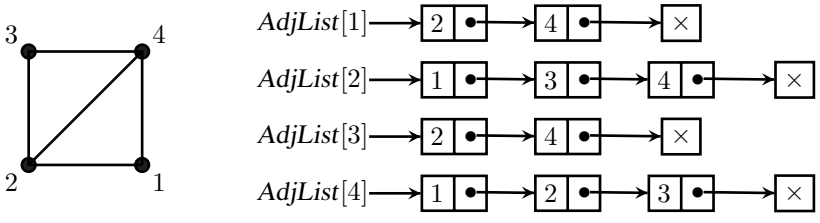


FIGURE 1.12
Adjacency lists of a graph

We can use another array of linked lists, $Pts[k]$, being a linked list of the vertices u_i whose degree-to-be $d_i = k$. With this data structure, Algorithm 1.2.1 can be written as follows:

Algorithm 1.2.1: GRAPHGEN(D)

comment: Assume D is not graphic.
comment: Create and initialize the linked lists $Pts[k]$.
graphic $\leftarrow$ **false**
for $k \leftarrow 0$ **to** $n-1$ **do** $Pts[k] \leftarrow$ NIL
for $k \leftarrow 1$ **to** n **do** LISTINSERT($Pts[D[k]], k$)
comment: Begin with vertex of largest degree.
for $k \leftarrow n-1$ **downto** 0
do while $Pts[k] \neq$ NIL
do
- **comment:** These points are to have degree k.
- $x \leftarrow Pts[k]$
- $u \leftarrow data\langle x\rangle$
- LISTREMOVEFIRST($Pts[k]$)
- **comment:** Join u to the next k vertices v of largest degree.
- **comment:** If this is not possible, then D is not graphic so exit.
- $i \leftarrow k$
- **for** $j \leftarrow 1$ **to** k
- **do**
  - **while** $Pts[i] =$ NIL **do** $i \leftarrow i-1$
  - **if** $i = 0$ **exit**
  - $x = Pts[i]$
  - $v = data\langle x\rangle$
  - LISTREMOVEFIRST($Pts[i]$)
  - LISTINSERT($AdjList[u], v$)
  - LISTINSERT($AdjList[v], u$)
  - LISTINSERT($TempList[i], v$)
- **comment:** For each such v joined to u if v is on list $Pts[j]$,
- **comment:** then transfer v to $Pts[j-1]$
- **for** $j \leftarrow k$ **downto** 1
- **do**
  - **while** $TempList[j] \neq$ NIL
  - **do**
    - $x = TempList[j]$
    - $v = data\langle x\rangle$
    - LISTREMOVEFIRST($TempList[j]$)
    - LISTINSERT($Pts[j-1], v$)
- **comment:** u is now completely joined. Choose the next point.

comment: Now every vertex has been successfully joined.
graphic $\leftarrow$ **true**

This program is illustrated in Figure 1.13 for the sequence $D = (4, 4, 2, 2, 2, 2)$, where $n = 6$. The diagram shows the linked lists before vertex 1 is joined to vertices 2, 3, 4, and 5, and the new configuration after joining. Care must be used in transferring the vertices v from $Pts[j]$ to $Pts[j-1]$, because we do not want to join u to v more than once. The purpose of the list $Pts[0]$ is to collect vertices which have been transferred from $Pts[1]$ after having been joined to u. The degrees $d_1, d_2, \ldots, d_n$

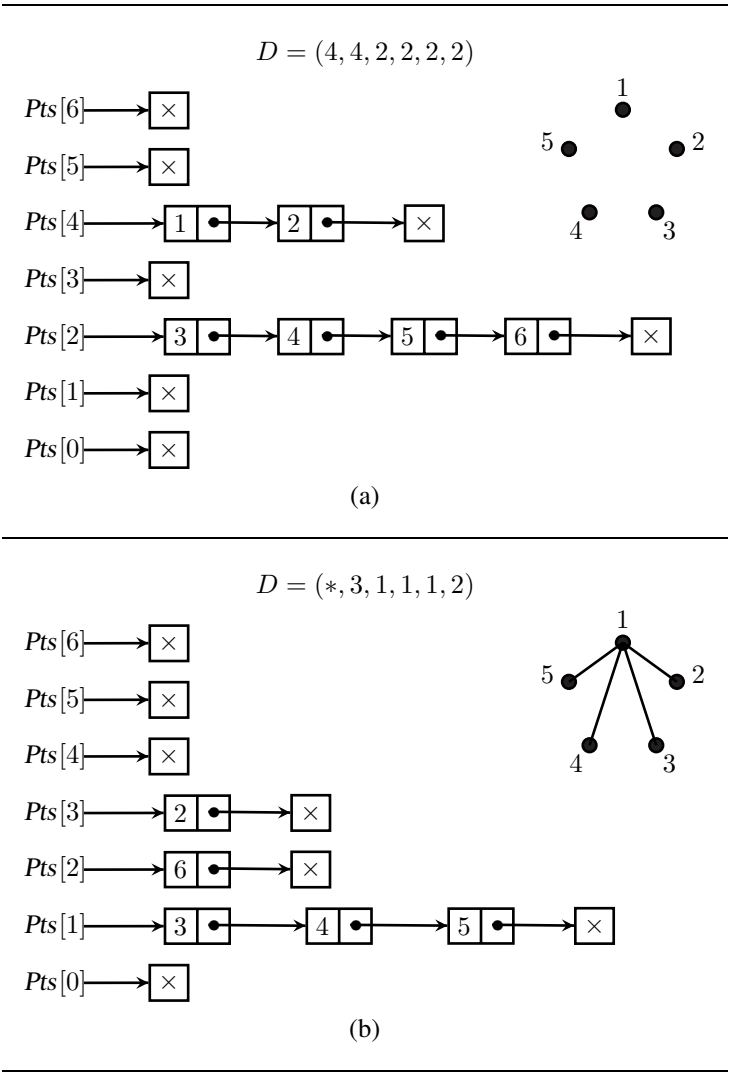


FIGURE 1.13
The linked lists $Pts[k]$. (a) Before 1 is joined to 2, 3, 4, and 5. (b) After 1 is joined to 2, 3, 4, and 5.

need not necessarily be in descending order for the program to work, because the points are placed in the lists $Pts[k]$ according to their degree, thereby sorting them into buckets. Upon completion of the algorithm vertex k will have degree d_k. However, when this algorithm is done by hand, it is much more convenient to begin with a sorted list of degrees; for example, $D = (4, 3, 3, 3, 2, 2, 2, 2, 1)$, where $n = 9$. We begin with vertex u_1, which is to have degree four. It will be joined to the vertices u_2, u_3, and u_4, all of degree three, and to one of u_5, u_6, u_7, and u_8, which have degree two. In order to keep the list of degrees sorted, we choose u_8. We then have $u_1 \longrightarrow \{u_2, u_3, u_4, u_8\}$, and D is reduced to $(*, 2, 2, 2, 2, 2, 2, 1, 1)$. We then choose u_2 and join it to u_6 and u_7, thereby further reducing D to $(*, *, 2, 2, 2, 2, 1, 1, 1, 1)$. Continuing in this way, we obtain a graph G.

In general, when constructing G by hand, when u_k is to be joined to one of u_i and u_j, where $d_i = d_j$ and $i < j$, then join u_k to u_j *before* u_i, in order to keep D sorted in descending order.

We still need to prove that Algorithm 1.2.1 works. It accepts a possible degree sequence

$$D = (d_1, d_2, \ldots d_n),$$

and joins u_1 to the d_1 vertices of largest remaining degree. It then reduces D to new sequence

$$D' = (d'_2, d'_3, \ldots d'_n).$$

1.2.1 Havel-Hakimi theorem

Theorem 1.5. (Havel-Hakimi theorem) *D is graphic if and only if D' is graphic.*

Proof. Suppose D' is graphic. Then there is a graph G' with degree sequence D', where $V(G') = \{u_2, u_3, \ldots, u_n\}$ with $\text{DEG}(u_i) = d'_i$. Furthermore

$$D' = (d'_2, d'_3, \ldots, d'_n)$$

consists of the degrees

$$\{d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n\}$$

arranged in descending order. Create a new vertex u_1 and join it to vertices of degree

$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1.$$

Then $\text{DEG}(u_1) = d_1$. Call the new graph G. Clearly the degree sequence of G is

$$D = (d_1, d_2, \ldots, d_n).$$

Therefore D is graphic.

Now suppose D is graphic. Then there is a graph G with degree sequence

$$D = (d_1, d_2, \ldots, d_n),$$

where $V(G) = \{u_1, u_2, \ldots, u_n\}$, with $\text{DEG}(u_i) = d_i$. If u_1 is adjacent to vertices

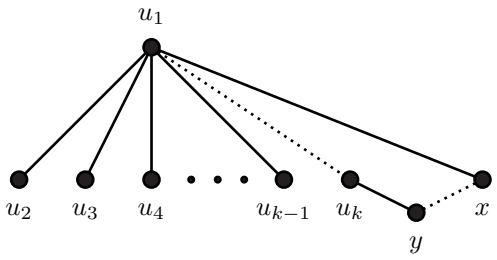


FIGURE 1.14
Vertices adjacent to u_1

of degree $d_2, d_3, \ldots, d_{d_1+1}$, then $G' = G - u_1$ has degree sequence D', in which case D' is graphic.

Otherwise, u_1 is not adjacent to vertices of degree $d_2, d_3, \ldots, d_{d_1+1}$. Let u_k (where $k \geq 2$) be the *first* vertex such that u_1 is not joined to u_k, but is joined to $u_2, u_3, \ldots, u_{k-1}$. (Maybe $k = 2$.)

Now $\text{DEG}(u_1) = d_1 \geq k$, so u_1 is joined to some vertex $x \neq u_2, u_3, \ldots u_{k-1}$. u_k is the vertex of next largest degree, so $\text{DEG}(u_k) \geq \text{DEG}(x)$. Now x is joined to u_1, while u_k is not. Therefore, there is some vertex y such that $u_k \longrightarrow y$ but $x \not\longrightarrow y$. Set $G \leftarrow G + xy + u_1u_k - u_1x - u_ky$.

The degree sequence of G has not changed, and now $u_1 \longrightarrow \{u_2, u_3, \ldots, u_k\}$. Repeat until $u_1 \longrightarrow \{u_2, u_3, \ldots, u_{d_1+1}\}$. Then $G' = G - u_1$ has degree sequence D', so that D' is graphic. (See Figure 1.14.)

□

Therefore we know the algorithm will terminate with the correct answer, because it reduces D to D'. So we have an algorithmic test to check whether D is graphic and to generate a graph whenever one exists.

1.2.2 Erdös-Gallai theorem

There is another way of determining whether D is graphic, without constructing a graph.

Theorem 1.6. (Erdös-Gallai theorem) *Let $D = (d_1, d_2, \ldots, d_n)$, where $d_1 \geq d_2 \geq \cdots \geq d_n$. Then D is graphic if and only if*

1. $\sum_{i=1}^{n} d_i$ *is even; and*
2. $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \text{MIN}(k, d_i)$, *for* $k = 1, 2, \ldots, n$.

Proof. Suppose D is graphic. Then $\sum_{i=1}^{n} d_i = 2\varepsilon$, which is even. Let V_1 contain the k vertices of largest degree, and let $V_2 = V(G) - V_1$ be the remaining vertices. See Figure 1.15.

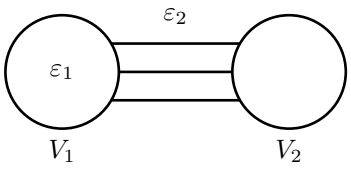


FIGURE 1.15
The vertices V_1 of largest degree and the remaining vertices v_2

Suppose that there are ε_1 edges within V_1 and ε_2 edges from V_1 to V_2. Then $\sum_{i=1}^{k} d_i = 2\varepsilon_1 + \varepsilon_2$, because each edge within V_1 is counted twice in the sum, once for each endpoint, but edges between V_1 and V_2 are counted once only. Now $\varepsilon_1 \leq \binom{k}{2}$, because V_1 can induce a complete subgraph at most. Each vertex $v \in V_2$ can be joined to at most k vertices in V_1, because $|V_1| = k$, but v can be joined to at most $\text{DEG}(v)$ vertices in V_1, if $\text{DEG}(v) < k$. Therefore ε_2, the number of edges between V_1 and V_2, is at most $\sum_{v \in V_2} \text{MIN}(k, \text{DEG}(v))$, which equals $\sum_{i=k+1}^{n} \text{MIN}(k, d_i)$. This now gives $\sum_{i=1}^{k} d_i = 2\varepsilon_1 + \varepsilon_2 \leq k(k-1) + \sum_{i=k+1}^{n} \text{MIN}(k, d_i)$.

Various proofs of the converse are available. Interesting proofs can be found in the books by HARARY [80] or BERGE [14]. Here we outline a proof by CHOUDUM [33]. The proof is by induction on $S = \sum_{i=1}^{n} d_i$. If $S = 2$, it is clear that the result is true. Without loss of generality, we can assume that $d_n \geq 1$. Let t be the smallest integer such that $d_t > d_{t+1}$, if there is one. Otherwise, if all d_i are equal, we take $t = n - 1$.

Construct $D' = (d'_1, \ldots, d'_n)$ from D as follows: if $i \neq t$ and $i \neq n$, then $d'_i = d_i$; if $i = t$ or $i = n$, then $d'_i = d_i - 1$. That is, we are looking for a graph with degree sequence D in which vertex t is adjacent to vertex n. Then $S' = S - 2$. If we can verify that D' satisfies the conditions of the theorem, with corresponding graph G', we can then construct a graph G with degree sequence D.

Consider $S_k = \sum_{i=1}^{k} d_i$ and $S'_k = \sum_{i=1}^{k} d'_i$. Let $T_k = k(k-1) + \sum_{i=k+1}^{n} \text{MIN}(d_i, k)$, and similarly for T'_k. If $k \geq t$, then $S'_k = S_k - 1 \leq T_k - 1 \leq T'_k$. Thus the conditions of the theorem are satisfied when $k \geq t$. If $k < t$ there are several cases to consider. Note that when $k < t$, $d_1 = d_2 = \ldots = d_k$, so that $S'_k = S_k = kd_k$.

If $k < t$ and $d_k < k$, then $S'_k = kd_k \leq k(k-1) \leq T'_k$.

If $k < t$ and $d_k = k$, then $S'_k = kd_k = k^2 = k(k-1) + k$. Now d'_{k+1} is either k or $k-1$. Therefore when $i > k$, $\text{MIN}(d'_i, k) = d'_i$. If $d'_{k+1} = k$, then $S'_k = k(k-1) + d'_{k+1} \leq T'_k$. Otherwise $d'_{k+1} = k-1$, and $S'_k = k(k-1) + d'_{k+1} + 1$. If $d'_{k+2} \geq 1$, we obtain $S'_k \leq T'_k$. Otherwise $n = k+2$ and $t = n-1$, so that $d_n = 1$, giving $S = (n-2)^2 + (n-2) + 1$, which must be even, so that n is odd. But then all degrees are odd, which is impossible.

If $k < t$ and $d_k > k$ and $d_n > k$, then $\text{MIN}(d_i, k) = \text{MIN}(d'_i, k) = k$ when $i > k$. $S'_k = kd_k = S_k \leq T_k = T'_k$.

If $k < t$ and $d_k > k$ and $d_n \leq k$, let r be the first integer such that $d_r \leq k$.

Then $r > t$ and $\text{MIN}(d_i, k) = d_i$ when $i \geq r$, so that $S'_k = S_k = kd_k \leq T_k = k(k-1) + k(r-k-1) + d_r + \ldots + d_n = k(r-2) + (d_r + \ldots + d_n)$. We have $T'_k = T_k - 1$.

Now $S_{k+1} = (k+1)d_k \leq T_{k+1} = (k+1)k + (k+1)(r-k-2) + d_r + \ldots + d_n$. Hence $d_k \leq (r-2) + (d_r + \ldots + d_n)/(k+1)$. Substituting this into the previous expression gives $kd_k \leq k(r-2) + \frac{k}{k+1}(d_r + \ldots + d_n) < T_k$. Therefore $S'_k = kd_k < T_k$. But $T'_k = T_k - 1$, so that $S'_k \leq T'_k$.

Thus, $S'_k \leq T'_k$ for all k. By induction we know that D' is graphic. Let G' be a simple graph with degree sequence D'. If vertices t and n are not adjacent in G', we add the edge $\{t, n\}$ to obtain a graph G with degree sequence D. If they are adjacent, then choose a vertex m such that $\{t, m\} \notin E(G')$. Because $d'_m \geq d'_n$, there is a vertex r adjacent to m in G' such that $\{r, n\} \notin E(G')$. Remove the edge $\{m, r\}$ and add the edges $\{t, m\}, \{r, n\}$ to obtain a graph G with degree sequence D.

□

Conditions 1 and 2 of Theorem 1.6 are known as the *Erdös-Gallai conditions*.

Exercises

1.2.1 Prove Theorem 1.2 for arbitrary graphs. That is, prove

Theorem 1.7. *For any graph G we have*

$$\sum_{u \in V(G)} Deg(u) + \ell = 2\varepsilon(G).$$

where ℓ is the number of loops in G and $\text{DEG}(u)$ is the number of edges incident on u. What formula is obtained if loops count two toward $\text{DEG}(u)$?

1.2.2 We know that a simple graph with n vertices has at least one pair of vertices of equal degree, if $n \geq 2$. Find all simple graphs with exactly one pair of vertices with equal degrees. What are their degree sequences? *Hint*: Begin with $n = 2, 3, 4$. Use a recursive construction. Can degree 0 or $n-1$ occur twice?

1.2.3 Program the GRAPHGEN() algorithm. Input the sequence $D = (d_1, d_2, \ldots, d_n)$ and then construct a graph with that degree sequence, or else determine that the sequence is not graphic. Use the following input data:

(a) 4 4 4 4 4
(b) 3 3 3 3 3 3
(c) 3 3 3 3 3 3 3 3
(d) 3 3 3 3 3 3 3 3 3
(e) 2 2 2 2 2 2 2 2 2 2
(f) 7 6 6 6 5 5 2 1

1.2.4 If G has degree sequence $D = (d_1, d_2, \ldots, d_n)$, what is the degree sequence of $\overline{G}$?

1.2.5 Let $D = (d_1, d_2, \ldots, d_n)$, where $d_1 \geq d_2 \geq \ldots \geq d_n$. Prove that there is a multigraph with degree sequence D if and only if $\sum_{i=1}^{n} d_i$ is even, and $d_1 \leq \sum_{i=2}^{n} d_i$.

1.3 Analysis

Let us estimate the number of steps that Algorithm 1.2.1 performs. Consider the loop structure

for $k \leftarrow n$ **downto** 1
  do while *Pts*$[k] \neq$ NIL
  do $\{\cdots$

The for-loop performs n iterations. For many of these iterations, the contents of the while-loop will not be executed, because *Pts*$[k]$ will be NIL. When the contents of the loop are executed, vertex u of degree-to-be k will be joined to k vertices. This means that k edges will be added to the adjacency lists of the graph G being constructed. This takes $2k$ steps, because an edge uv must be added to both *GraphAdj*$[u]$ and *GraphAdj*$[v]$. It also makes $\textsc{Deg}(u) = k$. When edge uv is added, v will be transferred from *Pts*$[j]$ to *Pts*$[j-1]$, requiring additional k steps. Once u has been joined, it is removed from the list. Write $\varepsilon = \frac{1}{2}\sum_i d_i$, the number of edges of G when D is graphic. Then, in all, the combination for-while-loop will perform exactly 2ε steps adding edges to the graph and a further ε steps transferring vertices to other lists, plus n steps for the n iterations of the for-loop. This gives a total of $3\varepsilon + n$ steps for the for-while-loop. The other work that the algorithm performs is to create and initialize the lists *Pts*$[\cdot]$, which takes $2n$ steps altogether. So we can say that in total, the algorithm performs $3\varepsilon + 3n$ steps.

Now it is obvious that each of these "steps" is composed of many other smaller steps, for there are various comparisons and assignments in the algorithm which we have not explicitly taken account of (they are subsumed into the steps we have explicitly counted). Furthermore, when compiled into assembly language, each step will be replaced by many smaller steps. Assembly language is in turn executed by the microprogramming of a computer, and eventually we come down to logic gates, flip-flops, and registers. Because of this fact, and because each computer has its own architecture and machine characteristics, it is customary to ignore the constant coefficients of the graph parameters ε and n, and to say that the algorithm has *order* $\varepsilon + n$, which is denoted by $O(\varepsilon + n)$, pronounced "big Oh of $\varepsilon + n$". A formal definition is provided by Definition 1.6. Even though the actual running time of a given algorithm depends on the architecture of the machine it is run on, the programmer can often make a reasonable estimate of the number of steps of some constant size (e.g., counting one assignment, comparison, addition, multiplication, etc. as one step), and

thereby obtain a formula like $3\varepsilon + 3n$. Such an algorithm will obviously be superior to one which takes $15\varepsilon + 12n$ steps of similar size. Because of this fact, we shall try to obtain formulas of this form whenever possible, as well as expressing the result in a form like $O(\varepsilon + n)$.

The *complexity* of an algorithm is the number of steps it must perform, in the worst possible case. That is, it is an *upper bound* on the number of steps. Because the size of each step is an unknown constant, formulas like $5n^2/6$ and $25n^2$ are both expressed as $O(n^2)$. We now give a formal definition of this notation.

DEFINITION 1.6: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that $f(n)$ is $O(g(n))$ provided that there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

In other words, $f(n)$ is $O(g(n))$ provided that $f(n)$ is bounded above by a constant factor times $g(n)$ for large enough n. For example, the function $5n^3 + 2n + 1$ is $O(n^3)$, because for all $n \geq 1$, we have

$$5n^3 + 2n + 1 \leq 5n^3 + 2n^3 + n^3 = 8n^3.$$

Hence, we can take $c = 8$ and $n_0 = 1$, and Definition 1.6 is satisfied.

The notation $f(n)$ is $\Omega(g(n))$ ("big omega") is used to indicate that $f(n)$ is bounded below by a constant factor times $g(n)$ for large enough n.

DEFINITION 1.7: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that $f(n)$ is $\Omega(g(n))$ provided that there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.

We say that $f(n)$ is $\Theta(g(n))$ ("big theta") when $f(n)$ is bounded above and below by constant factors times $g(n)$. The constant factors may be different. More precisely:

DEFINITION 1.8: Suppose $f : \mathbb{Z}^+ \to \mathbb{R}$ and $g : \mathbb{Z}^+ \to \mathbb{R}$. We say that $f(n)$ is $\Theta(g(n))$ provided that there exist constants $c, c' > 0$ and $n_0 \geq 0$ such that $0 \leq c \cdot g(n) \leq f(n) \leq c' \cdot g(n)$ for all $n \geq n_0$.

If $f(n)$ is $\Theta(g(n))$, then we say that f and g have the same *growth rate*.

The big O-notation is a method of indicating the *qualitative* nature of the formula, whether quadratic, linear, logarithmic, exponential, etc. Notice that "equations" involving $O(\cdot)$ are not really equations, because $O(\cdot)$ can only be used in this sense on the right hand side of the equals sign. For example, we could also have shown that $10n^2 + 4n - 4$ is $O(n^3)$ or that $10n^2 + 4n - 4$ is $O(2^n)$, but these expressions are not equal to each other. Given a complexity formula like $10n^2 + 4n - 4$, we want the smallest function $f(n)$ such that $10n^2 + 4n - 4$ is $O(f(n))$. Among the useful rules for working with the O-notation are the following sum and product rules.

Theorem 1.8. *Suppose that the two functions $f_1(n)$ and $f_2(n)$ are both $O(g(n))$. Then the function $f_1(n) + f_2(n)$ is $O(g(n))$.*

Theorem 1.9. *Suppose that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$. Then the function $f_1(n)\, f_2(n)$ is $O(g_1(n)\, g_2(n))$.*

As examples of the use of these notations, we have that n^2 is $O(n^3)$, n^3 is $\Omega(n^2)$, and $2n^2 + 3n - \sin n + 1/n$ is $\Theta(n^2)$.

Several properties of growth rates of functions that arise frequently in algorithm analysis follow. The first of these says that a polynomial of degree d, in which the high-order coefficient is positive, has growth rate n^d.

Theorem 1.10. *Suppose that* $a_d > 0$. *Then the function* $a_0 + a_1 n + \cdots + a_d n^d$ *is* $\Theta(n^d)$.

The next result says that *logarithmic growth* does not depend on the base to which logarithms are computed. It can be proved easily using the formula $\log_a n = \log_a b \cdot \log_b n$.

Theorem 1.11. *The function* $\log_a n$ *is* $\Theta(\log_b n)$ *for any* $a, b > 1$.

The next result can be proved using Stirling's formula. It gives the growth rate of the factorial function in terms of exponential functions.

Theorem 1.12. *The function* $n!$ *is* $\Theta(n^{n+1/2}e^{-n})$.

Exercises

1.3.1 Show that if G is a simple graph with n vertices and ε edges, then $\log \varepsilon = O(\log n)$.

1.3.2 Consider the following statements which count the number of edges in a graph, whose adjacency matrix is *Adj*.

```
Edges ← 0
for u ← 1 to n − 1
  do for v ← u + 1 to n
  do if Adj[u, v] = 1
  then Edges ← Edges + 1
```

Calculate the number of steps the algorithm performs. Then calculate the number of steps required by the following statements in which the graph is stored in adjacency lists:

```
Edges ← 0
for u ← 1 to n − 1
  do for each v ⟶ u
  do if u < v
  then Edges ← Edges + 1
```

What purpose does the condition $u < v$ fulfill, and how can it be avoided?

1.3.3 Use induction to prove that the following formulas hold:

(a) $1 + 2 + 3 + \cdots + n = \binom{n+1}{2}$

(b) $\binom{2}{2} + \binom{3}{2} + \binom{4}{2} + \cdots + \binom{n}{2} = \binom{n+1}{3}$.

(c) $\binom{t}{t} + \binom{t+1}{t} + \binom{t+2}{t} + \cdots + \binom{n}{t} = \binom{n+1}{t+1}$.

1.3.4 Show that $3n^2 + 12n = O(n^2)$; that is, find constants A and N such that $3n^2 + 12n \leq An^2$ whenever $n \geq N$.

1.3.5 Show that $\log(n+1) = O(\log n)$, where the logarithm is to base 2.

1.3.6 Use the answer to the previous question to prove that

$$(n+1)\log(n+1) = O(n \log n).$$

1.3.7 Prove that if $f_1(n)$ and $f_2(n)$ are both $O(g(n))$, then $f_1(n) + f_2(n)$ is $O(g(n))$.

1.3.8 Prove that if $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n)\, f_2(n)$ is $O(g_1(n)\, g_2(n))$.

1.4 Notes

Some good general books on graph theory are BERGE [14], BOLLOBÁS [20], BONDY and MURTY [23], CHARTRAND and LESNIAK [31], CHARTRAND and OELLERMANN [32], DIESTEL [44], GOULD [73], and WEST [189]. A very readable introductory book is TRUDEAU [172]. GIBBONS [66] is an excellent treatment of graph algorithms. A good book discussing the analysis of algorithms is PURDOM and BROWN [138]. AHO, HOPCROFT, and ULLMAN [1], SEDGEWICK [157] and WEISS [188] are all excellent treatments of data structures and algorithm analysis.

2

Paths and Walks

2.1 Introduction

Let u and v be vertices of a simple graph G. A *path* P from u to v is a sequence of vertices $u_0, u_1, \ldots, u_k$ such that $u = u_0$, $v = u_k$, $u_i \longrightarrow u_{i+1}$, and all the u_i are distinct vertices. The length of a path P is $\ell(P)$, the number of *edges* it uses. In this example, $\ell(P) = k$, and P is called a uv-path of length k. A uv-path of length 4 is illustrated in Figure 2.1, with dashed edges.

A *cycle* C is a sequence of vertices $u_0, u_1, \ldots, u_k$ forming a u_0u_k-path, such that $u_k \longrightarrow u_0$. The length of C is $\ell(C)$, the number of edges that it uses. In this case, $\ell(C) = k + 1$.

A uv-path P *connects* u to v. The set of all vertices connected to any vertex u forms a subgraph C_u, the *connected component* of G containing u. It will often be the case that C_u contains all of G, in which case G is a *connected graph*. $\omega(G)$ denotes the number of distinct connected components of G. The graph of Figure 2.1 is disconnected, with $\omega = 3$.

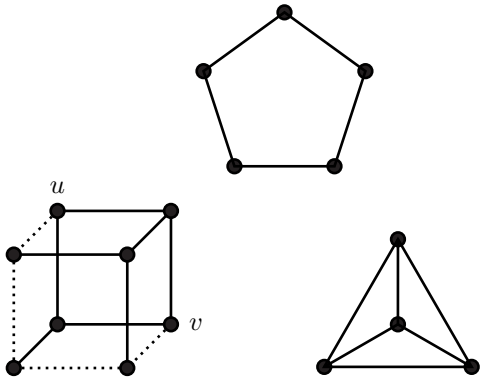


FIGURE 2.1
A graph with three components

There are several ways of finding the connected components of a graph G. One way to find the sets C_u for a graph G is as follows:

procedure COMPONENTS(G)
 for each $u \in V(G)$
  do initialize C_u to contain only u
 for each $u \in V(G)$
  do { **for each** $v \longrightarrow u$
  do if $C_u \neq C_v$ **then** MERGE(C_u, C_v)

The inner for-loop ensures that, upon completion, if $u \longrightarrow v$, then $C_u = C_v$, for any vertices u and v. Therefore, if $P = (u_0, u_1, \ldots, u_k)$ is any path, we can be sure that $C_{u_0} = C_{u_1} = \cdots = C_{u_k}$, so that when the algorithm terminates, each C_u will contain all the vertices connected to u by any path; that is, C_u will be the connected component containing u.

The complexity of the algorithm naturally depends upon the data structures used to program it. This algorithm is a perfect example of the use of the *merge-find* data structure. Initially, each $C_u = \{u\}$ and $C_v = \{v\}$. When the edge uv is examined, C_u and C_v are merged, so that now $C_u = C_v = \{u, v\}$. The two operations which need to be performed are to determine whether $C_u = C_v$, and to merge C_u and C_v into one. This can be done very efficiently by choosing a vertex in each C_u as *component representative*.

uRep $\leftarrow$ COMPREP(C_u)
vRep $\leftarrow$ COMPREP(C_v)
if *uRep* $\neq$ *vRep*
 then MERGE(C_u, C_v)

Initially, $C_u = \{u\}$, so that u begins as the representative of C_u. Associated with each vertex v is a pointer toward the representative of the component containing v. To find the representative of C_u, we start at u and follow these pointers, until we come to the component representative. The component representative is marked by a pointer that is negative. The initial value is -1. The pointers are easily stored as an array, *CompPtr*.

COMPREP() is a recursive procedure that follows the component pointers until a negative value is reached.

procedure COMPREP(u)
 if *CompPtr*$[u] < 0$
  then return (u)
  else { *theRep* $\leftarrow$ COMPREP(*CompPtr*$[u]$)
  CompPtr$[u] \leftarrow$ *theRep*
  return (*theRep*)

The assignment

$$\textit{CompPtr}[u] \leftarrow \textit{theRep}$$

is called *path compression*. It ensures that the next time *CompPtr*(u) is computed, the representative will be found more quickly. The algorithm COMPONENTS() can now be written as follows:

Algorithm 2.1.1: COMPONENTS(G)

$n \leftarrow |G|$
for $u \leftarrow 1$ **to** n
  do *CompPtr*$[u] \leftarrow -1$
for $u \leftarrow 1$ **to** n
  do { **for each** $v \longrightarrow u$
    do { *uRep* $\leftarrow$ COMPREP(u)
      vRep $\leftarrow$ COMPREP(v)
      if *uRep* $\neq$ *vRep*
        then MERGE(*uRep*, *vRep*)

The essential step in merging C_u and C_v is to assign either

$$\mathit{CompPtr}[\mathit{vRep}] \leftarrow \mathit{uRep}$$

or

$$\mathit{CompPtr}[\mathit{uRep}] \leftarrow \mathit{vRep}$$

The best one to choose is that which merges the smaller component onto the larger. We can determine the size of each component by making use of the negative values of *CompPtr*[*uRep*] and *CompPtr*[*vRep*]. Initially, *CompPtr*$[u] = -1$, indicating a component of size one.

procedure MERGE(*uRep*, *vRep*)
  uSize $\leftarrow -$*CompPtr*[*uRep*]
  vSize $\leftarrow -$*CompPtr*[*vRep*]
  if *uSize* $<$ *vSize*
    then { *CompPtr*[*uRep*] $\leftarrow$ *vRep*
      CompPtr[*vRep*] $\leftarrow -$(*uSize* + *vSize*)
    else { *CompPtr*[*vRep*] $\leftarrow$ *uRep*
      CompPtr[*uRep*] $\leftarrow -$(*uSize* + *vSize*)

When C_u and C_v are merged, the new component representative (either *uRep* or *vRep*) has its *CompPtr*$[\cdot]$ assigned equal to $-$(*uSize* + *vSize*). The component pointers can be illustrated graphically. They are shown in Figure 2.2 as arrows. The merge operation is indicated by the dashed line.

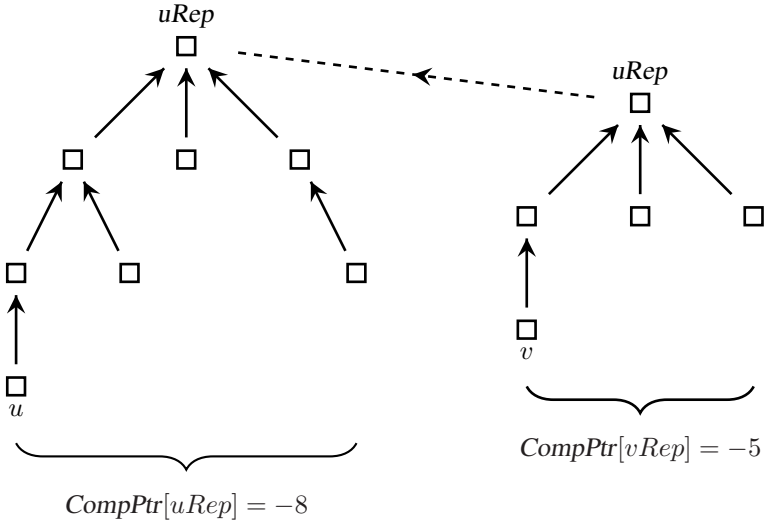


FIGURE 2.2
Component representatives

2.2 Complexity

The components algorithm is very efficient. The for-loop which initializes the *CompPtr* array requires n steps. If adjacency lists are used to store G, then the total number of times that the body of the main loop is executed is

$$\sum \text{DEG}(u) = 2\varepsilon.$$

Thus COMPREP() is called 4ε times. How many times is MERGE() called? At each merge, two existing components are replaced by one, so that at most $n-1$ merges can take place. Each merge can be performed using four assignments and a comparison. It takes n steps to initialize the *CompPtr* array. Thus the total number of steps is about $6n + 4\varepsilon$·(number of steps per call to COMPREP()). The number of steps each call to COMPREP() requires depends on the depth of the trees which represent the components. The depth is changed by path compression, and by merging. It is proved in AHO, HOPCROFT, and ULLMAN [1], that if there are a total of n points involved, the number of steps required is $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the function $A(n)$, defined recursively as follows.

$$\begin{aligned} A(1) &= 1 \\ A(k) &= 2^{A(k-1)} \end{aligned}$$

Thus, $A(2) = 2^1 = 2$, $A(3) = 2^2 = 4$, $A(4) = 2^4 = 16$, $A(5) = 2^{16} = 65536$, etc. It follows that $\alpha(n) \leq 5$, for all $n \leq 65536$. So the complexity of Algorithm 2.1.1 is almost linear, namely, $O(n + \varepsilon\alpha(n))$, where $\alpha(n) \leq 5$, for all practical values of n.

Exercises

2.2.1 Assuming the data structures described in Section 2.1, program the COMPONENTS() algorithm, merging the smaller component onto the larger. Include an integer variable *NComps* which contains the current number of components. Upon completion, its value will equal $\omega(G)$.

2.2.2 Algorithm 2.1.1 computes the connected components C_u using the array *CompPtr*. If we now want to print the vertices of each distinct C_u, it cannot be done very efficiently. Show how to use linked lists so that for each component, a list of the vertices it contains is available. Rewrite the MERGE() procedure to include this. Is the complexity thereby affected?

2.2.3 In the Algorithm 2.1.1 procedure, the for-loop

for $u \leftarrow 1$ **to** n **do**

executes the statement *uRep* $\leftarrow$ COMPREP(u) once for every $v \longrightarrow u$. Show how to make this more efficient by taking the statement *uRep* $\leftarrow$ COMPREP(u) out of the v-loop, and modifying the MERGE() procedure slightly. Calculate the new complexity.

2.2.4 Let $n = |G|$. Show that if $\varepsilon > \binom{n-1}{2}$, then G is connected. *Hint*: If G is disconnected, there is a component of size $x < n$. What is the maximum number of edges G can then have?

2.2.5 Show that if $\delta > \lfloor (n-1)/2 \rfloor$, then G is connected.

2.2.6 Show that if G is disconnected, then $\overline{G}$ is connected.

2.2.7 Show that if G is simple and connected but not complete, then G has three vertices u, v, and w such that $u \longrightarrow v, w$, but $v \not\longrightarrow w$.

2.2.8 A *longest path* in a graph G is any path P such that G contains no path longer than P. Thus a graph can have several different longest paths (all of the same length, though). Show that $\ell(P) \geq \delta(G)$, for any longest path. *Hint*: Consider an endpoint of P.

2.2.9 Show that every graph G has a cycle of length at least $\delta(G)+1$, if $\delta(G) \geq 2$. *Hint*: Consider a longest path.

2.2.10 Prove that in a connected graph, any two longest paths have at least one vertex in common.

2.3 Walks

Paths do not contain repeated vertices or edges. A *walk* in G is any sequence of vertices $u_0, u_1, \ldots, u_k$ such that $u_i \longrightarrow u_{i+1}$. Thus, in a walk, edges and vertices may be repeated. Walks are important because of their connection with the adjacency matrix of a graph. Let A be the adjacency matrix of G, where $V(G) = \{u_1, u_2, \ldots, u_n\}$, such that row and column i of A correspond to vertex u_i.

Theorem 2.1. *Entry $[i, j]$ of A^k is the number of walks of length k from vertex u_i to u_j.*

Proof. By induction on k. When $k = 1$, there is a walk of length 1 from u_i to u_j if and only if $u_i \longrightarrow u_j$, in which case entry $A[i, j] = 1$. Assume it's true whenever $k \leq t$ and consider A^{t+1}. Let W be a u_iu_j-walk of length $t + 1$, where $t \geq 2$. If u_l is the vertex before u_j on W, then W can be written as (W', u_l, u_j), where W' is a u_iu_l-walk of length t. Furthermore, every u_iu_l-walk of length t gives a u_iu_j-walk of length $t + 1$ whenever $u_l \longrightarrow u_j$. Therefore the number of u_iu_j-walks of length $t + 1$ is

$$\sum_l (\text{the number of } u_iu_l - \text{walks of length } t)(A[l, j]).$$

But the number of u_iu_l-walks of length t is $A^t[i, l]$, so that the number of u_iu_j-walks of length $t + 1$ is

$$\sum_{l=1}^{n} A^t[i, l]A[l, j],$$

which equals $A^{t+1}[i, j]$. Therefore the result is true when $k = t + 1$. By induction, it's true for all values of k. □

Notice that this result is also true for multigraphs, where now $A[i, j]$ is the number of edges joining u_i to u_j. For multigraphs, a walk W must be specified by giving the sequence of edges traversed, as well as the sequence of vertices, because there can be more than one edge joining the same pair of vertices.

Exercises

2.3.1 Show that $A^2[i, j]$ equals the number of u_iu_j-paths of length 2, if $i \neq j$, and that $A^2[i, i] = \text{DEG}(u_i)$.

2.3.2 Show that $A^3[i, i]$ equals the number of triangles containing vertex u_i. Find a similar interpretation of $A^3[i, j]$, when $i \neq j$. (A triangle is a cycle of length 3.)

2.3.3 A^k contains the number of walks of length k connecting any two vertices. Multiply A^k by x^k, the k^{th} power of a variable x, and sum over k, to get the matrix power series $I + Ax + A^2x^2 + A^3x^3 + \cdots$, where I is the identity matrix. The sum of this power series is a matrix whose ij^{th} entry

is a function of x containing the number of u_iu_j-walks of each length, as the coefficient of x^k. Because the power series expansion of $(1-a)^{-1}$ is $1 + a + a^2 + a^3 + \cdots$, we can write the above matrix as $(I - Ax)^{-1}$. That is, the inverse of the matrix $(I - Ax)$ is the *walk generating matrix*. Find the walk generating matrix for the graph of Figure 2.3.

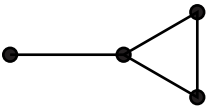


FIGURE 2.3
Compute the number of walks in this graph

2.4 The shortest-path problem

The *distance* from vertex u to v is $\textsc{Dist}(u, v)$, the length of the shortest uv-path. If G contains no uv-path, then $\textsc{Dist}(u, v) = \infty$. In this section we study the following two problems.

Problem 2.1:	Shortest Path
Instance:	a graph G and a vertex u.
Find:	$\textsc{Dist}(u, v)$, for all $v \in V(G)$.

Problem 2.2:	All Paths
Instance:	a graph G.
Find:	$\textsc{Dist}(u, v)$, for all $u, v \in V(G)$.

Given a vertex u, one way of computing $\textsc{Dist}(u, v)$, for all v, is to use a *breadth-first search* (BFS), as is done in procedure BFS().

procedure BFS(G, u)
comment: $\begin{cases} \textit{ScanQ} \text{ is a queue of vertices} \\ \textit{dist}[v] \text{ will equal DIST(u,v), upon completion} \end{cases}$
for each $v \in V(G)$
do $\textit{dist}[v] \leftarrow \infty$
$\textit{dist}[u] \leftarrow 0$
place u on *ScanQ*
repeat
select v for the head of *ScanQ*
for each $w \longrightarrow v$
do if w not on *ScanQ*
then $\begin{cases} \text{add } w \text{ to the end of } \textit{ScanQ} \\ \textit{dist}[w] \leftarrow \textit{dist}[v] + 1 \end{cases}$
advance *ScanQ*
until all of *ScanQ* has been processed

Procedure BFS() uses a type of data structure called a *queue*. A queue is an ordered list in which we usually access only the first or the *head* of the list and new items are only placed at the end or *tail* of the list. This is similar to one's experience of waiting in line at the checkout counter of a store. The person at the head of the line is processed first by the checker and the new customers enter at the end of the line. One of the most convenient ways to store a queue is as an array. For when an algorithm builds a queue on an array, all the vertices visited are on the array when the algorithm completes, ready for input to the next procedure. BFS() works in this way.

The breadth-first search (*BFS*) algorithm is a fundamental algorithm in graph theory. It appears in various guises wherever shortest paths are useful (e.g., network flows, matching theory, coset enumeration, etc.). Figure 2.4 shows the result of applying a BFS to the Petersen graph, where the vertices are numbered according to the order in which they were visited by the algorithm, and shaded according to their distance from vertex 1. The thicker edges show the shortest paths found by the algorithm.

Notice that the first vertex on the *ScanQ* is u, whose $\textit{dist}[u] = \text{DIST}(u, u) = 0$. The next vertices to be placed on the queue will be those adjacent to u, that is, those at distance 1. When they are placed on the queue, their distance will be computed as

$$\textit{dist}[\cdot] \leftarrow \textit{dist}[u] + 1.$$

So we can say that initially, that is, up to vertices of distance one, vertices are placed on the queue in order of their distance from u; and that when each vertex w is placed on *ScanQ*, $\textit{dist}[w]$ is made equal to $\text{DIST}(u, w)$. Assume that this is true for all vertices of distance k or less, where $k \geq 1$. Consider when v is chosen as the first vertex of distance k on *ScanQ*. The for-loop examines all vertices $w \longrightarrow v$. If w on *ScanQ* already, then there is a uw-path of length $\leq k$, and w is ignored. If w is not

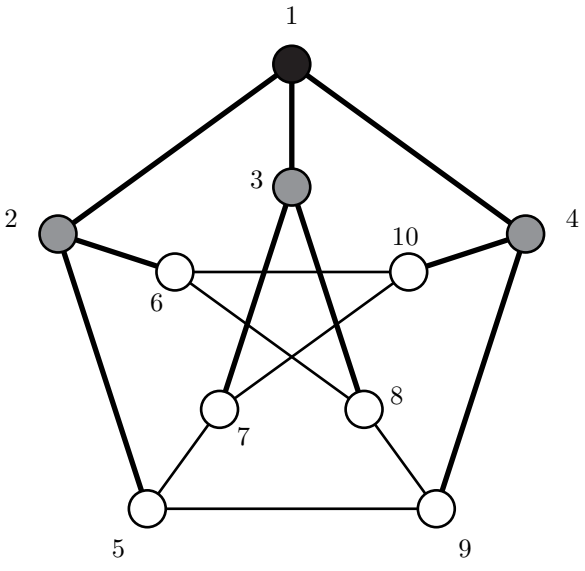


FIGURE 2.4
A breadth-first search

on *ScanQ*, then $\text{DIST}(u, w) > k$. The uw-path via v has length $k + 1$, so w is added to the queue, and $dist[w]$ is set equal to $dist[v] + 1 = k + 1$. Because every vertex at distance $k + 1$ is adjacent to a vertex at distance k, we can be sure that when all vertices v on *ScanQ* at distance k have been scanned, all vertices at distance $k + 1$ will be on the queue. Thus the assertion that vertices are placed on the queue in order of their distance from u, and that when each vertex w is placed on *ScanQ*, $dist[w]$ is made equal to $\text{DIST}(u, w)$, is true up to distance $k + 1$. By induction, it is true for all distances.

This proof that the BFS() algorithm works illustrates how difficult and cumbersome it can be to prove that even a simple, intuitively "obvious" algorithm works correctly. Nevertheless, it is important to be able to prove that algorithms work correctly, especially the more difficult algorithms. Writing down a proof for an "obvious" algorithm will often reveal hidden bugs that it contains. This proof also illustrates another feature, namely, proofs that algorithms work tend to use induction, often on the number of iterations of a main loop.

The complexity of the BFS() is very easy to calculate. The main operations which are performed are

1. Scan all $w \longrightarrow v$.
2. Select the next $v \in$ *ScanQ*.
3. Determine whether $w \in$ *ScanQ*.

The first operation is most efficiently done if G is stored in adjacency lists. We want the second and third operations to take a constant number of steps. We store *ScanQ* as an integer array, and also store a boolean array *onScanQ* to tell whether $w \in$ *ScanQ*. The revised algorithm is Algorithm 2.4.1.

Algorithm 2.4.1: BFS(G, u)

global n
for $v \leftarrow 1$ **to** n
  do { $dist[v] \leftarrow \infty$; $onScanQ[v] \leftarrow$ **false**
$dist[u] \leftarrow 0$
$ScanQ[1] \leftarrow u$
$onScanQ[u] \leftarrow$ **true**
$QSize \leftarrow 1$
$k \leftarrow 1$
repeat
 $v \leftarrow ScanQ[k]$
 for each $w \longrightarrow v$
  do if not $onScanQ[w]$
   then { $QSize \leftarrow QSize + 1$; $ScanQ[QSize] \leftarrow w$; $onScanQ[w] \leftarrow$ **true** ; $dist[w] \leftarrow dist[v] + 1$
 $k \leftarrow k + 1$
until $k > QSize$

The initialization takes $2n$ steps. The repeat-loop runs at most n times. At most n vertices are placed on the queue. The for-loop over all $w \longrightarrow v$ requires

$$\sum_v \text{DEG}(v) = 2\varepsilon$$

steps, all told. This assumes that the adjacent vertices are stored in a linked list – the for-loop traverses the adjacency list. Therefore the total number of steps executed is at most

$$3n + 2\varepsilon = O(n + \varepsilon) = O(\varepsilon).$$

Notice that in this program we could have dispensed with the array *onScanQ*, by using instead $dist[w] = \infty$ to determine w is on *ScanQ*. Because a breath-first search always uses a queue but not always a $dist[\cdot]$ array, we have kept the boolean array, too.

2.5 Weighted graphs and Dijkstra's algorithm

A breath-first search calculates $\textsc{Dist}(u, v)$ correctly because in a simple graph, each edge has "length" one; that is, the length of a path is the number of edges it contains. In a more general application where graphs are used to model a road network, or distribution network, etc., we may want to assign a length ≥ 1 to each edge. This is illustrated in Figure 2.5.

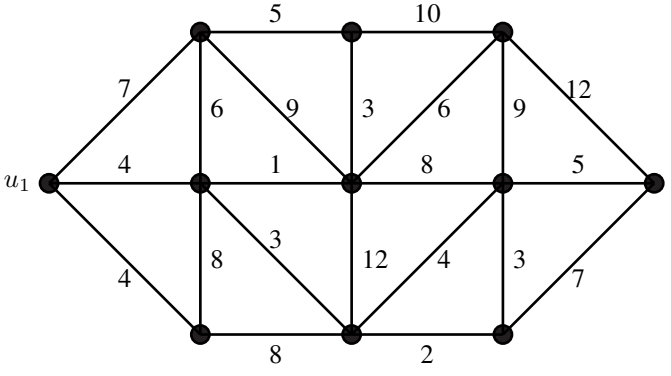


FIGURE 2.5
A weighted graph

This is an example of a *weighted graph*. Each edge $uv \in E(G)$ is assigned a *positive integral weight* $\textsc{Wt}(uv)$. $\textsc{Wt}(uv)$ may represent length, weight, cost, capacity, etc., depending on the application. In a weighted graph, the length of a path $P = (u_0, u_1, \ldots, u_k)$ is

$$\ell(P) = \sum_{i=0}^{k-1} \textsc{Wt}(u_i u_{i+1}).$$

The distance between two vertices is now defined as

$$\textsc{Dist}(u, v) = \textsc{min}\{\ell(P) : P \text{ is a } uv\text{-path}\}.$$

A breath-first search will not compute $\textsc{Dist}(u, v)$ correctly in a weighted graph, because a path with more edges may have the shorter length. There are many algorithms for computing shortest paths in a weighted graph. Dijkstra's algorithm is one.

procedure DIJKSTRA(u)

comment: { Compute DIST(u, v), for all $v \in V(G)$
dist$[v]$ will equal DIST(u, v) upon completion.
Vertices are chosen as $u_1, u_2, \ldots, u_n$,
in order of their distance from u.

$u_1 \leftarrow u$ "the nearest vertex to u "

for $k \leftarrow 1$ **to** $n - 1$

do {
  comment: { $u_1, u_2, \ldots, u_k$ are currently known
  in this iteration, u_{k+1} is selected
  select v, the nearest vertex to u_1, such that $v \notin \{u_1, u_2, \ldots, u_k\}$
  $u_{k+1} \leftarrow v$
  assign *dist*$[u_{k+1}]$
  comment: $u_1, u_2, \ldots, u_{k+1}$ are now known

comment: all *dist*$[u_i]$ are now known

Dijkstra's algorithm is an example of a so-called " greedy" or "myopic" algorithm, that is, an algorithm which always selects the next nearest, or next best, etc., on each iteration. Many problems can be solved by greedy algorithms.

We need to know how to choose v, the next nearest vertex to u_1, in each iteration. On the first iteration, it will be the vertex v adjacent to u_1 such that WT$(u_1 v)$ is minimum. This will give $\{u_1, u_2\}$ such that DIST(u_1, u_1) and DIST(u_1, u_2) are known. On the next iteration, the vertex v chosen will be adjacent to one of u_1 or u_2. The distance to u_1 will then be either

$$\text{DIST}(u_1, u_1) + \text{WT}(u_1 v)$$

or

$$\text{DIST}(u_1, u_2) + \text{WT}(u_2 v),$$

and v will be the vertex for which this sum is minimum.

In general, at the beginning of iteration k, vertices $u_1, u_2, \ldots, u_k$ will have been chosen, and for these vertices,

$$\text{DIST}[u_i] = \text{DIST}(u_1, u_i).$$

The next nearest vertex v must be adjacent to some u_i, so that the shortest $u_1 v$-path will have length dist$[u_i]$ + WT$(u_i v)$, for some i. v is chosen as the vertex for which this value is a minimum. This is illustrated in Figure 2.6. The refined code for Dijkstra's algorithm is Algorithm 2.5.1.

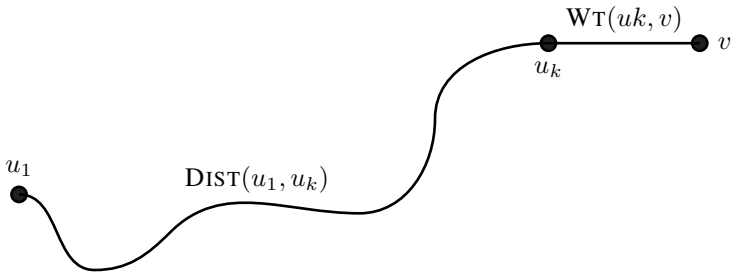


FIGURE 2.6
A shortest u_1v-path, via vertex u_k

Algorithm 2.5.1: DIJKSTRA(u)

comment: Compute DIST(u, v), for all $v \in V(G)$
dist$[v]$ will equal DIST(u, v) upon completion.
Vertices are chosen as $u_1, u_2, \ldots, u_n$,
in order of their distance from u

for each v
  do *dist*$[v] \leftarrow \infty$
$u_1 \leftarrow u$ " the nearest vertex to u "
dist$[u] \leftarrow 0$
for $k \leftarrow 1$ **to** $n - 1$
do
  comment: $u_1, u_2, \ldots, u_k$ are currently known
  in this iteration, u_{k+1} is selected
  for each $v \longrightarrow u_k$ such that $v \notin \{u_1, u_2, \ldots, u_k\}$
    do *dist*$[v] \leftarrow$ MIN(*dist*$[v]$, *dist*$[u_k]$ + WT(u_kv))
  pick $v \notin \{u_1, u_2, \ldots, u_k\}$ such that *dist*$[v]$ is minimum
  $u_{k+1} \leftarrow v$
  comment: *dist*$[u_{k+1}]$ now equals DIST(u_1, u_{k+1}), and
  $u_1, u_2, \ldots, u_{k+1}$ are now known

Exercises

2.5.1 Prove that Dijkstra's algorithm works. Use induction on the number k of iterations to prove that at the beginning of iteration k, each *dist*$[u_i]$=DIST(u_1, u_i), and that for all $v \neq u_i$, for any i, *dist*$[v]$ equals the length of a shortest u_1v-path using only the vertices $\{u_1, u_2, \ldots, u_{k-1}, v\}$. Conclude that after $n - 1$ iterations, all distances *dist*$[v]$ = DIST(u_1, v).

2.5.2 Assuming that G is stored in adjacency lists, and that the minimum $dist[v]$ is computed by scanning all n vertices, show that the complexity of Dijkstra's algorithm is $O(\varepsilon + n^2)$.

2.6 Data structures

When computing the distances $\text{DIST}(u_1, v)$, it would also be a good idea to store a shortest u_1v-path. All the u_1v-paths can easily be stored using a single array

$$PrevPt[v]\text{: the previous point to } v \text{ on a shortest } u_1v\text{-path.}$$

Initially, $PrevPt[u] \leftarrow 0$. When $dist[v]$ and $dist[u_k] + \text{WT}(u_kv)$ are compared, if the second choice is smaller, then assign $PrevPt[v] \leftarrow u_k$. The shortest u_1v-path can then be printed by the following loop:

repeat
 output (v)
 $v \leftarrow PrevPt[v]$
until $v = 0$

The complexity of Dijkstra's algorithm was calculated in Exercise 2.5.2 as $O(n^2 + \varepsilon)$. The term $O(n^2)$ arises from scanning up to n vertices in order to select the minimum vertex v. This scanning can be eliminated if we store the vertices in a partially ordered structure in which the minimum vertex is always readily available. A *heap* is such a structure. In a heap H, nodes are stored so that the smallest element is always at the top.

A heap is stored as an array, but is viewed as the partially ordered structure shown in Figure 2.7. Its elements are not sorted, but satisfy the *heap property*, namely that $H[i] \leq H[2i]$ and $H[i] \leq H[2i + 1]$; that is, the value stored in each node is less than or equal to that of either of its children. Therefore, $H[1]$ is the smallest entry in the array.

The heap in Figure 2.7 has depth four; that is, there are four levels of nodes. A heap of depth k can contain up to $2^k - 1$ nodes, so that the depth needed to store N values is the smallest value of k such that $2^k - 1 \geq N$, namely, $k = \lceil \log(N + 1) \rceil$, where the log is to base 2.

If the value stored in a node is changed so that the heap property is no longer satisfied, it is very easy to update the array so that it again forms a heap. For example, if $H[10]$ were changed to 4, then the following loop will return H to heap form. The movement of data is shown in Figure 2.8.

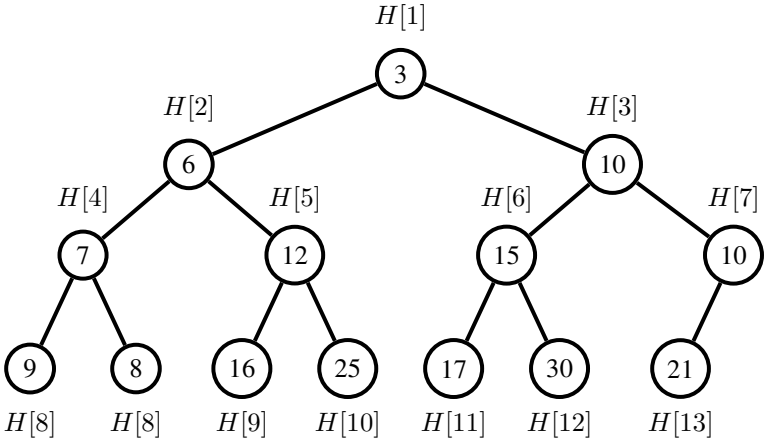


FIGURE 2.7
A heap

procedure FLOATUP(k)
comment: Element $H[k]$ floats up to its proper place in the heap

temp $\leftarrow H[k]$
$j \leftarrow k/2$
while *temp* $< H[j]$ **and** $j > 0$
do $\begin{cases} H[k] \leftarrow H[j] \\ k \leftarrow j \\ j \leftarrow k/2 \end{cases}$
$H[k] \leftarrow$ *temp*

Notice the circular movement of data when an altered element floats up to its proper place in the heap. If some entry in the heap were made larger, say $H[1]$ became equal to 10, then a similar loop (Procedure FLOATDOWN) would cause the new value to float down to its proper place. Because the depth of a heap containing N items is $\lceil \log(N+1) \rceil$, the number of items moved is at most $1 + \lceil \log(N+1) \rceil$.

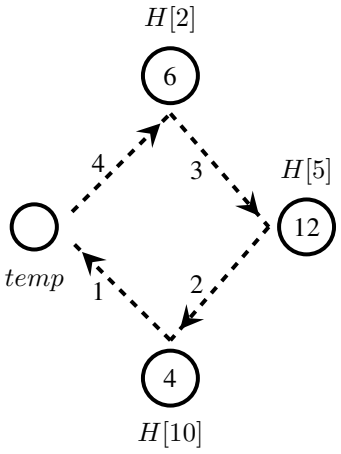


FIGURE 2.8
Updating a heap with FLOATUP

procedure FLOATDOWN(k)
comment: the entry at $H[k]$ has been increased – it now floats down the heap to its correct location. There are currently n entries in the heap, $H[1]$ to $H[n]$. The array $H[\cdot]$ has been dimensioned so that $H[0]$ is also available as a sentinel. $H[0]$ contains a large value bigger than any valid heap entry.

temp $\leftarrow H[k]$
while $k + k \leq n$
do
- $i \leftarrow k + k$ "the left child of $H[k]$"
- $j \leftarrow i + 1$ "the right child of $H[k]$"
- **if** $j > n$
  - **then** $j \leftarrow 0$ "the sentinel at $H[0]$"
- **if** $H[i] > H[j]$
  - **then** $i \leftarrow j$
- **comment:** $H[i]$ is now the smaller child
- **if** *temp* $\leq H[i]$
  - **then break** "break out of loop"
- $H[k] \leftarrow H[i]$
- $k \leftarrow i$

$H[k] \leftarrow$ *temp*

In order to extract the smallest item from a heap of N elements, we take its value from $H[1]$, and then perform the following steps:

$H[1] \leftarrow H[N]$
$N \leftarrow N - 1$
FLOATDOWN(1)

The new $H[1]$ floats down at most $1 + \lceil \log\ N \rceil$ steps.

There are two ways of building a heap.

procedure BUILDHEAPTOPDOWN(H, N)
comment: { The array H contains N entries
transform it into a heap
$k \leftarrow 1$
while $k < N$
do {
comment: the first k values in H already form a heap
$k \leftarrow k + 1$
FLOATUP(k)

Using this method, the elements in entries $1, 2, \ldots, k$ already form a subheap with k entries. On each iteration, a new entry is allowed to float up to its proper position so that the first $k + 1$ values now form a heap. There are two nodes on level two of the heap. The FLOATUP() operation for each of these may require up to $1 + 2 = 3$ data items to be moved. On level three there are four nodes. FLOATUP() may require up to $1 + 3 = 4$ data items to be moved for each one. In general, level k contains 2^{k-1} nodes, and FLOATUP() may need up to $1 + k$ data items to be moved for each. The total number of steps to create a heap with $d = \lceil \log(N + 1) \rceil$ levels in this way is therefore at most

$$S = 3 \cdot 2^1 + 4 \cdot 2^2 + 5 \cdot 2^3 + \cdots + (1 + d)2^{d-1} = \sum_{k=2}^{d-1} (1 + k)2^{k-1}.$$

Therefore

$$2S = 3.2^2 + 4.2^2 + 5.2^3 + \cdots + (1 + d)2^d,$$

so that

$$\begin{aligned} 2S - S &= (1 + d)2^d - 3 \cdot 2^1 - (2^2 + 2^3 + \cdots + 2^{d-1}) \\ &= (1 + d)2^d - 5 - (1 + 2 + 2^2 + 2^3 + \cdots + 2^{d-1}) \\ &= (1 + d)2^d - 5 - (2^d - 1) \\ &= d2^d - 4 \end{aligned}$$

Thus, it takes $O(N \log N)$ steps to build a heap in this way.

The second way of building a heap is to use FLOATDOWN().

procedure BUILDHEAPBOTTOMUP(H, N)

comment: {The array H contains N entries; transform it into a heap}

$k \leftarrow N/2$
while $k \geq 1$
do {
  comment: {the substructures at nodes $H[2k]$ and $H[2k+1]$ already form subheaps}
  FLOATDOWN(k)
  $k \leftarrow k - 1$
}

This way is much more efficient, requiring only $O(N)$ steps, as is proved in Exercise 2.7.1.

We can use a heap H to store the values *dist*$[v]$ in Dijkstra's algorithm. The main loop now looks like this.

$u_1 \leftarrow u$ "the nearest vertex to u"
for $k \leftarrow 1$ **to** $n - 1$
do {
  comment: $u_1, u_2, \ldots, u_k$ are currently known
  for each $v \longrightarrow u_k$ such that $v \notin u_1, u_2, \ldots, u_k$
  do {
    if *dist*$[v] >$ *dist*$[u_k] +$ WT$(u_k v)$
    then {
      dist$[v] \leftarrow$ *dist*$[u_k] +$ WT$(u_k v)$
      FLOATUP(v) "which entry corresponds to v?"
    }
  }
  choose u_{k+1} using $H[1]$
  $H[1] \leftarrow H[n-k]$
  remove last entry from H
  FLOATDOWN(1)
  comment: $u_1, u_2, \ldots, u_{k+1}$ are now known
}

Notice that the FLOATUP(v) operation requires that we also know which node $H[k]$ in the heap corresponds to the vertex v, and vice versa. This can be done with an array mapping vertices into the heap. Let us work out the complexity of Dijkstra's algorithm using this data structure. It is not possible to get an accurate estimate of the number of steps performed in this case, but only an upper bound. The initialization of the heap and *dist*$[\cdot]$ array take $O(n)$ steps. The inner for-loop executes a total of at most 2ε if-statements, so that at most 2ε FLOATUP()'s are performed, each requiring at most $1 + \lceil \log(n+1) \rceil$ steps. There are also $n - 1$ FLOATDOWN()'s performed. Thus the complexity is now

$$O((2\varepsilon + n)(1 + \lceil \log(n+1) \rceil)) = O(\varepsilon \log\ n).$$

This may be better or worse that the previous estimate of $O(n^2)$ obtained when the minimum vertex is found by scanning up to n vertices on each iteration. If the graph has few edges, say $\varepsilon \leq \Delta n/2$, where the maximum degree Δ is some fixed constant,

or a slowly growing function of n, then Dijkstra's algorithm will certainly be much more efficient when a heap is used. Furthermore, it must be remembered that the complexity estimate using the heap is very much an *upper bound*, whereas the other method will always take at least $O(n^2)$ steps. If the number of edges is large, say $\varepsilon = O(n^2)$, then the heap-version of Dijkstra's algorithm can spend so much time keeping the heap up-to-date, that no increase in efficiency is obtained.

2.7 Floyd's algorithm

Floyd's algorithm solves the All Paths Problem, computing a matrix of values $\mathit{Dist}[u, v] = \textsc{Dist}(u, v)$, for all $u, v \in V(G)$. Initially, $\mathit{Dist}[\cdot, \cdot]$ equals the weighted adjacency matrix A, where

$$A[u, v] = \begin{cases} \textsc{Wt}(u, v), & \textbf{if } u \longrightarrow v, \\ \infty, & \textbf{if } u \not\longrightarrow v, \\ 0, & \textbf{if } u = v. \end{cases}$$

Floyd's algorithm is extremely simple to program.

procedure FLOYD($Dist$)
comment: $\mathit{Dist}[u, v]$ will equal $\textsc{Dist}(u, v)$, upon completion

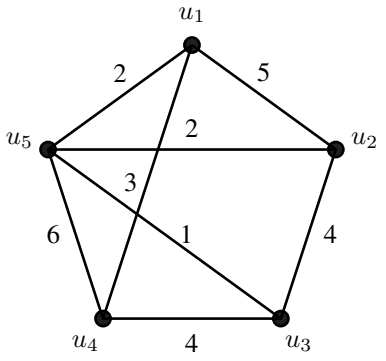
for $k \leftarrow 1$ **to** n
do { **for** $v \leftarrow 1$ **to** $n - 1$
do { **for** $w \leftarrow v + 1$ **to** n
do $\mathit{Dist}[v, w] \leftarrow \textsc{min}(\mathit{Dist}[v, w], \mathit{Dist}[v, u_k] + \mathit{Dist}[u_k, w])$

The for-loops for v and w together examine $\binom{n}{2}$ pairs vw for each value of u, so the complexity of the algorithm is

$$n\binom{n}{2} = \frac{1}{2}n^3 - \frac{1}{2}n^2 = O(n^3).$$

The graph is stored as a weighted adjacency matrix, in which non-adjacent vertices v, w can be considered to be joined by an edge of weight ∞. Figure 2.9 shows a weighted graph on which the reader may like to work Floyd's algorithm by hand.

Let the vertices of G be named $u_1, u_2, \ldots, u_n$. In order to prove that Floyd's algorithm works, we prove by induction, that at the end of k^{th} iteration of the for-loop for u, Dist$[v, w]$ is the length of the shortest vw-path which uses only vertices v, w, and $u_1, u_2, \ldots, u_k$. When $k = 0$, that is, before the first iteration, $\mathit{Dist}[v, w]$ is the length of the edge vw, that is, the length of the shortest path using only vertices v and w. At the end of the first iteration, $\mathit{Dist}[v, w] = \textsc{min}(\textsc{Wt}(v, w), \textsc{Wt}(v, u_1) + \textsc{Wt}(u_1, w))$. This is the length of the shortest vw-path using only vertices v, w, and



	u_1	u_2	u_3	u_4	u_5
u_1	0	5	∞	3	2
u_2	5	0	4	∞	2
u_3	∞	4	0	1	4
u_4	3	∞	1	0	6
u_5	2	2	4	6	0

FIGURE 2.9
A complete weighted graph and its weighted adjacency matrix

u_1, because that path either uses u_1, or else consists only of the edge vw. Thus, the statement is true when $k = 1$.

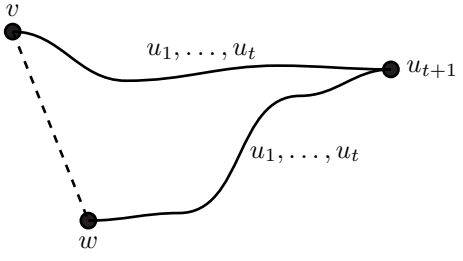


FIGURE 2.10
A path via $\boldsymbol{u_{t+1}}$

Assume that it is true whenever $k \leq t$, and consider iteration $t + 1$. At the end of the iteration, each

$$\textit{Dist}[v, w] = \text{MIN}(\textit{Dist}[v, w], \textit{Dist}[v, u_{t+1}] + Dist[u_{t+1}, w]). \tag{2.1}$$

If the shortest vw-path using only vertices $v, w, u_1, u_2, \ldots, u_{t+1}$ does not use u_{t+1}, then its length is the previous value of $Dist[v, w]$ from iteration t. If the path does use u_{t+1}, then the length is given by the second term of Equation 2.1. Therefore, at the end of the iteration, the value of $\textit{Dist}[v, w]$ is as required. By induction, it follows that at the end of the $n^{\textit{th}}$ iteration, $\textit{Dist}[v, w] = \text{DIST}(v, w)$, for all v and w. Floyd's algorithm finds all distances in the graph. It always takes $n\binom{n}{2} = O(n^3)$ steps, irrespective of the number of edges of G. When there are few edges, it is faster to use Dijkstra's algorithm n times, once for every starting vertex u. This gives a complexity of $O(\varepsilon n \log n)$, using a heap, which can be less than $O(n^3)$.

Exercises

2.7.1 Calculate the number of steps needed to construct a heap using the BUILDHEAPBOTTOMUP() procedure.

2.7.2 The repeat-loop of the FLOATUP() procedure described in Section 2.7 requires $k+2$ data items to be moved when an entry floats up k nodes in the heap. If FLOATUP() is programmed by swapping adjacent elements instead of moving them in a cycle, calculate the number of items moved when an entry floats up k nodes. Which is more efficient?

2.7.3 The type of heap discussed in Section 2.6 is called a *binary heap*, because each node $H[k]$ has two children, $H[2k]$ and $H[2k+1]$. The depth of a binary heap with N elements is $\lceil \log(N+1) \rceil$. In a *ternary heap*, node $H[k]$ has three children, $H[3k]$, $H[3k+1]$, and $H[3k+2]$. What is the depth of a ternary heap with N nodes? Calculate the number of steps needed to construct it using the BUILDHEAPBOTTOMUP() procedure.

2.7.4 Program Dijkstra's algorithm using a binary heap.

2.7.5 Show how to store a complete set of shortest paths in Floyd's algorithm, using a matrix *PrevPt*$[v,w]$, being the previous point to v on a shortest vw-path. What should the initial value of *PrevPt*$[v,w]$ be, and how and when should it be modified?

2.7.6 Ford's algorithm. Consider the following algorithm to find DIST(u,v), for a given vertex $u \in V(G)$ and all vertices $v \in V(G)$.

procedure FORD(u)
 for each $v \in V(G)$
  do *dist*$[v] \leftarrow \infty$
 dist$[u] \leftarrow 0$
 while there is an edge vw such that *dist*$[w] >$ *dist*$[v] +$ WT$[vw]$
  do *dist*$[w] \leftarrow$ *dist*$[v] +$ WT$[vw]$

Prove that Ford's algorithm correctly computes DIST(u,v). What data structures are necessary for an efficient implementation of Ford's algorithm? Analyze the complexity of Ford's algorithm. Give a numerical estimate of the number of steps, as well as a formula of the form $O(\cdot)$.

2.8 Notes

WEISS [188] contains an excellent treatment of the merge-find data structure and heaps. Dijkstra's shortest-path algorithm and Floyd's algorithm are described in most books on algorithms and data structures.

3

Subgraphs

3.1 Counting subgraphs

Given a graph G, we have seen two kinds of subgraphs – *induced subgraphs*, and *edge subgraphs*, also known as *partial subgraphs*. An induced subgraph is determined by a subset of $V(G)$. An edge-subgraph is determined by a subset of $E(G)$. We will also look at *mixed subgraphs* later. There are interesting relationships amongst these kinds of subgraphs.

If H is a graph with $|H| \leq |G|$, then following [113], we will write

$$\binom{G}{H}$$

for the number of induced subgraphs of G that are isomorphic to H. For example, $\binom{G}{K_2}$ counts the number of edges of G; $\binom{G}{K_3}$ counts the number of triangles of G. If P_2 represents a path of length two, then $\binom{G}{P_2}$ counts the number of *induced* paths of length two in G, etc.

Similarly, we use

$$\begin{bmatrix} G \\ H \end{bmatrix}$$

to denote the number of edge subgraphs of G that are isomorphic to H. We then have $\begin{bmatrix} G \\ K_2 \end{bmatrix}$ is also the number of edges of G, but $\begin{bmatrix} G \\ P_2 \end{bmatrix}$ is the number of paths of length two in G. Now an edge-subgraph isomorphic to P_2 can induce either a triangle, or a P_2. Therefore

$$\begin{bmatrix} G \\ P_2 \end{bmatrix} = \binom{G}{P_2} + 3\binom{G}{K_3}$$

where the coefficient 3 arises because K_3 contains three edge-subgraphs P_2. We see that the numbers of induced subgraphs and edge subgraphs are related. Let H be an edge subgraph of G, and let $m = |H|$.

Lemma 3.1.

$$\begin{bmatrix} G \\ H \end{bmatrix} = \sum_{|U|=m} \begin{bmatrix} G[U] \\ H \end{bmatrix} \binom{G}{G[U]}$$

Proof. Every subset $U \subseteq V(G)$ with m vertices induces a subgraph $G[U]$ of G.

$G[U]$ contains $\begin{bmatrix} G[U] \\ H \end{bmatrix}$ edge-subgraphs isomorphic to H. Each edge subgraph of G that is isomorphic to H occurs in exactly one subset U. Therefore the sum counts all edge subgraphs of G isomorphic to H. □

For example, if $|H| = 3$, there are four possible induced subgraphs of G with three vertices : $K_3, P_2, K_2 + K_1, 3K_1$. If we choose $H = P_2$, the lemma gives the formula previously found.

We now make a list of all graphs on m vertices, say $g_1, g_2, \ldots g_M$, for some M. If G is any graph with at least m vertices, then Lemma 3.1 gives M linear equations relating the M quantities $\begin{bmatrix} G \\ g_i \end{bmatrix}$ to the M quantities $\binom{G}{g_j}$.

Lemma 3.2.

$$\begin{bmatrix} G \\ g_i \end{bmatrix} = \sum_{j=1}^{M} \begin{bmatrix} g_j \\ g_i \end{bmatrix} \binom{G}{g_j}$$

Construct a matrix $\begin{bmatrix} g_j \\ g_i \end{bmatrix}$, with rows indexed by g_i and columns indexed by g_j. We order the graphs g_i in order of increasing number of edges. For example, if $m = 3$, we take $g_1 = 3K_1$, $g_2 = K_2 + K_1$, $g_3 = P_2$, and $g_4 = K_3$. The matrix is then given by

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that it is upper triangular, with a diagonal of ones, and therefore invertible. Moreover, the entries of the inverse are integers. When $m > 3$, there will be several of the graphs g_i with the same number of edges, maybe $g_i, g_{i+1}, g_{i+2}, \ldots$. The portion of the subgraph matrix they determine will be an identity sub-matrix appearing along the diagonal.

The quantities $\binom{G}{g_j}$ and $\begin{bmatrix} G \\ g_i \end{bmatrix}$ form two bases of the vector space of all possible linear combinations of subgraph counts of G. Lemma 3.2 can be viewed as a change of basis transformation.

3.1.1 Möbius inversion

The graphs $g_1, g_2, \ldots g_M$ on m vertices form a partially ordered set, where $g_i \leq g_j$ if and only if g_i is an edge subgraph of g_j. For example, when $m = 3$, the partial order is illustrated in Figure 3.1. Each g_i is represented by a node in the partial order, with a graph drawn beside it. There is an edge connecting the node representing g_i to the node for g_j below it if g_j is a subgraph of g_i, and there are no subgraphs between them. The ordered set for $m = 4$ is shown in Figure 3.2.

Given any graph G, we have two subgraph counting functions $\binom{G}{g_i}$ and $\begin{bmatrix} G \\ g_i \end{bmatrix}$ defined on each g_i in the ordered set. Lemma 3.2 says that if the values $\binom{G}{g_j}$ are

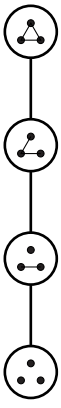


FIGURE 3.1
The graphs on three vertices

known, then the values $\left[\begin{smallmatrix} G \\ g_i \end{smallmatrix}\right]$ are determined by those g_j such that $g_j \geq g_i$ in the partial order. For example, when $m = 4$, we have the equation

$$\left[\begin{smallmatrix} G \\ \square\square\square\square \end{smallmatrix}\right] = \binom{G}{\square\square\square\square} + 4\binom{G}{\square\square\square\square} + 2\binom{G}{\square\square\square\square} + 6\binom{G}{\square\square\square\square} + 12\binom{G}{\square\square\square\square} \tag{3.1}$$

The equations of Lemma 3.2 can be inverted, using the partial order. It is clear that $\binom{G}{K_m} = \left[\begin{smallmatrix} G \\ K_m \end{smallmatrix}\right]$, so that the equation for K_m can be inverted. K_m is the "top" graph in the partial order. We then move to the graphs below it. Consider a graph g_i for which we want to determine $\binom{G}{g_i}$ in terms of $\left[\begin{smallmatrix} G \\ g_j \end{smallmatrix}\right]$. Given an occurrence of g_i as an edge subgraph of G, let U denote the subset of $V(G)$ that this occurrence of g_i spans. There may be a g_i such that $G[U]$ contains more edges than those of g_i, so that $\binom{G}{g_i} \leq \left[\begin{smallmatrix} G \\ g_i \end{smallmatrix}\right]$. If uv is any such edge, then $g_i + uv = g_j$, for some j, such that $g_i < g_j$ in the partial order. The subset U contains both g_i and g_j, so that the count $\left[\begin{smallmatrix} G \\ g_i \end{smallmatrix}\right]$ contains $\binom{G}{g_i}$, but also other induced subgraphs as well. In order to count only the induced copies of g_i, we must subtract the edge-subgraphs g_j contained in $\left[\begin{smallmatrix} G \\ g_i \end{smallmatrix}\right]$, Each copy of g_j contains $\left[\begin{smallmatrix} g_j \\ g_i \end{smallmatrix}\right]$ copies of g_i. Therefore we subtract $\left[\begin{smallmatrix} g_j \\ g_i \end{smallmatrix}\right]\left[\begin{smallmatrix} G \\ g_j \end{smallmatrix}\right]$, for each such g_j, giving

$$\left[\begin{matrix} G \\ g_i \end{matrix}\right] - \sum_j \left[\begin{matrix} g_j \\ g_i \end{matrix}\right]\left[\begin{matrix} G \\ g_j \end{matrix}\right] \tag{3.2}$$

where the sum is over all graphs g_j with one more edge than g_i. If every subset U containing a subgraph g_i induces either g_i or a graph with one more edge than g_i, then we now have $\binom{G}{g_i}$, thereby inverting the equation for g_i. This will always be the case when $g_j \cong K_m$.

Otherwise, let $xy \neq uv$ be another edge of $G[U]$ not contained in g_i, and let $g_\ell = g_i + xy$ and $g_k = g_\ell + uv = g_j + xy$. The occurrences of g_j and of g_ℓ

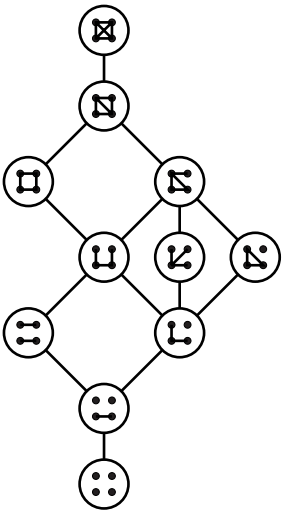


FIGURE 3.2
The graphs on four vertices

contained in $G[U]$ have already been subtracted in Equation (3.2). And because $G[U]$ also contains g_k, we find that g_k has been subtracted twice – once for g_j and once for g_ℓ. Therefore it must be added back once to Equation (3.2). Each copy of g_k contains $\left[\begin{smallmatrix} g_k \\ g_i \end{smallmatrix}\right]$ copies of g_i, and each of these copies of g_i can be extended to a copy of g_j and g_ℓ contained in g_k. This gives

$$\begin{bmatrix} G \\ g_i \end{bmatrix} - \sum_j \begin{bmatrix} g_j \\ g_i \end{bmatrix} \begin{bmatrix} G \\ g_j \end{bmatrix} + \sum_k \begin{bmatrix} g_k \\ g_i \end{bmatrix} \begin{bmatrix} G \\ g_k \end{bmatrix} \tag{3.3}$$

where the k sum is over all graphs g_k with two more edges than g_i. We then consider graphs g_r contained in $G[U]$ with three more edges than g_i. Each has been subtracted three times in Equation (3.2), but then added three times in Equation (3.3), and so must be subtracted again. This pattern continues for all subgraphs of $G[U]$ with more edges than g_i. The result is given in the following lemma.

Lemma 3.3.

$$\binom{G}{g_i} = \begin{bmatrix} G \\ g_i \end{bmatrix} + \sum_{j=1}^{M} (-1)^{\varepsilon(g_j)-\varepsilon(g_i)} \begin{bmatrix} g_j \\ g_i \end{bmatrix} \begin{bmatrix} G \\ g_j \end{bmatrix}$$

For example inverting Equation (3.1)

$$\binom{G}{\text{U}} = \begin{bmatrix} G \\ \text{U} \end{bmatrix} - 4\begin{bmatrix} G \\ \text{⊓} \end{bmatrix} + 2\begin{bmatrix} G \\ \text{N} \end{bmatrix} - 6\begin{bmatrix} G \\ \text{N} \end{bmatrix} + 12\begin{bmatrix} G \\ \text{⊠} \end{bmatrix}$$

is obtained.

This method of overcounting, then alternately subtracting and adding counts is known as *inclusion-exclusion*. It is a special case of inversion in a partially ordered set, known as *Möbius inversion*. See [15, 183] for further information.

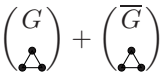
Combining Lemmas 3.2 and 3.3, we have

Corollary 3.4.

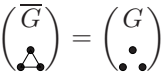
$$\sum_{k=1}^{M}(-1)^{\varepsilon(g_i)-\varepsilon(g_k)}\begin{bmatrix} g_i \\ g_k \end{bmatrix}\begin{bmatrix} g_k \\ g_j \end{bmatrix} = 0 \quad \text{if } i \neq j, \text{ or } 1 \text{ if } i = j$$

3.1.2 Counting triangles

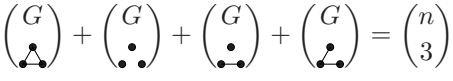
Let G be a graph with n vertices and ε edges, and complement $\overline{G}$. Suppose that we want to count the number of triangles in G and $\overline{G}$ combined. Clearly the number is



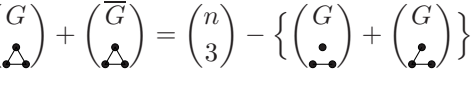
However



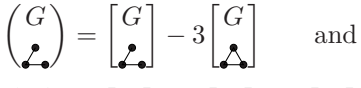
If we combine this with the identity

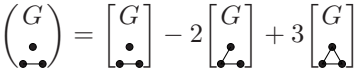


we obtain

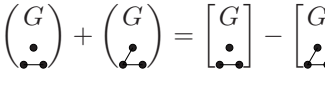


Now we can convert the induced subgraph counts to edge subgraph counts, obtaining

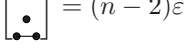




Adding these gives



Now the first of these is easy to evaluate, it consists of an edge and $n-2$ more vertices, so that



The second term is the total number of ways of choosing two incident edges at every vertex of G. If vertex u has degree d_u, then

$$\left[\begin{smallmatrix} G \\ \cdot^{\cdot}_{\cdot} \end{smallmatrix}\right] = \sum_u \binom{d_u}{2} = \frac{1}{2}\sum_u d_u^2 - \frac{1}{2}\sum_u d_u$$

The second sum is just 2ε. The first sum can be bounded using the Cauchy-Schwartz inequality

$$\sum_u d_u^2 \geq \frac{1}{n}(\sum_u d_u)^2 = \frac{4\varepsilon^2}{n}$$

Substituting into the equation for counting triangles gives

$$\binom{G}{\cdot^{\cdot}_{\cdot}} + \binom{\overline{G}}{\cdot^{\cdot}_{\cdot}} \geq \binom{n}{3} - \varepsilon(n-1) + \frac{2\varepsilon^2}{n}$$

as a bound on the number of triangles. This is Goodman's formula [72] for the combined number of triangles in G and $\overline{G}$. If n is fixed, this is a quadratic in ε, which has its minimum value when $\varepsilon = n(n-1)/4$, if $n \equiv 0$ or $n \equiv 1 \pmod 4$. Substituting this into the triangle count gives the minimum value $n(n-1)(n-5)/24$.

3.2 Multiplying subgraph counts

Either of the subgraph counts $\binom{G}{g_j}$ and $\left[\begin{smallmatrix} G \\ g_i \end{smallmatrix}\right]$ can be used as the basis of a vector space or module. In fact, products of the subgraph counts can be expressed as linear combinations, so that the subgraph counts form an algebra. If we choose two edges of a graph G, this can be indicated by $\binom{G}{K_2}\binom{G}{K_2}$. There are three possible outcomes — the same edge is chosen twice, two adjacent edges are chosen, or two independent edges are chosen. This can be expressed as

$$\binom{G}{K_2}\binom{G}{K_2} = \binom{G}{K_2} + 2\binom{G}{P_2} + 2\binom{G}{2K_2}$$

The coefficient of 2 arises because P_2 and $2K_2$ each have two edges, either of which can be chosen first.

This is a special case of a general theorem. Suppose that H_1 and H_2 are induced subgraphs of G. An *induced subgraph cover* of G by H_1 and H_2 is a pair of subsets $U_1, U_2 \subseteq V(G)$ such that $G[U_1] \cong H_1$ and $G[U_2] \cong H_2$, and $U_1 \cup U_2 = V(G)$. That is, U_1 and U_2 together cover all of $V(G)$, and they induce H_1 and H_2, respectively. Notice that it is not required that U_1 and U_2 be disjoint. The number of induced subgraph covers of G by H_1 and H_2 is denoted $\binom{G}{H_1,H_2}$.

Theorem 3.5. *Let G be a graph, and let H_1, H_2 be induced subgraphs of G. Let $m = |V(H_1)| + |V(H_2)|$, and let U be a subset of $V(G)$. Then*

$$\binom{G}{H_1}\binom{G}{H_2} = \sum_{|U|\le m} \binom{G[U]}{H_1, H_2}\binom{G}{G[U]}$$

Proof. The left-hand side of the equation asks for the number of induced subgraphs of G isomorphic to H_1, and the number of induced subgraphs isomorphic to H_2. If U_1 induces H_1, and U_2 induces H_2, then $U = U_1 \cup U_2$ will induce $\binom{G[U]}{H_1,H_2}$ copies of H_1 and H_2. Conversely, any induced subgraph $G[U]$ of G, where $|U| \le m$, which has an induced subgraph cover by H_1 and H_2 will contain $\binom{G[U]}{H_1,H_2}$ copies of H_1 and H_2. An induced cover by H_1 and H_2 can contain at most $m = |V(H_1)| + |V(H_2)|$ vertices. □

As an example of the use of this theorem, let $H_1 = P_3$ and let $H_2 = K_2$. Write (H_1) and (H_2) in place of $\binom{G}{H_1}$ and $\binom{G}{H_2}$. Theorem 3.5 gives the linear combination shown in Figure 3.3.

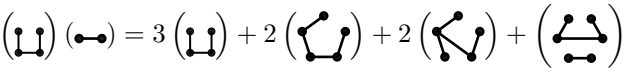


FIGURE 3.3
An example of Theorem 3.5

This theorem has the obvious extension to products of more than two subgraph counts, e.g.,

$$\binom{G}{H_1}\binom{G}{H_2}\binom{G}{H_3}\ldots = \sum_{|U|\le m} \binom{G[U]}{H_1, H_2, H_3\ldots}\binom{G}{G[U]}$$

A product of subgraph counts can always be written as a linear combination of subgraph counts.

An identical result holds for edge subgraphs. An *edge subgraph cover* of G by H_1 and H_2 is a pair of edge subgraphs $H_1' \cong H_1$ and $H_2' \cong H_2$ of G such that $E(H_1') \cup E(H_2') = E(G)$. That is, H_1' and H_2' together cover all the edges of G. The number of edge covers of G by H_1 and H_2 is denoted by $\left[\begin{smallmatrix} G \\ H_1,H_2 \end{smallmatrix}\right]$.

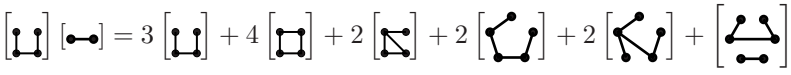
Theorem 3.6. *Let G be a graph, and let H_1, H_2 be edge subgraphs of G. Let $m = |E(H_1)| + |E(H_2)|$. Then*

$$\left[\begin{matrix} G \\ H_1 \end{matrix}\right]\left[\begin{matrix} G \\ H_2 \end{matrix}\right] = \sum_{\varepsilon(K)\le m} \left[\begin{matrix} K \\ H_1, H_2 \end{matrix}\right]\left[\begin{matrix} G \\ K \end{matrix}\right]$$

where the sum is over all edge subgraphs K of G with at most m edges.

Proof. The left-hand side of the equation asks for the number of edge subgraphs of G isomorphic to H_1, and the number of edge subgraphs isomorphic to H_2. Together H_1 and H_2 produce an edge cover of a subgraph K of G. Each edge subgraph K contains $\left[\begin{smallmatrix} G \\ H_1, H_2 \end{smallmatrix}\right]$ copies of H_1 and H_2. Clearly $\varepsilon(K) \leq m$. □

As an example of the use of this theorem, let $H_1 = P_3$ and let $H_2 = K_2$, the same graphs as in Figure 3.3. Write $[H_1]$ and $[H_2]$ in place of $\left[\begin{smallmatrix} G \\ H_1 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} G \\ H_2 \end{smallmatrix}\right]$. Theorem 3.6 gives the linear combination



It holds for any graph G.

This theorem also has the obvious extension to products of more than two subgraph counts, eg.,

$$\left[\begin{smallmatrix} G \\ H_1 \end{smallmatrix}\right]\left[\begin{smallmatrix} G \\ H_2 \end{smallmatrix}\right]\left[\begin{smallmatrix} G \\ H_3 \end{smallmatrix}\right]\ldots = \sum_{\varepsilon(K)\leq m} \left[\begin{smallmatrix} K \\ H_1, H_2, H_3 \ldots \end{smallmatrix}\right]\left[\begin{smallmatrix} G \\ K \end{smallmatrix}\right]$$

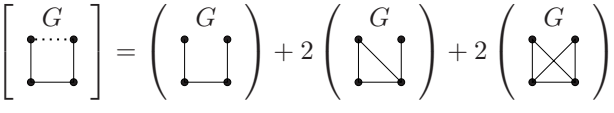
All products of subgraph counts can always be written as a linear combination of subgraph counts.

3.3 Mixed subgraphs

Given a graph H with vertex set $U = V(H)$, we consider edge subgraphs of G isomorphic to H, with certain edges forbidden. An edge-subgraph K of $\overline{H}$ is chosen such that $G[U]$ must not contain any edges of K. The pair (H, K) is called a *mixed* graph. The edges of H are *required* edges, and those of K are *forbidden* edges. So we are looking for edge subgraphs H of G such that K is an edge subgraph of $\overline{G}[V(H)]$. The number of mixed subgraphs of G isomorphic to (H, K) is denoted

$$\left\{\begin{smallmatrix} G \\ H, K \end{smallmatrix}\right\}$$

For example, suppose that $H \cong P_3$, a path of length three. Let the endpoints of the path be u and v, and let K be the edge uv. Then the mixed graph (H, K) contains a 3-path as required edges and a single forbidden edge, which together form a 4-cycle C_4. The diagonals of the cycle are neither required nor forbidden. H and K are both edge subgraphs, so that we can also denote $\left\{\begin{smallmatrix} G \\ H,K \end{smallmatrix}\right\}$ diagrammatically using solid edges for H and dotted edges for K, and use the notation for edge subgraphs:



The coefficients 2 arise because the graphs involved each contain two mixed subgraphs (H, K), that is, a P_3 whose endpoints are not adjacent.

Mixed subgraphs are a common generalization of induced subgraphs and edge subgraphs. For example, given H, we can take $K = \emptyset$. Then $(H, \emptyset)$ is a mixed subgraph of G if and only if H is an edge subgraph of G. But if we take $K = \overline{H}$, then $(H, \overline{H})$ is a mixed subgraph of G if and only if H is an induced subgraph of G.

Similar to Lemma 3.1, we have

Lemma 3.7. *Let G be a graph, and (H, K) a mixed graph. Let $m = |V(H)|$. Then*

$$\left\{ \begin{matrix} G \\ H, K \end{matrix} \right\} = \sum_{|U|=m} \left\{ \begin{matrix} G[U] \\ H, K \end{matrix} \right\} \binom{G}{G[U]}$$

Proof. Every mixed subgraph of G isomorphic to (H, K) is contained in a subset U of $V(G)$, which induces a subgraph $G[U]$. $G[U]$ accounts for $\left\{ \begin{matrix} G[U] \\ H,K \end{matrix} \right\}$ mixed subgraphs isomorphic to (H, K). □

Lemma 3.7 shows that the counts of mixed subgraphs can be written in terms of the basis of induced subgraphs. Equivalently, they can also be written in terms of the basis of edge-subgraphs. Identities similar to Theorems 3.5 and 3.6 can also be written.

3.4 Graph reconstruction

Ulam [182] asked whether the isomorphism type of a graph is determined by its subgraphs. This problem has become known as *Ulam's problem*, or the *graph reconstruction* problem. Given a graph G, we can form the vertex-deleted subgraphs $G - v$, where $v \in V(G)$. Ulam asked whether the isomorphism types of the $G - v$ determine the isomorphism type of G:

If G and H are graphs with $V(G) = V(H)$, and $G-v \cong H-v$, for all $v \in V(G)$, is $G \cong H$?

The answer to this question is still unknown. Graphs G and H which satisfy the hypothesis are said to be *reconstructions* of each other. If all reconstructions of G are isomorphic to G, then G is said to be *reconstructible*. It is easy to see that K_2 and $2K_1$ are reconstructions of each other. However, no other counterexamples are yet known. A few basic methods related to subgraph counting are presented here, beginning with *Kelly's lemma*.

Lemma 3.8. (Kelly's lemma) *Let g and G be graphs, with $|g| < |G|$. Then*

$$\binom{G}{g} = \frac{1}{|G| - |g|} \sum_{v \in V(G)} \binom{G - v}{g}$$

$$\begin{bmatrix} G \\ g \end{bmatrix} = \frac{1}{|G| - |g|} \sum_{v \in V(G)} \begin{bmatrix} G - v \\ g \end{bmatrix}$$

Proof. A given subgraph g of G is an induced or edge subgraph of $G - v$ whenever $v \in V(G) - V(g)$. □

Lemma 3.8 shows how to count all proper subgraphs of G. For example, the number of edges, the number of triangles, 4-cycles, etc., can all be easily counted. Some spanning subgraphs can then be counted using Theorems 3.5 and 3.6. For example, let $n = |G|$. A spanning tree of G is an acyclic graph with $n-1$ edges. Any subgraph with $n - 1$ edges is either a spanning tree, or else a graph with fewer than n vertices. We use Theorem 3.6 to construct a linear combination of edge subgraph counts for the expression

$$\binom{\begin{bmatrix} G \\ K_2 \end{bmatrix}}{n-1}$$

which counts the subgraphs of G containing $n - 1$ distinct edges. The result is a linear combination of subgraph counts for various edge subgraphs with fewer than n vertices, plus the number of spanning trees. We solve the equation to obtain the number of spanning trees. For example, when $n = 5$, we choose four distinct edges of G. Refer to Figure 3.2 to see that there are four graphs on four vertices with four edges. Any graph on five vertices with four edges is a spanning tree. Therefore the number of spanning trees can be written in terms of subgraphs on at most four edges. Hamilton cycles can be counted in a similar way. See [22] or [113] for further details.

The edge analogue of the graph reconstruction problem is:

Let G and H be graphs with $|G| = |H|$ and $\varepsilon(G) = \varepsilon(H)$, such that there is a one-to-one correspondence between $E(G) = \{e_1, e_2, \ldots, e_m\}$ and $E(H) = \{e'_1, e'_2, \ldots, e'_m\}$, where e_i corresponds with e'_i, and $m = \varepsilon(G)$. Suppose that $G - e_i \cong H - e'_i$, for all i. Is $G \cong H$?

Graphs satisfying this hypothesis are said to be *edge-reconstructions* of each other. If all reconstructions of G are isomorphic to G, then G is said to be *edge-reconstructible*. It is easy to see that $K_3 + K_1$ and $K_{1,3}$ are edge-reconstructions of each other. However, no other counterexamples are yet known. Many families of graphs have been proved to be edge-reconstructible.

There is a vast literature on graph reconstruction. See [22, 21, 131, 113, 139] for further information. One of the strongest results known is *Nash-Williams' lemma*, described in the next section.

3.4.1 Nash-Williams' lemma

Let G and H be edge-reconstructions of each other, with $V(G) = V(H)$, such that $G \ncong H$. Let $n = |V(G)|$. We count various mappings from G to H. Any permutation of $V(G)$ will usually map some edges of G to edges of H, and some edges of G to

non-edges of H. Given a graph K with $V(K) \subseteq V(G)$, we use $|G \to H|_K$ to denote the number of permutations of $V(G)$ such that only the edges of K map to edges of H. Similarly $|K \to H|$ indicates the number of one-to-one mappings of $V(K)$ into $V(H)$ that map all edges of K to edges of H. Clearly $|K \to H| = |\text{AUT}(K)| \cdot \begin{bmatrix} H \\ K \end{bmatrix}$. This is because *every* edge subgraph of H isomorphic to K can be permuted by $\text{AUT}(K)$ without changing it, and because K can be mapped to any edge subgraph of H that is isomorphic to K.

We now consider all graphs X that contain K as a spanning edge subgraph, and sum $|G \to H|_X$. This counts all those permutations of $V(G)$ such that at least the edges of K map to edges of H. Therefore

$$\sum_X |G \to H|_X = |K \to H| = |\text{AUT}(K)| \cdot \begin{bmatrix} H \\ K \end{bmatrix} \tag{3.4}$$

where the sum is over all graphs X that contain K as a spanning edge-subgraph.

But the graphs on n vertices form a partially ordered set, as in Figures 3.1 and 3.2, and we can use inclusion-exclusion, as in Theorem 3.3, to invert this formula. The result is

$$|G \to H|_K = \sum_X (-1)^{\varepsilon(X)-\varepsilon(K)} |\text{AUT}(X)| \cdot \begin{bmatrix} H \\ X \end{bmatrix} \tag{3.5}$$

where the sum is again over all graphs X containing K as a spanning edge subgraph. Now because G and H are edge-reconstructions of each other, we have $\begin{bmatrix} G \\ X \end{bmatrix} = \begin{bmatrix} H \\ X \end{bmatrix}$ for all edge subgraphs X with fewer than $\varepsilon(G)$ edges. And it is clear that both are zero if X has more than $\varepsilon(G)$ edges.

Lemma 3.9. (Nash-Williams' lemma) *Let G and H be edge-reconstructions of each other. Let K be a graph with $|G|$ vertices. Then*

$$|G \to G|_K - |G \to H|_K = (-1)^{\varepsilon(G)-\varepsilon(K)} |\text{AUT}(K)| \left\{ \begin{bmatrix} G \\ K \end{bmatrix} - \begin{bmatrix} H \\ K \end{bmatrix} \right\}$$

Proof. We subtract $|G \to G|_K$ and $|G \to H|_K$ using Equation (3.5). All terms cancel except possibly the terms with $X = K$ (when $K \cong G$ or $K \cong H$), thereby giving the result. □

We can choose $K = G$ in Lemma 3.9. Notice that if $G \cong H$, then $\begin{bmatrix} G \\ G \end{bmatrix} = \begin{bmatrix} H \\ G \end{bmatrix} = 1$, so that the equation of Lemma 3.9 is zero. But if $G \ncong H$, the equation determines $|G \to G|_G - |G \to H|_G = |\text{AUT}(G)| \neq 0$.

Consider now graphs G for which $\varepsilon(G) > \binom{n}{2}$. We choose K to be the empty graph on n vertices. Then $|G \to G|_K$ counts the number of permutations of $V(G)$ such that no edge of G is mapped to an edge of G, which must be zero. Therefore $|G \to G|_K = |G \to H|_K = 0$, which implies that $\begin{bmatrix} G \\ G \end{bmatrix} = \begin{bmatrix} H \\ G \end{bmatrix}$, so that $G \cong H$. This gives:

Corollary 3.10. *A graph G on n vertices with $\varepsilon(G) > \binom{n}{2}$ is edge-reconstructible.*

This result can be improved slightly. See [22] for further details.

Exercises

3.4.1 Construct the matrix $\begin{bmatrix} g_j \\ g_i \end{bmatrix}$ for the 11 graphs on four vertices. Find its inverse.

3.4.2 Calculate the minimum triangle count for Goodman's formula when $n \equiv 2$ and $n \equiv 3 \pmod 4$. Find graphs that achieve the minimum.

3.4.3 Construct the subgraph identities for $\binom{G}{P_2}\binom{G}{P_2}$ and $\begin{bmatrix} G \\ P_2 \end{bmatrix}\begin{bmatrix} G \\ P_2 \end{bmatrix}$, where P_2 is a path of length two.

3.4.4 Construct the subgraph identities for the number of spanning trees in a graph with six vertices.

3.4.5 Suppose that $G = L(H)$, the line-graph of H. If $e \in E(H)$, show that the vertex-deleted subgraph $G - e = L(H - e)$.

3.4.6 Show that the vertex-deleted subgraphs $G - v$ can be determined from the edge-deleted subgraphs $G - e$. Conclude that any graph which is vertex-reconstructible is also edge-reconstructible.

3.5 Notes

There are many surveys of the graph reconstruction problem. Some of them are BONDY [22], BONDY and HEMMINGER [21], NASH-WILLIAMS [131], LAURI and SCAPELLATO [113], and RAMACHANDRAN [139]. Theorems 3.5 and 3.6 are from KOCAY [106], and are also treated in BONDY [22] and LAURI and SCAPELLATO [113]. The description of Nash-Williams' lemma is based on BONDY [22]. It originally appeared in NASH-WILLIAMS [131].

4

Some Special Classes of Graphs

4.1 Bipartite graphs

A graph G is said to be *bipartite* if $V(G)$ can be divided into two sets X and Y such that each edge has one end in X and one end in Y. For example, the cube is a bipartite graph, where the bipartition (X, Y) is illustrated by the coloring of the nodes in Figure 4.1.

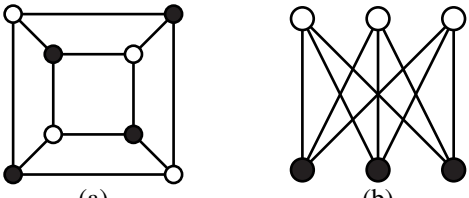


FIGURE 4.1
Two bipartite graphs

The maximum number of edges in a simple bipartite graph in which X and Y are the two sides of the bipartition is clearly $|X| \cdot |Y|$. The *complete bipartite* graph $K_{m,n}$ has $|X| = m$, $|Y| = n$, and $\varepsilon = mn$. For example, $K_{3,3}$ is illustrated in Figure 4.1.

Lemma 4.1. *A simple, bipartite graph G has at most $|G|^2/4$ edges.*

Proof. Let G have bipartition (X, Y), where $|X| = x$ and $|Y| = n - x$, where $n = |G|$. Then $\varepsilon \leq x(n - x) = nx - x^2 = n^2/4 - (n/2 - x)^2 \leq n^2/4$. □

If $C = (x_1, y_1, x_2, y_2, \ldots)$ is a cycle in a bipartite graph G, then consecutive vertices of C must be alternately in X and Y, the two sides of the bipartition. It follows that $\ell(C)$ is even. In fact, any graph in which all cycles have even length must be bipartite.

Theorem 4.2. *G is bipartite if and only if all cycles of G have even length.*

Proof. Let G be a connected graph in which all cycles have even length. Pick any

$x \in V(G)$ and set $X = \{v : \text{DIST}(x, v) \text{ is even}\}$, and $Y = V(G) - X$. Clearly X and Y partition $V(G)$ into two parts. We must show that there are no edges with both endpoints in X or Y. Suppose that uv is an edge with $u, v \in X$. Let P_u be a shortest xu-path, that is, a path of length $\text{DIST}(x, u)$, and let P_v be a shortest xv-path. Then $\ell(P_u)$ and $\ell(P_v)$ are both even. Say $\ell(P_u) \leq \ell(P_v)$. P_u and P_v both begin at point x. They do not both contain u, or $P_u uv$ would be a shortest xv-path of length $\ell(P_u) + 1$, an odd number. So let z be the last point in common to P_u and P_v. This defines the cycle $C = P_u[z, u]uvP_v[v, z]$. Here $P_u[z, u]$ denotes the portion of P_u from z to u and $P_v[v, z]$ denotes the portion of P_v from v to z. The length of C is then $\ell(P_u[z, u]) + \ell(P_u[v, z]) + 1 = \ell(P_u) + \ell(P_v) - 2\text{DIST}(x, z) + 1$, which is odd, a contradiction. Therefore no edge uv has both endpoints in X. Similarly, no edge uv has both endpoints in Y. Because a graph is bipartite if and only if every component is bipartite, this completes the proof. □

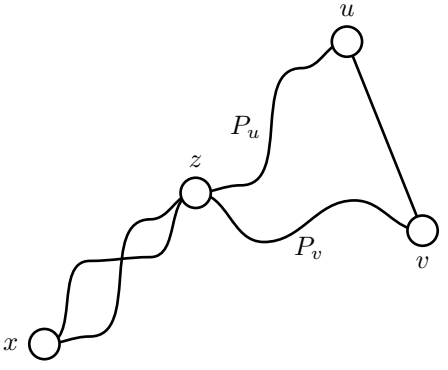


FIGURE 4.2
Two paths in a bipartite graph

Lemma 4.3. *If G is a k-regular bipartite graph, where $k > 0$, with bipartition (X, Y), then $|X| = |Y|$.*

Proof. Because each edge has one end in X, we can write $\varepsilon = \Sigma_{x \in X}\text{DEG}(x) = k \cdot |X|$. Similarly, $\varepsilon = \Sigma_{y \in Y}\text{DEG}(y) = k \cdot |Y|$. Therefore $k \cdot |X| = k \cdot |Y|$. Because $k > 0$, it follows that $|X| = |Y|$. □

Exercises

4.1.1 The *k-cube* Q_k is a graph whose vertex set consists of all binary vectors of length k:

$$V(Q_k) = \{(a_1, a_2, \ldots, a_k) : a_i \in \{0, 1\}\}$$

Thus there are 2^k vertices. The edges of Q_k are formed by joining two vertices $\overrightarrow{a} = (a_1, a_2, \ldots, a_k)$ and $\overrightarrow{b} = (b_1, b_2, \ldots, b_k)$ if $\overrightarrow{a}$ and $\overrightarrow{b}$

differ in exactly one coordinate, that is, $a_i = b_i$ for all i but one. Q_3 is displayed in Figure 4.3. Prove that Q_k is bipartite. Describe a bipartition of Q_k.

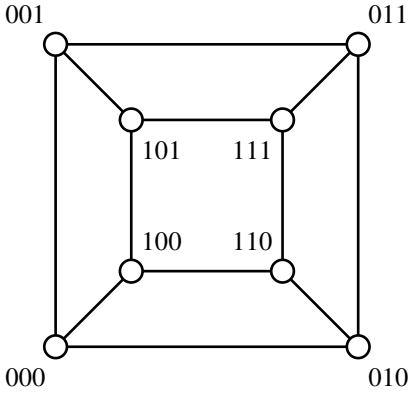


FIGURE 4.3
The 3-cube, $\boldsymbol{Q_3}$

4.1.2 Prove that $\varepsilon(Q_k) = k2^{k-1}$.

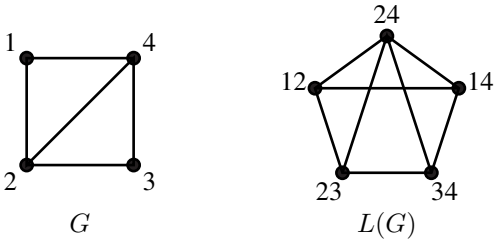
4.1.3 Describe in pseudo-code an algorithm to find a bipartition of G, or to determine that G is not bipartite. Describe the data-structures needed, and calculate the complexity (should be $O(\varepsilon)$).

4.1.4 Let G be a bipartite simple graph with bipartition (X, Y) and n vertices. Let δ_X be the minimum degree among the vertices of X, and δ_Y be the minimum degree among the vertices of Y. Show that if $\delta_X + \delta_Y > n/2$, then G is connected, where $\delta_X, \delta_Y > 0$.

4.2 Line graphs

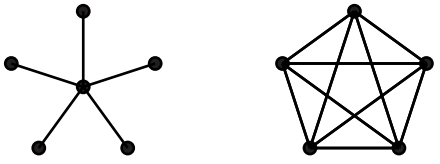
Two edges of a graph G are adjacent if they share a common endpoint. The *line-graph* of G is a graph $L(G)$ which describes the adjacencies of the edges of G. Thus, every vertex of $L(G)$ corresponds to an edge uv of G, so that $|L(G)| = \varepsilon(G)$. This is illustrated in Figure 4.4.

A line-graph can always be decomposed into complete subgraphs. For a vertex $v \in V(G)$ lies on $\textsc{Deg}(v)$ distinct edges all of which share the endpoint v. The $\textsc{Deg}(v)$ corresponding vertices of $L(G)$ form a complete subgraph containing $\binom{\textsc{Deg}(u)}{2}$ edges. Every edge of $L(G)$ is contained in exactly one such complete subgraph.



G $L(G)$

FIGURE 4.4
Constructing a line-graph



G $L(G)$

$\text{DEG}(v)$ edges $\binom{\text{DEG}(v)}{2}$ edges

FIGURE 4.5
Complete subgraph in a line-graph

This gives the following theorem:

Theorem 4.4. $\varepsilon(L(G)) = \sum_{u \in V(G)} \binom{\text{DEG}(u)}{2}$

Exercises

4.2.1 Find the line-graph of the cube.

4.2.2 Construct $\overline{L(K_5)}$ and show that it is isomorphic to the Petersen graph.

4.2.3 Let G be any graph. If we insert a vertex of degree two into each edge, we obtain a new graph $S(G)$, called the *subdivision graph* of G. For example, $S(K_4)$ is illustrated in Figure 4.6. Prove that $S(G)$ is always bipartite, and find a formula for $\varepsilon(S(G))$.

4.2.4 The graph P in Figure 4.7 is called the 3-prism. Find the line-graphs of the subdivision graphs of K_4 and P. Draw them as neatly as possible. What can you say in general about constructing the line-graph of the subdivision graph of a 3-regular graph?

4.2.5 We know that

$$\sum_u \text{DEG}(u) = 2\varepsilon(G),$$

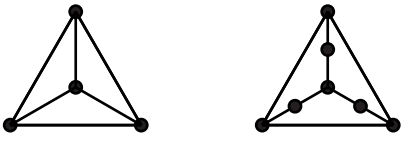


FIGURE 4.6
Subdivision graph of $\boldsymbol{K_4}$

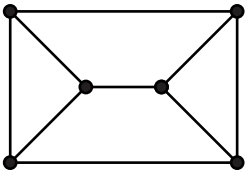


FIGURE 4.7
The 3-prism

and that

$$\sum_u \binom{\text{DEG}(u)}{2} = \varepsilon(L(G)).$$

Can you find a similar way of interpreting

$$\sum_u \binom{\text{DEG}(u)}{3}?$$

Assume first that there are no triangles in G.

4.2.6 Suppose that a graph G is represented by its adjacency matrix. Write a program to print out the adjacency lists of $L(G)$, but do not store either adjacency lists or an adjacency matrix for $L(G)$; just print it out. Also print out a list of the edges of G, in order to give a numbering to the vertices of $L(G)$.

4.2.7 Notice that $L(K_3) \cong L(K_{1,3}) \cong K_3$ (see Figure 4.8). Prove that if G and H are any other graphs, then $G \cong H$ if $L(G) \cong L(H)$.

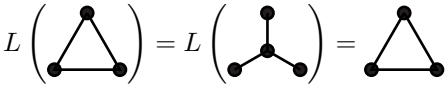


FIGURE 4.8
Two graphs with isomorphic line-graphs

4.3 Moore graphs

The length of the shortest cycle in a graph G is called its *girth*, denoted $\gamma(G)$. For example, the cube has girth four. Graphs with fixed degree k and fixed girth often have interesting properties. For example, let G be a k-regular graph of girth four, and pick any vertex u in G. There are k vertices at distance one from u. Because G has no triangles, there are at least $k-1$ vertices at distance two from u, as shown in Figure 4.9. Therefore, $|G| \geq 1+k+(k-1) = 2k$. There is only one such graph with $|G| = 2k$, and that is the complete bipartite graph $K_{k,k}$.

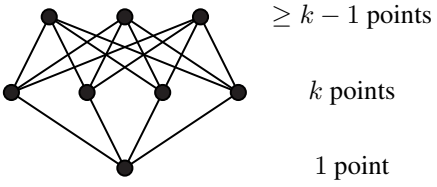


FIGURE 4.9
$K_{k,k}$

Now let G be a k-regular graph of girth five, and let u be any vertex. There are k vertices at distance one from u. Because G has no 4-cycles, each point at distance one is adjacent to $k-1$ more vertices at distance two, so that $|G| \geq 1+k+k(k-1) = k^2+1$.

Problem. Are there any k-regular graphs G of girth five with $|G| = k^2+1$?

These graphs are called *Moore graphs*. Let $n = |G|$. A 1-regular graph cannot have $\gamma = 5$, so $k \geq 2$. If $k = 2$, then $n = 2^2+1 = 5$. G is a cycle of length five. This is the unique Moore graph of degree two.

If $k = 3$, then $n = 3^2+1 = 10$. There are three vertices at distance one from u, and six at distance two, as illustrated in Figure 4.10. Consider vertex u_6. $u_6 \not\to u_5$, because this would create a triangle, whereas $\gamma = 5$. Without loss of generality, we can take $u_6 \to u_8$. Were we now to join $u_6 \to u_7$, this would create a 4-cycle (u_6, u_7, u_3, u_8), which is not allowed. Therefore, without loss of generality, we can take $u_6 \to u_9$. This is shown in Figure 4.10.

There is now only one way of completing the graph so as to maintain $\gamma = 5$. Vertex u_9 cannot be joined to u_5, u_8, or u_{10}. Therefore $u_9 \to u_7$. Similarly $u_8 \to u_{10}$, etc. The completed graph is shown in Figure 4.10, and has been redrawn in Figure 4.11 (check that this is the same graph).

Thus, we have proved the following.

Theorem 4.5. *The Petersen graph is the unique Moore graph of degree three.*

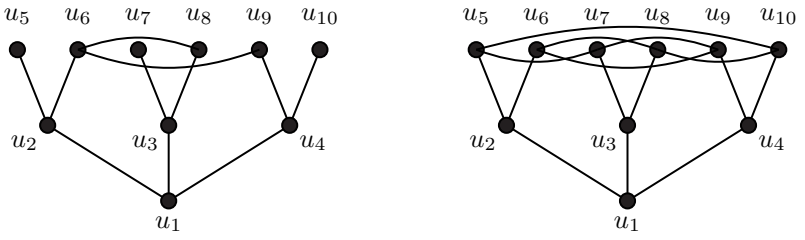


FIGURE 4.10
A Moore graph of degree three

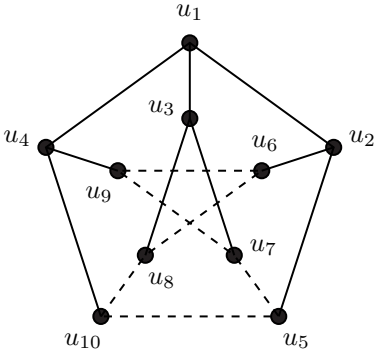


FIGURE 4.11
The Petersen graph

There is a very elegant theorem proving that Moore graphs can exist only for special values of k.

Theorem 4.6. *A Moore graph of degree k can exist only if $k = 2, 3, 7,$ or 57.*

Proof. Let G be a Moore graph with adjacency matrix A and consider A^2. Entry $[i, j]$ of A^2 is the number of u_iu_j-paths of length two. If $u_i \longrightarrow u_j$, then there is no 2-path from u_i to u_j, because $\gamma = 5$. Therefore, $[A^2]_{ij} = 0$ if $[A]_{ij} = 1$.

If $u_i \not\longrightarrow u_j$ then $\text{DIST}(u_i, u_j) > 1$. It is shown in Exercise 3.3.3 that $\text{DIST}(u_i, u_j)$ is always at most 2. Therefore, if $u_i \not\longrightarrow u_j$ there must be a 2-path connecting u_i to u_j. There cannot be two such 2-paths, for that would create a 4-cycle containing u_i and u_j. Therefore, $[A^2]_{ij} = 1$ if $[A]_{ij} = 0$.

It follows that the matrix $A^2 + A$ consists of all 1's off the diagonal. The diagonal elements all equal k, the degree of G. The number of vertices is $n = k^2 + 1$.

$$A^2 + A = \begin{bmatrix} k & & & 1 \\ & k & & \\ 1 & & \ddots & \\ & & & k \end{bmatrix}_{n\times n}$$

We can find the eigenvalues of this matrix. Write $B = A^2 + A$. If $\mathbf{x}$ is an eigenvector of A with eigenvalue α, then

$$B\mathbf{x} = (A^2 + A)\mathbf{x} = AA\mathbf{x} + A\mathbf{x} = \alpha A\mathbf{x} + \alpha\mathbf{x} = (\alpha^2 + \alpha)\mathbf{x}$$

so that $\beta = \alpha^2 + \alpha$ is an eigenvalue of B. To find the eigenvalues of B, we solve $\det(\lambda I - B) = 0$ for λ.

$$\det(\lambda I - B) = \begin{bmatrix} \lambda - k & & & \text{-1} \\ & \lambda - k & & \\ & \text{-1} & \ddots & \\ & & & \lambda - k \end{bmatrix}_{n\times n}$$

Adding rows 2 to n onto row 1 gives

$$\begin{vmatrix} \lambda - k - n + 1 & \lambda - k - n + 1 & \cdots & \\ -1 & \lambda - k & & \\ & & \ddots & \\ & & & \lambda - k \end{vmatrix}_{n\times n}$$

$$= (\lambda - k - n + 1) \begin{vmatrix} 1 & 1 & \cdots & \\ -1 & \lambda - k & & \\ & & \ddots & \\ & & & \lambda - k \end{vmatrix}_{n\times n}$$

Now add the first row to each row to get

$$(\lambda - k - n + 1) \begin{vmatrix} 1 & 1 & \cdots & \\ 0 & \lambda - k + 1 & & \\ & & \ddots & \\ & & & \lambda - k + 1 \end{vmatrix}_{n\times n}$$

$$= (\lambda - k - n + 1)(\lambda - k + 1)^{n-1} = 0.$$

Therefore the eigenvalues of B are

$$\lambda = \begin{cases} \beta_1 = k + n - 1 & (once), \\ \beta_2 = k - 1 & (\text{n-1}times). \end{cases}$$

Because $\beta = \alpha^2 + \alpha$, we can solve for $\alpha = \frac{1}{2}(-1 \pm \sqrt{1+4\beta})$. Should we take the plus or minus sign? Because $n = k^2 + 1$, the value $\beta_1 = k^2 + k + 1$ gives

$$\alpha = \frac{1}{2}(-1 \pm \sqrt{4k^2+4k+1}) = \frac{1}{2}\{-1 \pm (2k+1)\} = k \text{ or } -k-1.$$

Now β_1 occurs only once as an eigenvalue, so we must choose only one of these. G is k-regular, so that the rows of A all sum to k. Thus, if $\mathbf{x}$ is the vector of all 1's, then $A\mathbf{x} = k\mathbf{x}$, so that k is in fact an eigenvalue of A.

Consider now β_2. The corresponding eigenvalues of A are

$$\alpha = \begin{cases} \alpha_1 = \frac{1}{2}(-1+\sqrt{4k-3}) & (m_1 \text{ times}), \\ \alpha_2 = \frac{1}{2}(-1-\sqrt{4k-3}) & (m_2 \text{ times}). \end{cases}$$

The total multiplicity is $m_1 + m_2 = n - 1 = k^2$. Because the trace of A, that is, the sum of its diagonal elements, also equals the sum of its eigenvalues, we can write

$$\begin{aligned} m_1 + m_2 &= k^2 && \text{(sum of multiplicities)} \\ \alpha_1 m_1 + \alpha_2 m_2 + k &= 0 && \text{(sum of eigenvalues)} \end{aligned}$$

Solving these equations for m_1 and m_2 gives

$$m_1 = \frac{-1}{2}\left\{\frac{-k^2+2k}{\sqrt{4k-3}} - k^2\right\}$$

and

$$m_2 = \frac{1}{2}\left\{\frac{-k^2+2k}{\sqrt{4k-3}} + k^2\right\}$$

The multiplicities m_1 and m_2 are integers. Consider the fraction

$$\frac{-k^2+2k}{\sqrt{4k-3}}.$$

If $k = 2$, the numerator is 0. If $k \neq 2$, then $\sqrt{4k-3}$ must be an integer, so that $4k-3$ is a perfect square, say $4k-3 = s^2$. Then

$$k = \frac{1}{4}(s^2+3),$$

and

$$-k^2 + 2k = \frac{1}{16}(-s^4 + 2s^2 + 15).$$

This expression must be divisible by $\sqrt{4k-3} = s$. If s does not divide 15, it cannot be an integer, because the other 2 terms have no s in the denominator. Therefore $s = 1, 3, 5$, or 15. The corresponding values of k, m_1, m_2, α_1, and α_2 are shown in the following table:

s	k	n	m_1	m_2	α_1	α_2
1	1	2	1	0	0	-1
3	3	10	4	5	1	-2
5	7	50	21	28	2	-3
15	57	3250	1520	1729	7	-8

The value $k = 1$ does not correspond to a graph. $k = 3$ gives the Petersen graph. There is a unique Moore graph with $k = 7$ and $n = 50$, called the Hoffman-Singleton graph. It is not known whether a Moore graph with $k = 57$ and $n = 3250$ exists. The 5-cycle is a Moore graph with $k = 2$. Its eigenvalues are

$$\alpha_1 = \frac{1}{2}(-1 + \sqrt{5})$$

and

$$\alpha_2 = \frac{1}{2}(-1 - \sqrt{5}),$$

with multiplicities $m_1 = m_2 = 2$. □

The *diameter* of a graph is the maximum distance between any two vertices,

$$\text{diam}(G) = \max\{\textsc{Dist}(u, v) : u, v \in V(G)\}.$$

Thus, Moore graphs have diameter two.

Exercises

4.3.1 Let G be a Moore graph of degree k, with $n = k^2 + 1$ vertices. Let v be any vertex of G. Prove that there are exactly

$$\frac{k(k-1)^2}{2}$$

pentagons containing v. Conclude that G contains

$$\frac{k(k^2+1)(k-1)^2}{10}$$

pentagons, so that $k \not\equiv 4 \pmod 5$.

4.3.2 Let G be as in exercise 4.3.1. Prove that every $v \in V(G)$ is contained in exactly

$$\frac{k(k-1)^2(k-2)}{2}$$

hexagons and in

$$\frac{k(k-1)^2(k-2)(k-3)}{2}$$

heptagons.

4.3.3 Show that in a k-regular graph of girth five, with $n = k^2 + 1$ vertices, the distance $\textsc{Dist}(u, v)$ between any two vertices is at most two. *Hint*: Show that $\textsc{Dist}(u, v) = 3$ implies the existence of a 4-cycle.

4.4 Euler tours

Figure 4.12 shows a drawing of K_5 illustrating a walk in which each edge is covered exactly once.

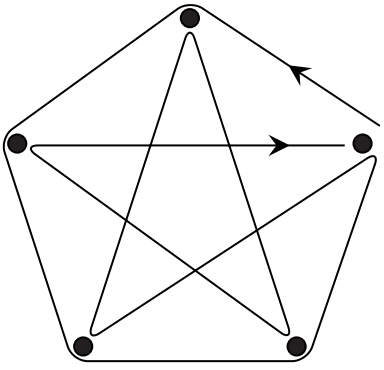


FIGURE 4.12
A traversal of $\boldsymbol{K_5}$

A walk which covers each edge of a graph G exactly once is called an *Euler trail* in G. A closed Euler trail is called an *Euler tour*. Thus Figure 4.12 shows that K_5 has an Euler tour and we say that K_5 is *Eulerian*. It is easy to prove that a graph is Eulerian when all its degrees are even.

Theorem 4.7. *A connected graph G has an Euler tour if and only if all degrees of G are even.*

Proof. Let W be a closed Euler trail in G, beginning at vertex v. Each time that W enters a vertex u, it also must exit it. Therefore W uses an even number of edges at each vertex $u \neq v$. Because the trail is closed, the same is true of v. Because W covers every edge of G exactly once, all degrees must be even.

Conversly suppose that all degrees of G are even. The proof is by induction on the number of edges of G. The smallest connected graphs with even degrees are K_1 and K_3, and both of these are Eulerian (for K_1, $W = \emptyset$ is an Euler trail). If the theorem is not true, let G be the smallest graph (i.e., smallest ε) with even degrees with no closed Euler trail. Clearly $\delta(G) \geq 2$, so that G contains a cycle, which is an Eulerian subgraph. Let C be the largest Eulerian subgraph which G contains. Then $\varepsilon(C) < \varepsilon(G)$. The complementary subgraph $G - C$ also has even degrees, and because it is smaller than G, each component of it must be Eulerian. Furthermore, C intersects each component of $G - C$. We can now make an Euler trail in G from C, by inserting into C Euler trails of each component K of $G - C$, as the walk in C reaches each K in turn. Therefore G is Eulerian. By induction, all connected graphs with even degrees are Eulerian. □

Notice that this theorem is true for multigraphs as well as simple graphs. If a connected graph G has exactly two vertices, u and v, of odd degree, then we can add an extra edge uv to G to get G', which will then have all even degrees. G' may now have multiple edges. If we now choose an Euler tour W in G' beginning at v, we can number the edges so that the new edge vu is the last edge traversed. Then $W - uv$ will be an Euler trail in G beginning at u and ending at v. This is illustrated in Figure 4.13.

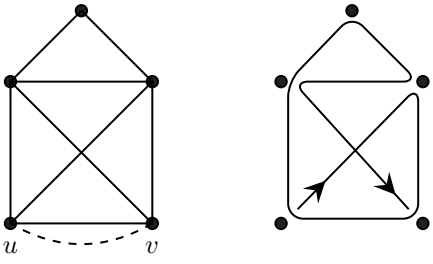


FIGURE 4.13
An Euler trail

If there are more than two vertices of odd degree, then it is clear that G cannot have an Euler trail.

4.4.1 An Euler tour algorithm

The proof of Theorem 4.7 is essentially an algorithm to find an Euler tour in a connected graph G. The algorithm works by building a walk from a starting vertex u. It takes the first edge $e_0 = uv$ incident on u and follows it. Then it takes the first edge e_1 incident on v and follows it, and so forth. Because the degrees are all even, it must eventually return to u. At this point, it will have found a closed walk in G that is a sub-tour of an Euler tour. All the vertices visited in the sub-tour are stored on an array called the *ScanQ*. It then repeats this process at u, finding another sub-tour. The sub-tours are then linked together to create one sub-tour. It continues like this until all the edges at u have been used up. It then moves to the next vertex on the *ScanQ* and builds a sub-tour there, always linking the sub-tours found into the existing tour. When the algorithm completes, all the vertices of G are in the *ScanQ* array, because G is connected. Therefore we have an Euler tour.

The Euler tour is stored as a linked list of edges. This makes it easy to insert a sub-tour into the list at any location. If $e = uv$ is an edge of G, then we write *nextEdge*$\langle e \rangle$ and *prevEdge*$\langle e \rangle$ for the next and previous edges in a tour, respectively. The adjacency list for vertex u is denoted by *Graph*$[u]$. This is a linked list of incident edges.

When the algorithm begins building a sub-tour at vertex u, it needs to know an edge at u currently in the Euler tour, if there is one. This is stored as *EulerEdge*$[u]$. It

allows the algorithm to insert a sub-tour into the existing tour at that location in the linked list.

Algorithm 4.4.1: EULERTOUR(G)

comment: Construct an Euler tour in G

$ScanQ[1] \leftarrow 1$
$QSize \leftarrow 1$
$k \leftarrow 1$
while $k \leq QSize$
do
- $u \leftarrow ScanQ[k]$
- **while** $Graph[u] \neq$ **null**
  - **do**
    - $e_0 \leftarrow Graph[u]$ "First edge at u."
    - $v \leftarrow$ other endpoint of e_0
    - Remove edge e_0 from the graph.
    - $e_1 \leftarrow e_0$
    - **while** $v \neq u$
      - **do**
        - **if** $v \notin ScanQ$
          - **then** $QSize \leftarrow QSize + 1$; $ScanQ[QSize] \leftarrow v$
        - $e_2 \leftarrow Graph[v]$ "First edge at v."
        - $nextEdge\langle e_1 \rangle \leftarrow e_2$
        - $prevEdge\langle e_2 \rangle \leftarrow e_1$
        - **if** $EulerEdge[v] =$ **null**
          - **then** $EulerEdge[v] \leftarrow e_2$
        - $e_1 \leftarrow e_2$
        - $v \leftarrow$ other endpoint of e_1
        - Remove edge e_1 from the graph.
    - **comment:** a sub-tour at u has just been completed
    - $prevEdge\langle e_0 \rangle \leftarrow e_1$
    - $nextEdge\langle e_1 \rangle \leftarrow e_0$
    - **if** $EulerEdge[u] = NULL$
      - **then** $EulerEdge[u] \leftarrow e_0$
      - **else**
        - **comment:** insert the sub-tour at u into the existing tour at u
        - $e_1 \leftarrow EulerEdge[u]$
        - $e_2 \leftarrow prevEdge\langle e_1 \rangle$
        - $e_3 \leftarrow prevEdge\langle e_0 \rangle$
        - $nextEdge\langle e_2 \rangle \leftarrow e_0$
        - $nextEdge\langle e_3 \rangle \leftarrow e_1$
- $k \leftarrow k + 1$

Algorithm 4.4.1 is very efficient. For each vertex u, all incident edges are considered. Each edge is linked into the Euler tour. This takes $\text{DEG}(u)$ steps. Several sub-tours at u may be linked into the Euler tour being constructed. There are at most $\text{DEG}(u)/2$ sub-tours at u. If follows that the complexity is determined by

$$\sum_u \text{DEG}(u) = 2\varepsilon(G) = O(\varepsilon).$$

Exercises

4.4.1 Program Algorithm $\text{EULERTOUR}(G)$.

4.4.2 Let G be a connected graph in which $2k$ of the vertices are of odd degree. Show that there are k trails $W_1, W_2, \ldots, W_k$ such that, taken together, $W_1, W_2, \ldots, W_k$ cover each edge of G exactly once.

4.4.3 Let W be an Euler tour in G. To what subgraph of the line-graph $L(G)$, does W correspond?

4.4.4 Show that any Euler tour of a graph G can be written as a union of cycles.

4.4.5 What does the following algorithm do when input a connected graph G? What is its complexity?

procedure $\text{TESTGRAPH}(G)$
ScanQ$[1] \leftarrow 1$
QSize $\leftarrow 1$
Tag$[1] \leftarrow 1$
$k \leftarrow 1$
while $k \leq$ *QSize*
do { **for all** $v \longrightarrow u$
do { **if** $v \notin$ *ScanQ*
then { *QSize* $\leftarrow$ *QSize* $+ 1$
ScanQ[*QSize*] $\leftarrow v$
Tag$[v] \leftarrow -$*Tag*$[u]$
else if *Tag*$[v] =$ *Tag*$[u]$
then return (**false**)
$k \leftarrow k + 1$
return (**true**)

4.5 Notes

The theorem on Moore graphs is due to HOFFMANN and SINGLETON [85]. The application of algebraic methods to graph theory is treated in BIGGS [17] and GODSIL and ROYLE [70]. The eigenvalues of graph adjacency matrices is a vast topic. See the surveys by HOFFMAN [84], SCHWENK and WILSON [156], or the book by CVETKOVIC, DOOB, and SACHS [39]. An excellent description of Euler tour algorithms can be found in GOULD [73].

5

Trees and Cycles

5.1 Introduction

A *tree* is a connected graph that has no cycles. Figure 5.1 shows a number of trees.

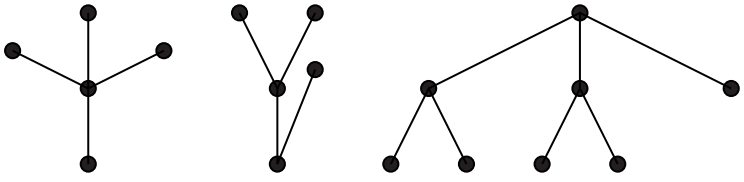


FIGURE 5.1
Various trees

Trees are the smallest connected graphs; remove any edge from a tree and it becomes disconnected. As well as being an important class of graphs, trees are important in computer science as data structures, and as objects constructed by search algorithms. A fundamental property of trees is that all trees on n vertices have the same number of edges.

Theorem 5.1. *If G is a tree, then $\varepsilon(G) = |G| - 1$.*

Proof. The proof is by induction on $|G|$. If $|G| = 1$, then $G = K_1$, which is a connected graph with no cycle, so that $\varepsilon = 0$. Similarly, if $|G| = 2$, then $G = K_2$, which has $\varepsilon = 1$. Assume that the result is true whenever $|G| \leq t$, and consider a tree G with $|G| = t + 1$. Now G must have a vertex of degree one, or it would contain a cycle, so let $v \in V(G)$ have degree one. Then $G' = G - v$ is still connected, and has no cycle, so it is a tree on t vertices. Therefore $\varepsilon(G') = |G'| - 1 = |G| - 2$. It follows that $\varepsilon(G) = |G| - 1$, so that the result is true when $|G| = t + 1$. By induction, it holds for all values of $|G|$. □

We saw in this proof that a tree G with $\varepsilon > 0$ must have a vertex of degree one. Consider a longest path P in G. The two endpoints of P can only be joined to vertices of P. Because G does not contain any cycles, we can conclude that the endpoints of a longest path have degree one. Therefore a tree has *at least two* vertices of degree one.

In a connected graph, any two vertices are connected by some path. A fundamental property of trees is that any two vertices are connected by a *unique* path. For if there were two uv-paths P and Q, where $P \neq Q$, then traveling from u to v on P we could find the first point of P which is not on Q. Continuing on P until we come to the first point which is again on both P and Q, we could now follow Q back toward u and so find a cycle, which, however, is not possible in a tree.

Every graph G has subgraphs that are trees. The most important of these are the *spanning trees*, that is, trees which span all the vertices of G.

Lemma 5.2. *Every connected graph has a spanning tree.*

Proof. Let G be a connected graph. If G has no cycles, then G is a spanning tree. Otherwise choose a cycle C, and remove any edge $xy \in C$ from G. G is still connected, because any uv-path which uses xy can now be replaced by a path using $C - xy$, so that every u and v are still connected by some path after xy has been removed. We repeat this as many times as necessary until the resulting graph has no cycles. It is a spanning tree of the original G. □

Exercises

5.1.1 Describe in pseudo-code an algorithm to find an edge on a cycle, if one exists.

5.1.2 Make a list of all isomorphism types of trees on 1, 2, 3, 4, 5, and 6 vertices.

5.1.3 Show that there is a tree on n vertices with degree sequence $(d_1, d_2, \ldots, d_n)$ if and only if

$$\sum_{i=1}^{n} d_i = 2(n-1).$$

5.2 Fundamental cycles

Figure 5.2 shows a spanning tree of the Petersen graph. If T is a spanning tree of G, let $G - T$ stand for the graph whose edges are $E(G) - E(T)$. Notice that if any edge $xy \in E(G - T)$ is added to T, then $T + xy$ contains a unique cycle C_{xy}. This is because x and y are connected by a unique path P_{xy} in T. $P_{xy} + xy$ creates a cycle, C_{xy}, called the *fundamental cycle* of xy with respect to T.

Exercises

5.2.1 Show that G has $\varepsilon - |G| + 1$ fundamental cycles with respect to any spanning tree T.

5.2.2 Let T be a spanning tree of G, and let C be a cycle of G containing exactly two edges xy and uv of $G - T$. Prove that $C = C_{xy} \oplus C_{uv}$, where $\oplus$ denotes the operation of *exclusive OR*.

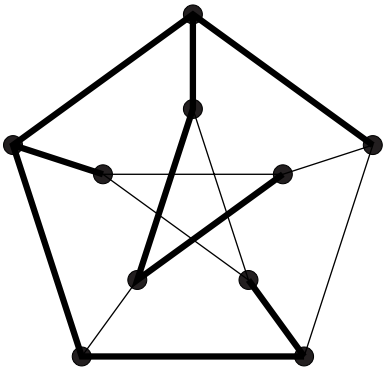


FIGURE 5.2
A spanning tree

Every cycle of G can be formed from the fundamental cycles of G with respect to any spanning tree T.

Theorem 5.3. *Let T be a spanning tree of G. Let C be any cycle containing k edges $u_1v_1, u_2v_2, \ldots, u_kv_k$ of $G-T$, where $k \geq 1$. Then $C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$.*

Proof. The proof is by induction on k. The result is certainly true when $k = 1$. Suppose that the edges u_iv_i occur on C in the order $i = 1, 2, \ldots, k$. These edges divide the remaining edges of C into a number of paths $P_1, P_2, \ldots, P_k$, where P_i connects v_i to u_{i+1}. This is shown in Figure 5.3.

Let $C_i = C_{u_iv_i}$ denote the i^{th} fundamental cycle. Consider C_1. It consists of the edge u_1v_1 and the unique path P of T connecting v_1 to u_1. P and P_1 both begin at vertex v_1. As we travel on P from v_1 toward u_1, we eventually come to the first vertex of P which is not on P_1. This is also the last vertex in common with P_1, because P and P_1 are both contained in T, which has no cycles. P may intersect several of the paths P_i. In each case the intersection must consist of a single segment, that is, a consecutive sequence of vertices of P_i, because T contains no cycles. The last path which P intersects is P_k, because both P and P_k end with u_1. This is illustrated in Figures 5.3 and 5.4.

Consider now $H = C_1 \oplus C$. It is a subgraph of G. It consists of that part of C which is not contained in C_1, plus that part of P which is not contained in C. Thus, the portions common to P and each P_i are discarded, but the new segments of P which are now added create one or more new cycles. Thus, H consists of one or more edge-disjoint cycles constructed from edges of T, plus the edges $u_2v_2, u_3v_3, \ldots, u_kv_k$. Because each of these cycles contains fewer than k edges of $G - T$, we can say that $H = C_{u_2v_2} \oplus C_{u_3v_3} \oplus \cdots \oplus C_{u_kv_k}$. We then have $C_1 \oplus H = (C_1 \oplus C_1) \oplus C = C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$. Therefore the result is true when C contains k edges of $G - T$. By induction the result is true for all values of k. □

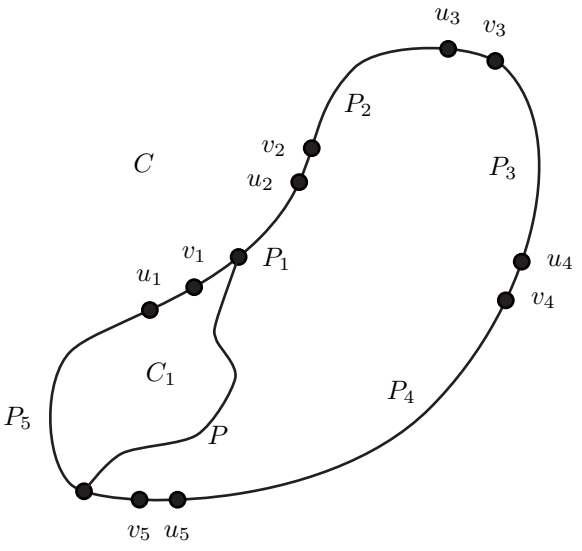


FIGURE 5.3
Decomposition into fundamental cycles

Thus the fundamental cycles of G with respect to any spanning tree T generate all the cycles of G.

5.3 Co-trees and bonds

Let G be a graph. If $S \subset V(G)$, then $\overline{S}$ denotes $V(G) - S$. The *edge-cut* $[S, \overline{S}]$ consists of all edges of G with one endpoint in S and one endpoint in $\overline{S}$. Notice that $G - [S, \overline{S}]$ is a disconnected graph. See Figure 5.5.

If T is a spanning tree of G, then the complementary graph $\widehat{T} = G - T$ is called the *co-tree* corresponding to T. Now a co-tree $\widehat{T}$ cannot contain any edge-cut of G. This is because $G - \widehat{T} = T$, which is connected. If uv is any edge of T, then $T - uv$ consists of two components, S_u, those vertices connected to u, and S_v, those vertices connected to v. $[S_u, S_v]$ is an edge-cut of G. It is contained in $\widehat{T} + uv$. This is illustrated in Figure 5.6.

$[S_u, S_v]$ is a *minimal edge-cut* of G, that is, it does not contain any smaller edge-cuts. For if xy is any edge, where $x \in S_u$ and $y \in S_v$, then $G - [S_u, S_v] + xy$ is connected. Therefore:

1. A co-tree $\widehat{T}$ contains no edge-cut.

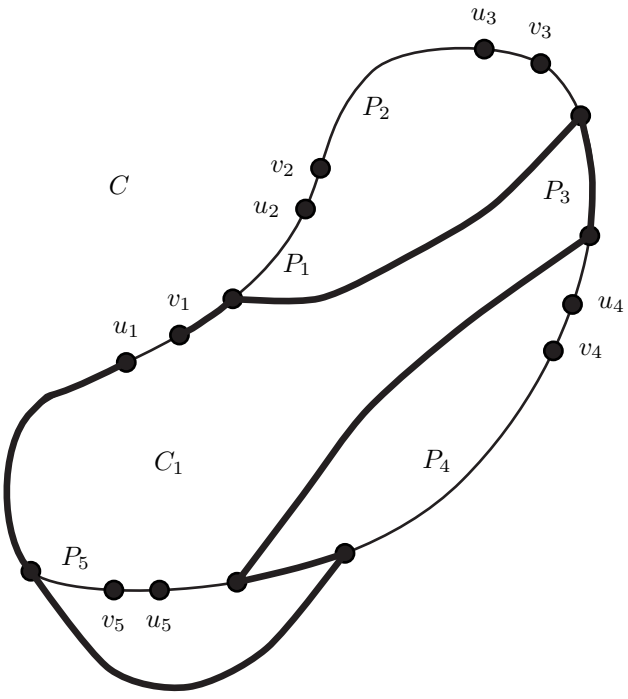


FIGURE 5.4
Decomposition into fundamental cycles

2. If uv is any edge of $G - \widehat{T}$, then $\widehat{T} + uv$ contains a unique minimal edge-cut $B_{uv} = [S_u, S_v]$.

Compare this with trees:

1. A tree T contains no cycle.
2. If uv is any edge of $G - T$, then $T + uv$ contains a unique fundamental cycle C_{uv}.

The unique edge-cut B_{uv} contained in $\widehat{T} + uv$ is called the *fundamental edge-cut* of uv with respect to $\widehat{T}$. Any minimal edge-cut of G is called a *bond*. There is a duality between trees and co-trees, and cycles and bonds (bonds are sometimes called *co-cycles*). There is a linear algebra associated with every graph, in which cycles and bonds generate orthogonal vector spaces, called the *cycle space* and *bond space* of G. Theorem 5.3 shows that the fundamental cycles with respect to any spanning tree form a basis for the cycle space. Similarly, the fundamental edge-cuts form a basis for the bond space. See BONDY and MURTY [23] for more information.

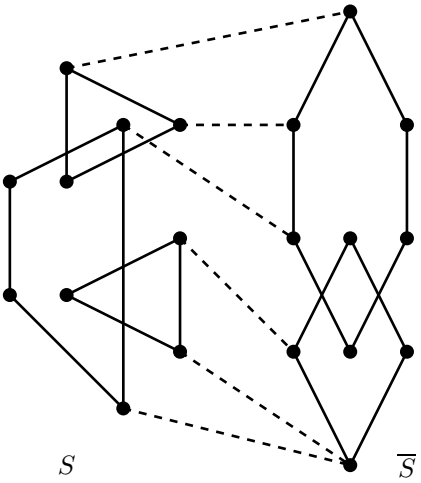


FIGURE 5.5
An edge-cut

Exercises

5.3.1 How may fundamental edge-cuts does G have with respect to a co-tree $\widehat{T}$? What are the dimensions of the cycle space and bond space of G?

5.3.2 Let T be a spanning tree of G, and let edge $uv \notin T$. Let xy be any edge of the fundamental cycle C_{uv}, such that $xy \neq uv$. Then $T + uv - xy$ is also a spanning tree of G. Thus, spanning trees of G are adjacent via fundamental cycles. The *tree graph* of G is $\mathit{Tree}(G)$. Its vertices are the spanning trees of G, and they are adjacent via fundamental cycles. Show that $\mathit{Tree}(G)$ is a connected graph.

5.3.3 Show that trees (co-trees) are also adjacent via fundamental edge-cuts.

5.3.4 Let T be a spanning tree of G, and let W be a closed walk in G such that W uses edges of T and edges $u_1v_1, u_2v_2, \ldots, u_kv_k$ of $G - T$. Describe the subgraph $H = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_kv_k}$. What is its relation to W?

5.3.5 Let $[S, \overline{S}]$ be an edge-cut of G. Prove that $[S, \overline{S}]$ is a bond if and only if $G[S]$ and $G[\overline{S}]$ are connected graphs.

5.3.6 Let $B_1 = [S_1, \overline{S}_1]$ be an edge-cut of G, and let $B_2 = [S_2, \overline{S}_2]$ be a bond contained in $[S_1, \overline{S}_1]$; that is, $B_1 \subset B_2$. (Note: S_2 will generally *not* be a subset of S_1.) Prove that $[S_1, \overline{S}_1] - [S_2, \overline{S}_2]$ is also an edge-cut.

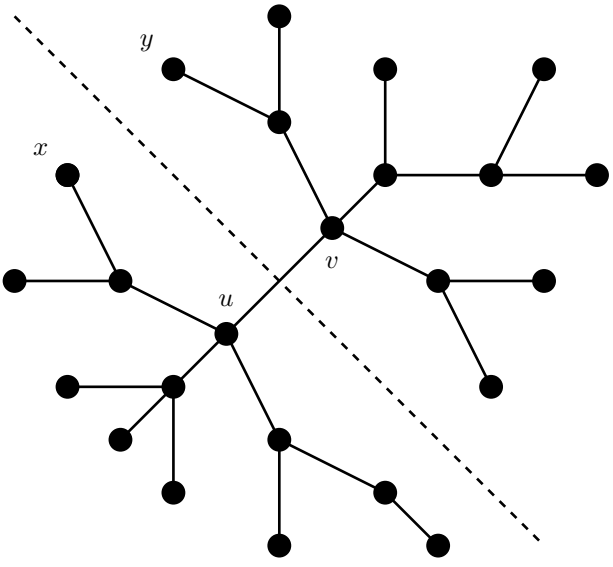


FIGURE 5.6
Co-trees and edge-cuts

5.3.7 Use the previous question to prove that every edge-cut can be decomposed into a disjoint union of bonds.

5.3.8 Find a decomposition of the edge-cut $[S, \overline{S}]$ in the graph shown in Figure 5.7 into bonds. The set S is marked by the shading. Is the decomposition unique? (*Hint*: Redraw the graph so that edges do not cross each other.)

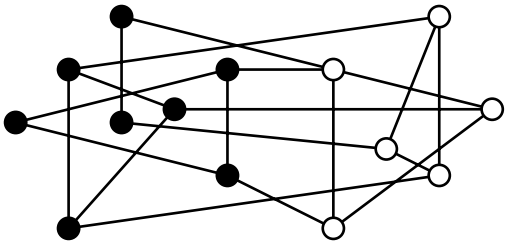


FIGURE 5.7
Find a decomposition into bonds

5.3.9 Let T be a spanning tree of G. Let uv and xy be edges of T, with corresponding bonds $B_{uv} = [S_u, S_v]$ and $B_{xy} = [S_x, S_y]$, where $B_{uv} \cap B_{xy} \neq \emptyset$. Prove that $B_{uv} \oplus B_{xy}$ is a bond.

5.3.10 Prove that any cycle and any bond must intersect in an even number of edges.

5.4 Spanning tree algorithms

One of the easiest ways to construct a spanning tree of a graph G is to use a breadth-first search. The following code is adapted from Algorithm 2.4.1. The statements marked with $(\star)$ have been added.

Algorithm 5.4.1: BFSEARCH(G, u)

comment: build a breadth-first spanning tree of G

for $v \leftarrow 1$ **to** $|G|$ **do** $\mathit{OnScanQ}[v] \leftarrow$ **false**
$\mathit{ScanQ}[1] \leftarrow u$
$\mathit{OnScanQ}[u] \leftarrow$ **true**
$\mathit{QSize} \leftarrow 1$
$k \leftarrow 1$
$\mathit{Parent}[u] \leftarrow 0$ $(\star)$
$\mathit{BFNum}[u] \leftarrow 1$ $(\star)$
$\mathit{Count} \leftarrow 1$ $(\star)$
$\mathit{Tree} \leftarrow$ empty list $(\star)$
repeat
  $v \leftarrow \mathit{ScanQ}[k]$
  for each $w \longrightarrow v$
    do if not $\mathit{OnScanQ}[w]$
    then
      $\mathit{QSize} \leftarrow \mathit{QSize} + 1$
      $\mathit{ScanQ}[\mathit{QSize}] \leftarrow w$
      $\mathit{OnScanQ}[w] \leftarrow$ **true**
      $\mathit{Parent}[w] \leftarrow v$ $(\star)$
      $\mathit{Count} \leftarrow \mathit{Count} + 1$ $(\star)$
      $\mathit{BFNum}[w] \leftarrow \mathit{Count}$ $(\star)$
      add edge vw to Tree $(\star)$
  $k \leftarrow k + 1$
until $k > \mathit{QSize}$

A number of arrays *ScanQ*, *OnScanQ*, *Parent*, and *BFNum* are used in this algorithm, as well as the counters *QSize* and *Count*, and the list of edges *Tree* of the spanning tree constructed.

BFSEARCH(G, u) visits each vertex of the connected graph G, beginning with u. The order in which the vertices are visited defines a numbering of the vertices, called the *breadth-first numbering*. It is saved in $\mathit{BFNum}[\cdot]$. This is illustrated in Figure 5.8.

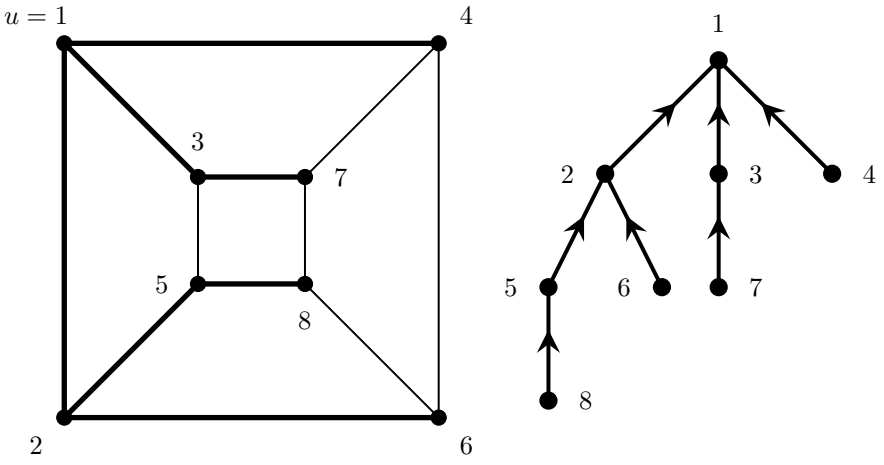


FIGURE 5.8
A breadth-first tree

The search begins at node u, called the *root* of the spanning tree. In the example provided in Figure 5.8, $u = 1$. As each node w is placed on the *ScanQ*, its *parent* in the search tree is saved in $\mathit{Parent}[w]$. This is represented by the arrows on the tree in the diagram. Thus, beginning at any node in the graph, we can follow the $\mathit{Parent}[\cdot]$ values up to the root of the tree. The breadth-first numbering defines a traversal of the tree, which goes level by level, and from left to right in the drawing. A great many graph algorithms are built around the breadth-first search. The important property of breadth-first spanning trees is that the paths it constructs connecting any vertex w to the root of the tree are *shortest* paths.

In a weighted graph, different spanning trees will have different weights, where

$$\textsc{Wt}(T) = \sum_{uv \in T} \textsc{Wt}(uv).$$

We now want to find a spanning tree T of *minimum* weight. This is called the *minimum spanning tree problem*. There are many algorithms which solve it. We present some of them here.

5.4.1 Prim's algorithm

The idea here is to pick any $u \in V(G)$ and "grow" a tree on it; that is, at each iteration, we add one more edge to the current tree, until it spans all of $V(G)$. We must do this in such a way that the resulting tree is minimum.

Algorithm 5.4.2: PRIM(G)

comment: { *Tree* is a list of edges in a minimum spanning tree.
VT are the vertices in the current tree being grown.

initialize *Tree* to contain no edges
$t \leftarrow 0$ "the number of edges in *Tree*"
choose any $u \in V(G)$
initialize VT to contain u
comment: the *Tree* now has 1 node and 0 edges

while $t < |G| - 1$
do {
choose an edge xy of *minimum* weight, with $x \in VT$ and $y \notin VT$
add xy to *Tree*
add y to VT
$t \leftarrow t + 1$

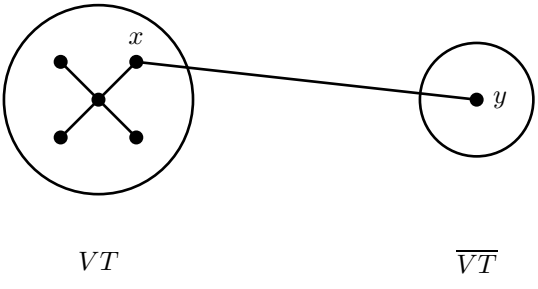


FIGURE 5.9
Growing a tree with Prim's algorithm

We first prove that Prim's algorithm does in fact produce a minimum spanning tree. Initially, VT contains one vertex, and *Tree* contains no edges. On each iteration an edge xy with $x \in VT$ and $y \notin VT$ is added to *Tree*, and y is added to VT. Therefore, the edges of *Tree* always form a tree which spans VT. After $n - 1$ iterations, it is a spanning tree of G. Call the tree produced by Prim's algorithm T, and suppose that it consists of edges $e_1, e_2, \ldots, e_{n-1}$, chosen in that order. If it is not a minimum spanning tree, then choose a minimum tree T^* which agrees with T on the first k iterations, but not on iteration $k + 1$, where k is as large as possible. Then $e_1, e_2, \ldots, e_k \in T^*$, but $e_{k+1} \notin T^*$. Consider iteration $k + 1$, and let $e_{k+1} = xy$, where $x \in VT$ and $y \in \overline{VT}$. Then $T^* + xy$ contains a fundamental cycle C_{xy}.

C_{xy} must contain another edge uv with $u \in VT$ and $v \in \overline{VT}$. Because Prim's algorithm chooses edges by weight, we know that $\text{WT}(xy) \leq \text{WT}(uv)$. Now $T' = T^* + xy - uv$ is also a spanning tree, and $\text{WT}(T') \geq \text{WT}(T^*)$, because T^* is a minimum tree. But $\text{WT}(T') = \text{WT}(T^*) + \text{WT}(xy) - \text{WT}(uv) \leq \text{WT}(T^*)$. Therefore, $\text{WT}(T') = \text{WT}(T^*)$ and $\text{WT}(xy) = \text{WT}(uv)$. It follows that T' is also a minimum

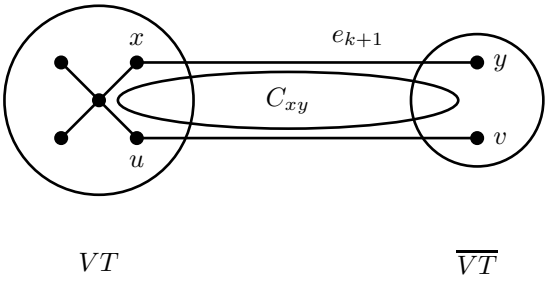


FIGURE 5.10
A fundamental cycle in Prim's algorithm

tree, and that T' contains $e_1, e_2, \ldots, e_{k+1}$; that is, it agrees with T on $k+1$ iterations, a contradiction. Consequently, Prim's tree T is also a minimum spanning tree.

5.4.1.1 Data structures

The main operation performed in Prim's algorithm is to select the edge xy, of minimum weight, with $x \in VT$ and $y \in \overline{VT}$. One way to do this is to store two values for each vertex $y \in \overline{VT}$:

MinWt$[y]$: the minimum weight $\text{WT}(xy)$, over all $x \longrightarrow y$, where $x \in VT$
MinPt$[y]$: that vertex $x \in VT$ with $\text{WT}(xy) =$ *MinWt*$[y]$.

Then to select the minimum edge xy, we need only perform the following steps:

select $y \in \overline{VT}$ with smallest *MinWt*[y] value
$x \leftarrow$ *MinPt*$[y]$
for each $w \longrightarrow y$ **do** { **if** $w \in \overline{VT}$ **then** update *MinWt*$[w]$

Let $n = |G|$. If we scan the set $\overline{VT}$ in order to select the minimum vertex y on each iteration, then the first iteration requires scanning $n-1$ vertices, the second iteration requires $n-2$ steps, etc., requiring $1+2+\cdots+(n-1) = \binom{n}{2}$ in total. The total number of steps needed to update the *MinWt*$[\cdot]$ values is at most $\sum_y \text{DEG}(y) = 2\varepsilon$ steps, over all iterations. Thus, the complexity when Prim's algorithm is programmed like this is

$$O(2\varepsilon + \frac{n^2}{2}) = O(\varepsilon + n^2).$$

In order to remove the $O(n^2)$ term, we could store the vertices $\overline{VT}$ in a heap H. Selecting the minimum now requires approximately $\log n$ steps. For each $w \longrightarrow y$ we may also have to update H, requiring at most an additional $\text{DEG}(y) \log n$ steps per iteration. The total number of steps performed over all iterations is now at most

$$\sum_{k=1}^{n-1} \log n + \sum_{y} \text{DEG}(y) \log n \leq n \log n + 2\varepsilon \log n.$$

The complexity of Prim's algorithm using a heap is therefore $O(n \log n + \varepsilon \log n)$. If ε is small, this will be better than the previous method. But if ε is large, this can be worse, depending on how much time is actually spent updating the heap. Thus we can say that Prim's algorithm has complexity:

- $O(\varepsilon + n^2)$, if the minimum is found by scanning.
- $O(n \log n + \varepsilon \log n)$, if a heap is used.

Exercises

5.4.1 Work Prim's algorithm by hand on the graph in Figure 5.11, starting at the shaded vertex.

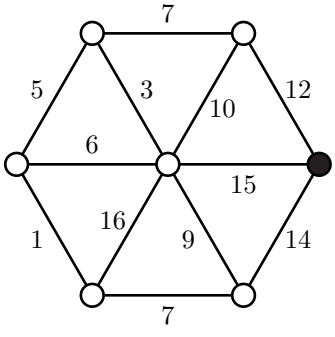


FIGURE 5.11
Find a spanning tree

5.4.2 Consider Dijkstra's shortest-path algorithm, which finds a shortest uv-path for all $v \in V(G)$. For each v, let P_v be the shortest path found. Show that the collection of paths, $\bigcup_v P_v$, defines a spanning tree of G. Is it a minimum spanning tree? (*Hint*: Use induction.)

5.4.3 Program Prim's algorithm, storing the vertices $\overline{VT}$ in a heap.

5.4.4 Modify the breadth-first search algorithm to find the fundamental cycles of G with respect to a BF-tree. Print out the edges on each fundamental cycle. What is the complexity of the algorithm?

5.4.5 Let G be a weighted graph in which all edge-weights are distinct. Prove that G has a unique minimum spanning tree.

5.4.2 Kruskal's algorithm

A *forest* is a graph which need not be connected, but whose every component is a tree. Prim's algorithm constructs a spanning tree by growing a tree from some initial vertex. Kruskal's algorithm is quite similar, but it begins with a spanning forest and adds edges until it becomes connected. Initially the forest has $n = |G|$ components and no edges. Each component is a single vertex. On each iteration, an edge which connects two distinct components is added, and the two components are merged. When the algorithm terminates the forest has become a tree.

Algorithm 5.4.3: KRUSKAL(G)

comment: { *Tree* is a list of edges in a minimum spanning tree.
{ T_u is the component of the forest which contains u.

initialize *Tree* to contain no edges
for each $u \in V(G)$ **do** initialize T_u to contain only u
$t \leftarrow 0$ "the number of edges in *Tree*"
comment: the forest currently has $|G|$ nodes and 0 edges

while $t < |G| - 1$
do { Select the next edge xy of *minimum* weight, and determine
{ which components x and y are in, say $x \in T_u$ and $y \in T_v$
{ **if** $T_u \neq T_v$
then { merge T_u and T_v
{ add xy to *Tree*
{ $t \leftarrow t + 1$

Initially the forest has n components, and each one is a tree with no edges. Each edge that is added connects two distinct components, so that a cycle is never created. Whenever an edge is added the two components are merged, so that the number of components decreases by one. After $n-1$ iterations, there is only one component left, which must be a tree T. If T is not a minimum tree, then we can proceed as we did in Prim's algorithm. Let T consist of edges $e_1, e_2, \ldots, e_{n-1}$, chosen in that order. Select a minimum tree T^* which contains $e_1, e_2, \ldots, e_k$, but not e_{k+1}, where k is as large as possible. Consider the iteration in which $e_{k+1} = xy$ was selected. $T^* + xy$ contains a fundamental cycle C_{xy}, which must contain another edge ab incident on T_x. Because Kruskal's algorithm chooses edges in order of their weight, $\text{WT}(xy) \leq \text{WT}(ab)$. Then $T' = T^* + xy - ab$ is a spanning tree for which $\text{WT}(T') \leq \text{WT}(T^*)$. But T^* is a minimum tree, so that $\text{WT}(T') = \text{WT}(T^*)$, and T' is also a minimum tree. T' contains edges $e_1, e_2, \ldots, e_{k+1}$, a contradiction. Therefore, Kruskal's tree T is a minimum spanning tree.

5.4.2.1 Data structures and complexity

The main operations in Kruskal's algorithm are:

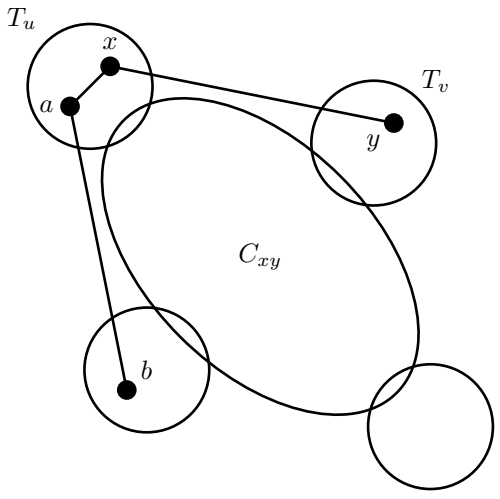


FIGURE 5.12
Growing a forest with Kruskal's algorithm

1. Choose the next edge xy of minimum weight.
2. Determine that $x \in T_u$ and $y \in T_v$.
3. Merge T_u and T_v.

The edges could either be completely sorted by weight, which can be done in $O(\varepsilon \log \varepsilon)$ steps, or they could be kept in a heap, which makes it easy to find the minimum edge. Because we may have a spanning tree T before all the edges have been considered, it is usually better to use a heap. The components T_u can easily be stored using the merge-find data structure described in Chapter 2.

Each time an edge xy is selected from the heap, it requires approximately $\log \varepsilon$ steps to update the heap. In the worse case we may need to consider every edge of G, giving a bound of $\varepsilon \log \varepsilon$ steps. Similarly, $O(\varepsilon \alpha(n))$ steps are needed to build the components, where $n = |G|$. Thus, Kruskal's algorithm can be programmed with a complexity of $O(\varepsilon \log n + \varepsilon \alpha(n))$, where we have used $\log \varepsilon < 2 \log n$. Notice that this can be slightly better than Prim's algorithm. This is because the term $\alpha(n)$ is essentially a constant, and because the heap does not need to be constantly updated as the *MinWt*$[\cdot]$ value changes.

5.4.3 The Cheriton-Tarjan algorithm

The Cheriton-Tarjan algorithm is a modification of Kruskal's algorithm designed to reduce the $O(\varepsilon \log \varepsilon)$ term. It also grows a spanning forest, beginning with a forest of $n = |G|$ components each consisting of a single node. Now the term $O(\varepsilon \log \varepsilon)$ comes from selecting the minimum edge from a heap of ε edges. Because every

component T_u must eventually be connected to another component, this algorithm keeps a separate heap PQ_u for each component T_u, so that initially n smaller heaps are used. Initially, PQ_u will contain only $\text{DEG}(u)$ edges, because T_u consists only of vertex u. When T_u and T_v are merged, PQ_u and PQ_v must also be merged. This requires a modification of the data structures, because *heaps cannot be merged efficiently*. This is essentially because merging heaps reduces to building a new heap. Any data structure in which a minimum element can be found efficiently is called a *priority queue*. A heap is one form of priority queue, in which elements are stored as an array, but viewed as a binary tree. There are many other forms of priority queue. In this algorithm, PQ_u will stand for a priority queue which can be merged. The Cheriton-Tarjan algorithm can be described as follows.

It stores a list *Tree* of the edges of a minimum spanning tree. The components of the spanning forest are represented as T_u and the priority queue of edges incident on vertices of T_u is stored as PQ_u.

Algorithm 5.4.4: CHERITONTARJAN(G)

initialize *Tree* to contain no edges
for each $u \in V(G)$
do {
  initialize T_u to contain only u
  create PQ_u
  for each $v \longrightarrow u$
    do add uv to PQ_u
  comment: each edge will appear in two priority queue
}
comment: the forest currently has $|G|$ nodes and 0 edges
$t \leftarrow 0$
while $t < |G| - 1$
do {
  select a component T_u
  repeat
    select the minimum edge $xy \in PQ_u$ and determine
    which components x and y are in, say $x \in T_u$ and $y \in T_v$
  until $T_u \neq T_v$
  comment: xy connects two different components
  merge T_u and T_v
  merge PQ_u and PQ_v
  add xy to *Tree*
  $t \leftarrow t + 1$
}

Exercises

5.4.1 Prove that the Cheriton-Tarjan algorithm constructs a minimum spanning tree.

5.4.2 Show that a heap is best stored as an array. What goes wrong when the attempt is made to store a heap with pointers?

5.4.3 Show that heaps cannot be merged efficiently. Describe an algorithm to merge two heaps, both with n nodes, and work out its complexity.

5.4.4 Program Kruskal's algorithm, using a heap to store the edges, and the merge-find data structure to store the components.

5.4.4 Leftist binary trees

A *leftist binary tree* (LB-tree) is a modification of a heap which allows efficient merging. A node x in an LB-tree has the following four fields:

1. *Value*$\langle x \rangle$: the value stored.
2. *Left*$\langle x \rangle$: a pointer to the left subtree.
3. *Right*$\langle x \rangle$: a pointer to the right subtree.
4. *rPath*$\langle x \rangle$: the right-path distance.

An LB-tree satisfies the heap property; namely the entry stored in any node has value less than or equal to that of its two children:

$$\mathit{Value}\langle x \rangle \quad \leq \quad \mathit{Value}\langle \mathit{Left}\langle x \rangle \rangle$$

and

$$\mathit{Value}\langle x \rangle \quad \leq \quad \mathit{Value}\langle \mathit{Right}\langle x \rangle \rangle$$

Therefore the smallest entry in the tree occurs in the top node. Thus, a heap is a special case of an LB-tree. The distinctive feature of LB-trees is contained in field *rPath*$[x]$. If we begin at any node in an LB-tree and follow *Left* and *Right* pointers in any sequence, we eventually reach a nil pointer. In an LB-tree, the shortest such path is *always the rightmost path*. This is true for every node in the tree. The length of the path for a node x is the *rPath*$\langle x \rangle$ value. In the tree shown in Figure 5.13, the *rPath* values are shown beside each node.

In summary, an LB-tree is a binary tree which satisfies the heap property, and whose shortest path to a nil pointer from any node is always the rightmost path. This means that LB-trees will tend to have more nodes on the left than the right; hence, the name leftist binary trees.

The rightmost path property makes it possible to merge LB-trees efficiently. Consider the two trees A and B in Figure 5.14 which are to be merged into a tree T. The top node of T is evidently taken from A, because it has the smaller minimum. This breaks A into two subtrees, L and R. The three trees B, L, and R are now to be made into two subtrees of T. The easiest way to do this is first to merge R and B into a single tree P, and then take P and L as the new right and left subtrees of T, placing the one with the smaller *rPath* value on the right. The recursive merge procedure is described in Algorithm 5.4.5, and the result of merging A and B of Figure 5.14 is shown in Figure 5.15.

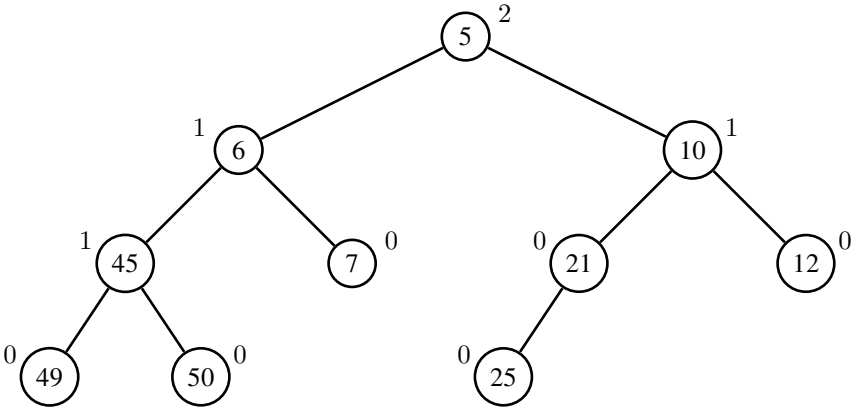


FIGURE 5.13
A leftist binary tree

Algorithm 5.4.5: LBMERGE(A, B)

comment: Merge non-null LB-trees A and B

if *Value*$\langle A\rangle > $ *Value*$\langle B\rangle$ **then** swap A and B
if *Right*$\langle A\rangle =$ **null** **then** $P \leftarrow B$
else $P \leftarrow$ LBMERGE(*Right*$\langle A\rangle, B$)
comment: choose the tree with the smaller *rPath* as right subtree

if *Left*$\langle A\rangle =$ **null**
then { *Right*$\langle A\rangle \leftarrow$ **null**; *Left*$\langle A\rangle \leftarrow P$; *rPath*$\langle A\rangle \leftarrow 0$ }
else { **if** *rPath*$\langle P\rangle \leq$ *rPath*$\langle$*Left*$\langle A\rangle\rangle$
then *Right*$\langle A\rangle \leftarrow P$
else { *Right*$\langle A\rangle \leftarrow$ *Left*$\langle A\rangle$; *Left*$\langle A\rangle \leftarrow P$ }
rPath$\langle A\rangle \leftarrow$ *rPath*$\langle$*Right*$\langle A\rangle\rangle + 1$ }
return (A)

Notice that when the top node of A is removed, thereby splitting A into two subtrees L and R, the left subtree L subsequently becomes one of the subtrees of T. That is, L is not decomposed in any way, it is simply transferred to T. Furthermore, L is usually the larger of the two subtrees of A. Let us estimate the number of steps necessary to merge A and B, with *rPath* values r_1 and r_2, respectively. One step is needed to choose the smaller node A, say, as the new top node. The right subtree R will have rightmost path length of $r_1 - 1$. When R and B are merged, one of them

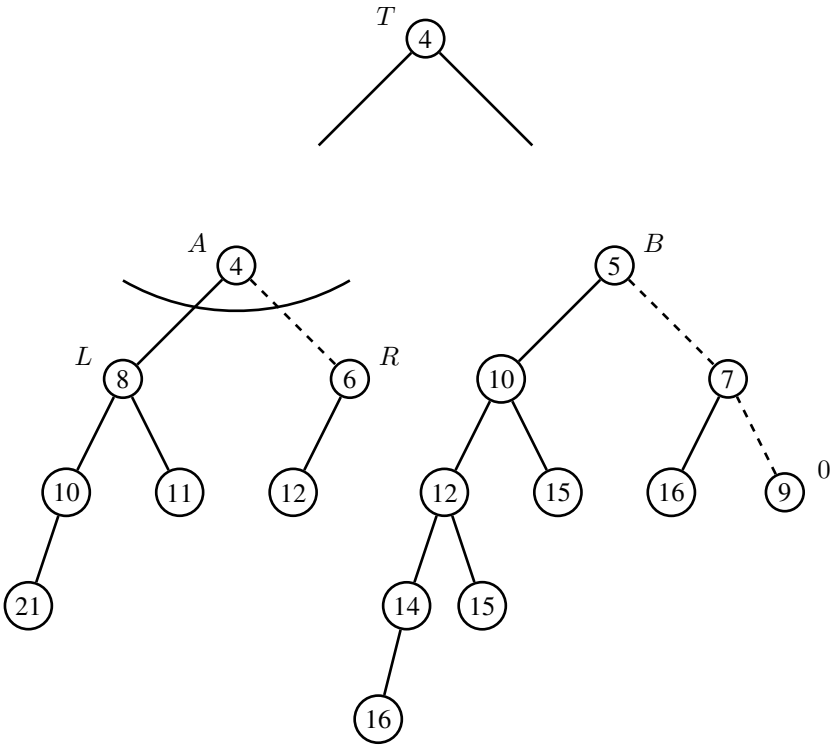


FIGURE 5.14
Merging two leftist binary trees

will be similarly decomposed into a left and right subtree. The left subtree is never broken down. At each step in the recursion, the smaller value is chosen as the new top node, and its *Right* becomes the next subtree to be considered; that is, LBMERGE() follows the rightmost paths of A and B, always choosing the smaller entry of the two paths. Thus, the rightmost paths of A and B are merged into a single path (see Figure 5.15). Therefore, the depth of the recursion is at most $r_1 + r_2$. At the bottom of the recursion the *rPath* values may both equal zero. It then takes about five steps to merge the two trees. Returning up through the recursion, LBMERGE() compares the *rPath* values of L and P, and makes the smaller one into the new right subtree. Also, the new *rPath* value is assigned. All this takes about four steps per level of recursion, so that the total number of steps is at most $5(r_1 + r_2 + 1)$.

What is the relation between the rightmost path length of an LB-tree and the number of nodes it contains? If the *rPath* value of an LB-tree T is r, then beginning at the top node, every path to a nil pointer has length at least r. Therefore, T contains at least a full binary tree of r levels; that is, T has at least $2^{(r+1)} - 1$ nodes. The

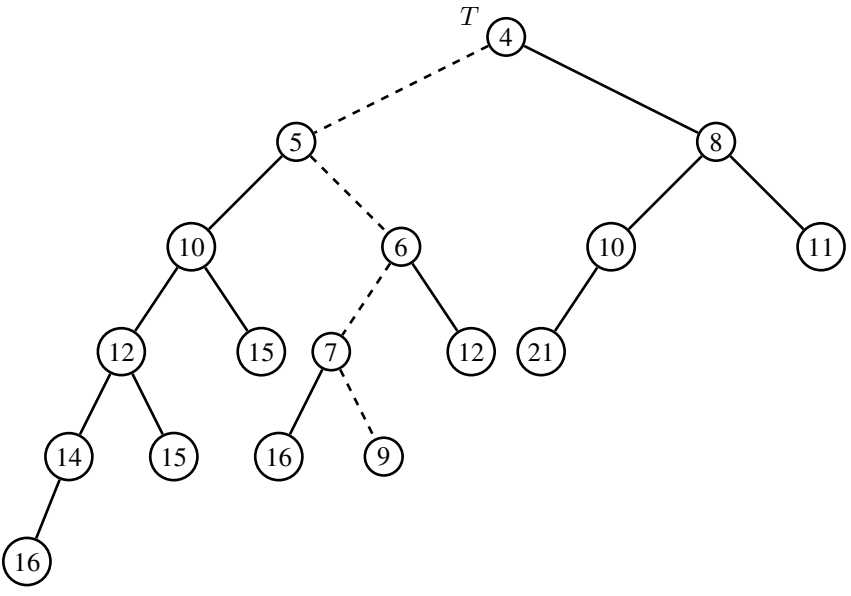


FIGURE 5.15
The merged LB-tree

largest rightmost path length possible if T is to store n nodes is the smallest value of r such that $2^{(r+1)} - 1 \geq n$, or $r \leq \lceil \log \frac{n+1}{2} \rceil$.

If A and B both contain at most n nodes, then $r_1, r_2 \leq \lceil \log \frac{n+1}{2} \rceil$, and LBMERGE(A, B) takes at most $5(r_1 + r_2 + 1) \leq 10 \cdot \lceil \log \frac{n+1}{2} \rceil + 5 = O(\log n)$ steps. Thus LB-trees can be merged quite efficiently.

We can use this same method to extract the minimum entry from an LB-tree A, using at most $O(\log n)$ steps:

select minimum as *Value*$\langle A \rangle$
$A \leftarrow$ LBMERGE(*Left*$\langle A \rangle$, *Right*$\langle A \rangle$)

Consider now how to construct an LB-tree. In Chapter 2 we found that there are two ways of building a heap, one much more efficient than the other. A similar situation holds for LB-trees. The most obvious way to build one is to merge successively each new node into an existing tree:

initialize A, a new LB-tree with one node
repeat
  get next value
  create and initialize a new LB-tree, B
  $A \leftarrow$ LBMERGE(A, B)
until all values have been inserted

However, this can easily create LB-trees that are really linear linked lists, as shown in Figure 5.16. This algorithm then becomes an insertion sort, taking $O(n^2)$ steps, where n is the number of nodes inserted.

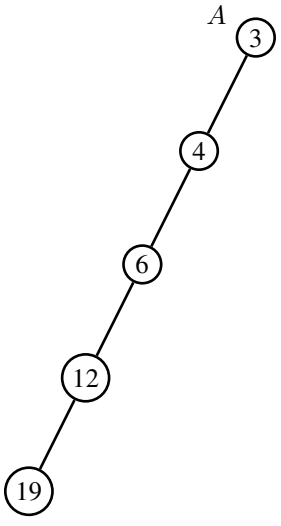


FIGURE 5.16
An LB-tree

A better method is to create n LB-trees, each containing only one node, and then merge them two at a time, until only one tree remains. The trees are kept on a queue, called the *MergeQ*.

Algorithm 5.4.6: BUILDLBTREE(*MergeQ*)

repeat
select A and B, the first two trees of *MergeQ*
$A \leftarrow$ LBMERGE(A, B)
put A at end of *MergeQ*
until *MergeQ* contains only one tree
return (A)

How many steps are needed to build an LB-tree in this way, if we begin with n trees of one node each? There will be $\lfloor n/2 \rfloor$ merges of pairs 1-node trees, each of which takes at most 5 steps. This will leave $\lceil n/2 \rceil$ trees on the Merge$Q$, each with at most two nodes. These will be taken two at a time, giving $\lfloor n/4 \rfloor$ merges of up to 2-node trees. Similarly there will be $\lfloor n/8 \rfloor$ merges of up to 4-node trees, etc. This is

summarized in the following table:

tree size	# pairs	max *rPath*	max $r_1 + r_2 + 1$
1	$\lfloor n/2 \rfloor$	0	1
2	$\lfloor n/4 \rfloor$	0	1
4	$\lfloor n/8 \rfloor$	1	3
8	$\lfloor n/16 \rfloor$	2	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^k	$\lfloor n/2^{k+1} \rfloor$	$k-1$	$2k-1$

The last step will merge two trees with roughly $n/2$ nodes each. The maximum *rPath* value for these trees will be $\leq \lceil \log \frac{n/2+1}{2} \rceil$, or approximately $\lceil \log n \rceil - 1$. The total number of steps taken to build the LB-tree is then at most

$$5\lfloor \frac{n}{2} \rfloor + \sum_{k=1}^{\lfloor \log n \rfloor - 1} 5(2k-1)\lfloor \frac{n}{2^{k+1}} \rfloor \leq \frac{5n}{2} + 5n \sum_{k=1}^{\lfloor \log n \rfloor - 1} \frac{2k-1}{2^{k+1}}.$$

We can sum this using the same technique as in the heap analysis of Chapter 2, giving a sum of $10n - \frac{5nr}{2^r}$, where $r = \lfloor \log n \rfloor - 1$. Thus, an LB-tree can be built in $O(n)$ steps.

We can now fill in the details of the Cheriton-Tarjan spanning tree algorithm. There are three different kinds of trees involved in the algorithm:

1. A minimum spanning tree is being constructed.
2. The components T_u are merge-find trees.
3. The priority queues PQ_u are LB-trees.

At the beginning of each iteration, a component T_u is selected, and the minimum edge $xy \in PQ_u$ is chosen. How is T_u selected? There are several possible strategies. If we choose the same T_u on each iteration, then the algorithm grows a tree from u; that is, it reduces to Prim's algorithm. If we choose the component T_u incident on the minimum remaining edge, then the algorithm reduces to Kruskal's algorithm. We could choose the smallest component T_u, but this would add an extra level of complication, because we would now have to keep a heap of components in order to find the smallest component quickly. The method which Cheriton and Tarjan recommend is *uniform selection*; that is, we keep a queue, *TreeQ*, of components. Each entry on the *TreeQ* contains T_u and PQ_u. On each iteration, the component T_u at the head of the queue is selected and the minimum $xy \in PQ_u$ is chosen, say $x \in T_u$ and $y \in T_v$, where $T_u \neq T_v$. Once T_u and T_v, PQ_u and PQ_v have been merged, they are moved to the end of the *TreeQ*. Thus, the smaller components will tend to be at the head of the queue. So the algorithm uses two queues, the *MergeQ* for constructing LB-trees and merging them, and the *TreeQ* for selecting components.

The complexity analysis of the Cheriton-Tarjan algorithm is beyond the scope of this book. If analyzed very carefully, it can be shown to be $O(\varepsilon \log\log \varepsilon)$, if programmed in a very special way.

Minimum spanning tree algorithms are a good illustration of the process of algorithm development. We begin with a simple algorithm, like growing a spanning tree from an initial vertex, and find a complexity of $O(n^2)$. We then look for a data structure or programming technique that will allow the n^2 term to be reduced, and obtain a new algorithm, with complexity $O(\varepsilon \log n)$, say. We then ask how the ε or $\log n$ term can be reduced, and with much more effort and more sophisticated data structures, obtain something like $O(\sqrt{\varepsilon} \log n)$ or $O(\varepsilon \log \log n)$. Invariably, the more sophisticated algorithms have a higher constant of proportionality, so that improvements in running time are only possible when n and ε become very large. However, the sophisticated algorithms also indicate that there are theoretical limits of efficiency for the problem at hand.

Exercises

5.4.1 Prove that the result of LBMERGE(A,B) is always an LB-tree, where A and B are non-nil LB-trees.

5.4.2 Let A be an LB-tree with n nodes and let B be an arbitrary node in the tree. Show how to update A if:

(a) *Value*$\langle B \rangle$ is increased.

(b) *Value*$\langle B \rangle$ is decreased.

(c) Node B is removed.

5.4.3 **Delayed Merge.** When (T_u, PQ_u) is selected from the *TreeQ*, and merged with (T_v, PQ_v), the result is moved to the end of the queue. It may never come to the head of the *TreeQ* again. In that case, it would not really be necessary to perform the LBMERGE(PQ_u, PQ_v). Cheriton and Tarjan delay the merging of the two by creating a new dummy node D and making PQ_u and PQ_v into its right and left subtrees. D can be marked as a dummy by setting *rPath*$\langle D \rangle$ to -1. Several dummy nodes may accumulate at the top of the trees PQ_u. Should a tree with a dummy node come to the head of the queue, its dummy nodes must be removed before the minimum edge $xy \in PQ_u$ can be selected. Write a recursive tree traversal which removes the dummy nodes from an LB-tree, and places its non-dummy subtrees on the *MergeQ*. We can then use BUILDLBTREE() to combine all the subtrees on the *MergeQ* into one.

5.4.4 Program the Cheriton-Tarjan algorithm, using leftist binary trees with delayed merge, to store the priority queues.

5.5 Notes

An excellent description of the cycle space and bond space can be found in BONDY and MURTY [23]. Kruskal's and Prim's algorithms are standard algorithms for mini-

mum spanning trees. They are described in most books on algorithms and data structures. The Cheriton-Tarjan algorithm is from CHERITON and TARJAN [29]. Leftist binary trees are from KNUTH [103], and are also described in WEISS [188].

6

The Structure of Trees

6.1 Introduction

The structure of trees is naturally recursive. When trees are used as data structures, they are typically processed by recursive procedures. Similarly, exhaustive search programs working by recursion also construct trees as they follow their search paths. These trees are always *rooted trees*; that is, they begin at a distinguished node, called the *root vertex*, and are usually built outwards from the root. Figure 6.1 shows several rooted trees, where the root vertex is shaded black.

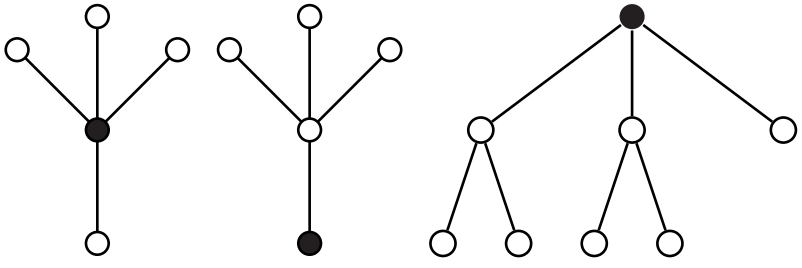


FIGURE 6.1
Several rooted trees

If T is a tree, then any vertex v can be chosen as the root, thereby making T into a rooted tree. A rooted tree can always be decomposed into *branches*. The tree T shown in Figure 6.2 has three branches B_1, B_2, and B_3.

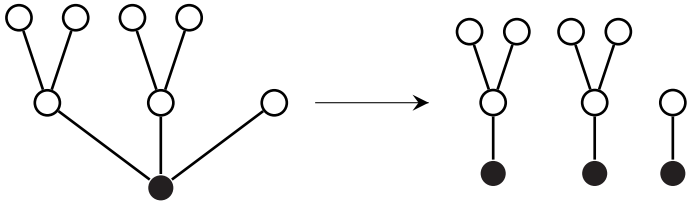


FIGURE 6.2
Decomposition into branches

DEFINITION 6.1: Let T have root vertex v. The *branches* of T are the maximal subtrees in which v has degree one.

Thus, the root is in every branch, but the branches have no other vertices in common. The number of branches equals the degree of the root vertex. If we know the branches of some tree T, then we can easily recombine them to get T. Therefore, two rooted trees have the same structure; that is, they are isomorphic, if and only if they have the same number of branches, and their branches have the same structure.

Any vertex of a tree which has degree one is called a *leaf*. If the root is a leaf, then T is itself a branch. In this case, let u be the unique vertex adjacent to v, the root of T. Then $T' = T - v$ is a rooted tree, with root u. This is illustrated in Figure 6.3. T' can then be further broken down into branches, which can in turn be reduced to rooted trees, etc. This gives a recursive decomposition of rooted trees into branches, and branches into rooted trees.

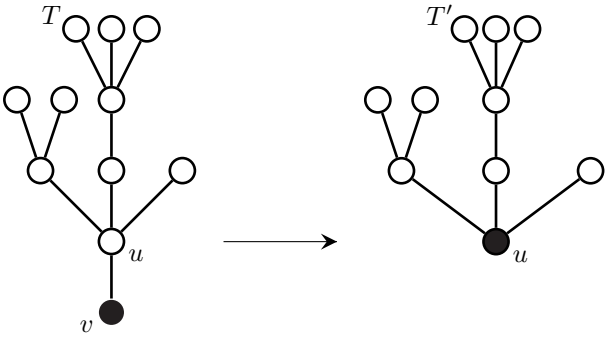


FIGURE 6.3
Reducing a branch to a rooted tree

This technique can be developed into a method for determining when two rooted trees have the same structure.

6.2 Non-rooted trees

All non-rooted trees on five and fewer vertices are displayed in Figure 6.4. Table 6.1 gives the number of trees up to 10 vertices.

If a leaf is removed from a tree on n vertices, a tree on $n-1$ vertices is obtained. Thus, one way to list all the trees on n vertices is to begin with a list of those on $n-1$ vertices, and add a leaf in all possible ways, discarding duplicates. How can we recognize when two trees have the same structure? We shall see that non-rooted trees can always be considered as rooted trees, by choosing a special vertex as root, in the *center* of T, denoted $\text{CTR}(T)$. The center is defined recursively.

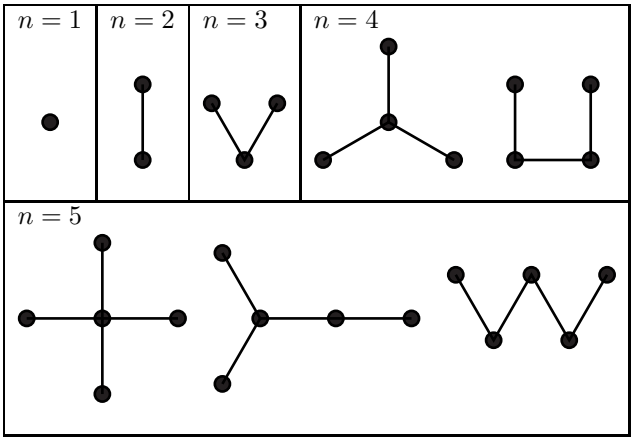


FIGURE 6.4
The trees on five and fewer vertices

DEFINITION 6.2: Let T be a tree on n vertices.

1. If $n = 1$, say $V(T) = \{u\}$, then $\text{CTR}(T) = u$.
2. If $n = 2$, say $V(T) = \{u, v\}$, then $\text{CTR}(T) = uv$.
3. If $n > 2$, then T has at least two leaves. Delete all the leaves of T to get a tree T'. Then $\text{CTR}(T) = \text{CTR}(T')$.

Thus the center of a tree is either a vertex or an edge, because eventually case (1) or (2) of the definition is used in determining the center of T. Trees whose center consists of a single vertex are called *central trees*. Trees with two vertices in the center (i.e., $\text{CTR}(T)$ is an edge) are called *bicentral trees*. Figure 6.5 shows two trees, one central and one bicentral.

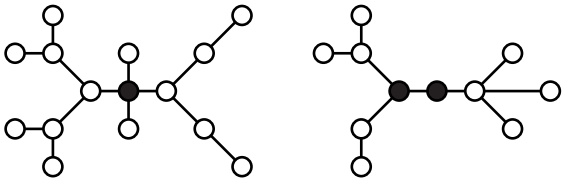


FIGURE 6.5
A central tree and a bicentral tree

A central tree can always be considered a rooted tree, by taking the center as the root. A bicentral tree can also be considered a rooted tree, but we must have a means of deciding which of two vertices to take as the root. Thus we can say that every tree is a rooted tree.

TABLE 6.1
The number of trees up to 10 vertices

n	# trees
2	1
3	1
4	2
5	3
6	6
7	11
8	23
9	47
10	106

Exercises

6.2.1 Find the center of the trees shown in Figures 6.1 and 6.3.

6.2.2 Prove that any longest path in a tree T contains the center.

6.2.3 Prove that T is central if $\text{DIAM}(T)$ is even, and bicentral if $\text{DIAM}(T)$ is odd.

6.2.4 A *binary tree* is a rooted tree such that the root vertex has degree two, and all other vertices which are not leaves have degree three. Show that if T is a binary tree on n vertices, that n is odd, and that T has $(n+1)/2$ leaves.

6.3 Read's tree encoding algorithm

There are a number of interesting algorithms for encoding trees. Here we present one of Read's algorithms. It is basically an algorithm to find $\text{CTR}(T)$, keeping certain information for each vertex as it progresses. When the center is reached, a root node is uniquely chosen. Read's algorithm encodes a tree as an integer. Its description reveals a number of interesting properties satisfied by trees.

Let T be a tree whose center is to be found. Instead of actually deleting the leaves of T, let us simply draw a circle around each one. Draw the circle in such a way that it encloses all the circles of any adjacent nodes which have been previously circled. The last vertices to be circled form the center.

This system of nested circles can be redrawn in various ways.

Each circle corresponds to a vertex of T. The largest circle which encloses the entire system corresponds to the center of T. Two circles correspond to adjacent vertices if and only if one circle is nested inside the other. The circles not containing a nested circle are the leaves of T. If we cut off the top and bottom of each circle in

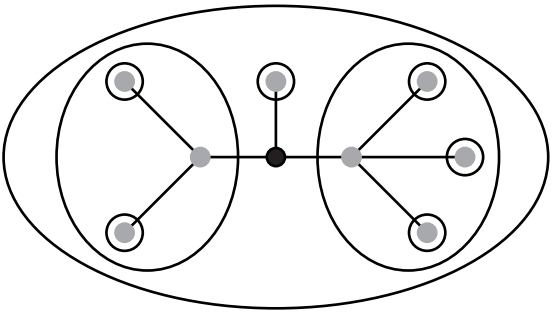


FIGURE 6.6
A circled tree

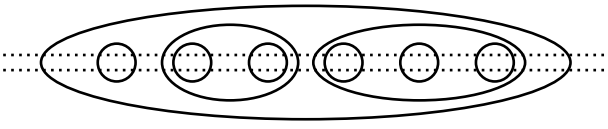


FIGURE 6.7
Nested circles

Figure 6.7, we are left with a set of matched parentheses: (()(()())(()()())). By writing 0 for each left parenthesis and 1 for each right parenthesis, this can be considered a binary number, 0010010110010101111, which represents an integer.

The internal circles in Figure 6.6 have been sorted and arranged in order of increasing complexity. For example, the first inner circle can be denoted 01. This is less than the second circle, which can be denoted 001011, which in turn is less than the third circle 00101011, considered as binary numbers. Thus, there is a natural ordering associated with these systems of nested circles.

The binary number associated with each vertex v is called its *tag*, denoted $t(v)$. Initially each leaf has a tag of 01. The algorithm to find $\textsc{Ctr}(T)$ constructs the vertex tags as it proceeds.

Algorithm 6.3.1: TREEENCODE(T)

comment: construct an integer tree code to represent the tree T

repeat
  construct $L(T)$, the set of leaves of T, stored on an array
  for each leaf $v \in L(T)$
  do {
    comment: suppose that $v \longrightarrow \{u_1, u_2, \ldots, u_k\} \subseteq L(T)$
    sort the tagged vertices adjacent to v by tag,
     say $t(u_1) \leq t(u_2) \leq \cdots \leq t(u_k)$
    $t(v) \leftarrow 0t(u_1)t(u_2)\cdots t(u_k)1$ "concatenate them"
  $T \leftarrow T - L(T)$ "just *mark* the vertices deleted"
until all of T has been tagged
if $L(T)$ contains one vertex u
  then return $(t(u))$
  else {
    comment: $L(T)$ contains two vertices u and v, say $t(u) \leq t(v)$
    return $(0t(u)t'(v))$

On the last iteration, when the center was found, either one or two vertices will have been tagged. They form the center of T. If T is a central tree, with $\text{CTR}(T) = u$, we choose u as the root of T. Then $t(u)$, the tag of the center, represents the entire system of nested circles. It is chosen as the encoding of T.

If T is a bicentral tree, with center uv, we must decide which vertex to choose as the root of T. We arbitrarily choose the one with the larger tag. Suppose that $t(u) \leq t(v)$, so that v is chosen as the root. The code for the entire tree is formed by altering the enclosing circle of v so as to enclose the entire tree. This is illustrated in Figure 6.8.

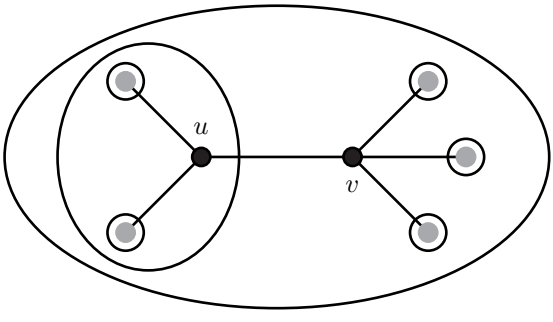


FIGURE 6.8
Choosing the root vertex

Thus, the tree code for T in the bicentral case is the concatenation $0t(u)t'(v)$, where $t'(v)$ is formed from $t(v)$ by dropping one initial 0-bit.

If $t(u) = t(v)$ for some bicentral tree T, then we can obviously select either u or v as the root vertex.

The easiest way to store the tags $t(v)$ is as an integer array $t[v]$. We also need to store the length $\ell[v]$, of each tag, that is, its length in bits. Initially each leaf v has $t[v] = 1$ and $\ell[v] = 2$. To concatenate the tags $0t(u_1)t(u_2)\cdots t(u_k)1$ we use a loop.

$t[v] \leftarrow 0$
for $i \leftarrow 1$ **to** k
do { shift $t[v]$ left $\ell[v]$ bits
$t[v] \leftarrow t[v] + t[u_i]$
$\ell[v] \leftarrow \ell[v] + \ell[u_i]$
$t[v] \leftarrow 2t[v] + 1$
$\ell[v] \leftarrow \ell[v] + 2$

If a primitive left shift operation is not available, one can always store a table of powers of 2, and use multiplication by $2^{\ell[u_i]}$ in order to shift $t[v]$ left by $\ell[u_i]$ bits.

6.3.1 The decoding algorithm

If the circles of Figure 6.6 are unnested, they can be redrawn so as to emphasize their relation to the structure of T.

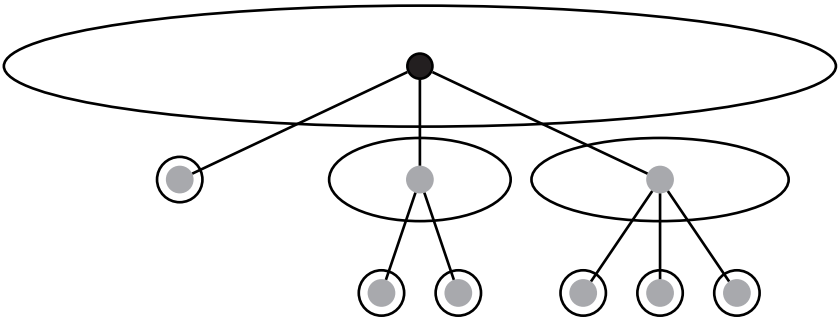


FIGURE 6.9
Unnested circles

The decoding algorithm scans across the system of nested circles. Each time a new circle is entered, a vertex is assigned to it. The first circle entered is that corresponding to the root vertex. The decoding algorithm uses a global vertex counter k, which is initially zero, and constructs a global tree T. It can be programmed to scan the tree code from right to left as follows:

Algorithm 6.3.2: TREEDECODE($Tcode, v_i$)

comment: a new circle has just been entered, from circle v_i.

$k \leftarrow k+1$ "create a new vertex"
join $v_k \longrightarrow v_i$
$Tcode \leftarrow Tcode/2$ "shift right 1 bit"
while *Tcode* is odd
do { **comment:** the rightmost bit = 1, a new circle is entered
TREEDECODE($Tcode, v_k$)
$Tcode \leftarrow Tcode/2$ "shift right 1 bit"

The easiest way to use Algorithm 6.3.2 is to create a dummy vertex v_0 which will only be joined to the root vertex, v_1, and then delete v_0 once the tree has been constructed.

$k \leftarrow 0$
TREEDECODE($Tcode, v_0$)
delete v_0

Exercises

6.3.1 Encode the trees of Figure 6.5 into nested circles by hand. Write down their tree codes.

6.3.2 Work through Algorithm 6.3.2 by hand, for the tree codes 001011 and 0001011011.

6.3.3 Write Algorithm 6.3.2 so as to scan the tree code from left to right, using multiplication by 2 to shift *Tcode* to the right, and using the sign bit of the code to test each bit. Assume a word length of 32 bits.

6.3.4 Program the encoding and decoding algorithms.

6.3.5 If T has n vertices, what is the total length of its tree code, in bits? How many 1's and 0's does the code contain? How many leading 0's does the code begin with? What is the maximum value that n can be if T is to be encoded in 32 bits?

6.3.6 Let T be a tree. Prove that the tree obtained by decoding TREEENCODE(T), using the decoding algorithm, is isomorphic to T.

6.3.7 Let T_1 and T_2 be two trees. Prove that $T_1 \cong T_2$ if and only if TREEENCODE(T_1) = TREEENCODE(T_2).

6.3.8 Consider the expression $x_1x_2 \cdots x_{n+1}$, where $x_1, x_2, \ldots, x_{n+1}$ are variables. If parentheses are inserted so as to take exactly two terms at a time, we obtain a valid bracketing of the expression, with n pairs of matched parentheses (e.g., $((x_1(x_2x_3))x_4)$, where $n = 3$). Each pair of matched parentheses contains exactly two terms. Describe the type of rooted tree on n vertices that corresponds to such a valid bracketing. They are called

binary plane trees. Each leaf of the tree corresponds to a variable x_i and each internal node corresponds to a pair of matched parentheses, giving $2n + 1$ vertices in total.

6.3.9 Let p_n denote the number of binary plane trees with n leaves (e.g., $p_0 = 0, p_1 = 1, p_2 = 1, p_3 = 2$, etc.). We take $p_1 = 1$ corresponding to the tree consisting of a single node. Let $p(x) = p_0 + p_1x + p_2x^2 + \cdots$ be the *generating function* for the numbers p_n, where x is a variable. If T_1 and T_2 are two binary plane trees with n_1 and n_2 leaves, respectively, then they can be combined by adding a new root vertex, adjacent to the roots of T_1 and T_2. This gives a binary plane tree with $n_1 + n_2$ leaves. There are $p_{n_1}p_{n_2}$ ways of constructing a tree in this way. This term arises from $p^2(x)$ as part of the coefficient of $x^{n_1+n_2}$. This holds for all values of n_1 and n_2. Therefore, we can write $p(x) = x + p^2(x)$. Solve this identity for $p(x)$ in terms of x, and then use the binomial theorem to write it as a power series in x. Finally, obtain a binomial expression for p_n in terms of n. The numbers p_n are called the *Catalan numbers*. (The answer should be $p_n = \frac{1}{2n-1}\binom{2n-1}{n}$.)

6.4 Generating rooted trees

One way to generate a list of all the trees on n vertices would be to add a new leaf to the trees on $n - 1$ vertices in all possible ways, and to discard duplicates, using the tree codes to identify isomorphic trees. However, they can also be generated directly, one after the other, with no duplicates.

Let T be a central tree, rooted at its center v. Decompose T into its branches $B_1, B_2, \ldots, B_k$. Each branch B_i is also rooted at v. Write $T = (B_1, B_2, \ldots, B_k)$ to indicate the decomposition into branches. Because v is the center of T, it is the middle vertex of every longest path in T. Therefore, the two "tallest" branches of T will have equal height, where we define the *height of a branch* B rooted at v as $h(B) = \textsc{Max}\{\textsc{Dist}(v, w) \mid w \in B\}$. If the branches of the central tree T have been ordered by height, so that $h(B_1) \geq h(B_2) \geq \cdots \geq h(B_k)$, then we know that $h(B_1) = h(B_2)$. Any rooted tree for which the two highest branches have equal height is necessarily rooted at its center. Therefore, when generating central trees, the branches must be ordered by height.

Generating the rooted trees on n vertices in a sequence implies a linear ordering on the set of of all rooted trees on n vertices. In order to construct a data structure representing a rooted tree T as a list of branches, we also require a linear order on the set of all branches. Then we can order the branches of T so that $B_1 \geq B_2 \geq \cdots \geq B_k$. This will uniquely identify T, as two trees with the same set branches will have the same ordered set of branches. The smallest possible branch will evidently be of height one, and have two vertices – it is K_1 rooted at a vertex. The tree shown in Figure 6.10 has five branches of height one; call them *elementary branches*. The

next smallest branch is of height two, and has three vertices – it is the path P_2 rooted at a leaf.

Associated with each branch B is a rooted tree, as shown in Figure 6.3, constructed by advancing the root to its unique adjacent vertex, and deleting the original root. We will use the ordering of rooted trees to define recursively an ordering of all branches, and the ordering of all branches to define recursively an ordering of all rooted trees. We know that all branches on at most three vertices have already been linearly ordered.

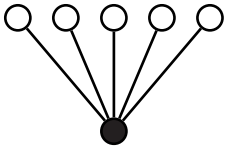


FIGURE 6.10
A tree with all branches of height one

DEFINITION 6.3: Suppose that all branches on at most $m \geq 2$ vertices have been linearly ordered. Let $T = (B_1, B_2, \ldots, B_k)$ and $T' = (B'_1, B'_2, \ldots, B'_\ell)$, where $k+\ell \geq 3$, be any two distinct rooted trees with given branches, such that each branch has at most m vertices, ordered so that $B_1 \geq B_2 \geq \cdots \geq B_k$ and $B'_1 \geq B'_2 \geq \cdots \geq B'_\ell$. Suppose that $|T| \leq |T'|$. Then T and T' are compared as follows:

1. If $|T| < |T'|$, then $T < T'$.
2. Otherwise, compare $(B_1, B_2, \ldots, B_k)$ and $(B'_1, B'_2, \ldots, B'_\ell)$ lexicographically. That is, find i, the first subscript such that $B_i \neq B'_i$; then $T < T'$ if $B_i < B'_i$.

The first condition is to ensure that all rooted trees on n vertices precede all trees on $n+1$ vertices in the linear order. The second condition defines an ordering of trees based on the lexicographic ordering of branches. Notice that if $k \neq \ell$, there must be an i such that $B_i \neq B'_i$; for if every $B_i = B'_i$, but $k \neq \ell$, then T and T' would have different numbers of vertices, so that condition (1) would apply. This defines a linear ordering on all rooted trees whose branches all have at most m vertices. This includes all trees on $m+1$ vertices with at least two branches. In fact, it includes all rooted trees on $m+1$ vertices, except for the path P_m, rooted at a leaf. As this is a branch on $m+1$ vertices, it is handled by Definition 6.4.

We now have an ordering of rooted trees with at least two branches, based on the ordering of branches. We can use it in turn to extend the ordering of branches. Let B and B' be two distinct branches. In order to compare B and B', we advance their roots, as in Figure 6.3, to the unique adjacent vertex in each, and delete the original root. Let the rooted trees obtained in this way be T and T', respectively. Then $B < B'$ if $T < T'$. In summary, branches are compared as follows:

DEFINITION 6.4: Suppose that all rooted trees on $\leq m-1$ vertices have been linearly ordered, and that all rooted trees on m vertices with at least two branches have also been linearly ordered. Let B and B' be branches on m vertices, with corresponding rooted trees T and T'. Suppose that $|B| \leq |B'|$ and that if $|B| = |B'|$, then B is the branch of smaller height. Then B and B' are compared as follows:

1. If $|B| < |B'|$, then $B < B'$.
2. Otherwise, if $h(B) < h(B')$, then $B < B'$.
3. Otherwise, $B < B'$ if $T < T'$.

We have a recursive ordering which compares trees by the ordering of their branches, and branches by the ordering of their trees. We must prove that the definition is valid.

Theorem 6.1. *Definitions 6.3 and 6.4 determine a linear order on the set of all rooted trees.*

Proof. Notice that rooted trees have a sub-ordering based on the number of vertices – all rooted trees on n vertices precede all rooted trees on $n+1$ vertices. Branches have an additional sub-ordering based on height – all branches of height h on n vertices precede all branches of height $h+1$ on n vertices. A branch is a special case of a rooted tree, in which the root vertex has degree one. If a branch B and tree T on n vertices are compared, where T has at least two branches, then that first branch of T has fewer than n vertices, so that $T < B$, by Definition 6.3. Therefore all trees whose root vertex has degree two or more precede all branches on n vertices.

The definitions are clearly valid for all rooted trees on ≤ 3 vertices. Suppose that the set of all rooted trees on $\leq n$ vertices is linearly ordered by these definitions, and consider two distinct rooted trees

$$T = (B_1, B_2, \ldots, B_k)$$

and

$$T' = (B'_1, B'_2, \ldots, B'_\ell)$$

on $n+1$ vertices. If $k = \ell = 1$, then T and T' are both branches. The trees formed by advancing their root vertices have only n vertices, and so can be compared by Definition 6.3. Otherwise at least one of T and T' has two or more branches. Therefore at least one of each pair B_i and B'_i of branches has $\leq n$ vertices. Therefore the branches B_i and B'_i can be compared by Definition 6.4. The conclusion follows by induction. □

In this ordering of branches and trees, the first rooted tree on n vertices is a *star* consisting of the tree $K_{1,n-1}$ rooted at its center. The last rooted tree on n vertices is a path P_n, rooted at a leaf. The first branch on n vertices is $K_{1,n-1}$, rooted at a leaf, and the last branch on n vertices is also the path P_n, rooted at a leaf. The first few rooted trees are shown in Figure 6.11.

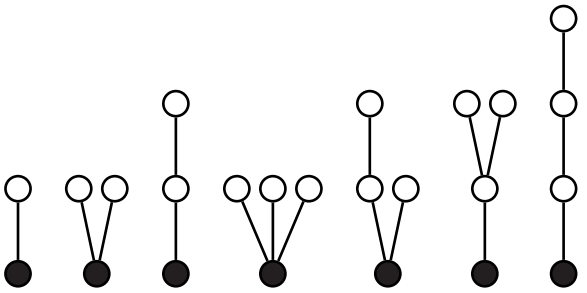


FIGURE 6.11
The beginning of the linear order of rooted trees

Let $T = (B_1, B_2, \ldots, B_k)$ be the list of branches of a rooted tree T, with root vertex v. The recursive data structure we use to represent T is a linked list of branches. Each branch B_i also has root v. It is in turn represented in terms of the rooted tree T', whose root vertex is the unique vertex adjacent to v. Thus, a record representing a tree T has four fields:

- *NodeNum*$\langle T \rangle$, the node number of the root, which is used for printing.
- *nNodes*$\langle T \rangle$, the number of vertices in the tree.
- *FirstBranch*$\langle T \rangle$, a pointer to the first branch.
- *LastBranch*$\langle T \rangle$, a pointer to the last branch.

Each branch B of T has a corresponding rooted tree T'. B is represented as a record having four fields:

- *Height*$\langle B \rangle$, the height of the branch.
- *NextRoot*$\langle B \rangle$, a pointer to the rooted tree T'.
- *NextBranch*$\langle B \rangle$, a pointer to the next branch of T.
- *PrevBranch*$\langle B \rangle$, a pointer to the previous branch of T.

It is not necessary to store the number of vertices of a branch B, as it is given by *nNodes*$\langle$*NextRoot*$\langle B \rangle\rangle + 1$. The functions which compare two trees and branches are given as follows. They return an integer, whose value is one of three constants, *LessThan*, *EqualTo*, or *GreaterThan*.

Algorithm 6.4.1: COMAPARETREES(T_1, T_2)

comment: T_1 and T_2 both have at least one branch

if $|T_1| < |T_2|$ **then return** (*LessThan*)
if $|T_1| > |T_2|$ **then return** (*GreaterThan*)
comment: otherwise $|T_1| = |T_2|$

$B_1 \leftarrow$ *FirstBranch*$\langle T_1 \rangle$
$B_2 \leftarrow$ *FirstBranch*$\langle T_2 \rangle$
Result $\leftarrow$ COMPAREBRANCHES(B_1, B_2)
while *Result* $=$ *EqualTo*
do {
  if $B_1 =$ *LastBranch*$\langle T_1 \rangle$ **then return** (*EqualTo*)
  $B_1 \leftarrow$ *NextBranch*$\langle B_1 \rangle$
  $B_2 \leftarrow$ *NextBranch*$\langle B_2 \rangle$
  Result $\leftarrow$ COMPAREBRANCHES(B_1, B_2)
return (*Result*)

Algorithm 6.4.2: COMPAREBRANCHES(B_1, B_2)

comment: B_1 and B_2 both have a unique vertex adjacent to the root

if $|B_1| < |B_2|$ **then return** (*LessThan*)
if $|B_1| > |B_2|$ **then return** (*GreaterThan*)
comment: otherwise $|B_1| = |B_2|$

if *Height*$\langle B_1 \rangle <$ *Height*$\langle B_2 \rangle$ **then return** (*LessThan*)
if *Height*$\langle B_1 \rangle >$ *Height*$\langle B_2 \rangle$ **then return** (*GreaterThan*)
comment: otherwise *Height*$\langle B_1 \rangle =$ *Height*$\langle B_2 \rangle$

if *Height*$\langle B_1 \rangle = 1$ **then return** (*EqualTo*)
$T_1 \leftarrow$ *NextRoot*$\langle B_1 \rangle$
$T_2 \leftarrow$ *NextRoot*$\langle B_2 \rangle$
return (COMPARETREES(T_1, T_2))

Using these functions we can generate all rooted trees on n vertices, one after the other, beginning with the first tree, which consists of a root vertex and $n - 1$ elementary branches of height one, until the last tree is reached, which has only one branch, of height $n - 1$. Written in pseudo-code, the technique is as follows, where NEXTTREE(T) is a procedure which replaces T with the next tree, and returns **true** unless T was the last tree (see Algorithm 6.4.3). FIRSTTREE(n) is a procedure that constructs the first tree on n vertices.

$T \leftarrow$ FIRSTTREE(n)
repeat
  PRINTTREE(T)
until not NEXTTREE(T)

Suppose that T has branch decomposition $(B_1, B_2, \ldots, B_k)$, where $B_1 \geq B_2 \geq \cdots \geq B_k$. The procedure NEXTTREE$(T)$ works by finding the last branch B_i such that $B_i \neq B_{i-1}$. Then $B_i, B_{i+1}, \ldots, B_k$ is a sequence of isomorphic branches. So long as B_i is not simply a path of length $h(B_i)$, there is a larger branch with the same number of vertices. B_i is then replaced with the next larger branch and the subsequent branches $B_{i+1}, B_{i+2}, \ldots, B_k$ are replaced with a number of elementary branches. This gives the lexicographically next largest tree. This is illustrated in Figure 6.12. Here, B_2 was replaced with a branch of height three, and B_3 was replaced with three elementary branches.

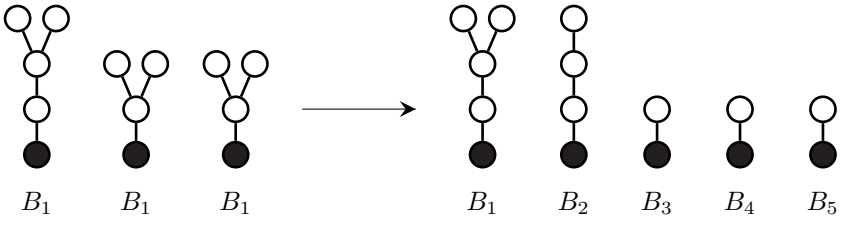


FIGURE 6.12
Constructing the next tree

But if B_i is simply a path, then it is the last branch with $|B_i|$ vertices. In order to get the next branch we must add another vertex. B_i is then replaced with the first branch with one more vertex. This is the unique branch with $|B_i| + 1$ vertices and height two. T is then filled in with as many elementary branches as needed. This is illustrated in Figure 6.13.

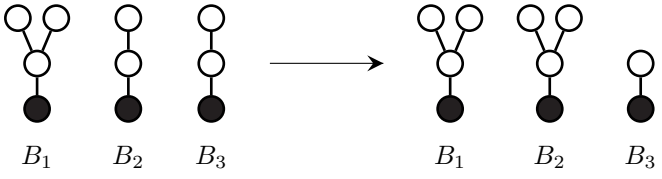


FIGURE 6.13
Constructing the next tree

The procedure DELETEBRANCHES(T, B_1) destroys all branches of T following B_1, and returns the number of nodes deleted. Similarly DESTROYTREE(T) is a procedure that destroys all branches of T, and returns the total number of nodes deleted.

Algorithm 6.4.3: NEXTTREE(T)

$B_1 \leftarrow \mathit{LastBranch}\langle T\rangle$
if $B_1 = \mathit{FirstBranch}\langle T\rangle$
  then { **comment:** only one branch – advance the root
  if $\mathit{nNodes}\langle T\rangle = \mathit{Height}\langle B_1\rangle + 1$ **then return** (**false**)
  return (NEXTTREE($\mathit{NextRoot}\langle B_1\rangle$))
comment: otherwise at least two branches
$B_2 \leftarrow \mathit{PrevBranch}\langle B_1\rangle$
while COMPAREBRANCHES(B_1, B_2) = *EqualTo*
  do { $B_1 \leftarrow B_2$
  if $B_1 = \mathit{FirstBranch}\langle T\rangle$ **then go to** 1
  $B_2 \leftarrow \mathit{PrevBranch}\langle B_2\rangle$
1 : **comment:** delete the branches of T following B_1
$N \leftarrow$ DELETEBRANCHES(T, B_1) "N nodes are deleted"
comment: replace B_1 with next branch, if possible
if $\mathit{nNodes}\langle \mathit{NextRoot}\langle B_1\rangle\rangle > \mathit{Height}\langle B_1\rangle$
  then { NEXTTREE($\mathit{NextRoot}\langle B_1\rangle$)
  fill in T with N elementary branches
  return (**true**)
comment: otherwise construct the first branch with one more node
if $N > 0$ **then**
{ $M \leftarrow$ DESTROYTREE($\mathit{NextRoot}\langle B_1\rangle$) "$M$ nodes are deleted"
$\mathit{NextRoot}\langle B_1\rangle \leftarrow$ FIRSTTREE($M + 1$)
fill in T with $N - 1$ elementary branches
return (**true**)
comment: otherwise there's no branch following B_1 to take a node from
repeat
  $B_1 \leftarrow B_2$
  if $B_1 = \mathit{FirstBranch}\langle T\rangle$ **then go to** 2
  $B_2 \leftarrow \mathit{PrevBranch}\langle B_1\rangle$
until COMPAREBRANCHES(B_1, B_2) $\neq$ *EqualTo*
2 : **comment:** delete the branches of T following B_1
$N \leftarrow$ DELETEBRANCHES(T, B_1) "N nodes are deleted"
comment: replace B_1 with next branch
if $\mathit{nNodes}\langle \mathit{NextRoot}\langle B_1\rangle\rangle > \mathit{Height}\langle B_1\rangle$
  then NEXTTREE($\mathit{NextRoot}\langle B_1\rangle$)
  else $\mathit{NextRoot}\langle B_1\rangle \leftarrow$ FIRSTTREE($\mathit{nNodes}\langle \mathit{NextRoot}\langle B_1\rangle\rangle + 1$)
fill in T with elementary branches
return (**true**)

Theorem 6.2. *Let T be a tree on n vertices. Algorithm* NEXTTREE(T) *constructs the next tree on n vertices after T in the linear order of trees, if there is one.*

Proof. The proof is by induction on n. It is easy to check that it works for trees on $n = 2$ and $n = 3$ vertices. Suppose that it holds up to $n - 1$ vertices, and let T have n vertices. Let $T = (B_1, B_2, \ldots, B_k)$ be the branches of T, where $B_1 \geq B_2 \geq \cdots \geq B_k$. The algorithm first checks whether there is only one branch. If so, and T is a branch of height $n - 1$, it returns **false** . Otherwise let T' be the rooted tree corresponding to B_1 by advancing the root. The algorithm calls NEXTTREE(T'). Because T' has $n - 1$ vertices, this gives the next branch following B_1 in the linear order, as required.

Otherwise, T has at least two branches. It finds the branch B_i such that $B_i = B_{i+1} = \ldots = B_k$, but $B_{i-1} \neq B_i$, if there is one (possibly $i = 1$). The first tree following T must differ from T in B_i, unless $i = k$ and B_i is a path. In the first case, the algorithm replaces B_i with the next branch in the linear order, and fills in the remaining branches of T with elementary branches. This is the smallest tree on n vertices following T. Otherwise $i = k$ and B_i is a path, so that there is no tree following B_i. Because there are at least two branches, the algorithm finds the branch B_j such that $B_j = B_{j+1} = \ldots = B_{k-1} > B_k$ (possibly $j = 1$). It then replaces B_j with the first branch following it, and fills in T with elementary branches. In each case the result is the next tree after T. □

Exercises

6.4.1 Work through the NEXTTREE(T) algorithm by hand to construct all the rooted trees on 4, 5, 6, and 7 vertices.

6.4.2 Write the recursive procedure PRINTTREE(T) to print out a tree as shown in Figure 6.14, according to the distance of each vertex from the root.

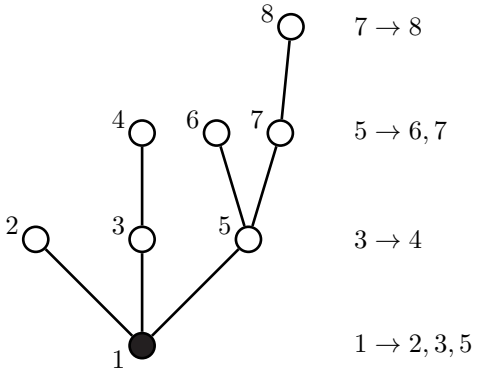


FIGURE 6.14
Printing a rooted tree

6.4.3 Write the recursive functions DESTROYTREE(T) and DELETEBRANCHES(T, B_1), both of which return the number of vertices deleted.

6.4.4 Program the NEXTTREE(T) algorithm, and use it to find the number of rooted trees up to 10 vertices.

6.5 Generating non-rooted trees

The NEXTTREE(T) algorithm generates the rooted trees on n vertices in sequence, beginning with the first tree of height 1 and ending with the tree of height $n - 1$. In order to generate non-rooted trees, we must root them in the center. Because every non-rooted tree can be viewed as a rooted tree, all non-rooted trees also occur in the linear order of trees. Central trees can be generated by modifying NEXTTREE(T) so that the two highest branches are always required to have equal height. This can be done with another procedure, NEXTCENTRALTREE(T), which in turn calls NEXTTREE(T) when forming the next branch. Bicentral trees are slightly more difficult, because the highest branches B_1 and B_2 satisfy $h(B_1) = h(B_2) + 1$. If we generate trees in which the heights of the two highest branches differ by one, then most bicentral trees will be constructed twice, once for each vertex in the center. For example, Figure 6.15 shows two different branch decompositions of the same bicentral tree. It would therefore be generated twice, because it has different branch decompositions with respect to the two possible roots.

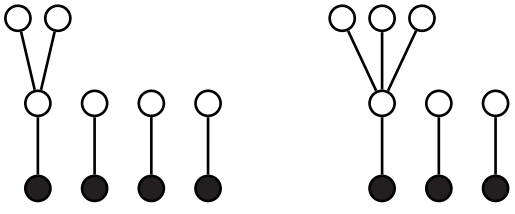


FIGURE 6.15
Two decompositions of a bicentral tree

The easiest solution to this is to subdivide the central edge with a new vertex, taking it as the root. Then each bicentral tree on n vertices corresponds to a unique central tree on $n + 1$ vertices, with exactly two branches. We can construct these by generating rooted trees with only two branches, which have equal height, and then ignoring the extra root vertex.

Exercises

6.5.1 Write and program the procedures NEXTCENTRALTREE(T) and NEXTBICENTRALTREE(T), and use them to construct all the non-rooted trees on n vertices, up to $n = 15$.

6.6 Prüfer sequences

Read's algorithm encodes a tree according to its isomorphism type, so that isomorphic trees have the same code. This can be used to list all the isomorphism types of trees on n vertices. A related question is to make a list all the trees on the n vertices $V_n = \{1, 2, \ldots, n\}$. These are sometimes referred to as *labeled trees*. For example, Figure 6.16 illustrates the three distinct, or labeled trees on three vertices, which are all isomorphic to each other.

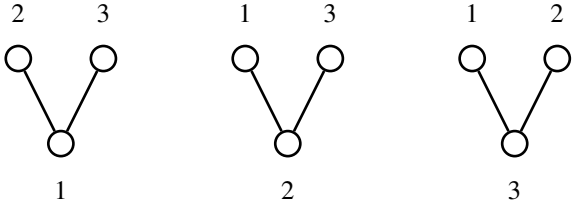


FIGURE 6.16
Three distinct trees on three vertices

Let T be a tree with $V(T) = \{1, 2, \ldots, n\}$. A *Prüfer sequence* for T is a special encoding of T as an integer sequence. For example, the tree of Figure 6.17 with $n = 9$ has Prüfer sequence $t = (6, 9, 1, 4, 4, 1, 6)$.

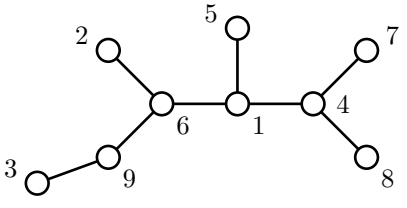


FIGURE 6.17
Finding the Prüfer sequence of a tree

This is constructed as follows. The leaves of T are $L(T) = \{2, 3, 5, 7, 8\}$. The numerically smallest leaf is 2. Because $2 \to 6$, we take $t_1 = 6$ as the first member of

t. We now set $T := T-2$, and find $L(T) = \{3, 5, 7, 8\}$. We again choose the smallest leaf, 3, and because $3 \to 9$, we take $t_2 = 9$ as the second member of the sequence, and set $T := T - 3$. Notice that when 3 is deleted, 9 becomes a leaf. Therefore, on the next iteration we will have $L(T) = \{5, 7, 8, 9\}$. The general technique is the following:

for $k \leftarrow 1$ **to** $n-2$
do $\begin{cases} \text{find } L(T) \\ \text{select } v \in L(T), \text{ the smallest leaf} \\ t_k \leftarrow \text{ the unique vertex adjacent to } v \\ T \leftarrow T - v \end{cases}$
comment: T now has 2 vertices left

This always gives a sequence of $n-2$ integers $t=(t_1, t_2, \ldots, t_{n-2})$, where each $t_i \in V_n$. Notice that at each step, a leaf of T is deleted, so that T is always a tree throughout all the steps. Because T is a tree, we can always choose a leaf to delete. When T reduces to a single edge, the algorithm stops. Therefore no leaf of T is ever chosen as any t_k. In fact, if $\text{DEG}(v) \geq 2$, then v will appear in t each time a leaf adjacent to v is deleted. When the degree drops to one, v itself becomes a leaf, and will appear no more in t. Therefore, each vertex v appears in t exactly $\text{DEG}(v) - 1$ times.

Theorem 6.3. *Let $t = (t_1, t_2, \ldots, t_{n-2})$ be any sequence where each $t_i \in V_n = \{1, 2, \ldots, n\}$. Then t is the Prüfer sequence of a tree T on n vertices.*

Proof. The sequence t consists of $n-2$ integers of V_n. Therefore at least two members of V_n are not used in t. Let L be those numbers not used in t. If t were formed by encoding a graph using the above technique, then the smallest element $v \in L$ must have been a leaf adjacent to t_1. So we can join $v \to t_1$, and discard t_1 from t. Again we find L, the numbers not used in t, and pick the smallest one, etc. This is summarized in the following pseudo-code:

$N \leftarrow \{1, 2, \ldots, n\}$
for $k \leftarrow 1$ **to** $n-2$
do $\begin{cases} \text{construct } L, \text{ those numbers of } N \text{ not used in } t \\ \text{select the smallest } v \in L \\ \text{join } v \longrightarrow t_k \\ \text{discard } t_k \text{ from } t \\ N \leftarrow N - v \end{cases}$
comment: T now has 2 vertices left, u and v

join $u \longrightarrow v$

This creates a graph T with $n-1$ edges. It is the *only* graph which could have produced the Prüfer sequence t, using the above encoding technique. Must T be a tree? If T were not a tree, then it could not be connected, because T has only $n-1$ edges. In that case, some component of T would have a cycle C. Now the encoding technique only deletes leaves. No vertex on a cycle could ever be deleted by this

method, for the degree of every $u \in C$ is always at least two. This means that a graph containing a cycle would not produce a Prüfer sequence of length $n-2$. Therefore T can have no cycle, which means that it must be a tree. □

Thus we see that every tree with vertex set $\{1, 2, \ldots, n\}$ corresponds to a unique Prüfer sequence, and that every sequence t can only be obtained from one tree. The corollary is that the number of trees equals the number of sequences. Now it is clear that there are n^{n-2} such sequences, because each of the $n-2$ elements t_k can be any of the numbers from 1 to n. This result is called Cayley's theorem.

Theorem 6.4. (Cayley's theorem) *The number of distinct trees on n vertices is n^{n-2}.*

6.7 Spanning trees

Consider the problem of making a list of all the spanning trees of a graph G. If $G \cong K_n$, then there are n^{n-2} spanning trees, and each one corresponds to a Prüfer sequence. If $G \ncong K_n$, then we can find all the spanning trees of G as follows. Choose any edge uv of G. First find all the spanning trees that use uv and then find all the trees that do not use uv. This gives all spanning trees of G. Write $\tau(G)$ for the number of spanning trees of G. The spanning trees T that do not use edge uv are also spanning trees of $G - uv$, and their number is $\tau(G - uv)$. If T is a spanning tree that does use uv, then we can *contract* the edge uv, identifying u and v so that they become a single vertex. Let $T \cdot uv$ denote the reduced tree. It is a spanning tree of $G \cdot uv$. Every spanning tree of $G \cdot uv$ is a contraction $T \cdot uv$ of some spanning tree T of G; for just expand the contracted edge back into uv to get T. This gives:

Lemma 6.5. *Let G be any graph. Then $\tau(G) = \tau(G - uv) + \tau(G \cdot uv)$.*

This applies equally well to simple graphs and multigraphs. It is illustrated in Figures 6.18 and 6.19.

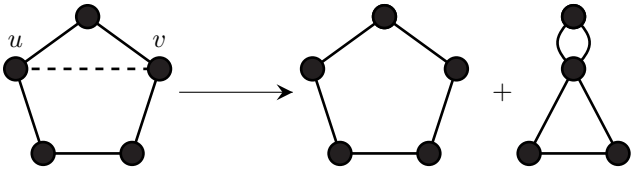


FIGURE 6.18
Deletion and contraction of edge uv

Notice that even when G is a simple graph, $G \cdot uv$ will often be a multigraph, or have loops. Now loops can be discarded, because they cannot be part of any spanning

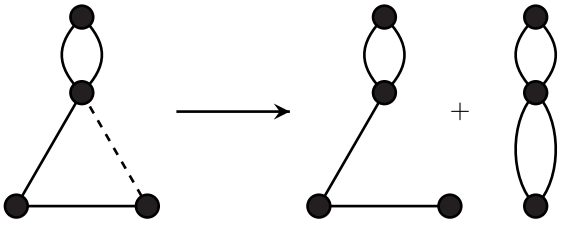


FIGURE 6.19
Finding the number of spanning trees

tree. However multiple edges must be kept, because they correspond to different spanning trees of G.

In the example of Figure 6.18, the 5-cycle obviously has five spanning trees. The other graph is then decomposed, giving "trees" which contain some multiple edges (i.e., replacing the multiple edges with single edges gives a tree). The two such "trees" shown clearly have two and four spanning trees each, respectively. Therefore the original graph has $5 + 2 + 4 = 11$ spanning trees.

In general, if G has an edge of multiplicity k joining vertices u and v, then deleting any one of the equivalent k edges will give the same number of spanning trees. Contracting any one of them forces the rest to collapse into loops, which are then discarded. This gives the following lemma:

Lemma 6.6. *Let edge uv have multiplicity k in G. Replace the multiple edges having endpoints u and v by a single edge uv to get a graph G_{uv}. Then*

$$\tau(G) = \tau(G_{uv} - uv) + k\tau(G_{uv} \cdot uv).$$

This gives a recursive technique for finding the number of spanning trees of a connected graph G. G is stored as a weighted simple graph, for which the weight of an edge represents its multiplicity.

Algorithm 6.7.1: SPTREES(G)

find an edge uv on a cycle
if there is no such edge
then { **comment:** G is a tree
return (product of edge weights)
else return (SPTREES($G - uv$) + WT(uv) $*$ SPTREES($G \cdot uv$))

This can be expanded to make a list of all spanning trees of G. However, if only the *number* of spanning trees is needed, there is a much more efficient method.

6.8 The matrix-tree theorem

The number of spanning trees of G can be computed as the determinant of a matrix. Let $A(G)$ denote the adjacency matrix of G. The *degree matrix* of G is $D(G)$, all of whose entries are 0, except for the diagonal, which satisfies $[D]_{uu} = \text{DEG}(u)$, for vertex u. The *Kirchhoff matrix* of G is $K(G) = D - A$. This matrix is sometimes also called the *Laplacian matrix* of G. The number of spanning trees is found from the Kirchhoff matrix.

First, notice that the row and column sums of K are all 0, because the row and column sums of A are the degrees of G. Therefore, $\det(K) = 0$. Consider the expansion of $\det(K)$ into cofactors along row u. Write

$$\det(K) = \sum_{v=1}^{n} (-1)^{u+v} k_{uv} \ \det(K_{uv}).$$

Here k_{uv} denotes the entry in row u and column v of K, and K_{uv} denotes the submatrix formed by crossing out row u and column v. The *cofactor* of k_{uv} is $(-1)^{u+v} \det(K_{uv})$. There are n vertices.

Theorem 6.7. (Matrix-Tree Theorem) *Let K be the Kirchhoff matrix of G. Then $\tau(G) = (-1)^{u+v} \det(K_{uv})$, for any row index u and any column index v.*

Proof. Notice that the theorem says that all cofactors of K have the same value, namely, the number of spanning trees of G. The proof is by induction on the number of vertices and edges of G. Suppose first that G is a disconnected graph; let one of the components be H. Order the vertices so that vertices of H come before the rest of G. Then $K(G)$ is a block matrix, as shown in Figure 6.20.

$$K(G) = \begin{array}{|c|c|} \hline K(H) & 0 \\ \hline 0 & K(G-H) \\ \hline \end{array}$$

FIGURE 6.20
Kirchhoff matrix of a disconnected graph

If row u and column v, where $u, v \in V(H)$, are crossed off, then the row and column sums of $G-H$ will be all 0, so that the cofactor corresponding to K_{uv} will be

zero. Similarly, if any other row and column are crossed off, the remaining cofactor will be zero. Therefore, if G is disconnected, the theorem is true, because $\tau(G) = 0$.

Suppose now that G is a tree. Choose a leaf v and let $v \longrightarrow u$. Without loss of generality, we can order the vertices so that v is the last vertex. Write $K \cdot uv = K(G \cdot uv)$. The two Kirchhoff matrices are shown in Figure 6.21, where $a = k_{uu}$.

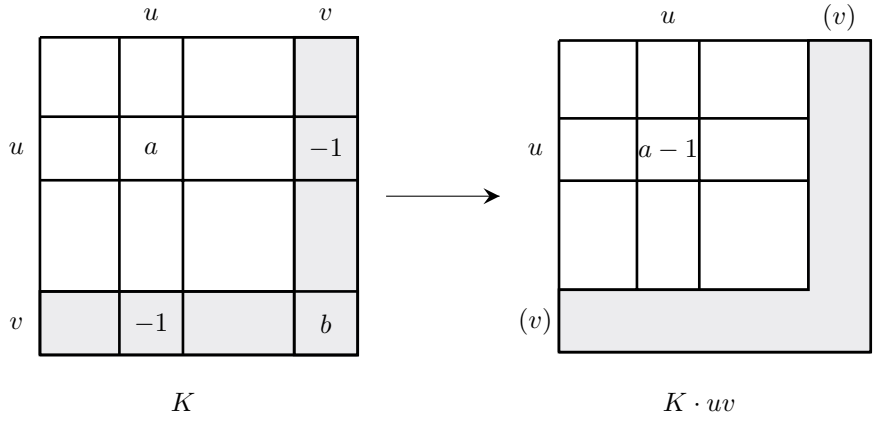


FIGURE 6.21
Kirchhoff matrices for a tree

If $n = 2$, there is only one tree, with Kirchhoff matrix $\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$. All the cofactors have value ± 1, as desired. If $n > 2$, we assume that the matrix-tree theorem holds for all trees with at most $n-1$ vertices, and form $K \cdot uv$. Now $\det(K \cdot uv) = 0$, because it is a Kirchhoff matrix. The submatrix K_{vv} differs from $K \cdot uv$ only in the single term a instead of $a - 1$ in entry uu. When we expand $\det(K_{vv})$ along row u, all the terms are identical to expanding $\det(K \cdot uv)$ along row u, except for this one. Therefore $\det(K_{vv}) - \det(K \cdot uv)$ equals the uu-cofactor in $K \cdot uv$. By the induction hypothesis, this is $\tau(G \cdot uv) = 1$. Therefore $\det(K_{vv}) = 1$. Striking off row and column u from K, and expanding along row v, shows that $\det(K_{uu})$ again equals the uu-cofactor in $K \cdot uv$, which is 1. Therefore, the uu-cofactor in K equals $\tau(G)$. Consider next the cofactor $\det(K_{xy})$, where neither x nor y equals u or v. Strike off row x and column y of K. In order to evaluate $(-1)^{x+y} \det(K_{xy})$, first add row v to row u, and then expand the determinant along column v. The value clearly equals the xy-cofactor of $K \cdot uv$, which is $\tau(G) = 1$. If $x = u$ but $y \neq u$ or v, a similar argument shows that the cofactor equals 1. If $x = v$ but $y \neq v$, then expand along column v to evaluate $(-1)^{x+y} \det(K_{xy})$. The result is $(-1)^{x+y}(-1)(-1)^{u+x-1}$ times the determinant of $(K \cdot uv)_{uy}$. This reduces to $(-1)^{u+y} \det((K \cdot uv)_{uy}) = \tau(G)$. Thus, in all cases, the cofactors equal $\tau(G) = 1$. By induction, the matrix-tree theorem is true for all trees.

If G is a multigraph with $n = 2$ vertices, then it consists of m parallel edges, for some $m \geq 1$, so that $\tau(G) = m$. It is easy to see that the theorem is true in this case, as the Kirchhoff matrix is $\begin{bmatrix} m & -m \\ -m & m \end{bmatrix}$. Suppose now that it holds for all multigraphs

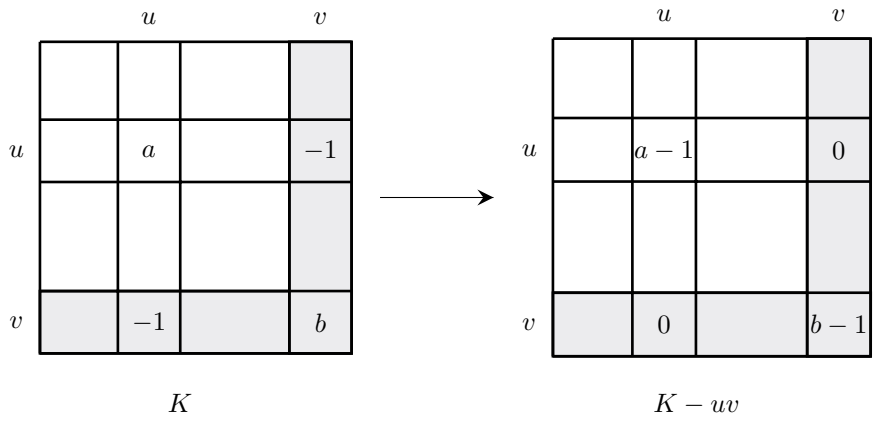


FIGURE 6.22
Kirchhoff matrices $\boldsymbol{K}$ and $\boldsymbol{K - uv}$

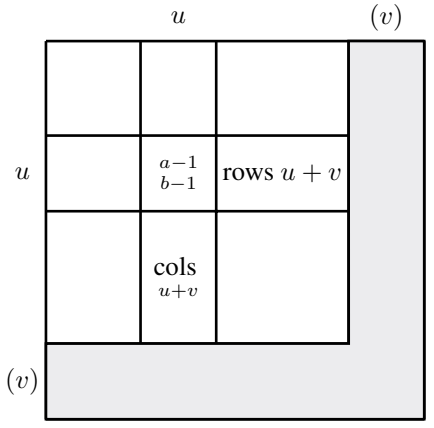
with fewer than n vertices, where $n > 2$, and for all multigraphs on n vertices with less than ε edges, where $\varepsilon > n-1$, because we know that it holds for trees. Choose an edge uv of G, and write $\tau(G) = \tau(G-uv)+\tau(G\cdot uv)$. The corresponding Kirchhoff matrices are illustrated in Figure 6.22, where we write $K - uv$ for $K(G - uv)$.

The diagram is drawn as though the edge uv had multiplicity 1, but the proof is general, and holds for any multiplicity $m \geq 1$. Let a denote the entry k_{uu} and b the entry k_{vv}.

Consider the vv-cofactor of K. It is nearly identical to the vv-cofactor of $K - uv$, differing only in the uu-entry, which is a in K but $a - 1$ in $K - uv$. Expanding $\det(K_{vv})$ along row u shows that $\det(K_{vv}) - \det((K - uv)_{vv})$ equals the uu-cofactor of $K - uv$, with row and column v removed. Comparison with Figure 6.23 shows that this is identical to the uu-cofactor of $K \cdot uv$. Therefore $\det(K_{vv}) - \det((K - uv)_{vv}) = \det((K \cdot uv)_{uu})$. By the induction hypothesis, this gives $\det(K_{vv}) = \tau(G \cdot uv) + \tau(G - uv) = \tau(G)$, as desired.

Consider now K_{vu}, formed by striking off row v and column u of K. This matrix is almost identical to that formed by striking off row v and column u from $K - uv$. The only difference is in the uv-entry. Expanding along row u shows that the difference of the determinants, $\det(K_{vu}) - \det((K-uv)_{vu})$, is $(-1)^{u+v-1}(-1)\det((K\cdot uv)_{uu})$. Therefore $(-1)^{v+u}\det(K_{vu}) = (-1)^{v+u}\det((K - uv)_{vu}) + \det((K\cdot uv)_{uu}) = \tau(K-uv)+\tau(K\cdot uv) = \tau(G)$. Thus, the vu-cofactor and the vv-cofactor both equal $\tau(G)$.

Finally, we show that the remaining entries in row v also have cofactors equal to $\tau(G)$. Consider any entry k_{vw}, where $w \neq u, v$. Strike off row v and column w of K and of $K - uv$. In order to evaluate $\det(K_{vw})$ and $\det((K - uv)_{vw})$, first add the remaining part of column v to column u in both matrices. This is illustrated in Figure 6.24.



$K \cdot uv$

FIGURE 6.23
Kirchhoff matrix $\boldsymbol{K \cdot uv}$

K_{vw} and $(K - uv)_{vw}$ are now identical, except for the uv-entry. Expand $\det(K_{vw})$ along row u. All terms are equal to the corresponding terms in the expansion of $\det((K - uv)_{vw})$ along row u, except for the last term. The difference is $(-1)^{u+v-1}(-1)\det((K \cdot uv)_{uw})$. Therefore $(-1)^{v+w}\det(K_{vw}) = (-1)^{v+w}\det((K - uv)_{vw}) + (-1)^{u+w}\det((K \cdot uv)_{uw})$. As before, we get $(-1)^{v+w}\det(K_{vw}) = \tau(G)$. Thus, all the cofactors of K from row v have equal value, namely, $\tau(G)$. Because v could be any vertex, all the cofactors of K have this value. This completes the proof of the matrix-tree theorem. □

A nice illustration of the use of the matrix-tree theorem is to compute $\tau(K_n)$. The Kirchhoff matrix is

$$K(K_n) = \begin{bmatrix} n-1 & -1 & -1 & \cdots & -1 \\ -1 & n-1 & -1 & \cdots & -1 \\ -1 & -1 & n-1 & & \vdots \\ \vdots & \vdots & & \ddots & -1 \\ -1 & -1 & \cdots & -1 & n-1 \end{bmatrix}$$

Strike off the last row and column. In order to evaluate the determinant, add all the rows to the first row, to get

$$\begin{bmatrix} 1 & 1 & \cdots & \\ -1 & n-1 & & \\ \vdots & & \ddots & \\ & & & n-1 \end{bmatrix}_{(n-1)\times(n-1)}$$

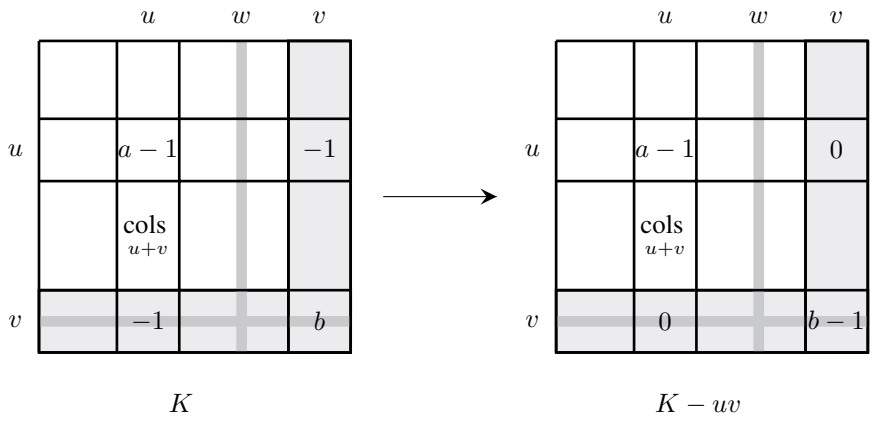


FIGURE 6.24
Evaluating $\det(K_{vw})$

Now add the first row to each row in turn, in order to get n's on the diagonal and 0's off the diagonal. Thus, the determinant is n^{n-2}, as expected.

The Kirchhoff matrix was first used to solve electrical circuits. Consider a simple electrical network consisting of resistors and a battery producing voltage V. Let the nodes in the network be $u_1, u_2, \ldots, u_n$, and suppose that the resistance connecting u_i to u_j is r_{ij}. The battery causes current to flow in the network, and so sets up a voltage V_i at each node u_i. The current from u_i to u_j is $(V_i - V_j)/r_{ij}$. This is illustrated in Figure 6.25.

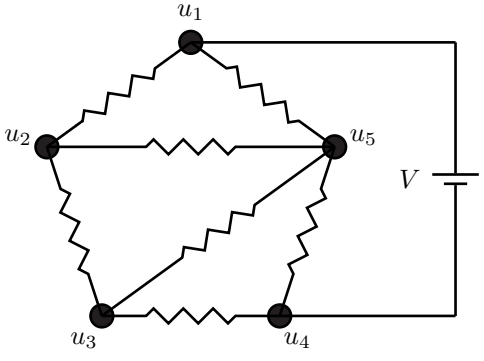


FIGURE 6.25
A simple network of resistors

The law of conservation of charge says that the total current flowing out of node u_i must equal the total current flowing in, that is, not counting the current flowing

through the battery,

$$\sum_{u_j \to u_i} \frac{(V_i - V_j)}{r_{ij}} = 0,$$

for all nodes u_i. The battery maintains a constant voltage difference of V across nodes u_1 and u_n, say. Let I denote the current through the battery. Then the u_1-equation must be modified by setting the right-hand side to I instead of 0; and the u_n-equation requires the right-hand side to be $-I$. This gives a system of linear equations in the variables V_i and I. If we consider the network as a multigraph in which u_i is joined to u_j by $1/r_{ij}$ parallel edges, then the diagonal entries of the matrix corresponding to the equations are the degrees of the nodes. The off-diagonal entries form the negated adjacency matrix of the network. Thus, this is the Kirchhoff matrix of the network. Because the Kirchhoff matrix has determinant zero, there is no unique solution to the system. However, it is voltage *differences* that are important, and we know that the battery maintains a constant voltage difference of V. Therefore, we can arbitrarily set $V_n = 0$ and $V_1 = V$, so that we can cross off the n^{th} column from the matrix. The rows are linearly dependent, so that we can also discard any row. The system then has a unique solution, because striking off a row and column from the Kirchhoff matrix leaves the spanning tree matrix. Notice that once the current in each edge is known, each spanning tree of the network will determine the voltage distribution uniquely, because a spanning tree has a unique path connecting any two vertices.

Exercises

6.8.1 Find $\tau(K_{3,3})$ using the recursive method.

6.8.2 Find $\tau(K_n - uv)$, where uv is any edge of K_n, using the matrix-tree theorem.

6.8.3 Find $\tau(C_n)$, where C_n is the cycle of length n, using the matrix-tree theorem.

6.8.4 What is the complexity of finding the determinant of an $n \times n$ matrix, using Gaussian elimination? Accurately estimate an upper bound on the number of steps needed.

6.8.5 Let G be a graph with n vertices. Replace each edge of G with m multiple edges to get a graph G_m. Prove that $\tau(G_m) = m^{n-1}\tau(G)$.

6.8.6 Program the recursive algorithm to find the number of spanning trees. Use a breadth-first search to find an edge on a cycle.

6.8.7 Solve the electrical circuit of Figure 6.25, taking all resistances equal to one. Solve for the voltage V_i at each node, the current in each edge, and the total current I, in terms of the battery voltage V.

6.9 Notes

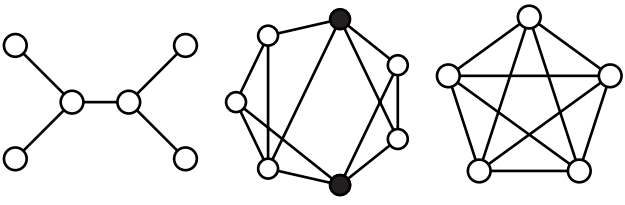
Read's tree encoding algorithm is from READ [142]. Prüfer sequences date back to 1918 – PRÜFER [137]. They are described in several books, including BONDY and MURTY [23]. The matrix-tree theorem is one of the most fundamental theorems in graph theory.

7

Connectivity

7.1 Introduction

Trees are the smallest connected graphs. For deleting any edge will disconnect a tree. The following figure shows three graphs in order of increasing connectivity.



$\kappa = 1$	$\kappa = 2$	$\kappa = 4$
$\kappa' = 1$	$\kappa' = 3$	$\kappa' = 4$

FIGURE 7.1
Three graphs with increasing connectivity

The second graph can be disconnected by deleting the two shaded vertices, but three edges must be deleted in order to disconnect it. The third graph is complete and cannot be disconnected by deleting any number of vertices. However, the deletion of four edges will do so. Thus, connectivity is measured by what must be deleted from a graph G in order to disconnect it. Because one can delete vertices or edges, there will be two measures of connectivity.

The *vertex-connectivity* of G is $\kappa(G)$, the minimum number of vertices whose deletion disconnects G. If G cannot be disconnected by deleting vertices, then $\kappa(G) = |G| - 1$. A disconnected graph requires the deletion of 0 vertices, so it has $\kappa = 0$. The complete graph has $\kappa(K_n) = n-1$. Hence, $\kappa(K_1) = 0$, but all other connected graphs have $\kappa \geq 1$. Any set of vertices whose deletion disconnects G is called a *separating set* or *vertex cut* of G.

The *edge-connectivity* of G is $\kappa'(G)$, the minimum number of edges whose deletion disconnects G. If G has no edges, then $\kappa'(G) = 0$. A disconnected graph does not need any edges to be deleted, and so it has $\kappa' = 0$. K_1 also has $\kappa' = 0$ because it has no edges, but all other connected graphs have $\kappa' \geq 1$.

The edge-connectivity is always at most $\delta(G)$, because deleting the δ edges incident on a vertex of minimum degree will disconnect G. The following inequality always holds.

Theorem 7.1. $\kappa \leq \kappa' \leq \delta$.

Proof. We know that $\kappa' \leq \delta$. We prove that $\kappa \leq \kappa'$ by induction on κ'. If $\kappa' = 0$, then either G has no edges, or else it is disconnected. In either case, $\kappa = 0$. Suppose that it is true whenever $\kappa' \leq m$, and consider $\kappa' = m + 1$. If $\kappa' = |G| - 1$, then $\delta = \kappa'$ and thus $\kappa \leq \kappa'$; so suppose that $\kappa' < |G| - 1$. Let $[S, \overline{S}]$ be an edge-cut containing $m + 1$ edges. Pick any edge $uv \in [S, \overline{S}]$ and form $H = G - uv$. Then $[S, \overline{S}] - uv$ is an edge-cut of H containing m edges, so $\kappa'(H) \leq m$. By the induction hypothesis, $\kappa(H) \leq m$. Let $U \subseteq V(H)$ be a minimum separating set of H. Then $|U| \leq m$, and $H - U$ consists of two or more components. We now want to put the edge uv back. Where does it go?

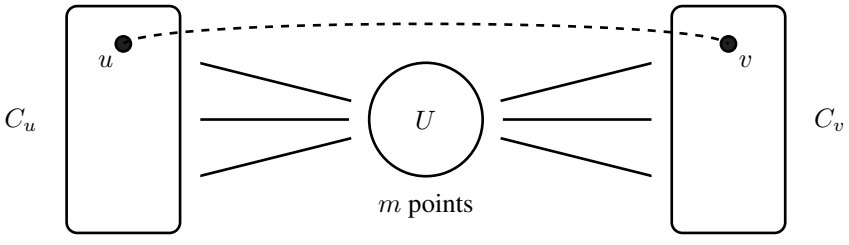


FIGURE 7.2
A minimum separating set of $\boldsymbol{H}$

If $H - U$ had three or more components, then U would also be a separating set of G, in which case $\kappa(G) \leq |U| = m$. If $H - U$ has exactly two components, C_u and C_v, containing u and v, respectively, then U will not be a separating set of G, for the edge uv will keep it connected. However, $\kappa'(G) < |G| - 1$, so that $m = \kappa' - 1 < |G| - 2$. Therefore, one of C_u and C_v contains two or more vertices, say C_u does. Then $U' = U \cup \{u\}$ is a separating set of G with $m + 1$ vertices, so that $\kappa(G) \leq \kappa'(G)$. By induction, the theorem is true for all values of κ'. □

Except for this inequality, the parameters κ, κ', and δ are free to vary considerably, as shown by CHARTRAND and HARARY [30]. For example, the graph of Figure 7.3 has $\kappa = 2$, $\kappa' = 3$, and $\delta = 4$.

Given any three non-negative integers a, b, and c satisfying $a \leq b \leq c$, we can easily make a graph with $\kappa = a$, $\kappa' = b$, and $\delta = c$, as illustrated in Figure 7.3. Take two complete graphs G' and G'', isomorphic to K_{c+1}. They have minimum degree $\delta = c$. Choose any a vertices $U' \subseteq V(G')$, and a corresponding vertices $U'' \subseteq V(G'')$. Join them up in pairs, using a edges. Then U' is a separating set of the graph, containing a vertices. Now add $b - a$ edges connecting G' to G'', such that

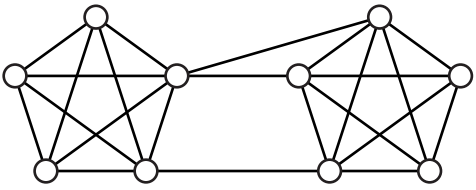


FIGURE 7.3
A graph with $\boldsymbol{\kappa = 2}$, $\boldsymbol{\kappa' = 3}$, and $\boldsymbol{\delta = 4}$

every edge added has one endpoint in U'. Clearly the graph constructed has $\kappa = a$, $\kappa' = b$, and $\delta = c$.

Exercises

7.1.1 Let G be connected and let $uv \in E(G)$. Prove that uv is in every spanning tree of G if and only if uv is a cut-edge of G.

7.1.2 Show that a connected graph with exactly two vertices that are not cut-vertices is a tree. *Hint*: Consider a spanning tree of G.

7.1.3 Prove that if G is a k-regular bipartite graph with $k > 1$ then G has no cut-edge.

7.1.4 Prove that if G is connected, with all even degrees, then $\omega(G - v) \leq \frac{1}{2}\text{DEG}(v)$, for any $v \in V(G)$, where $\omega(G)$ is the number of connected components of G.

7.1.5 Let G be a 3-regular graph.

(a) If $\kappa = 1$, show that $\kappa' = 1$.

(b) If $\kappa = 2$, show that $\kappa' = 2$.

Conclude that $\kappa = \kappa'$ for 3-regular graphs.

7.1.6 Let G be a 4-regular graph with $\kappa = 1$. Prove that $\kappa' = 2$.

7.1.7 Let $(d_1, d_2, \ldots, d_n)$, where $d_1 \leq d_2 \leq \ldots \leq d_n$, be the degree sequence of a graph G. Prove that if $d_j \geq j$, for $j = 1, 2, \ldots, n - 1 - d_n$, then G is connected.

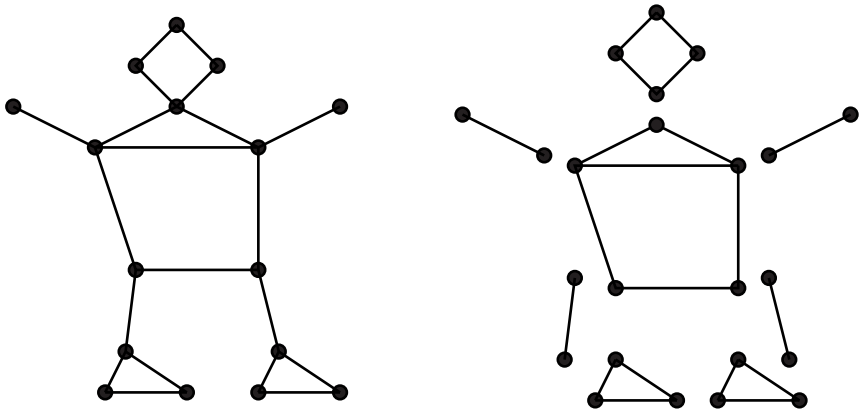
7.1.8 Give another proof that $\kappa \leq \kappa'$, as follows. Let $[S, \overline{S}]$ be a minimum edge-cut of G, containing κ' edges. Construct a set $U \subseteq S$ consisting of all vertices $u \in S$, such that there is an edge $uv \in [S, \overline{S}]$. Then $|U| \leq \kappa'$. If $U \neq S$, then U is a separating set of G with $\leq \kappa'$ vertices. Therefore $\kappa \leq \kappa'$. Show how to complete the proof when $U = S$.

7.2 Blocks

Any graph G with $\kappa \geq 1$ is connected. Consequently G is said to be 1-connected. Similarly, if $\kappa \geq 2$, then at least two vertices must be deleted in order to disconnect G, so G is said to be 2-connected. It is usually easier to determine a lower bound, such as $\kappa \geq 2$ or $\kappa \geq 3$, than to compute the exact value of κ. In general, G is said to be *k-connected* if $\kappa \geq k$, for some integer k.

If G is a disconnected graph, then its structure is determined by its components, that is, its maximal connected subgraphs. A component which is an isolated vertex will have $\kappa = 0$, but all other components will be 1-connected.

If a connected graph G has a cut-vertex v, then it is said to be *separable*, because deleting v separates G into two or more components. A separable graph has $\kappa = 1$, but it may have subgraphs which are 2-connected, just as a disconnected graph has connected subgraphs. We can then find the maximal non-separable subgraphs of G, just as we found the components of a disconnected graph. This is illustrated in Figure 7.4.



(a) (b)

FIGURE 7.4
A graph (a) and its blocks (b)

The maximal non-separable subgraphs of G are called the *blocks* of G. The graph illustrated in Figure 7.4 has eight blocks, held together by cut-vertices. Every separable graph will have two or more blocks. Any 2-connected graph is non-separable. However, K_2, a graph which consists of a single edge, is also non-separable, because it has no cut-vertex. Therefore every edge of G is a non-separable subgraph, and so will be contained in some maximal non-separable subgraph. Can an edge be contained in two distinct blocks? We first describe some properties of 2-connected graphs.

Notice that cycles are the smallest 2-connected graphs, because a connected graph with no cycle is a tree, which is not 2-connected. Any two vertices u and v on a cycle C divide C into two distinct paths with only the endpoints u and v in common. Paths which have only their endpoints in common are said to be *internally disjoint*; see Figure 7.5.

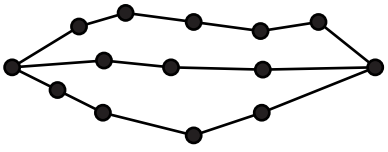


FIGURE 7.5
Three internally disjoint paths

Theorem 7.2. *A graph G with three or more vertices is 2-connected if and only if every pair of vertices is connected by at least two internally disjoint paths.*

Proof. Suppose that every pair of vertices of G is connected by at least two internally disjoint paths. If a vertex w is deleted, then every remaining pair of vertices is still connected by at least one path, so that w is not a cut-vertex. Therefore $\kappa \geq 2$.

Conversly suppose that G be 2-connected, and let $u, v \in V(G)$. We prove by induction on $\text{DIST}(u, v)$ that u and v are connected by two internally disjoint paths. If $\text{DIST}(u, v) = 1$, then $G - uv$ is still connected, because $\kappa' \geq \kappa \geq 2$. Therefore $G - uv$ contains a uv-path P, so that G has two uv-paths, P and uv. Suppose that the result holds when $\text{DIST}(u, v) \leq m$ and consider $\text{DIST}(u, v) = m + 1$. Let P be a uv-path of length $m + 1$ and let w be the last vertex before v on this path. Then $\text{DIST}(u, w) = m$, because P is a shortest path. By the induction hypothesis, G contains internally disjoint uw-paths P_w and Q_w.

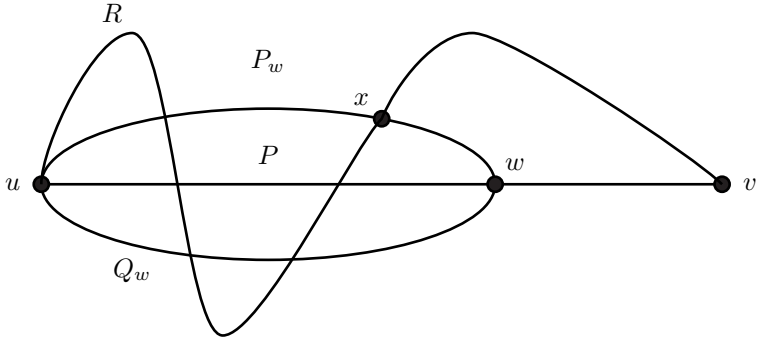


FIGURE 7.6
Internally disjoint paths $\boldsymbol{P_w}$ and $\boldsymbol{Q_w}$

Because G is 2-connected, $G - w$ is still connected, and so has a uv-path R. R has the endpoint u in common with both P_w and Q_w. Let x be the last vertex common to R and either of P_w or Q_w, say $x \in P_w$. Then $P_w[u, x]R[x, v]$ and $Q_w wv$ are two internally disjoint uv-paths. By induction, the result holds for all pairs u, v of vertices. □

So in a 2-connected graph, every pair u, v, of vertices are connected by at least two internally disjoint paths P and Q. Because P and Q together form a cycle, we know that *every pair of vertices lies on a cycle*. Another consequence of this theorem is that every pair of *edges* also lies on a cycle.

Corollary 7.3. *A graph G with three or more vertices is 2-connected if and only if every pair of edges lies on a cycle.*

Proof. Let G be 2-connected and pick edges $uv, xy \in E(G)$. Subdivide uv with a new vertex w, and xy with a new vertex z to get a graph G'. Now G has no cut-vertex, so neither does G'. By the previous theorem, w and z lie on a cycle in G', so that uv and xy lie on a cycle in G.

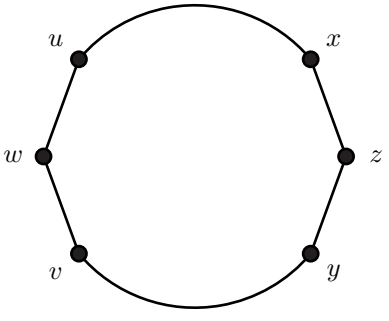


FIGURE 7.7
Two edges on a cycle

Conversly suppose now that every pair of edges lies on a cycle. Then every vertex has degree at least two, because no cycle could pass through an edge incident on a vertex of degree one. Choose any two vertices u and x. Choose any vertex v adjacent to u and a vertex y adjacent to x, such that $y \neq v$. Then the edges uv and xy must lie on a cycle C. Clearly C contains u and x, so that every pair u, x, of vertices lies on a cycle. It follows that G is 2-connected. □

Lemma 7.4. *Each edge uv of G is contained in a unique block.*

Proof. Let uv be an edge in a graph G, and let B be a maximal 2-connected subgraph of G containing uv. If B' is another maximal 2-connected subgraph containing uv, where $B \neq B'$, then choose any edge $xy \in B' \setminus B$. B' contains a cycle C containing both uv and xy, because B' is 2-connected. The subgraph $B \cup C$ is 2-connected, and

is larger than B, a contradiction. Therefore, each edge uv is contained in exactly one block of G. □

7.3 Finding the blocks of a graph

The first algorithm to find the blocks of a graph was discovered by READ [143]. It uses the fundamental cycles with respect to a spanning tree T. Because each edge of G is contained in a unique block B_{uv}, the algorithm begins by initializing B_{uv} to contain only uv and uses the merge-find data structure to construct the full blocks B_{uv}. For each edge $uv \notin T$, the fundamental cycle C_{uv} is found. Because C_{uv} is 2-connected, all its edges are in one block. So upon finding C_{uv}, we merge all the blocks B_{xy}, where $xy \in C_{uv}$, into one. Any spanning tree T can be used. If we choose a breadth-first tree, we have Algorithm 7.3.1.

Algorithm 7.3.1: BLOCKS(G)

comment: G is a connected graph
for each $uv \in E(G)$
  do initialize B_{uv} to contain uv
pick any vertex $x \in V(G)$
place x on *ScanQ*
repeat
 select u from head of *ScanQ*
 for each $v \longrightarrow u$
  do if $v \notin$ *ScanQ*
   then { **comment:** uv forms part of the spanning tree T
            add edge uv to T
            add v to *ScanQ*
   else { **comment:** uv creates a fundamental cycle
            construct C_{uv}
            for each edge $xy \in C_{uv}$
              do $B_{uv} \leftarrow B_{uv} \cup B_{xy}$
 advance *ScanQ*
until all vertices on *ScanQ* are processed
comment: each B_{uv} now consists of the unique block containing uv

Lemma 7.5. *At the beginning of each iteration of the repeat loop, each B_{uv} is either a single edge, or else is 2-connected.*

Proof. The proof is by induction on the number of iterations of the repeat loop. At the beginning of the first iteration it is certainly true. Suppose that it is true at the beginning of the k^{th} iteration. If the edge uv chosen forms part of the spanning tree T, it will also be true for the $(k+1)^{st}$ iteration, so suppose that uv creates a fundamental cycle C_{uv}. Each B_{xy} for which $xy \in C_{uv}$ is either a single edge, or else 2-connected. The new B_{uv} is formed by merging all the B_{xy} into one, say $B_{uv} = B_{x_1y_1} \cup B_{x_2y_2} \cup \cdots \cup B_{x_my_m}$. Pick any two edges $ab, cd \in B_{uv}$, say $ab \in B_{x_iy_i}$ and $cd \in B_{x_jy_j}$. We show that B_{uv} contains a cycle containing both ab and cd. If $ab, cd \in C_{uv}$, it is certainly true. Otherwise, notice that each $B_{x_iy_i}$ contains some edge of C_{uv}. Without loss of generality, we can suppose that the edges $x_ly_l \in C_{uv}$, for $l = 1, 2, \ldots, m$. Because $B_{x_iy_i}$ is 2-connected, it contains a cycle C_i containing both x_iy_i and ab, and $B_{x_jy_j}$ contains a cycle C_j containing both x_jy_j and cd. This is illustrated in Figure 7.8. Then $C_{uv} \oplus C_i \oplus C_j$ is a cycle contained in B_{uv} and containing ab and cd. By Corollary 7.3, the new B_{uv} is 2-connected. Therefore the result is true at the beginning of the $(k+1)^{st}$ iteration. By induction it is true for all iterations. □

Corollary 7.6. *Upon completion of Algorithm 7.3.1, each B_{uv} contains the unique block containing uv.*

Proof. By the previous lemma, each B_{uv} will either be a single edge, or else 2-connected. If B_{uv} is not the unique block B containing uv, then pick some edge $xy \in B - B_{uv}$. B contains a cycle C containing both uv and xy. By Theorem 5.3, C can be written in terms of fundamental cycles with respect to the spanning tree T constructed by Algorithm 7.3.1, $C = C_{u_1v_1} \oplus C_{u_2v_2} \oplus \cdots \oplus C_{u_mv_m}$. But each of the fundamental cycles $C_{u_iv_i}$ will have been processed by the algorithm, so that all edges of C are contained in one B_{uv}, a contradiction. Therefore, each B_{uv} consists of the unique block containing uv. □

Exercises

7.3.1 Given an edge uv which creates a fundamental cycle C_{uv}, describe how to find C_{uv} using the *Parent*$[\cdot]$ array created by the BFS.

7.3.2 Let $(d_1, d_2, \ldots, d_n)$, where $d_1 \leq d_2 \leq \ldots \leq d_n$, be the degree sequence of a graph G. Prove that if $d_j \geq j+1$, for $j = 1, 2, \ldots, n-1-d_{n-1}$, then G is 2-connected.

7.3.3 Program the BLOCKS() algorithm. One way to store the merge-find sets B_{uv} is as an n by n matrix *BlockRep*$[\cdot, \cdot]$. Then the two values $x \leftarrow$ *BlockRep*$[u, v]$ and $y \leftarrow$ *BlockRep*$[v, u]$ together define the edge xy representing uv. Another way is to assign a numbering to the edges, and use a linear array.

7.3.4 Try to estimate the complexity of the algorithm BLOCKS(). It is difficult to obtain a close estimate because it depends on the sum of the lengths of all $\varepsilon - n + 1$ fundamental cycles of G, where $n = |G|$.

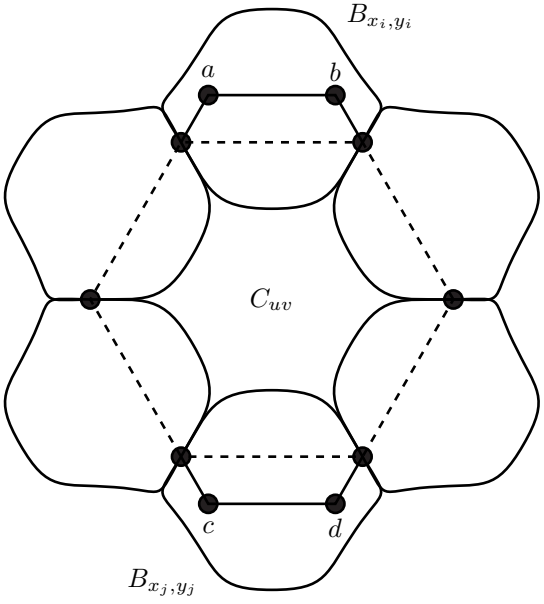


FIGURE 7.8
Merging the B_{xy}

7.3.5 *The Block-Cut-Vertex Tree.* (See HARARY [80].) Let G be a connected separable graph. Let $\mathcal{B}$ denote the set of blocks of G and $\mathcal{C}$ denote the set of cut-vertices. Each cut-vertex is contained in two or more blocks, and each block contains one or more cut-vertices. We can form a bipartite graph $\mathcal{BC}(G)$ with vertex-set $\mathcal{B} \cup \mathcal{C}$ by joining each $B \in \mathcal{B}$ to the cut-vertices $v \in \mathcal{C}$ that it contains.

(a) Show that $\mathcal{BC}(G)$ has no cycles, and consequently is a tree.

(b) In the block-cut-vertex tree $\mathcal{BC}(G)$, the degree of each $v \in \mathcal{C}$ is the number of blocks of G containing v. Denote this value by $b(v)$, for any vertex $v \in V(G)$. Show that

$$\sum_{v \in V(G)} b(v) - 1 = \sum_{v \in \mathcal{C}} b(v) - 1 = |\mathcal{B}| - 1,$$

so that the number of blocks of G is given by

$$|\mathcal{B}| = 1 + \sum_{v \in V(G)} b(v) - 1.$$

(c) Prove that every separable graph has at least two blocks which contain only one cut-vertex each.

7.4 The depth-first search

There is an easier, more efficient way of finding the blocks of a graph than using fundamental cycles. It was discovered by Hopcroft and Tarjan. It uses a *depth-first search* (*DFS*) to construct a spanning tree. With a depth-first search the fundamental cycles take a very simple form – essentially we find them for free, as they require no extra work. The basic form of the depth-first search follows. It is a recursive procedure, usually organized with several global variables initialized by the calling procedure. The example following uses a global counter *DFCount*, and two arrays *DFNum*$[v]$ and *Parent*$[v]$. Each vertex v is assigned a number *DFNum*$[v]$, being the order in which the DFS visits the vertices of G, and a value *Parent*$[v]$, being the vertex u from which DFS(v) was called. It is the parent of v in the rooted spanning tree constructed. Initially all *DFNum*$[\cdot]$ values are set to 0.

Algorithm 7.4.1: DFS(u)

comment: extend a depth-first search from vertex u

DFCount $\leftarrow$ *DFCount* $+ 1$
DFNum$[u]$ $\leftarrow$ *DFCount*
for each $v \longrightarrow u$
  do if *DFNum*$[v] = 0$
    then { **comment:** v is not visited yet
              add edge uv to the spanning tree
              Parent$[v] \leftarrow u$
              DFS(v)
    else { **comment:** uv creates a fundamental cycle

The calling procedure can be written as follows:

DFCount $\leftarrow 0$
for $u \leftarrow 1$ **to** n **do** *DFNum*$[u] \leftarrow 0$
select a staring vertex u
DFS(u)

Figure 7.9 shows a depth-first search in a graph. The numbering of the vertices shown is that of *DFNum*$[\cdot]$, the order in which the vertices are visited.

Notice that while visiting vertex u, DFS(v) is called immediately, for each $v \longrightarrow u$ discovered. This means that before returning to node u, all vertices that can be reached from v on paths that do not contain u will be visited; that is, all nodes of $G - u$ that are reachable from v will be visited. We state this important property as a lemma. For any vertex u, let $A(u)$ denote u, plus all ancestors of u in the depth-first tree, where an ancestor of u is either its parent, or any vertex on the unique spanning tree path from u to the root vertex.

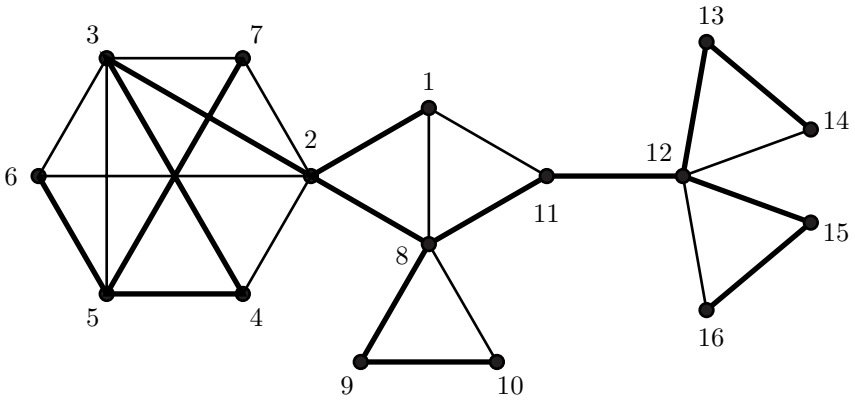


FIGURE 7.9
A depth-first search

Lemma 7.7. *Suppose that* $\mathrm{DFS}(v)$ *is called while visiting node* u*. Then* $\mathrm{DFS}(v)$ *visits every vertex in* $V(G) - A(u)$ *reachable from* v *before returning to node* u*.*

Proof. The statement

if *DFNum*$[u] = 0$ **then** $\cdots$

ensures that no vertex of $A(u)$ will be visited before returning to node u. To show that every vertex of $G - A(u)$ connected to v is visited, let w be a vertex of $V(G) - A(u)$, with $\textsc{Dist}(v, w) = k$. The proof is by induction on k. It is clear that all $w \longrightarrow v$ will be visited before returning to u, so that the statement is true when $k = 1$. If $k > 1$, let P be a vw-path of length k, and let $x \longrightarrow v$ be the first vertex of P. Now x will certainly be visited before returning to node u. When x is visited, either w will already have been visited, or else some $\mathrm{DFS}(y)$ called from node x will visit w before returning to v, because $\textsc{Dist}(x, w) = k - 1$. Therefore all vertices of $G - u$ connected to v will be visited before returning to u. □

This makes it possible to detect when u is a cut-vertex. It also means that a spanning tree constructed by a depth-first search will tend to have few, but long, branches. The following diagram shows the DF-tree constructed by the DFS in Figure 7.9.

Suppose that while visiting node u, a vertex $v \longrightarrow u$ with *DFNum*$[v] \neq 0$ is encountered. This means that v has already been visited, either previously to u, or as a descendant of u. While visiting v, the edge uv will have been encountered. Therefore, if v was visited previously, either $\mathrm{DFS}(u)$ was called from node v, so that *Parent*$[u] = v$, or else $\mathrm{DFS}(u)$ was called from some descendant w of v. So we can state the following fundamental property of depth-first searches:

Lemma 7.8. *Suppose that while visiting vertex* u *in a depth-first search, edge* uv *creating a fundamental cycle is encountered. Then either* v *is an ancestor of* u*, or else* u *is an ancestor of* v*.*

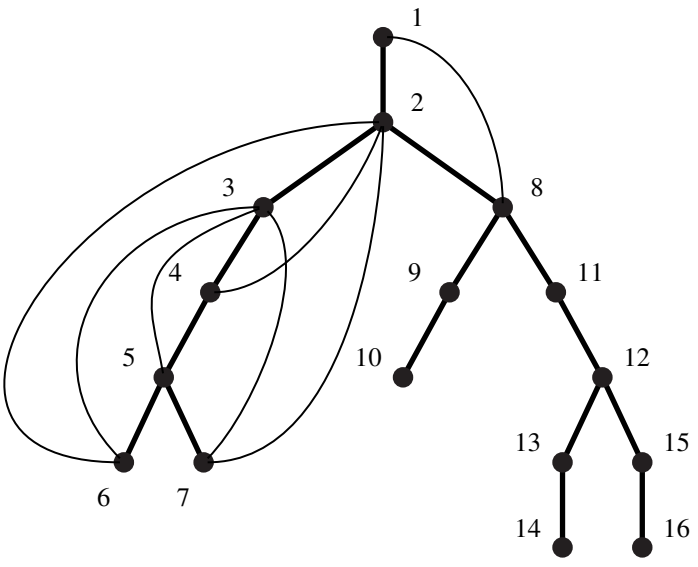


FIGURE 7.10
A depth-first tree with fronds

Edges which create fundamental cycles with respect to a depth-first spanning tree T are called *fronds* of T. Some of the fronds of the graph of Figure 7.9 are shown in Figure 7.10.

Now suppose that G is a separable graph with a cut-vertex v. v will occur somewhere in T, say that DFS(v) was called from node u, and that node v in turn calls DFS(w), where u and w are in different blocks. Thus, edges uv and vw do not lie on any cycle of G. This is illustrated in Figure 7.11. Consider any descendant x of w and a frond xy discovered while visiting node x, where y is an ancestor of x. Now y cannot be an ancestor of v, for then the fundamental cycle C_{xy} would contain both edges uv and vw, which is impossible. Therefore either $y = v$, or else y is a descendant of v. *So, for every frond xy, where x is a descendant of w, either $y = v$, or else y is a descendant of v.* We can recognize cut-vertices during a DFS in this way. Ancestors of v will have smaller *DFNum*$[\cdot]$ values than v, and descendants will have larger values. For each vertex v, we need to consider the endpoints y of all fronds xy such that x is a descendant of v.

DEFINITION 7.1: Given a depth-first search in a graph G. The *low-point* of a vertex v is *LowPt*$[v]$, the minimum value of *DFNum*$[y]$, for all edges vy and all fronds xy, where x is a descendant of v.

We can now easily modify DFS(u) to compute low-points and find cut-vertices. In addition to the global variables *DFCount*, *DFNum*$[\cdot]$, and *Parent*$[\cdot]$, the algorithm keeps a stack of edges. Every edge encountered by the algorithm is placed on the

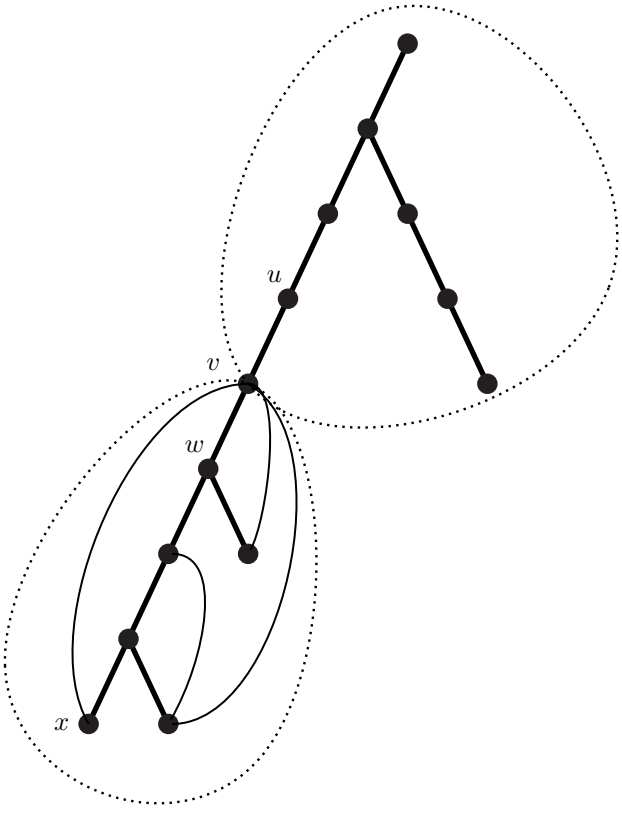


FIGURE 7.11
A cut-vertex v in a DF-tree

stack. When a cut-vertex is discovered, edges on the top of the stack will be the edges of a block of G.

The procedure DFSEARCH(u) considers all $v \longrightarrow u$. If v has been previously visited, there are two possibilities, either $v = \mathit{Parent}[u]$, or else uv is a frond. When uv is a frond, there are also two possibilities, either u is an ancestor of v, or v is an ancestor of u. These two cases are shown in Figure 7.12. The algorithm only needs those fronds uv for which v is an ancestor of u in order to compute $\mathit{LowPt}[u]$.

Algorithm 7.4.2: DFBLOCKS(G)

```
comment: { DFS to find the blocks of a connected graph G,
         { on n vertices.

procedure DFSEARCH(u)
  comment: extend a depth-first search from u
  DFCount ← DFCount + 1
  DFNum[u] ← DFCount
  for each v ⟶ u do
  { if DFNum[v] = 0
    { then { comment: { v is not visited yet,
           {          { add uv to the spanning tree
           { Parent[v] ← u
           { stack uv
           { LowPt[v] ← DFNum[u]        "initial value"
           { DFSEARCH(v)
           { comment: LowPt[v] is now known
           { if LowPt[v] = DFNum[u]
           {   then { comment: u is a cut-vertex
           {        { unstack all edges up to, and including, uv
           {   else { comment: otherwise LowPt[v] < DFNum[u]
           {        { if LowPt[v] < LowPt[u]
           {        {   then LowPt[u] ← LowPt[v]
    { else { comment: v has already been visited
           { if v ≠ Parent[u]  then
           { { comment: uv is a frond, it creates a fund. cycle
           { { if DFNum[v] < DFNum[u]
           {   then { comment: v is an ancestor of u
           {        { stack uv
           {        { if DFNum[v] < LowPt[u]
           {        {   then LowPt[u] ← DFNum[v]

main
  DFCount ← 0
  for u ← 1 to n do DFNum[u] ← 0
  select a vertex u
  LowPt[u] ← 1
  DFSEARCH(u)
```

The next thing to notice is that the *LowPt*$[u]$ is correctly computed. For if u is a leaf-node of the search tree, then all edges uv are fronds, and v is an ancestor of u. The algorithm computes *LowPt*$[u]$ as the minimum *DFNum*$[v]$, for all such v. There-

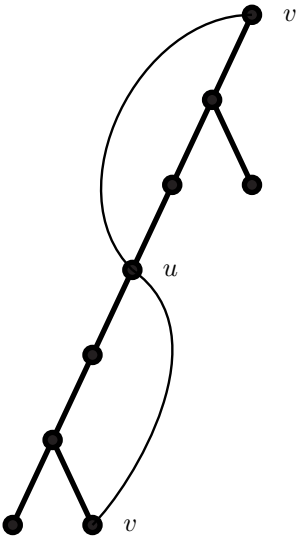


FIGURE 7.12
Two kinds of fronds

fore, if u is a leaf-node, the low-point is correctly computed. We can now use induction on the depth of the recursion. If u is not a leaf-node, then some DFSEARCH(v) will be called from node u. The depth of the recursion from v will be less than that from u, so that we can assume that DFSEARCH(v) will correctly compute *LowPt*$[v]$. Upon returning from this recursive call, *LowPt*$[v]$ is compared with the current value of *LowPt*$[u]$, and the minimum is taken, for every unsearched $v \longrightarrow u$. Therefore, after visiting node u, *LowPt*$[u]$ will always have the correct value.

So far the algorithm computes low-points and uses them to find the cut-vertices of G. We still need to find the edges in each block. While visiting node u, all new edges uv are stacked. If it is discovered that *LowPt*$[v]$ = *DFNum*$[u]$; so that u is a cut-vertex, then the edges in the block containing uv are all those edges on the stack up to, and including, uv.

Theorem 7.9. *Each time that*

$$\textit{LowPt}[v] = \textit{DFNum}[u]$$

occurs in DFSEARCH(u)*, the block containing* uv *consists of those edges on the stack up to and including* uv*.*

Proof. Let B_{uv} denote the block containing uv. The proof is by induction on the number of times that *LowPt*$[v]$ = *DFNum*$[u]$ occurs. Consider the first time it occurs. DFSEARCH(v) has just been called, while visiting node u. Edge uv has been placed on the stack. DFSEARCH(v) constructs the branch of the search tree at u containing v. By Lemma 7.7, this contains all vertices of $G - u$ connected to v. Call this

set of vertices B. By the definition of the low-point, there are no fronds joining v or any descendant of v to any ancestor of u. So u separates B from the rest of the graph. Therefore $B_{uv} \subseteq G[B \cup \{u\}]$.

Suppose now that B contained a cut-vertex w. No leaf-node of the DF-tree can be a cut-vertex, so some DFSEARCH(x) is called while visiting node w. It will visit all nodes of $G - w$ connected to x. Upon returning to node w, it would find that $\mathit{LowPt}[x] = \mathit{DFNum}[w]$, which is impossible, because this occurs for the first time at node u. Therefore B_{uv} consists of exactly those edges encountered while performing DFSEARCH(v); that is, those on the stack.

Upon returning from DFS(v) and detecting that $\mathit{LowPt}[v] = \mathit{DFNum}[u]$, all edges on the stack will be unstacked up to, and including, uv. This is equivalent to removing all edges of B_{uv} from the graph. The remainder of the DFS now continues to work on $G - B$. Now B_{uv} is an *end-block* of G (i.e., it has at most one cut-vertex) for u is the first vertex for which a block is detected. If G is 2-connected, then $G = B_{uv}$, and the algorithm is finished. Otherwise, $G - B$ is connected, and consists of the remaining blocks of G. It has one less block than G, so that each time $\mathit{LowPt}[v] = \mathit{DFNum}[u]$ occurs in $G - B$, the edges on the stack will contain another block. By induction, the algorithm finds all blocks of G. □

Each time the condition $\mathit{LowPt}[v] = \mathit{DFNum}[u]$ occurs, the algorithm has found the edges of a block of G. In this case, u will usually be a cut-vertex of G. The exception is when u is the root of the DF-tree, because u has no ancestors in the tree. Exercise 7.4.3 shows how to deal with this situation.

7.4.1 Complexity

The complexity of DFBLOCKS() is very easy to work out. For every $u \in V(G)$, all $v \longrightarrow u$ are considered. This takes

$$\sum_u \text{DEG}(u) = 2\varepsilon$$

steps. Each edge is stacked and later unstacked, and a number of comparisons are performed in order to compute the low-points. So the complexity is $O(\varepsilon)$.

Exercises

7.4.1 Can the spanning tree shown in the graph illustrated in Figure 7.13 be a DF-tree, with the given root-node? If so, assign a DF-numbering to the vertices.

7.4.2 Program the DFBLOCKS() algorithm to find all the blocks of a connected graph G. Print a list of the edges in each block. Choose the starting vertex to be sometimes a cut-vertex, sometimes not.

7.4.3 Modify your program to print also a list of the cut-vertices of G, by storing them on an array. A vertex u is a cut-vertex if $\mathit{LowPt}[v] = \mathit{DFNum}[u]$ occurs while visiting edge uv at node u. However, if u is the root-node

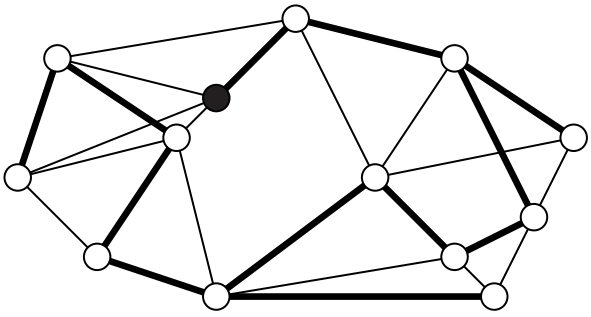


FIGURE 7.13
Is this a DF spanning tree?

of the DF-tree, then it will also satisfy this condition, even when G is 2-connected. Find a way to modify the algorithm so that it correctly determines when the root-node is a cut-vertex.

7.4.4 A separable graph has $\kappa = 1$, but can be decomposed into blocks, its maximal non-separable subgraphs. A tree is the only separable graph which does not have a 2-connected subgraph, so that every block of a tree is an edge. Suppose that G has $\kappa = 2$. In general, G may have 3-connected subgraphs. Characterize the class of 2-connected graphs which do not have any 3-connected subgraphs.

7.4.5 Let G be 3-connected. Prove that every pair of vertices is connected by at least three internally disjoint paths.

7.4.6 Let G have $\kappa = 2$, and consider the problem of finding all separating pairs $\{u, v\}$ of G using a DFS. Prove that for every separating pair $\{u, v\}$, one of u and v is an ancestor of the other in any DF-tree. Refer to Figure 7.14.

7.4.7 Suppose that deleting $\{u, v\}$ separates G into two or more components. Let G_1 denote one component and G_2 the rest of G. Show that there are two possible ways in which a DFS may visit u and v, as illustrated in Figure 7.14. Devise a DFS which will find all pairs $\{u, v\}$ which are of the first type. (*Hint*: You will need *LowPt2*$[v]$, the second low-point of v. Define it and prove that it works.)

7.5 Sections and modules

Suppose that G is a disconnected graph, with connected components $G_1, G_2, \ldots, G_k$. Algorithms applied to G can usually be applied successively to each G_i. The complement $\overline{G}$ is connected, but each subgraph $\overline{G}_i$ is connected to all other $\overline{G}_j$'s

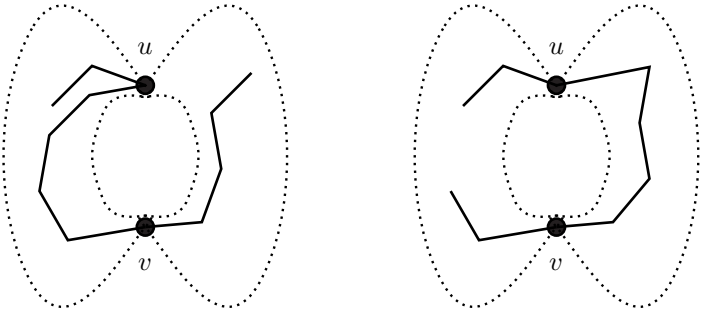


FIGURE 7.14
DFS with separating set $\{u, v\}$

by all possible edges. Therefore algorithms applied to $\overline{G}$ can also be applied successively to each $\overline{G}_i$. We say that each subset $V(G_i) \subseteq V(G)$ is a *section* of G and of $\overline{G}$. *Sections* are defined recursively.

DEFINITION 7.2: Given a graph G, with complement $\overline{G}$, and vertex set V.

1. V is a section of G and of $\overline{G}$.
2. If $U \subseteq V$ induces a connected component of G or of $\overline{G}$, then U is a section of G and $\overline{G}$.
3. Let U be a section of G and $\overline{G}$. Then any section of $G[U]$ is also a section of G and $\overline{G}$.

If $G = K_n$, then $\overline{G}$ has n connected components, so that each vertex is a section. Consider $G = C_4$. Then $\overline{G}$ consists of $2K_2$. The vertices of each K_2 form a section. And each K_2 itself consists of two sections. Thus, C_4 can be completely reduced by its sections. In general, the set of sections of G and $\overline{G}$ form a decomposition tree. The tree of sections of C_4 is shown in Figure 7.15.

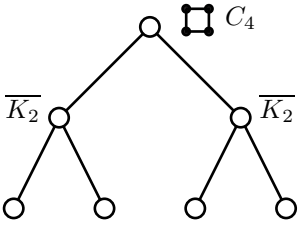


FIGURE 7.15
The sections of C_4

Let G denote the complement of $K_4 \cup K_{3,3}$. This is a connected graph, whose complement has two connected components, K_4 and $K_{3,3}$, each of which is a section of G. K_4 has four sections consisting of single vertices. And the complement of $K_{3,3}$ is disconnected, consisting of two triangles K_3. Each K_3 in turn has three sections consisting of single vertices. The complete tree is shown in Figure 7.16. We see that the decomposition tree of G is required in order to combine the sections and retrieve G.

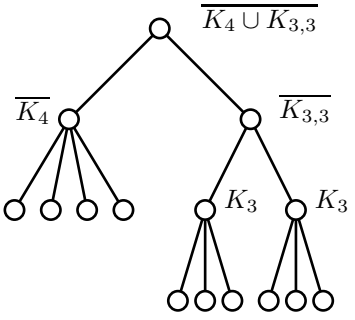


FIGURE 7.16
The sections of the complement of $\boldsymbol{K_4 \cup K_{3,3}}$

An algorithm for finding the sections of a graph G derives from Definition 7.2. We start by finding the connected components $G_1, G_2, \ldots, G_k$ of G. If $k > 1$, then each component is a section. The algorithm then proceeds recursively, taking in turn the complement of each G_i. But if G is connected, the algorithm finds the connected components of $\overline{G}$ and then proceeds recursively. The decomposition tree can be simultaneously constructed.

A *proper section* of G is a section U, where $U \neq V(G)$. Clearly a connected graph G whose complement is also connected has no proper sections. Let v be a vertex in such a G. We now take an arbitrary graph H, and alter G by replacing v with H. Any edge uv of G is replaced by all possible edges uw, where $w \in V(H)$. The set $V(H)$ now has the property that if $ux \in E(G)$ is any edge where $u \notin V(H)$ and $x \in V(H)$, then $uw \in E(G)$, for all $w \in V(H)$. Such a set is called a *module* or *autonomous set* of G.

DEFINITION 7.3: Let G be a graph and let $U \subseteq V(G)$ be a set of vertices of G. Then U is a *module* of G if it has the property: if $ux \in E(G)$ where $u \notin U$ and $x \in U$, then $uw \in E(G)$, for all $w \in U$.

It is clear from the definition that $V(G)$ is a module of G. Similarly every $\{v\}$, where $v \in V(G)$ is a module, and $\emptyset$ is a module of G. And every section of G is also a module of G. Every module of G is also a module of $\overline{G}$.

Lemma 7.10. *Let U_1 and U_2 be modules of G. Then*
1. $U_1 \cap U_2$ is a module of G.

2. If $U_1 \cap U_2 \neq \emptyset$ *then* $U_1 \cup U_2$ *is a module of* G.
3. If $U_1 \cap \overline{U}_2 \neq \emptyset$, *then* $\overline{U}_1 \cap U_2$ *is a module of* G.

Modules provide a further reduction for graphs with no proper sections. If U is a module of G, then because every $\{v\}$ is also a module of G, we see that $V(G)$ can always be written as a disjoint union of modules, one of which is U. If the set U is "shrunk" to a single vertex, a smaller graph is obtained. G is then said to be *decomposable*. A graph is *indecomposable* if every module is either the empty set, a singleton, or all of $V(G)$.

Exercises

7.5.1 Let U be a section of G. Show that $\overline{U}$ is also a section of G.

7.5.2 Let U be a module of G. Show that $\overline{U}$ need not be a module of G.

7.5.3 Prove Lemma 7.10.

7.5.4 Let G be a graph with n vertices and no proper sections. Can there be a module of $n-2$ vertices ? Can there be a module of $n-3$ vertices ?

7.5.5 Program the algorithm to find the sections and decomposition tree of a graph G.

7.5.6 Develop an algorithm to find the modules of a graph with no proper sections.

7.6 Notes

The example of Figure 6.3 is based on HARARY [80]. Read's algorithm to find the blocks of a graph is from READ [143]. The depth-first search algorithm is from HOPCROFT and TARJAN [89]. See also TARJAN [166]. Hopcroft and Tarjan's application of the depth-first search to find the blocks of a graph was a great breakthrough in algorithmic graph theory. The depth-first search has because been applied to solve a number of difficult problems, such as determining whether a graph is planar in linear time, and finding the 3-connected components of a graph.

Algorithms for finding the connectivity and edge-connectivity of a graph are described in Chapter 8. An excellent reference for connectivity is TUTTE [177], which includes a detailed description of the 3-connected components of a graph. A depth-first seach algorithm to find the 3-connected components of a graph can be found in HOPCROFT and TARJAN [90]. This algorithm has been carefully analyzed and improved by GUTWENGER and MUTZEL [77].

Sections were introduced by CORNEIL, LERCHS, BURLINGHAM [37], and also by GOLDBERG [71], who used them as a reduction in a graph isomorphism algorithm. Modules are presented in a very general setting by SCHMERL and TROTTER

in [155]. See also ILLE [91]. Modules are also known as *autonomous sets* or *intervals*. A fast algorithm to find the modules of a graph appears in BUER and MÖHRING [26].

8

Graphs and Symmetry

8.1 Groups

Groups are necessary for understanding of the *symmetry* of a graph, and for constructing graphs with prescribed symmetry. We outline here the main concepts of group theory which are used in this book. They are used for constructing various graphs. Consider the graph of the cube, illustrated in Figure 8.1. Given a 3-dimensional cube, it could be rotated in several ways, and the cube would still look the same. A rotation of the cube can be represented as a permutation of the vertices. For example, if we imagine an axis of rotation through the front and rear faces of the cube in Figure 8.1, then a clockwise rotation through $\pi/2$ could be represented by the permutation $(1,3,5,7)(2,4,6,8)$, where the parentheses indicate that the vertices move in two cycles of four vertices each: 1 maps to 3, which maps to 5, which maps to 7, which maps to 1, etc.

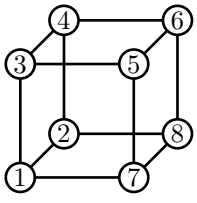


FIGURE 8.1
The graph of a cube

An *automorphism* of a graph G is a permutation of $V(G)$, (i.e., a one-to-one mapping of $V(G)$ onto $V(G)$), that maps $E(G)$ onto itself. Given a permutation θ, we write u^θ for the image of u under θ. (The functional notation would be $\theta(u)$, however, the exponential notation turns out to be more convenient for permutations and groups.) If $uv \in E(G)$ then θ maps edge uv onto $(uv)^\theta = u^\theta v^\theta \in E(G)$. Because an automorphism θ is bijective, given an edge $xy \in E(G)$, there can be only one edge uv such that $(uv)^\theta = xy$. It follows that θ also maps non-edges to non-edges.

Given two automorphsims, θ and ϕ, they can be composed to obtain $\theta\phi$: vertex u maps to u^θ under θ, then to $u^{\theta\phi}$ under ϕ. So the mapping $\theta\phi$ means "first θ, then ϕ".

The composition of permutations is also called the *product* of the permutations, and is evaluated from left to right.

The set of all automorphisms of a graph G is denoted $\text{AUT}(G)$. Observe that $\text{AUT}(G)$ has the following properties:

1. (identity) There is an *identity* permutation, denoted $\text{I} \in \text{AUT}(G)$, with the property that $\text{I}\theta = \theta\text{I}$, for all $\theta \in \text{AUT}(G)$.
2. (inverses) Given any $\theta \in \text{AUT}(G)$, the inverse θ^{-1} is also in $\text{AUT}(G)$.
3. (closure) If $\theta, \phi \in \text{AUT}(G)$, then the product $\theta\phi \in \text{AUT}(G)$.
4. (associativity) If $\theta, \phi, \rho \in \text{AUT}(G)$, then $(\theta\phi)\rho = \theta(\phi\rho)$.

These properties are abstracted and become the axioms of a *group*: a *group* is a set of elements which have a product defined, satisfying the above properties. Two common types of groups are permutation groups, and groups of invertible matrices with the operation of matrix multiplication. There are numerous possibilities for a group's "product" operation – it could be composition of permutations, multiplication of matrices, addition or multiplication of numbers, etc. The *symmetric group* S_n consists of all $n!$ permutations of n objects. It is easy to see that if a graph G has n vertices, and $\text{AUT}(G) = S_n$, then either $G = K_n$, or $G = \overline{K}_n$. The *order* of a group Γ is $|\Gamma|$, its number of elements. We will always use I for the identity element in a group — *all* groups will have an element named I.

Many graphs with interesting properties can be constructed from groups. We outline some of the key ideas required.

Let Γ be a group, say $\Gamma = \text{AUT}(G)$, for some graph G. A subset $K \subseteq \Gamma$ that is also a group is called a *subgroup*. Clearly every subgroup of Γ contains the identity element I. One way to form a subgroup of Γ is to choose an element $\theta \in \Gamma$, and construct its powers $\theta, \theta^2, \theta^3, \ldots$, where θ^2 means $\theta\theta$, etc. Eventually we find that $\theta^k = \text{I}$, for some non-negative integer k, called the *period* of θ. We write $\langle\theta\rangle$ for the subgroup *generated* by θ, i.e., all powers of θ. Another common way to form a subgroup of Γ is to choose a vertex $v \in V(G)$, and consider all elements that map v to itself. It is easy to verify that this defines a subgroup, called the *stabilizer* subgroup of v, denoted Γ_v. Similarly, given a subset $U \subseteq V(G)$, the set of elements of Γ that map U to U is a subgroup, the *set-wise stabilizer* of U.

Given a subgroup $K \subseteq \Gamma$, and an element $\theta \in \Gamma$, the right *coset* $K\theta$ consists of the set of elements of K, followed by θ, namely $K\theta = \{\phi\theta \mid \phi \in K\}$. It is easy to see that two cosets $K\theta_1$ and $K\theta_2$ are either identical (if $\theta_1\theta_2^{-1} \in K$), or disjoint (if $\theta_1\theta_2^{-1} \notin K$). It follows that Γ can be written as a disjoint union of distinct cosets, $\Gamma = K\theta_1 + K\theta_2 + \ldots + K\theta_m$, where it is customary to write the disjoint union of cosets as a sum. The elements $\theta_1, \ldots, \theta_m$ are called *coset representatives*. Therefore $|\Gamma| = m|K|$, i.e., the *order of a subgroup divides the order of the group*. This is called *Lagrange's theorem*. Left cosets θK are defined similarly, giving $\Gamma = \phi_1 K + \phi_2 K + \ldots + \phi_m K$, although we will mostly use right cosets.

We now choose a subgroup K to be Γ_v, the stabilizer of a vertex v, and write $\Gamma = K\theta_1 + K\theta_2 + \ldots + K\theta_m$. Every element in the coset $K\theta_i$ maps v to v^{θ_i}.

Hence, the vertices $\{v^{\theta_1}, v^{\theta_2}, \ldots, v^{\theta_m}\}$ consists of all vertices that Γ maps v to. This is called the *orbit* of v. It is denoted by v^{Γ}, or by $\mathrm{orb}(v)$. We have

Lemma 8.1. $|\mathrm{orb}(v)| \cdot |\Gamma_v| = |\Gamma|$

We now have sufficient techniques to find $\mathrm{AUT}(G)$ when G is the graph of the cube in Figure 8.1. Choose $v = 1$. We see that $\theta = (1,3,5,7)(2,4,6,8) \in \mathrm{AUT}(G)$ and that $\phi = (1,2)(3,4)(5,6)(7,8) \in \mathrm{AUT}(G)$. Therefore $|\mathrm{orb}(v)| = 8$, from which $|\mathrm{AUT}(G)| = 8 \cdot |\Gamma_1|$. Referring to the diagram, we see that Γ_1 must map $\{2,3,7\}$ to $\{2,3,7\}$, it must map $\{4,5,8\}$ to $\{4,5,8\}$, and it must map 6 to 6. Clearly $\alpha = (2,3,7)(4,5,8) \in \Gamma_1$. This is a rotation of the cube along the axis through vertices 1 and 6. We could also fix the edges 12 and 56, and perform an interchange $\gamma = (3,7)(4,8)$. This is an automorphism of the graph of the cube, although a physical cube cannot be transformed like this. We conclude that $|\Gamma_1| = 6$, so that $|\Gamma| = 48$. Furthermore, we have found *generators* for Γ, that is, every element of Γ can be written as a product of θ, ϕ, α, and γ, in some order. We write $\langle \theta, \phi, \alpha, \gamma \rangle$ for the group generated by these permutations.

Suppose now that a permutation group Γ acting on a set $V = \{1, 2, \ldots, n\}$ has been given. Usually generators for Γ will be known. For example, Γ could be given by the generators $\theta = (1,3,5,7)(2,4,6,8)$ and $\phi = (1,3,4,6,8,7)(2,5)$. One of the easiest ways to construct a graph G for which Γ acts as a group of automorphisms is to use algorithm SYMMETRICGRAPH.

procedure SYMMETRICGRAPH(Γ, n, uv)
comment: A group Γ acting on $V = \{1, 2, \ldots, n\}$ is given by a set of generators
comment: Construct a graph G containing edge uv such that $\Gamma \subseteq \mathrm{AUT}(G)$
comment: *ScanQ*$[1, 2, \ldots]$ is a queue of edges

$k \leftarrow 1$
$M \leftarrow 1$
ScanQ$[1] \leftarrow uv$
$E(G) \leftarrow \emptyset$
add uv to $E(G)$
while $k \leq M$
do
- $uv \leftarrow$ *ScanQ*$[k]$
- **for each** generator θ of Γ
  - **do**
    - $xy \leftarrow (uv)^{\theta}$
    - **if** $uv \notin E(G)$
      - **then**
        - add uv to $E(G)$
        - $M \leftarrow M + 1$
        - *ScanQ*$[M] \leftarrow xy$
- $k \leftarrow k + 1$

An edge uv is given, and the generators of Γ are applied successively to uv to determine all edges to which elements of Γ can map uv. If algorithm SYMMETRICGRAPH is applied using the group Γ generated by $\theta = (1,3,5,7)(2,4,6,8)$ and $\phi = (1,3,4,6,8,7)(2,5)$ on the starting edge $uv = 13$, then the graph of the cube of Figure 8.1 is constructed. Clearly every generator of Γ is an automorphism of the graph G constructed, so that $\Gamma \subseteq \text{AUT}(G)$.

DEFINITION 8.1: A permutation group Γ with just one orbit is said to be *transitive*. A graph G such that $\text{AUT}(G)$ is transitive is said to be *vertex transitive*. If the action of $\text{AUT}(G)$ on $E(G)$ is transitive, then G is said to be *edge transitive*.

The algorithm SYMMETRICGRAPH builds G from an edge uv, hence, it is clear that G is edge-transitive. The algorithm could easily be extended to start with several different edges.

Various algorithms are available for finding generators for $\text{AUT}(G)$, given a graph G. Several references for constructing and programming these algorithms are [107, 111, 124]. Graph isomorphism software can also be downloaded from the internet.

8.2 Cayley graphs

Let Γ be an arbitrary finite group. We construct an edge-colored digraph G from Γ as follows. The vertices of G are the elements of Γ. Choose a set of elements $S = \{\theta_1, \theta_2, \ldots, \theta_k\}$ that generate Γ. The edge colors will be $1, 2, \ldots, k$. Vertex $\gamma \in V(G) = \Gamma$ is joined by a directed edge of color i to $\gamma\theta_i$. The result is called a *Cayley digraph* of Γ, denoted $\overrightarrow{\text{Cay}}(\Gamma, S)$, For example, let Γ be the group generated by the permutations $\theta_1 = (1,2,3,4)$ and $\theta_2 = (2,4)$. By finding all products of θ_1, θ_2 we find that $|\Gamma| = 8$. The elements of Γ are:

$$\begin{array}{llll} \text{I}, & \theta_1 = (1,2,3,4), & \theta_1^2 = (1,3)(2,4), & \theta_1^3 = (1,4,3,2) \\ \theta_2 = (2,4), & \theta_2\theta_1 = (1,2)(3,4), & \theta_2\theta_1^2 = (1,3), & \theta_2\theta_1^3 = (1,4)(2,3) \end{array}$$

The resulting Cayley digraph is shown in Figure 8.2. The solid edges represent θ_1. The dotted edges represent θ_2, they are actually pairs of edges directed in opposite directions, because θ_2 has period two.

Consider the structure of a Cayley color-digraph. It is a graphical representation of the group structure of Γ. Choose any vertex γ. There is an arc from γ to $\gamma\theta_1$, and then to $\gamma\theta_1^2$, then to $\gamma\theta_1^3$, etc., finally returning to γ. That is, each vertex of G is part of a directed cycle of color i, whose length is the period of θ_i.

An automorphism of an edge-colored digraph is a digraph-automorphism that maps each arc to an arc of the same color.

Theorem 8.2. *Let Γ be a group with generators $S = \{\theta_1, \ldots, \theta_m\}$. Let $G = \overrightarrow{\text{Cay}}(\Gamma, S)$. Every element of $\text{AUT}(G)$ corresponds to an element of Γ.*

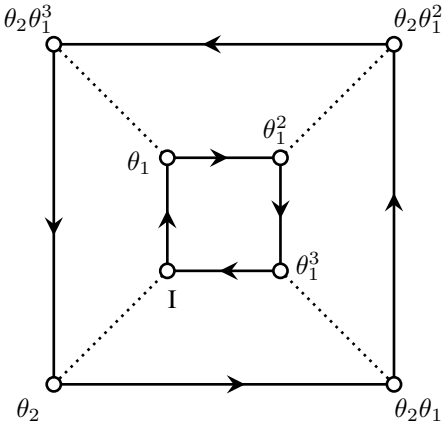


FIGURE 8.2
A Cayley color digraph for $\boldsymbol{\Gamma = \langle \theta_1, \theta_2 \rangle}$

Proof. Let $\gamma \in \Gamma$, and let α be a vertex of G. Left multiplication by γ gives a vertex $\gamma\alpha$ of G. Now G contains an arc $(\alpha, \alpha\theta_i)$ for each color i. It also contains an arc $(\gamma\alpha, \gamma\alpha\theta_i)$ for each color i. Therefore left multiplication by any element $\gamma \in \Gamma$ is an automorphism of G.

Now the permutations $\theta_1, \ldots, \theta_m$ generate Γ, so that α can be mapped to any $\beta \in V(G)$ by some γ. Thus $\text{AUT}(G)$ is transitive on $V(G)$. We now look at the stabilizer $\text{AUT}(G)_\alpha$. Vertex α has exactly one out-arc of color i, for each i, and exactly one in-arc of color i, for each i. Therefore, the stabilizer has order one, so that all automorphisms of $G = \overrightarrow{\text{Cay}}(\Gamma, S)$ derive from Γ by left multiplication. □

Another property of the Cayley color-digraph is that for each θ_i, the vertices of G are distributed uniformly into cycles whose length is the period of θ_i. In Figure 8.2, θ_1 produces cycles of length 4, and θ_2 produces cycles of length 2. Let θ_1^* denote the permutation of $V(G)$ that derives from right multiplication by θ_1. Let θ_2^* denote the corresponding permutation for θ_2. For every product of θ_1 and θ_2 in Γ, there is a corresponding product of θ_1^* and θ_2^*. We see that θ_1^* and θ_2^* generate a group Γ^*, it is called the *right regular representation* of Γ. Here "right" refers to right multiplication, and "regular" refers to the fact that each permutation consists of cycles of uniform length.

Given any edge-colored digraph, there corresponds a graph obtained by ignoring the edge-colors and direction of the arcs.

DEFINITION 8.2: Let Γ be a group with generators $S = \{\theta_1, \ldots, \theta_m\}$. The *Cayley graph* $\text{Cay}(\Gamma, S)$ is formed from the Cayley color-digraph by ignoring the edge-colors and arc-directions.

The Cayley graph for Figure 8.2 is the graph of the cube. We have already discovered that its automorphism group has order 48, whereas $|\Gamma| = 8$. Thus, a Cayley

graph can have more automorphisms than those arising from the group used to construct it.

DEFINITION 8.3: A Cayley graph $G = \mathrm{Cay}(\Gamma, S)$ for a group Γ is a *graphical regular representation* (GRR), if $|\mathrm{AUT}(G)| = |\Gamma|$.

Much work on graphical regular representations has been done by Godsil, Watkins, and Imrich.

8.3 Coset diagrams

A Cayley graph is constructed by allowing the elements of a group Γ to act on Γ by right multiplication. Each element of Γ can be viewed as a coset of the subgroup $\{\mathrm{I}\}$. If K is any subgroup of Γ, it is easy to see that the elements of Γ also permute the right cosets of K by right multiplication. Let $\Gamma = K\gamma_1 + \ldots + K\gamma_m$ be a decomposition of Γ into right cosets. Let $\theta_1, \ldots \theta_k$ be generators for Γ. The *coset diagram* $\Gamma \bmod K$ is an edge-colored directed graph whose vertices are the cosets of K, with an edge of color i from vertex $K\gamma_j$ to vertex $K\gamma_j\theta_i = K\gamma_\ell$, for some ℓ.

As an example, consider the group Γ of Figure 8.2 with generators $\theta_1 = (1, 2, 3, 4)$ and $\theta_2 = (2, 4)$. Let $K = \langle\theta_2\rangle$, the group generated by θ_2. Clearly $|K| = 2$, so that there are four cosets. The cosets can be listed as K, $K\theta_1$, $K\theta_1^2$, and $K\theta_1^3$. The coset diagram $\Gamma \bmod K$ is shown in Figure 8.3. Notice that a coset digraph usually contains loops. For if a generator $\theta_i \in K$, then $K\theta_i = K$, producing a loop of color i.

Walks in the coset diagram correspond to products of generators of Γ. For example, if we start at the coset labeled K in Figure 8.3 and successively follow the edges corresponding to θ_1, then θ_2, then θ_1, we obtain the product $\theta_1\theta_2\theta_1$. Starting at coset K, this gives the result $K\theta_1\theta_2\theta_1 = K$, so that $\theta_1\theta_2\theta_1 \in K$. Because $\theta_1, \ldots, \theta_k$ were chosen as *generators* for Γ, it follows that the coset digraph is connected. There is directed path from any vertex (coset) to any other. Coset diagrams are the basis of many algorithms for permutation groups. See [27, 158].

Each θ_i now gives rise to a permutation θ_i^* of the cosets, namely θ_i^* maps coset $K\gamma_j$ to $K\gamma_j\theta_i = K\gamma_\ell$, for some ℓ. Once again we have a representation of Γ by permutations, $\Gamma^* = \langle\theta_1^*, \theta_2^*\rangle$. In the example of Figure 8.3, the original permutations are reproduced.

As a larger example, let Γ be the permutation group generated by the permutations $\theta = (1, 2, 3, 4, 5, 6)$, and $\phi = (1, 2)$. Γ is the symmetric group S_6 consisting of all permutations of six objects. $|\Gamma| = 6! = 720$. Let K be the subgroup generated by $(1, 4, 2, 5, 3, 6)$ and $(1, 6, 2, 5, 3, 4)$, which has order 36. There are $720/36 = 20$ cosets. The coset diagram $\Gamma \bmod K$ is shown in Figure 8.4. The solid lines represent θ^*, and the dotted lines represent ϕ^*. We see that θ^* has two cycles of length 6, two cycles of length 3, and two fixed points. And ϕ^* consists of ten cycles of length two. If we number the cosets $1, 2, \ldots, 20$, we have a permutation representa-

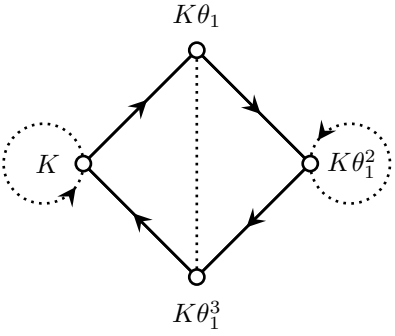


FIGURE 8.3
A coset diagram $\boldsymbol{\Gamma \bmod K}$

tion $S_6^* = \langle \theta^*, \phi^* \rangle$ of S_6 acting on 20 cosets. It can be checked that this representation of S_6 also has order 720, although it is not always the case that a representation of a group Γ by cosets of a subgroup has the same order as Γ.

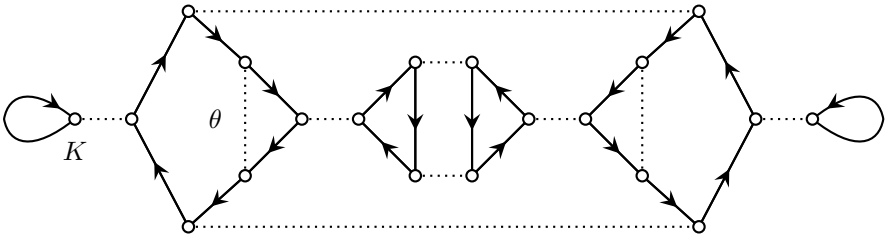


FIGURE 8.4
A coset diagram $\boldsymbol{\Gamma \bmod K}$

The Cayley graph construction, and the algorithm SYMMETRICGRAPH() both construct a graph with a given group acting as a group of automorphisms. Sometimes it is convenient to combine them as follows.

Beginning with a group Γ and generators $\{\theta_1, \ldots, \theta_k\}$, and a subgroup K, we construct a Cayley graph G for Γ. A decomposition of Γ into cosets of K is then found, $\{K\gamma_1, \ldots, K\gamma_m\}$. New vertices $u_1, \ldots, u_m$ are then added to G, such that u_i is adjacent to those vertices of G that belong to coset $K\gamma_i$. The result is a graph for which Γ permutes the vertices of the Cayley graph, and simultaneously permutes the vertices representing the cosets of K. An algorithm for finding a coset decomposition of a permutation group can be found in [27].

8.3.1 Double cosets

We are now ready to make *double coset graphs*. Let Γ be a group with a subgroup K, and let $K\gamma_1, \ldots, K\gamma_m$ be a coset decomposition of Γ. Let $\phi_1, \ldots, \phi_k$ be generators for K. Consider the elements of K, acting on these cosets by right multiplication. Coset $K\gamma_i$ will be mapped to $K\gamma_i\phi_j$ by generator ϕ_j. As ϕ_j is varied, all cosets of K contained in $K\gamma_i K$ will form one orbit. $K\gamma_i K$ is called a *double coset* of K. It is constructed by allowing K to act on its own cosets in Γ. Notice that if $\gamma = \mathrm{I}$, then $K\gamma K = K$, so that K is also a double coset, it forms an orbit of one coset. Most double cosets $K\gamma_i K$ will contain several cosets of K. Notice that the graph of all cosets acted on by the generators of K is not connected, because $\phi_1, \ldots, \phi_k$ do not generate all of Γ. Each connected component is a double coset of K. We state this as a lemma, from which it is obvious that double cosets $K\gamma_1 K$ and $K\gamma_2 K$ are either identical or disjoint.

Lemma 8.3. *The double cosets $K\gamma K$ consist of the connected components of the graph of right cosets of K in Γ, with K acting by right multiplication on its own cosets.*

The double cosets produce interesting graphs. Let a group Γ with a subgroup K be given. We construct a graph G whose vertices $V(G)$ are the cosets of K in Γ. Choose a coset $K\gamma$, for some $\gamma \in \Gamma$, where $\gamma \neq \mathrm{I}$. Suppose that $\{K, K\gamma\}$ is chosen as an edge of G. We want Γ to act as a group of symmetries on G. So for each $\theta \in \Gamma$, vertex $K\theta$ of G will be adjacent to vertex $K\gamma\theta$. Now Γ acts transitively on the cosets of K, so that G is a vertex-transitive graph. We now look at the stabilizer subgroup Γ_K. It consists of those elements $\phi \in \Gamma$ such that $K\phi = K$, i.e., $\Gamma_K = K$. Now when Γ_K acts on coset $K\gamma$, it produces the double coset $K\gamma K$. Therefore K is adjacent in G to $K\alpha_1, K\alpha_2, \ldots, K\alpha_m$, the right cosets of K contained in $K\gamma K$. G is called the *double coset graph*, denoted $\Gamma(K, \gamma)$.

There can also be other edges incident on K. Given $\theta \in \Gamma$, there is an edge $\{K\theta, K\gamma\theta\}$ in G. It is possible that $K\gamma\theta = K$. This implies that $\gamma\theta \in K$, so that $\theta \in \gamma^{-1}K$, from which we find that $K\theta$ is a right coset contained in the double coset $K\gamma^{-1}K$. Now it can be that $K\gamma^{-1}K = K\gamma K$. In this case, the graph G is regular of degree m. But if $K\gamma^{-1}K \neq K\gamma K$, then G will be regular of degree $2m$. We summarize these facts.

DEFINITION 8.4: Let Γ be a group with a subgroup K. Choose a coset $K\gamma$, where $\gamma \neq \mathrm{I}$. The *double coset graph* $\Gamma(K, \gamma)$ has vertex set equal to the right cosets of K in Γ, and edges $\{K\theta, K\gamma\theta\}$, where θ is any element of Γ.

Theorem 8.4. *The double coset graph $\Gamma(K, \gamma)$ is vertex transitive. Let m be the number of cosets of K in $K\gamma K$. Then $\Gamma(K, \gamma)$ is regular of degree m or $2m$.*

We can say more about the number m of right cosets of K contained in $K\gamma K$. It is easy to see that if K is a subgroup of Γ, then so is $\gamma^{-1}K\gamma$. Let $H = K \cap \gamma^{-1}K\gamma$. Then H is a subgroup of K. Consider a coset $K\gamma\alpha_i$ in $K\gamma K$, where $\alpha_i \in K$, and suppose that ϕ is some element of $H\alpha_i$. Then $\phi \in \gamma^{-1}K\gamma\alpha_i$, so that $\gamma\phi \in K\gamma\alpha_i$.

It follows that $K\gamma\phi = K\gamma\alpha_i$. But if $\phi \notin H\alpha_i$, then $K\gamma\phi$ and $K\gamma\alpha_i$ are different cosets. It follows that m is the number of cosets of $H = K \cap \gamma^{-1}K\gamma$ in K. A special case occurs when $\gamma^{-1}K\gamma = K$. In this case there is just one coset $K\gamma$ in $K\gamma K$, so that the double coset graph will have degree one or two.

When $K\gamma^{-1}K = K\gamma K$, the graph $\Gamma(K, \gamma)$ has degree m. Furthermore, the stabilizer Γ_K is transitive on the vertices adjacent to K. But when $K\gamma^{-1}K \neq K\gamma K$, the degree is $2m$, and the stabilizer is not transitive on the adjacent cosets, instead they divide into two orbits.

As an example of this technique, let Γ be the group of permutations acting on 20 points, with generators $\theta = (2, 5, 6)(3, 10, 11)(4, 15, 7)(8, 14, 16)(9, 20, 12)(13, 19, 17)$ and $\phi = (1, 5, 4, 3, 2)(6, 10, 9, 8, 7)(11, 15, 14, 13, 12)(16, 20, 19, 18, 17)$. Γ is the group of automorphisms of the dodecahedron. It has order 60. The subgroup K generated by $(1, 13)(2, 9)(3, 4)(5, 8)(6, 18)(7, 14)(10, 12)(11, 19)(15, 17)(16, 20)$ and $(1, 18)(2, 17)(3, 16(4, 20)(5, 19)(6, 13)(7, 12)(8, 11)(9, 15)(10, 14)$ is chosen. It has order 4, so that there are 15 cosets. The double coset $K\gamma K$ is chosen, where $\gamma = (2, 5, 6)(3, 10, 11)(4, 15, 7)(8, 14, 16)(9, 20, 12)(13, 19, 17)$. It contains four cosets of K. The resulting double coset graph $\Gamma(K, \gamma)$ is shown in Figure 8.5. The same group Γ acts on this graph of 15 vertices, however it is now permuting cosets. If we find the automorphism group of the graph, we find it has order 120. We can say that this graph is a graphical representation of the group Γ. In general, a group Γ can have many different representations by various graphs. For a group Γ has many subgroups K, of various orders, each with a number of possible double cosets. The result is many different graphs acted on by the same Γ. The graphs constructed can be very different, yet they are related through their automorphism groups.

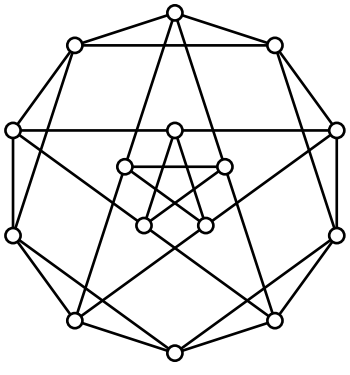


FIGURE 8.5
A double coset graph related to the dodecahedron.

The double cosets $K\gamma K$ are a special case of a more general double coset. If K and H are two subgroups of Γ, then $K\gamma H$ is the set of right cosets of K of the form

$K\alpha$, where $\alpha \in H$. Similarly, it is also the set of left cosets of H of the form βH, where $\beta \in K$.

8.4 Conjugation, Sylow subgroups

In the previous section, we started with a group Γ and a subgroup K, and allowed the generators of either Γ or K to act on the cosets of K. This produced a number of permutation representations of Γ, together with families of vertex-transitive graphs. There is another way to use the generators of either Γ or K to construct permutation representations. Given elements $\gamma, \theta \in \Gamma$, the *conjugate* of γ by θ is $\theta^{-1}\gamma\theta$. It is convenient to denote this also by γ^θ. We see that θ induces a permutation of Γ whereby every $\gamma \in \Gamma$ is mapped to γ^θ, its conjugate under θ. Similarly, if K is a subgroup of Γ, then $\theta^{-1}K\theta = K^\theta = \{\phi^\theta \mid \phi \in K\}$ is the conjugate of K by θ. The familiar notions of orbits and stabilizers produce interesting concepts using conjugation.

Let K be a subgroup of the permutation group Γ, acting on the points $V = \{1, 2, \ldots, n\}$, and let $\gamma, \theta \in \Gamma$.

Lemma 8.5. *K^θ is a subgroup of Γ. Let $\Delta \subseteq V$ be an orbit of K. Then Δ^θ is an orbit of K^θ.*

Proof. K^θ maps Δ^θ to $(\theta^{-1}\Delta\theta)^{\theta^{-1}K\theta} = \theta^{-1}\Delta^K\theta = \Delta^\theta$. □

Lemma 8.6. *Let $(u_1, u_2, \ldots, u_m)$ be a cycle of γ. Then $(u_1^\theta, u_2^\theta, \ldots, u_m^\theta)$ is a cycle of γ^θ.*

Proof. Point u_i^θ is mapped by γ^θ to $u_i^{\theta\theta^{-1}\gamma\theta} = u_{i+1}^\theta$. □

We see that conjugate permutations have an identical cycle structure. The stabilizer of γ under conjugation is $\{\theta \mid \gamma^\theta = \gamma\}$. This is a subgroup of Γ. We have $\gamma^\theta = \gamma$ if and only if $\gamma\theta = \theta\gamma$. So the stabilizer consists of all elements of Γ that commute with γ. This is called the *centralizer* of γ, and denoted $C(\gamma)$. By Lemma 8.1 we have

$$|\mathrm{orb}(\gamma)| \cdot |C(\gamma)| = |\Gamma|$$

The orbit of γ under conjugation consists of those permutations of Γ with the same cycle structure as γ to which γ can be conjugated. In the case when $\Gamma = S_n$, this will be *all* permutations with the same cycle structure as γ, called a *conjugacy class* of S_n.

The stabilizer of a subgroup K under conjugation is $\{\theta \mid K^\theta = K\}$. This is called the *normalizer* of K, and denoted $N(K)$. By Lemma 8.1 we have

$$|\mathrm{orb}(K)| \cdot |N(K)| = |\Gamma|$$

So the number of subgroups conjugate to K is determined by the normalizer. It is a subgroup of Γ, and we can write the decomposition of Γ into right cosets of $N(K)$

as $\Gamma = N(K)\gamma_1 + \ldots + N(K)\gamma_m$. Then it is easy to see that $K^{N(K)\gamma_i} = K^{\gamma_i}$, so that $K^{\gamma_1}, K^{\gamma_2}, \ldots, K^{\gamma_m}$ are all the conjugates of K in Γ. Conjugation of K by elements of Γ induces a permutation of the subgroups $K^{\gamma_1}, K^{\gamma_2}, \ldots, K^{\gamma_m}$, such that $K^{\gamma} = K^{\theta}$ if and only if γ and θ are in the same coset of $N(K)$.

When $N(K) = \Gamma$, only K is conjugate to itself, i.e., $K^{\theta} = K$, for all $\theta \in \Gamma$. K is then said to be a *normal subgroup*, denoted $K \triangleleft \Gamma$. In this case the right and left cosets of K are identical, $\theta^{-1}K\theta = K$, so that $\theta K = K\theta$. A reduced group Γ/K can then be constructed whose elements are the cosets of K:

$$\Gamma/K = \{K\alpha_1, K\alpha_2, \ldots, K\alpha_k\}$$

with the product rule $K\alpha_i K\alpha_j = K\alpha_i\alpha_j$. Because $K\alpha_i K\alpha_j = \alpha_i K K\alpha_j = \alpha_i K\alpha_j = K\alpha_i\alpha_j$, the product is well defined, and independent of the coset representatives chosen. We see that $|\Gamma/K| = |\Gamma|/|K|$. The identity in Γ/K is $\mathrm{I} = K$. The group Γ/K is called a *quotient group* or *factor group* of Γ.

It is clear that K is a normal subgroup of its normalizer $N(K)$, as $K^{\gamma} = K$, for all $\gamma \in N(K)$. $N(K)$ is the largest subgroup of Γ in which K is a normal subgroup.

Suppose now that p is a prime dividing $|\Gamma|$. Let p^k be the highest power of p in $|\Gamma|$, and consider the case when there is a subgroup K of order p^k. Such a subgroup K is called a *Sylow p-subgroup*. We will soon see that there always is such a subgroup. Γ acts by conjugation on K, so let $K^{\gamma_1}, K^{\gamma_2}, \ldots, K^{\gamma_m}$ be all the subgroups of Γ that are conjugate to K. Then $m = |\Gamma|/|N(K)|$. Now K is a subgroup of $N(K)$, so that $|N(K)| = p^k\ell$, where $\ell \geq 1$ and $|\Gamma| = p^k\ell m$.

Theorem 8.7. *Let K be a Sylow p-subgroup of Γ. The number of conjugates of K by elements of Γ is congruent to one, mod p.*

Proof. We allow K to act on the subgroups $K^{\gamma_1}, K^{\gamma_2}, \ldots, K^{\gamma_m}$ by conjugation, and find its orbits. Without loss of generality, let $K^{\gamma_1} = K$. Clearly $K^{\phi} = K$, for any $\phi \in K$, so that the subgroup K^{γ_1} is an orbit containing just one subgroup under conjugation. If $N(K) = \Gamma$, then $K \triangleleft \Gamma$, so that there is only one conjugate, and we are done.

Otherwise there are $m > 1$ conjugates of K in Γ, corresponding to the m cosets of $N(K)$. Coset $N(K)\gamma_i$ conjugates K to K^{γ_i}. Now $K \triangleleft N(K)$, so that $N(K)/K$ is a group of order $|N(K)|/|K| = \ell$, where $\gcd(\ell, p) = 1$. Therefore $N(K)/K$ has no elements whose period is divisible by p. It follows that the period of any element of $\theta \in N(K)$, where $\theta \notin K$ is not divisible by p. Consequently K is the only one of the conjugates K^{γ_i} that is contained in $N(K)$. It is easy to see that $N(K^{\gamma_i}) = N(K)^{\gamma_i}$. Now if $K \neq K^{\gamma_i}$, then K contains an element $\theta \notin K^{\gamma_i}$, so that $\theta \notin N(K^{\gamma_i})$. It follows that θ conjugates K^{γ_i} to some K^{γ_j} where $i \neq j$. That is, when K acts by conjugation on $K^{\gamma_1}, K^{\gamma_2}, \ldots, K^{\gamma_m}$, there is only one orbit of length one, namely K. All other orbits have length dividing $|K|$, so that all other orbits have length, a non-zero power of p. Therefore $m \equiv 1 \pmod{p}$.

□

We would now like to show that a Sylow p-subgroup always exists. First note that if $|\Gamma|$ is divisible by p, then Γ must contain an element γ whose period is divisible

by p. Let the period of γ be $p^i\ell$, where $\gcd(p,\ell) = 1$ and $i \geq 1$. Then the period of γ^ℓ is p^i. It follows that Γ always contains an element of order p^i. If $i > 1$, then it contains elements of order $p, p^2, \ldots, p^i$, and therefore it also contains subgroups of these orders.

Theorem 8.8. *Let p^k be the highest power of the prime p dividing $|\Gamma| = p^k m$, where $k \geq 1$, and let K be a subgroup of Γ of order p^i, where $1 \leq i < k$. Then K is contained in a subgroup of order p^{i+1}.*

Proof. We have already seen that Γ contains a subgroup of order p. The proof is by induction on k. Consider $N(K)$. Let $|N(K)| = p^j\ell$ where $j \geq i$ and $\gcd(p,\ell) = 1$. If $j > i$, then $N(K)/K$ is a group of order $p^{j-i}\ell$, and therefore contains an element of order p. Let $K\gamma \in N(K)/K$ be an element of order p. Then $\{K, K\gamma, \ldots, K\gamma^{p-1}\}$ is a subgroup of $N(K)/K$ of order p. If we add γ as a generator to K, we obtain a subgroup of order p^{i+1}.

Otherwise $j = i$. There are $p^{k-i}m/\ell$ conjugates of K in Γ. When K acts by conjugation on them, K is an orbit of size one. The number of conjugates in any non-singleton orbit is divisible by p. But p divides the number of conjugates, so that there is another singleton orbit, say $K^\gamma \neq K$. Therefore $K \subseteq N(K^\gamma)$. It follows that $N(K)$ contains an element not in K, but whose period is a power of p, so that $j > i$, a contradiction.

□

It follows from Theorem 8.8 that Γ contains a chain of subgroups $K_0, K_1, \ldots, K_k$ where $|K_i| = p^i$ and each K_i is a subgroup of K_{i+1}, up to K_k. Notice that if $|K| = p^i$ and K is a subgroup of a group H of order p^{i+1}, then there are p cosets of K in H. Let H act by right multiplication on these cosets. The resulting group is a transitive permutation group acting on p objects, whose order is a power of p. The stabilizer of any object must fix all p objects, for if the stabilizer has a non-singleton orbit, the order of H will not be a power of p. It follows that the order is p, giving $K \triangleleft H$. So Γ contains a chain of subgroups, $K_0 \triangleleft K_1 \triangleleft \ldots \triangleleft K_k$.

Sylow subgroups are a source of many interesting vertex-transitive graphs. Given a group Γ, it is convenient to choose a subgroup K as a Sylow p-subgroup for some prime p. Then a double coset graph for K can be an interesting graph. For example, the subgroup K of order 4 used in Figure 8.5 is a Sylow 2-subgroup of the automorphism group of the dodecahedron. Algorithms for finding Sylow subgroups of a permutation group Γ can be found in [27, 158].

8.5 Homomorphisms

We have already seen some examples of homomorphisms. Given a group Γ and a subgroup K, the action of Γ on the cosets of K gives a representation Γ^* of Γ, such that to each $\gamma \in \Gamma$, there corresponds an $h(\gamma) = \gamma^* \in \Gamma^*$ which permutes the right

cosets of K by right multiplication. This mapping $h : \Gamma \to \Gamma^*$ has the property that $h(\gamma\theta) = h(\gamma)h(\theta)$, for all $\gamma, \theta \in \Gamma$. A mapping h from a group Γ to a group Ψ with this property is called a *homomorphism*. It is easy to see that $h(\mathrm{I}) = \mathrm{I}$ and that $h(\gamma^{-1}) = h(\gamma)^{-1}$ for any homomorphism h.

When K is a normal subgroup of Γ, there is a natural homomorphism h from Γ to $\Gamma/K = \{K\alpha_1, K\alpha_2, \ldots, K\alpha_k\}$, such that every $\gamma \in K\alpha_i$ is mapped to $h(\gamma) = K\alpha_i$. Then every $\gamma \in K$ is mapped to $\mathrm{I} = K$.

DEFINITION 8.5: Let $h : \Gamma \to \Psi$ be a homomorphism. The *kernel* of h is $\mathrm{Ker}(h) = \{\gamma \in \Gamma \mid h(\gamma) = \mathrm{I} \in \Psi\}$.

Thus, when K is a normal subgroup, the kernel of this natural homomorphism is K. It is easy to prove that $\mathrm{Ker}(h)$ is always a normal subgroup of Γ.

Another example of a homomorphism is when Γ permutes its elements or subgroups by conjugation. Let $\theta \in \Gamma$ and consider the permutation group induced by conjugation, whereby every $\gamma \in \Gamma$ is mapped to γ^θ. Let the permutation of Γ induced by conjugation be denoted θ', so that $h(\theta) = \theta'$ is a homomorphism. The kernel of h is $Z = \{\theta \in \Gamma \mid \theta\gamma = \gamma\theta, \forall\theta \in \Gamma\}$. Z is called the *center* of Γ.

Another example of a homomorphism is when Γ permutes the right cosets of a subgroup K by right multiplication. Every $\theta \in \Gamma$ is mapped to a permutation of the cosets. The kernel of this homomorphism is $\{\theta \in \Gamma \mid K\gamma\theta = K\gamma, \forall\gamma \in \Gamma\}$. This is the largest subgroup of K that is normal in Γ. It is called the *core* of K. Algorithms for finding the center and core can be found in [27, 158].

8.6 Primitivity and block systems

Let Γ be a transitive permutation group acting on a set $V = \{1, 2, \ldots, n\}$. It will often be the case that Γ will be an automorphism group of a graph with vertex set V. Γ induces an action on the set of all pairs $\{u, v\}$, where $u, v \in V$. Let Γ^2 denote this induced action on $\binom{V}{2}$. It can have several orbits, even though Γ is transitive. For example, let Γ be the automorphism group of the graph of the cube, generated by $\theta = (1,3,5,7)(2,4,6,8)$ and $\phi = (1,3,4,6,8,7)(2,5)$. Each orbit of Γ^2 is the edge-set of a graph. For example, one of the orbits of Γ^2 is the graph of the cube, shown in Figure 8.1. There are two other orbits, shown in Figure 8.6.

DEFINITION 8.6: Let Γ be a transitive permutation group. If every orbit of Γ^2 is a connected graph, then Γ is said to be *primitive*. Otherwise Γ is *imprimitive*.

We see that the automorphism group of the graph of the cube is an imprimitive group. When an orbit of Γ^2 is a disconnected graph G, each automorphism of G must map a connected component of G to a connected component. Because $\Gamma \subseteq \mathrm{AUT}(G)$, this property also applies to Γ. Therefore Γ must permute the connected components. Given a disconnected orbit G, the collection of vertex-sets of the connected components of G is called a *block system* for Γ.

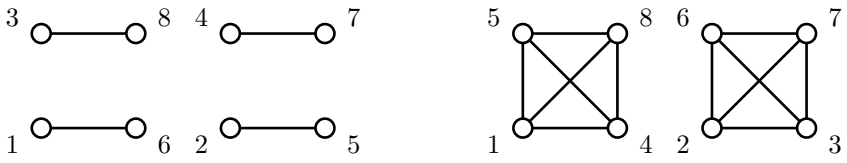


FIGURE 8.6
Two orbits of $\mathbf{\Gamma^2}$, finding a block system

DEFINITION 8.7: Let Γ be an imprimitive transitive permutation group acting on a set V. A *block system* for Γ consists of a collection of disjoint subsets $V_1, V_2, \ldots, V_m \subseteq V$, whose union is V, such that Γ induces a permutation of the V_i.

Two block systems for the above Γ are evident from Figure 8.6. One is $\{1,6\},\{2,5\},\{3,8\},\{4,7\}$. The other is $\{1,4,5,8\},\{2,3,6,7\}$. It is easy to see how to find a block system for a group Γ — we simply find the orbits of Γ^2, and test if they are connected graphs.

Let $V_1, V_2, \ldots, V_m$ be a block system for Γ. Let H be the subgroup that maps each V_i to V_i, i.e., $H = \{\gamma \in \Gamma \mid V_i^\gamma = V_i, \forall i\}$. Take any $\theta \in \Gamma$ and consider H^θ. Choose any i and suppose that $V_j^\theta = V_i$. Then H^θ maps V_i to $V_i^{\theta^{-1}H\theta} = V_j^{H\theta} = V_j^\theta = V_i$. It follows that $H^\theta = H$, for all $\theta \in \Gamma$, so that H is a normal subgroup. Therefore when there is a block system, Γ can be factored into a smaller group acting only on the blocks. If G is any graph for which $\Gamma = \text{AUT}(G)$, then G can also be factored into a *block graph*. The vertices of each block V_i are identified into a single vertex. Two blocks V_i and V_j are adjacent in the block graph if some $u_i \in V_i$ is adjacent to some $u_j \in V_j$ in G. The two block systems for the group of the cube of Figure 8.6 give two block graphs. The one with four blocks consists of a 4-cycle, the other consists of a single edge.

When the automorphism group of a vertex-transitive graph has a block system, the resulting block graph is also vertex-transitive. This is one way of reducing a vertex-transitive graph to another smaller graph, while preserving much of the symmetry.

Exercises

8.6.1 Use the stabilizer subgroup to find generators for $\text{AUT}(G)$ when G is the Petersen graph. Also find $|\text{AUT}(G)|$.

8.6.2 For the group Γ of Figure 8.2, find all elements of the right regular representation.

8.6.3 The permutations $\theta_1 = (1,2,3,4)$ and $\theta_2 = (1,2)$ generate S_4, which has order 24. Construct the Cayley color-digraph for generators θ_1, θ_2.

8.6.4 The permutations $\theta_1 = (1,7,5,2)(3,6,8,4)$ and

$\theta_2 = (1,6)(2,3)(4,8)(5,7)$ generate a group Γ of order 24 acting on 8 points. Construct the Cayley color-digraph for generators θ_1, θ_2.

8.6.5 Construct a coset diagram and permutation representation for the group Γ of the previous question, with subgroup K generated by θ_2.

8.6.6 Using the generators θ and ϕ, assign cosets to the vertices of the coset digraph of Figure 8.4.

8.6.7 Generators for the automorphism group of the dodecahedron are given near Figure 8.4. Determine whether it is a primitive group. Find a block system if there is one, and find its block graph.

8.6.8 Let θ be a permutation of $V = \{1, 2, \ldots, n\}$, stored as an array: $\theta[i]$ is the vertex that i is mapped to. Write a loop to find θ^{-1}.

8.6.9 Let θ and γ be permutations of $V = \{1, 2, \ldots, n\}$, stored as arrays. Write a loop to find γ^θ.

8.7 Self-complementary graphs

A graph G is said to be *self-complementary* if $G \cong \overline{G}$. In Chapter 1 we saw that a self-complementary graph on n vertices must satisfy $n \equiv 0$ or 1, (mod 4). Suppose that G is a self-complementary graph, and let $\theta : V(G) \to V(G)$ be an isomorphism mapping G to $\overline{G}$. Choose any edge $uv \in E(G)$. Then $(uv)^\theta \in E(\overline{G})$, so that $(uv)^\theta \notin E(G)$. Consequently $(uv)^{\theta^2} \notin E(\overline{G})$, so that $(uv)^{\theta^2} \in E(G)$, and so forth. This gives:

Lemma 8.9. *Let G be a self-complementary graph, and let θ be an isomorphism mapping G to $\overline{G}$. Then $\theta^2 \in \mathrm{AUT}(G)$.*

Such a θ is called a *complementing permutation* for G. It follows that $uv, (uv)^{\theta^2}, (uv)^{\theta^4}, \ldots$ is a sequence of edges of G, and that $(uv)^\theta, (uv)^{\theta^3}, \ldots$ is a sequence of edges of $\overline{G}$. Now θ is a permutation of $V(G)$, so that the vertices of G are decomposed into cycles of θ. Let $(u_1, u_2, \ldots, u_m)$ be a cycle of θ, so that $(u_i)^\theta = u_{i+1}$, if $i < m$, and $(u_m)^\theta = u_1$. Suppose that $m \geq 2$. The pair u_1u_2 is either an edge of G or of $\overline{G}$, so that successive pairs $u_1u_2, u_2u_3, \ldots$ are alternately edges of G and of $\overline{G}$. It follows that if $m > 1$, then m is even. We then consider the pair $u_1u_{m/2+1}$. It is mapped by $\theta^{m/2}$ to $u_{m/2+1}u_1$, which is in G. This requires that $m/2$ be even, so that $m \equiv 0$, (mod 4).

Theorem 8.10. *Let G be a self-complementary graph, and let θ be an isomorphism mapping G to $\overline{G}$. Then the length of every cycle of θ is either 1, or a multiple of 4. There is at most one cycle of length 1.*

Proof. We have already seen that if a cycle has length more than 1, then the length is a multiple of 4. If there were two cycles of length 1, say (u) and (v), then we find

that $uv \in E(G)$ if and only if $(uv)^\theta = uv \in E(\overline{G})$, which is not possible. Therefore there is at most one cycle of length one. □

An example of Theorem 8.10 is shown in Figure 8.7. Here the complementing permutation is $\theta = (1, 2, 3, 4, 5, 6, 7, 8)$. It is easy to map each edge uv of G by θ and see that it is a non-edge of $\overline{G}$.

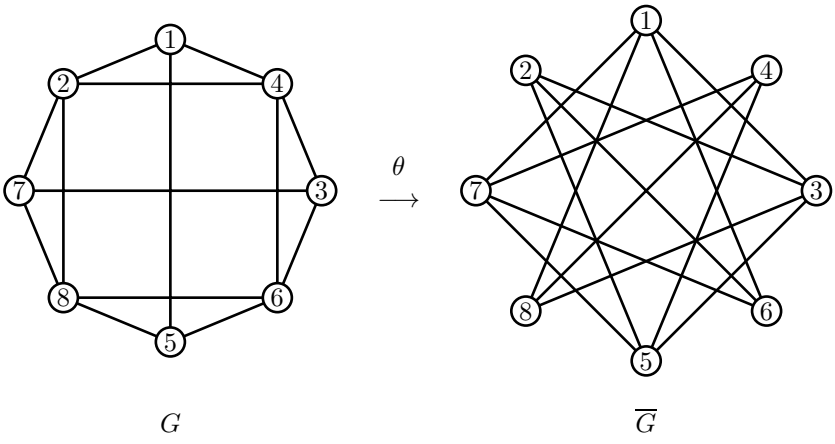


FIGURE 8.7
A self-complementary graph with $\boldsymbol{\theta = (1, 2, 3, 4, 5, 6, 7, 8)}$

Theorem 8.10 gives a simple algorithm to construct self-complementary graphs.

procedure SCGRAPH(n)
comment: n is an integer congruent to 0 or 1 (mod 4)

Create vertices $u_1, u_2, \ldots, u_n$
Choose a complementing permutation θ satisfying Theorem 8.10
Mark all pairs $u_i u_j$ as "unused"
while there is an unused pair
do
- $uv \leftarrow$ an unused pair
- assign uv to G
- $xy \leftarrow (uv)^\theta$
- assign xy to $\overline{G}$
- $xy \leftarrow (xy)^\theta$
- **while** $xy \neq uv$
  - **do**
    - assign xy to G
    - $xy \leftarrow (xy)^\theta$
    - assign xy to $\overline{G}$
    - $xy \leftarrow (xy)^\theta$

The graphs of Figure 8.7 were constructed using this algorithm. The pair 12 was chosen as an edge of G, then θ was used to alternately assign edges to G and $\overline{G}$. Then 13 was chosen as an edge of G, and so forth, until every pair uv had been assigned to be an edge of either G or $\overline{G}$.

Every self-complementary graph can be constructed using this algorithm. However, isomorphic copies of each one will be produced very many times. It is possible to make an algorithm that constructs each self-complementary graph exactly once, by using a general technique of B.D. McKay [123]. The reader is referred to this reference for further information.

Some additional techniques for constructing self-complementary graphs can also be useful. Let θ be a complementing permutation mapping the graph G to $\overline{G}$. Suppose that θ has one or more cycles whose length is a multiple of four. For example, suppose that $(u_1, \ldots, u_8)(u_9, \ldots, u_{16})(u_{17}, \ldots, u_{20})$ comprise two cycles of length eight and a cycle of length four of θ. Let $U_1 = \{u_1, \ldots, u_8\}, U_2 = \{u_9, \ldots, u_{16}\}$, and $U_3 = \{u_{17}, \ldots, u_{20}\}$. Then θ maps $G[U_i]$ to $\overline{G}[U_i]$, so that each $G[U_i]$ is a self-complementary graph. If H is *any* self-complementary graph with $V(H) = U_i$, then a new self-complementary graph can be constructed from G and $\overline{G}$ by replacing $G[U_i]$ with H, and also replacing $\overline{G}[U_i]$ with $\overline{H}$.

Similarly the edges $G[U_i, U_j]$ between U_i and U_j form part of a bipartite self-complementary graph. They can also be substituted with the edges of another bipartite self-complementary graph with the same bipartition. We state these facts as lemmas.

Lemma 8.11. *Let G be a self-complementary graph with complementing permutation θ, and let U be the vertices contained in one or more cycles of θ. Let H be a self-complementary graph with $|U|$ vertices. Then if $G[U]$ is replaced by H, the result is a self-complementary graph.*

Lemma 8.12. *Let G be a self-complementary graph with complementing permutation θ, and let U_1, U_2 be the vertices contained in two distinct cycles of θ. Let H be a bipartite self-complementary graph with bipartition (U_1, U_2). Then if $G[U_1, U_2]$ is replaced by $H[U_1, U_2]$, the result is a self-complementary graph.*

8.8 Pseudo-similar vertices

Let G be a graph with a non-identity automorphism θ. Choose a vertex u such that $u^\theta = v \neq u$. Then it is easy to see that $G-u \cong G-v$, as $(G-u)^\theta = G^\theta - u^\theta = G-v$. Vertices u and v are said to be *similar vertices*. There are graphs G with vertices u, v such that $G - u \cong G - v$, but there is no automorphism relating u to v. Such vertices are called *pseudo-similar vertices*. It turns out that a graph G with pseudo-similar vertices u and v is always an induced subgraph of a graph in which u and v are similar. An example of a graph with pseudo-similar vertices u and v is shown in Figure 8.8.

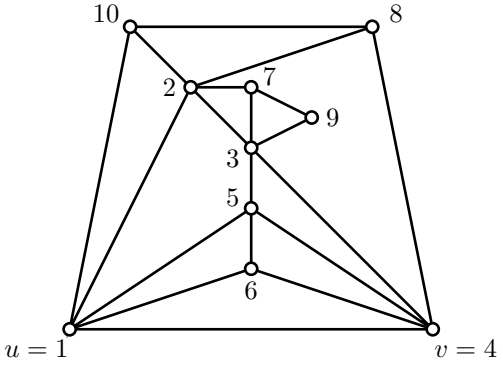


FIGURE 8.8
A graph with pseudo-similar vertices $\boldsymbol{u = 1}$ and $\boldsymbol{v = 4}$.

Suppose that u and v are pseudo-similar in G. Let θ be an isomorphism mapping $G - u$ to $G - v$. Then u^θ is not defined, and there is no vertex w such that $w^\theta = v$. Construct the sequence of vertices $U = \{v, v^\theta, v^{\theta^2}, \ldots\}$. Now θ is a one-to-one mapping, and there is no vertex w such that $w^\theta = v$. Therefore the sequence must terminate. It can only terminate at u, as this is the only vertex with no image under θ. Therefore $u = v^{\theta^m}$, for some $m \geq 1$. Let $v_i = u^{\theta^i}$, for $i = 0, 1, \ldots m$. We can express the sequence of vertices U as $<v_0, v_1, \ldots, v_m>$, where the angle brackets indicate that the sequence does not form a cycle. In the example of Figure 8.8 there is only one possible θ, and this sequence is $<4, 3, 2, 1>$.

Let $W = V(G) - U$ be the remaining vertices of G, if any. They fall into cycles of θ, so that θ acts as a permutation on W. In the example of Figure 8.8, $W = \{5, 6, 7, 8, 9, 10\}$, on which θ has a single cycle, namely $(5, 7, 10, 6, 9, 8)$. We write $\theta = <4, 3, 2, 1> (5, 7, 10, 6, 9, 8)$. Let p be the period of θ acting on W. (If $W = \emptyset$, we take $p = 1$.) We now choose an integer N which is a multiple of p, such that $N \geq 2(m + 1)$. We then define new vertices $v_{m+1}, v_{m+2}, \ldots, v_{N-1}$, and extend θ to θ' such that $v_i^{\theta'} = v_{i+1}$, for $i = m, m + 1, \ldots, N - 2$, and $v_{N-1}^{\theta'} = v_0$. For all other vertices, θ and θ' are defined identically.

Theorem 8.13. *Let θ be extended to θ' as above. Then θ' is an automorphism of a graph H such that G is an induced subgraph of H, and u and v are similar in H.*

Proof. The vertices of H are $V(H) = V(G) \cup \{v_{m+1}, v_{m+2}, \ldots, v_{N-1}\}$, so that θ' is a permutation of $V(H)$. In G, vertex v is adjacent to a subset $X \subseteq W$, and to a subset $Y \subseteq U$. The edges of H include $E(G)$, plus a number of other edges. As θ maps $G - u$ to $G - v$, it follows that v_i is adjacent to the subset $X^{\theta^i} \subseteq W$ in G when $i \leq m$. In H, each v_i, where $i = 0, 1, \ldots, N - 1$, is adjacent to X^{θ^i}. Because N was chosen as a multiple of p, it follows that $X^{\theta'^N} = X$, so that $v_0 = v_0^{\theta'^N}$ is adjacent to X in both G and H.

And each v_i is also adjacent to Y^{θ^i} in H. Suppose that v_i is adjacent to v_j in G, where $i \leq j$. Then $j - i \leq m$. Now N was chosen so that $N \geq 2(m+1)$. Therefore $N + i - j > m$. It follows that any edge of $H[U]$ is also an edge of $G[U]$. Therefore G is an induced subgraph of H. We see that $\theta' \in \text{AUT}(H)$, and $v^{\theta'^m} = u$, so that u and v are similar in H. □

This technique of extending a graph using a mapping θ is often useful in constructing graphs with certain properties. If we use the graph of Figure 8.8 as an example, we have $\theta = <4, 3, 2, 1> (5, 7, 10, 6, 9, 8)$. It has period 6 on U, so that 8 new vertices are added to obtain $V(H)$. It is not always necessary to choose $N \geq 2m$. This is done in the proof to ensure that $G[U] = H[U]$. But often $N = m + 1$ is sufficient, so long as it is a multiple of p. In general, there may be several possible choices for $\theta : G - u \to G - v$. Different choices of θ will give different graphs H.

Kimble, Schwenk and Stockmeyer [98] showed that it is possible for all vertices of a graph to be pseudo-similar. Their construction starts with a group Γ of odd order, and a GRR (graphical regular representation) of Γ, i.e., a Cayley graph H such that $\text{AUT}(H) \cong \Gamma$. Choose a vertex $w \in V(H)$, and let $G = H - w$.

Lemma 8.14. *Let H be a GRR of a group Γ of odd order. Let $w \in V(H)$ and let $G = H - w$. Then every vertex of G is pseudo-similar to another vertex of G.*

Proof. A Cayley graph is vertex-transitive, so that for every $u \in V(G)$, there is an automorphism θ_u mapping u to w. The period of θ_u is odd, because $|\Gamma|$ is odd. Therefore $v = w^{\theta_u} \neq u$, but $(G-u)^{\theta_u} = (H-\{u,w\})^{\theta_u} = H^{\theta_u} - \{w, v\} = G - v$. The vertices adjacent to w in H have different degree in G from the other vertices. Therefore every $\gamma \in \text{AUT}(G)$ belongs to the stabilizer of w in H. But the stabilizer has order one. Therefore vertices u and v are not similar in G, but pseudo-similar. □

It is known that most non-abelian groups of odd order have GRR's. In particular non-abelian groups of order p^3, where p is an odd prime always have GRR's with one exception [92]. Taking $p = 3$, there is a group of order 27 with a GRR, giving a graph on 26 vertices in which all vertices are pseudo-similar.

It is also possible to have *pseudo-similar edges* in a graph G. Edges uv and xy are pseudo-similar if they are not similar, and there is an isomorphism θ mapping $G - uv$ to $G - xy$. Note that θ is a permutation of $V(G)$. We follow the mapping θ to discover the structure of G. Edge xy is an edge of $G - uv$, so that $(xy)^\theta$ is an edge of $G - xy$. If $xy \neq uv$, then $(xy)^\theta$ is an edge of $G - uv$, so that $(xy)^{\theta^2}$ is an edge of $G - xy$. This argument is repeated until $(xy)^{\theta^m} = uv$, for some $m \geq 1$. Without loss of generality, we can take $x^{\theta^m} = u$ and $y^{\theta^m} = v$.

Now uv is not an edge of $G - uv$, so that $(uv)^\theta$ is not an edge of $G - xy$. If $(uv)^\theta \neq xy$, then $(uv)^\theta$ is also not an edge of $G - uv$. Therefore $(uv)^{\theta^2}$ is not an edge of $G - xy$. If $(uv)^{\theta^2} \neq xy$, then $(uv)^{\theta^2}$ is also not an edge of $G - uv$. This argument is repeated until $(uv)^{\theta^k} = xy$, for some $k \geq 1$.

We have a sequence of edges of G: $xy, (xy)^\theta, (xy)^{\theta^2}, \ldots, (xy)^{\theta^{m-1}}$, and a sequence of non-edges of G: $uv, (uv)^\theta, (uv)^{\theta^2}, \ldots, (uv)^{\theta^{k-1}}$. If the pairs

$uv, (uv)^{\theta}, (uv)^{\theta^2}, \ldots, (uv)^{\theta^{k-1}}$ are added to $E(G)$, a graph H is obtained such that $(xy)^{\theta^m} = uv$ and $(uv)^{\theta^k} = xy$, so that uv and xy are similar in H. This gives:

Theorem 8.15. *Let edges uv and xy be pseudo-similar in G. Then G is a subgraph of a graph H, where $V(H) = V(G)$, such that uv and xy are similar in H.*

Note that it is also possible to have similar or pseudo-similar non-edges in a graph. In Theorem 8.15, it would also be possible to remove the edges $xy, (xy)^{\theta}, (xy)^{\theta^2}, \ldots, (xy)^{\theta^{m-1}}$ from G to obtain a graph with similar non-edges uv and xy.

Exercises

8.8.1 Use $\theta = (1,2,3,4)(5,6,7,8)$ as a complementing permutation to construct a number of self-complementary graphs.

8.8.2 In the graph of Figure 8.9, vertices u and v are pseudo-similar. Determine all possible isomorphisms $\theta : G - u \to G - v$. For each possible θ, find a graph H using Theorem 8.13 such that G is an induced subgraph of H, and u and v are similar in H.

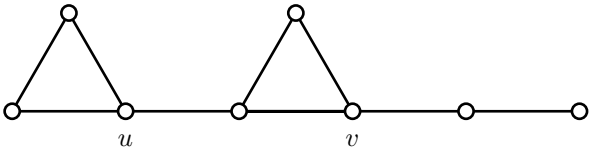


FIGURE 8.9
A graph with pseudo-similar vertices

8.8.3 When applying Theorem 8.13 to the graph of Figure 8.8, an additional 8 vertices were added to G. Determine whether a smaller number will suffice.

8.8.4 Use Theorem 8.15 to construct a graph with a pair of pseudo-similar edges.

8.9 Notes

Some sources for permutation groups and groups acting on graphs are GARDINER [63], GODSIL and ROYLE [70], HALL [79], and ROTMAN [151]. A representative selection of papers on GRR's is GODSIL [68], IMRICH [92], and WATKINS [187]. Two excellent sources for algorithms for permutation groups are BUTLER [27] and

SERESS [158], for readers who are interested in programming. The theory of self-complementary graphs was first developed by SACHS [154] and RINGEL [147]. Theorems 8.13 and 8.15 are from GODSIL and KOCAY [69].

9

Alternating Paths and Matchings

9.1 Introduction

Matchings arise in a variety of situations as assignment problems, in which pairs of items are to be matched together, for example, if people are to be assigned jobs, if sports teams are to matched in a tournament, if tasks are to be assigned to processors in a computer, whenever objects or people are to be matched on a one-to-one basis.

In a graph G, a *matching* M is a set of edges such that no two edges of M have a vertex in common. Figure 9.1 illustrates two matchings M_1 and M_2 in a graph G.

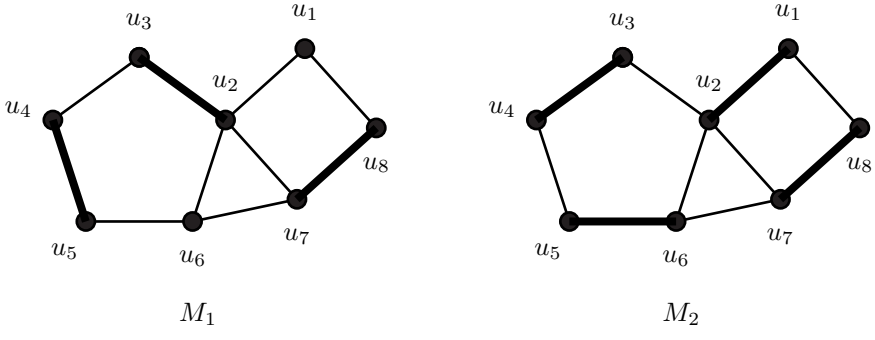


FIGURE 9.1
Matchings

Let M have m edges. Then $2m$ vertices of G are matched by M. We also say that a vertex u is *saturated* by M if it is matched, and *unsaturated* if it is not matched. In general, we want M to have as many edges as possible.

DEFINITION 9.1: M is a *maximum matching* in G if no matching of G has more edges.

For example, in Figure 9.1, $|M_1| = 3$ and $|M_2| = 4$. Because $|G| = 8$, M_2 is a maximum matching. A matching which saturates every vertex is called a *perfect matching*. Obviously a perfect matching is always a maximum matching. M_1 is not a maximum matching, but it is a *maximal matching*; namely, M_1 cannot be extended

by the addition of any edge uv of G. However, there is a way to build a bigger matching out of M_1. Let P denote the path $(u_1, u_2, \ldots, u_6)$ in Figure 9.1.

DEFINITION 9.2: Let G have a matching M. An *alternating path* P with respect to M is any path whose edges are alternately in M and not in M. If the endpoints of P are unsaturated, then P is an *augmenting path*.

So $P = (u_1, u_2, \ldots, u_6)$ is an augmenting path with respect to M_1. Consider the subgraph formed by the *exclusive or* operation $M = M_1 \oplus E(P)$ (also called the *symmetric difference*, $(M_1 - E(P)) \cup (E(P) - M_1)$). M contains those edges of P which are not in M_1, namely, u_1u_2, u_3u_4, and u_5u_6. M is a bigger matching than M_1. Notice that $M = M_2$.

Lemma 9.1. *Let G have a matching M. Let P be an augmenting path with respect to M. Then $M' = M \oplus E(P)$ is a matching with one more edge than M.*

Proof. Let the endpoints of P be u and v. M' has one more edge than M, because u and v are unsaturated in M, but saturated in M'. All other vertices that were saturated in M are still saturated in M'. So M' is a matching with one more edge. □

The key result in the theory of matchings is the following:

Theorem 9.2. (Berge's theorem) *A matching M in G is maximum if and only if G contains no augmenting path with respect to M.*

Proof. If M were a maximum matching and P an augmenting path, then $M \oplus E(P)$ would be a larger matching. So there can be no augmenting path if M is maximum.

Conversly suppose that G has no augmenting path with respect to M. If M is not maximum, then pick a maximum matching M'. Clearly $|M'| > |M|$. Let $H = M \oplus M'$. Consider the subgraph of G that H defines. Each vertex v is incident on at most one M-edge and one M'-edge, so that in H, $\text{DEG}(v) \leq 2$. Every path in H alternates between M-edges and M'-edges. So H consists of alternating paths and cycles, as illustrated in Figure 9.2.

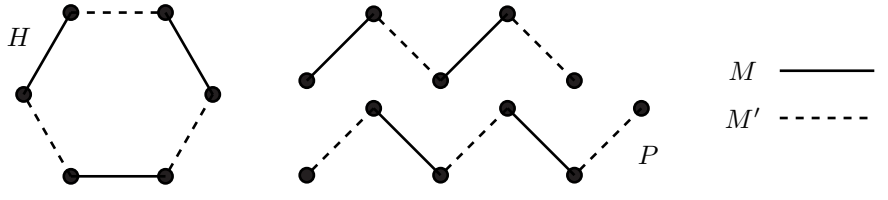


FIGURE 9.2
Alternating paths and cycles

Each cycle must clearly have even length, with an equal number of edges of M and M'. Because $|M'| > |M|$, some path P must have more M'-edges than M-edges. It can only begin and end with an M'-edge, so that P is augmenting with respect to M. But we began by assuming that G has no augmenting path for M. Consequently, M was initially a maximum matching. □

This theorem tells us how to find a maximum matching in a graph. We begin with some matching M. If M is not maximum, there will be an unsaturated vertex u. We then follow alternating paths from u. If some unsaturated vertex v is reached on an alternating path P, then P is an augmenting uv-path. Set $M \leftarrow M \oplus E(P)$, and repeat. If the method that we have chosen to follow alternating paths is sure to find all such paths, then this technique is guaranteed to find a maximum matching in G.

In bipartite graphs it is slightly easier to follow alternating paths and therefore to find maximum matchings, because of their special properties. Let G have bipartition (X, Y). If $S \subseteq X$, then the *neighbor set* of S is $N(S)$, the set of Y-vertices adjacent to S. Sometimes $N(S)$ is called the *shadow set* of S. If G has a perfect matching M, then every $x \in S$ will be matched to some $y \in Y$ so that $|N(S)| \geq |S|$, for every $S \subseteq X$. HALL [79] proved that this necessary condition is also sufficient.

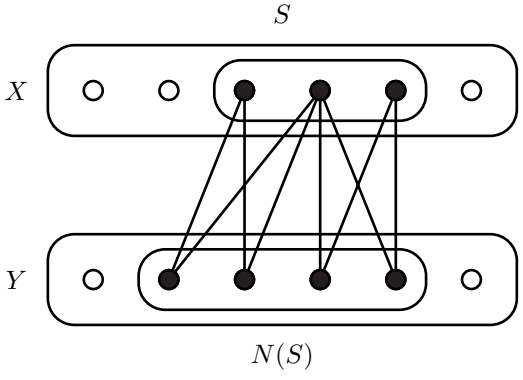


FIGURE 9.3
The neighbor set

Theorem 9.3. (Hall's theorem) *Let G have bipartition (X, Y). G has a matching saturating every $x \in X$ if and only if $|N(S)| \geq |S|$, for all $S \subseteq X$.*

Proof. We have already discussed the necessity of the conditions. For the converse suppose that $|N(S)| \geq |S|$, for all $S \subseteq X$. If M does not saturate all of X, pick an unsaturated $u \in X$, and follow all the alternating paths beginning at u. (See Figure 9.4.)

Let $S \subseteq X$ be the set of X-vertices reachable from u on alternating paths, and let T be the set of Y-vertices reachable. With the exception of u, each vertex $x \in S$ is matched to some $y \in T$, for S was constructed by extending alternating paths from $y \in T$ to $x \in S$ whenever xy is a matching edge. Therefore $|S| = |T| + 1$.

Now there may be other vertices $X - S$ and $Y - T$. However, there can be no edges $[S, Y - T]$, for such an edge would extend an alternating path to a vertex of $Y - T$, which is not reachable from u on an alternating path. So every $x \in S$ can only be joined to vertices of T; that is, $T = N(S)$. It follows that $|S| > |N(S)|$, a contradiction. Therefore every vertex of X must be saturated by M. □

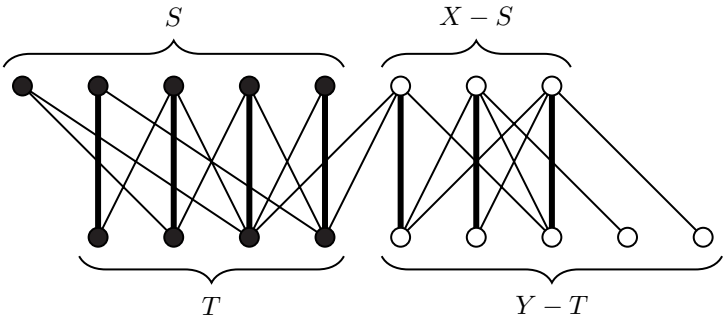


FIGURE 9.4
Follow alternating paths

Corollary 9.4. *Every k-regular bipartite graph has a perfect matching, if $k > 0$.*

Proof. Let G have bipartition (X, Y). Because G is k-regular, $\varepsilon = k \cdot |X| = k \cdot |Y|$, so that $|X| = |Y|$. Pick any $S \subseteq X$. How many edges have one end in S? Exactly $k \cdot |S|$. They all have their other end in $N(S)$. The number of edges with one endpoint in $N(S)$ is $k \cdot |N(S)|$. So $k \cdot |S| \leq k \cdot |N(S)|$, or $|S| \leq |N(S)|$, for all $S \subseteq X$. Therefore G has a perfect matching. □

Exercises

9.1.1 Find a formula for the number of perfect matchings of K_{2n} and $K_{n,n}$.

9.1.2 (Hall's theorem.) Let A_1, A_2, …, A_n be subsets of a set S. A *system of distinct representatives* for the family $\{A_1, A_2, \ldots, A_n\}$ is a subset $\{a_1, a_2, \ldots, a_n\}$ of S such that $a_1 \in A_1$, $a_2 \in A_2$, …, $a_m \in A_m$, and $a_i \neq a_j$, for $i \neq j$. Example:

$A_1 =$ students taking computer science 421

$A_2 =$ students taking physics 374

$A_3 =$ students taking botany 464

$A_4 =$ students taking philosophy 221

The sets A_1, A_2, A_3, A_4 may have many students in common. Find four distinct students a_1, a_2, a_3, a_4, such that $a_1 \in A_1$, $a_2 \in A_2$, $a_3 \in A_3$, and $a_4 \in A_4$ to represent each of the four classes.

Show that $\{A_1, A_2, \ldots, A_n\}$ has a system of distinct representatives if and only if the union of every combination of k of the subsets A_i contains at least k elements, for all $k = 1, 2, \ldots, n$. (*Hint*: Make a bipartite graph $A_1, A_2, \ldots, A_n$ versus all $a_j \in S$, and use Hall's theorem.)

9.2 The Hungarian algorithm

We are now in a position to construct an algorithm which finds a maximum matching in bipartite graphs, by following alternating paths from each unsaturated $u \in X$. How can we best follow alternating paths? Let $n = |G|$. Suppose that we store the matching as an integer array $\mathit{Match}[x]$, $x = 1, 2, \ldots, n$, where $\mathit{Match}[x]$ is the vertex matched to x (so $\mathit{Match}[\mathit{Match}[x]] = x$, if x is saturated). We use $\mathit{Match}[x] = 0$ to indicate that x is unsaturated. We could use either a DFS or BFS to construct the alternating paths. A DFS is slightly easier to program, but a BF-tree tends to be shallower than a DF-tree, so that a BFS will likely find augmenting paths more quickly, and find shorter augmenting paths, too. Therefore the BFS is used for matching algorithms.

The array used to represent parents in the BF-tree can be used in combination with the $\mathit{Match}[\cdot]$ array to store the alternating paths. We write $\mathit{PrevPt}[v]$ for the parent of v in a BF-tree. It is the previous point to v on an alternating path to the root. This is illustrated in Figure 9.5.

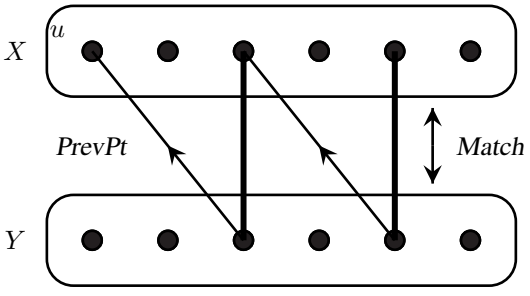


FIGURE 9.5
Storing the alternating paths

We also need to build the sets S and $N(S)$ as queues, which we store as the arrays *ScanQ* and NS, respectively. The algorithm for finding a maximum matching in bipartite graphs is Algorithm 9.3.1. It is also called the *Hungarian algorithm* for maximum matchings in bipartite graphs.

Algorithm 9.2.1: MAXMATCHING(G)

comment: Hungarian algorithm. G has bipartition (X, Y), and n vertices.

for $i \leftarrow 1$ **to** n **do** $\mathit{Match}[i] \leftarrow 0$
for each $u \in X$
do {
  comment: u is currently unsaturated
  $\mathit{ScanQ}[1] \leftarrow u$
  $\mathit{QSize} \leftarrow 1$
  comment: construct alternating paths from u using a BFS
  for $i \leftarrow 1$ **to** n **do** $\mathit{PrevPt}[i] \leftarrow 0$
  $k \leftarrow 1$
  repeat
    $x \leftarrow \mathit{ScanQ}[k]$
    for each $y \longrightarrow x$ **do**
      if $y \notin NS$
      then {
        add y to NS
        $\mathit{PrevPt}[y] \leftarrow x$
        if y is unsaturated
        then {
          comment: augmenting path found
          AUGMENT(y)
          go to 1 "u is now saturated"
        }
        add $\mathit{Match}[y]$ **to** ScanQ
      }
    $k \leftarrow k + 1$ "advance ScanQ"
  until $k > \mathit{QSize}$
  comment: {
    ScanQ now contains a set S, and NS contains
    the neighbor-set $N(S)$ such that $|S| = |N(S)| + 1$,
    no matching can saturate all of S
  }
  delete S and $N(S)$ from the graph
  1 :
}
comment: $\mathit{Match}[\cdot]$ now contains a maximum matching

Notice that the algorithm needs to be able to determine whether $y \in NS$. This can be done by storing a boolean array. Another possibility is to use $\mathit{PrevPt}[v] = 0$ to indicate that $v \notin N(S)$. We can test if y is unsaturated by checking whether $\mathit{Match}[y] = 0$. AUGMENT(y) is a procedure that computes $M \leftarrow M \oplus E(P)$, where P is the augmenting path found. Beginning at vertex y, it alternately follows $\mathit{PrevPt}[\cdot]$ and $\mathit{Match}[\cdot]$ back to the initial unsaturated vertex, which is the root-node of the BF-tree being constructed. This is illustrated in Figure 9.6.

Algorithm 9.2.2: AUGMENT(y)

comment: follow the augmenting path, setting $M \leftarrow M \oplus E(P)$

repeat
$w \leftarrow PrevPt[y]$
$Match[y] \leftarrow w$
$v \leftarrow Match[w]$
$Match[w] \leftarrow y$
$y \leftarrow v$
until $y = 0$

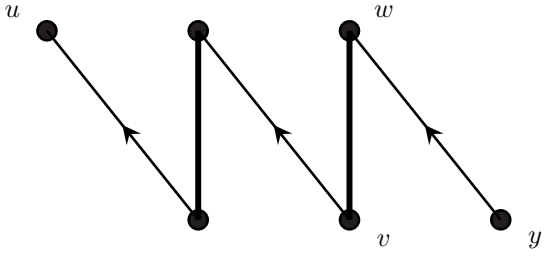


FIGURE 9.6
Augmenting the matching

The BFS constructs an alternating search tree. It contains all vertices reachable from the root-node u on alternating paths. Vertices at even distance from u in the tree form the set S, and those at odd distance form $N(S)$. The vertices of S are sometimes called *outer vertices*, and those of $N(S)$ *inner vertices*. All the actual searching is done from the outer vertices.

Theorem 9.5. *The Hungarian algorithm constructs a maximum matching in a bipartite graph.*

Proof. Let G have bipartition (X, Y). If the algorithm saturates every vertex of X, then it is certainly a maximum matching. Otherwise some vertex u is not matched. If there is an augmenting path P from u, it must alternate between X and Y, because G is bipartite. The algorithm constructs the sets S and $N(S)$, consisting of all vertices of X and Y, respectively, that can be reached on alternating paths. So P will be found if it exists. If u cannot be saturated, then we know that $|S| = |N(S)|+1$. Every vertex of S but u is matched. S and $N(S)$ are then deleted from the graph. Does the deletion of these vertices affect the rest of the algorithm? As in Hall's theorem, there are no edges $[S, Y - N(S)]$. Suppose that alternating paths from a vertex $v \in X$ were being constructed. If such a path were to reach a vertex y in the deleted $N(S)$, it could only extend to other vertices of S and $N(S)$. It could not extend to an augmenting path.

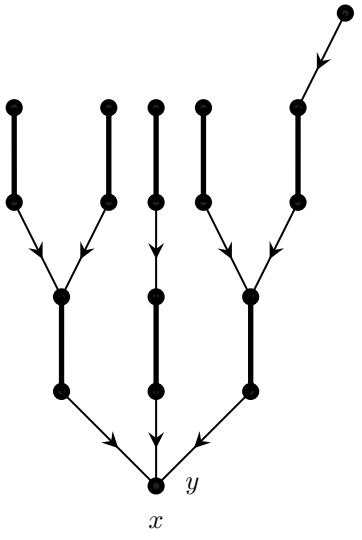


FIGURE 9.7
The alternating search tree

Therefore these vertices can be deleted. Upon completion, the algorithm will have produced a matching M for which there are no augmenting paths in the graph. By Theorem 9.2, M is a maximum matching. □

9.2.1 Complexity

Suppose that at the beginning of the for-loop, M has m edges. The largest possible size of S and $N(S)$ is then $m+1$, and m, respectively. The number of edges $[S, N(S)]$ is at most $m(m+1)$. In the worst possible case, S and $N(S)$ will be built up to this size, and $m(m+1)$ edges between them will be encountered. If an augmenting path is now found, then m will increase by one to give a worst case again for the next iteration. The length of the augmenting path will be at most $2m+1$, in case all m matching edges are in the path. The number of steps performed in this iteration of the for-loop will then be at most $m(m+1)+(2m+1)$. Because $|X|+|Y|=n$, the number of vertices, one of $|X|$ and $|Y|$ is $\leq n/2$. We can take X as the smaller side. Summing over all iterations then gives

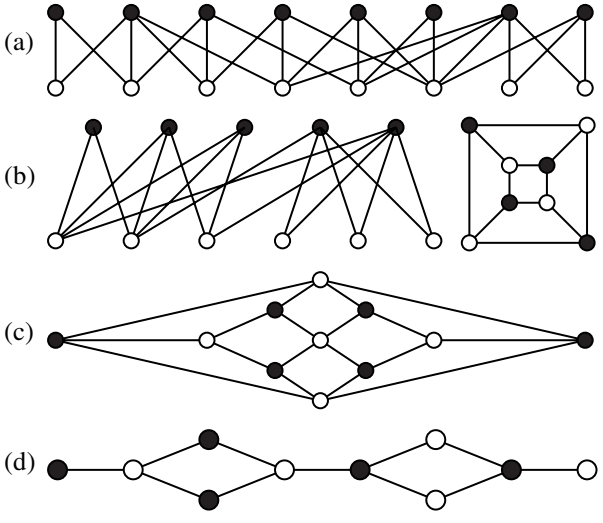
$$\sum_{m=0}^{\frac{n}{2}-1} m(m+1)+(2m+1) = \sum 2\binom{m+1}{2} + (2m+1)$$

$$= 2\binom{n/2+1}{3} + 2\binom{n/2}{2} + \frac{n}{2}.$$

The leading term in the expansion is $n^3/24$, so that the algorithm is of order $O(n^3)$, with a small constant coefficient. It can be improved with a more careful choice of augmenting paths. HOPCROFT and KARP [87] maintain several augmenting paths, and augment simultaneously on all of them to give $O(n^{2.5})$. This can also be accomplished with network flow techniques.

Exercises

9.2.1 Program the Hungarian matching algorithm. The output should consist of a list of the edges in a maximum matching. If there is no matching saturating the set X, this should be indicated by printing out the sets $S \subseteq X$ found whose neighbor set $N(S)$ is smaller than S. Use the four sample graphs listed below for input. The set X is marked by shaded dots, and Y by open dots.



9.3 Edmonds' algorithm, blossoms

When G is not bipartite, the Hungarian algorithm cannot be used to find a maximum matching in G. Edmonds [47] discovered how to find maximum matchings efficiently in non-bipartite graphs. Consider the graph G shown in Figure 9.8. Here the Hungarian algorithm is being used to construct an alternating tree from vertex 1, so that vertex 1 can be matched.

The alternating tree has been built to its full extent, but no augmenting path was found. By Berge's theorem 9.2, the matching can be augmented on the alternating path $[1, 3, 5, 7, 9, 8, 6, 10]$. However the algorithm does not find this path. Notice the odd cycle $(5, 6, 8, 9, 7)$ containing two matching edges. The key to finding all

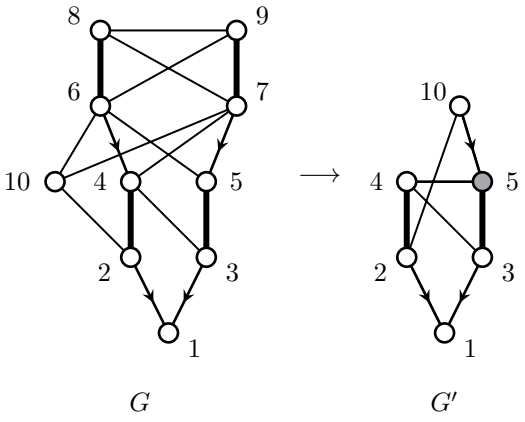


FIGURE 9.8
A matching in a non-bipartite graph

augmenting paths in a non-bipartite graph is to find certain odd cycles in which all vertices but one are matched within the cycle. Such a cycle is called a *blossom*.

DEFINITION 9.3: Let graph G have a matching M. Let C be the edges of an odd cycle of G such $M \cap C$ is a matching in C containing all vertices of C, but one. Then C is called a *blossom* of G with respect to M. The vertex of C which is not matched by $M \cap C$ is called the *blossom base*.

The Hungarian algorithm can be modified to detect blossoms as the alternating tree is built, and "shrink" them. If C is a blossom in G, then to *shrink* C means to change G into a new graph in which all vertices of C are identified into one vertex. The result of shrinking the blossom $(5, 6, 8, 9, 7)$ of Figure 9.8 is shown in the same diagram, where the "meta"-vertex representing the shrunken blossom is shaded. The reduced graph G' now has an alternating path $[1, 3, 5, 10]$ to vertex 10. When the matching in G' is augmented by following the alternating path from vertex 10 to 1, there is a corresponding alternating path in G that "travels through the blossom" to its base. In this case it is $[10, 6, 8, 9, 7, 5]$.

While the alternating tree is being constructed, a blossom may be detected, in which case it is immediately shrunk, and the algorithm continues. Another blossom containing the meta-vertex of the previous blossom may be later detected, and immediately shrunk — *blossoms are actually recursive structures, they may contain previously shrunken blossoms, which may in turn contain previously shrunken blossoms, etc.* We see that in Definition 9.3, some vertices of the graph G may actually represent blossoms that have already been shrunk.

In order to program Edmonds' algorithm effectively, we need a data structure that can store blossoms, which can contain recursively shrunken blossoms. We use a variation of the *Merge-Find* data structure, which was described in Chapter 2. Each blossom is represented by the vertex which is its base. There is an array *BasePtr*$[u]$

which points towards the base of the blossom containing vertex u. Initially the vertices are not in any blossom, so we set $BasePtr[u] = 0$. When a vertex u becomes part of a blossom, $BasePtr[u]$ will be set to the blossom base. There is a procedure BLOSSOMBASE(u) which returns the vertex which is the base of the blossom containing u. Similar to the procedure COMPREP() of Chapter 2, we have :

```
procedure BLOSSOMBASE(u)
 if BasePtr[u] = 0
   then return (0)       "not in any blossom"
 if BasePtr[u] = u
   then return (u)       "u is the blossom base"
 theRep ← BLOSSOMBASE(BasePtr[u])
 BasePtr[u] ← theRep
 return (theRep)
```

Edmonds' algorithm can now be presented as a breadth-first search to build an alternating tree, while shrinking blossoms as they are discovered. As before, $Match[v]$ indicates the vertex that v is matched to, it is initially set to 0. $PrevPt[v]$ indicates the previous vertex in a path from v to the root of the search tree. $PrevPt[v] = 0$ is used to indicate that v has not yet been visited. Blossoms are only meaningful with respect to the alternating search tree as it is being constructed. Therefore all blossoms are re-initiallized before each iteration of the algorithm.

The vertices of an alternating search tree, as in Figure 9.7, can be classified as *inner* vertices or *outer* vertices. All searching is done from outer vertices. The base of a blossom is always an outer vertex. The inner vertices are not in any blossom. Therefore when the search tree is built, and a vertex u becomes an outer vertex, a blossom is created for it, by assigning $BasePtr[u] = u$. This ensures that BLOSSOMBASE(u) will return u as the base of its blossom. All outer vertices are at an even distance from the root of the search tree.

An odd cycle is detected when an edge vw, where v and w are both outer vertices, is discovered. The alternating paths from v and w through the search tree towards the root are followed to find the base b of the new blossom, and $BasePtr[x] = b$ is assigned for each x on these paths. All vertices in the new blossom now become outer vertices. This is how blossoms are shrunk. The base of a blossom is either the root of the search tree, or it is matched to vertex $Match[v]$, which is an inner vertex that is not in any blossom. All outer vertices are placed on a queue called *ScanQ*. It is an array containing vertices from which the search tree is extended. When a blossom is detected and shrunk, all vertices in the blossom are placed on the *ScanQ*, so that their incident edges will be visited.

Algorithm 9.3.1: MAXMATCHING(G)

comment: Edmonds' matching algorithm. G has n vertices.

for $u \leftarrow 1$ **to** n **do** *Match*$[u] \leftarrow 0$
for $u \leftarrow 1$ **to** n **do** *BasePtr*$[u] \leftarrow 0$
for each $u \in X$ **do if** *Match*$[u] = 0$
then
- **comment:** u is currently unsaturated
- *ScanQ*$[1] \leftarrow u$
- *BasePtr*$[u] \leftarrow u$ "create a blossom containing u"
- *QSize* $\leftarrow 1$
- **for** $v \leftarrow 1$ **to** n **do** *PrevPt*$[v] \leftarrow 0$
- **comment:** construct alternating paths from u using a BFS
- $k \leftarrow 1$
- **repeat**
  - $x \leftarrow$ *ScanQ*$[k]$
  - *xBase* $\leftarrow$ BLOSSOMBASE(x)
  - **for each** $y \longrightarrow x$
    - **do**
      - *yBase* $\leftarrow$ BLOSSOMBASE(y)
      - **if** *yBase* $=$ *xBase* **then go to** 1 "ignore edge xy"
      - **if** *yBase* $\neq 0$
        - **then**
          - **comment:** x and y are in different blossoms
          - SHRINKBLOSSOM($x, y,$ *xBase*, *yBase*)
          - **go to** 1
      - **comment:** otherwise y is not in a blossom
      - **if** *PrevPt*$[y] \neq 0$ **then go to** 1 "y is already in the tree"
      - *PrevPt*$[y] \leftarrow x$ "add y to the tree"
      - **if** *Match*$[y] = 0$
        - **then**
          - **comment:** augmenting path found
          - AUGMENT(y)
          - **break**
      - **comment:** otherwise y is already matched
      - $v \leftarrow$ *Match*$[y]$
      - add v to *ScanQ*
      - *BasePtr*$[v] \leftarrow v$ "create a blossom containing v"
      - $1:$
  - $k \leftarrow k + 1$ "advance *ScanQ*"
- **until** $k >$ *QSize*

comment: *Match*$[\cdot]$ now contains a maximum matching

When a blossom has been shrunk, all vertices v in the blossom become outer vertices, to allow the search tree to be extended from each v. If an unmatched vertex

w is adjacent to one of these blossom vertices, then the algorithm must be able to follow the alternating path from w to the root of the search tree, in order to augment the matching. Consider the example shown in Figure 9.9.

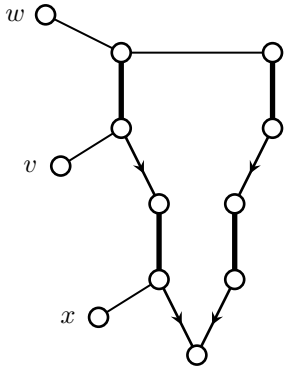


FIGURE 9.9
An augmenting path through a blossom

In this example, there is an alternating path from each of v, w, x through the blossom to the root of the search tree. In order to augment the matching, the algorithm must be able to follow any of these paths. In general, an alternating path from an inner vertex y in a search tree is followed by successively iterating

$u \leftarrow \mathit{PrevPt}[y]$
$y \leftarrow \mathit{Match}[u]$

until the root of the tree is reached, as in Algorithm 9.2.2. This will also work with blossoms, so long as *PrevPt* is assigned correctly. In the trees of Figures 9.7 and 9.8, arrows are used to indicate *PrevPt*, so that alternating paths can be followed down the tree towards the root by using the above statements. In Figure 9.9, the alternating path from w simply descends the tree to the root, as in Figure 9.7. But the alternating path from v or x in Figure 9.9 must first ascend the tree through the blossom, then travel around the blossom, and down the other side towards the root. This can easily be effected by extending the definition of *PrevPt*. When an edge yz in a blossom has $\mathit{PrevPt}[y] = z$, being indicated as an arrow pointing down the tree, we can also assign $\mathit{PrevPt}[z] = y$ when the blossom is shrunk. This will ensure that alternating paths will be correctly followed through a blossom to its base, from any vertex in the blossom. This method will clearly work for blossoms which are odd cycles. When an odd cycle blossom becomes part of a larger blossom, it will still work, because the alternating path through a blossom always goes to the base, which is matched to an inner vertex outside the blossom. Therefore this will also work for larger blossoms. Consequently, Algorithm 9.3.1 will always find an augmenting path from each vertex u, if one exists, and will therefore always find a maximum matching.

9.3.1 Complexity

The maximum number of steps will occur if the algorithm builds the alternating tree as large as possible on each iteration, and then finds an augmenting path. In this case there will be $n/2$ iterations of the main loop, and the i^{th} iteration will build a search tree with $i-1$ matching edges. The length of the alternating path found will be at most $2i+1$, in case all current matching edges are part of the path. The number of steps required to build the search tree and augment, over all iterations, will then be at most $O(n^2)$, not counting the steps to shrink blossoms.

The *Merge-Find* data structure has a near linear time performance — for all values of n within practical bounds, the performance is effectively $O(m)$, where m is the number of calls to BLOSSOMBASE(). The number of calls will be at most $i+1$ on iteration i, so that over all iterations, approximately $O(n^2)$ calls will be made. When a blossom is shrunk upon finding an edge vw that creates an odd cycle, the alternating paths from v and w to the root must be followed to find the base of the new blossom. This can be done in i steps on iteration i, because there are $i-1$ edges currently in the matching. This also results in $O(n^2)$ steps, taken over all iterations. Thus, Edmonds' algorithm can be programmed to take $O(n^2)$ steps, for all practical values of n. The algorithm is not difficult to program, and is an excellent exercise in programming graph algorithms.

9.4 Perfect matchings and 1-factorizations

Given any graph G and positive integer k, a *k-factor* of G is a spanning subgraph that is k-regular. Thus a perfect matching is a 1-factor. A 2-factor is a union of cycles that covers $V(G)$, as illustrated in Figure 9.10.

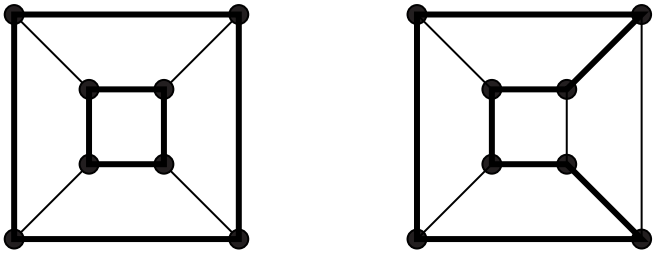


FIGURE 9.10
2-factors of the cube

The reason for this terminology is as follows. Associate indeterminates x_1, x_2, ..., x_n with the n vertices of a graph. An edge connecting vertex i to j can be represented by the expression $x_i - x_j$. Then the entire graph can be represented (up to sign) by the product $P(G) = \prod_{ij \in E(G)}(x_i - x_j)$. For example, if G is the 4-

cycle, this product becomes $(x_1 - x_2)(x_2 - x_3)(x_3 - x_4)(x_4 - x_1)$. Because the number of terms in the product is $\varepsilon(G)$, when it is multiplied out, there will be ε x's in each term. A 1-factor of $P(G)$, for example, $(x_1 - x_2)(x_3 - x_4)$, is a factor that contains each x_i exactly once and corresponds to a perfect matching in G. A k-factor of $P(G)$, is a factor that contains each x_i exactly k-times.

With some graphs it is possible to decompose the edge set into perfect matchings. For example, if G is the cube, we can write $E(G) = M_1 \cup M_2 \cup M_3$, where

$$\begin{aligned} M_1 &= \{12, 34, 67, 85\}, \\ M_2 &= \{23, 14, 56, 78\}, \\ M_3 &= \{15, 26, 37, 48\}, \end{aligned}$$

as shown in Figure 9.11. Each edge of G is in *exactly one* of M_1, M_2, or M_3. Also $P(G) = \pm F_1 F_2 F_3$ where

$$\begin{aligned} F_1 &= (x_1 - x_2)(x_3 - x_4)(x_6 - x_7)(x_8 - x_5), \\ F_2 &= (x_2 - x_3)(x_1 - x_4)(x_5 - x_6)(x_7 - x_8), \\ F_3 &= (x_1 - x_5)(x_2 - x_6)(x_3 - x_7)(x_4 - x_8). \end{aligned}$$

In general, a *k-factorization* of a graph G is a decomposition of $E(G)$ into $H_1 \cup H_2 \cup \ldots \cup H_m$, where each H_i is a k-factor, and each H_i and H_j have no edges in common. If a graph G has a k-factorization, we say that G is *k-factorable*. The graph G is k-factorable if and only if $P(G)$ can be written as a product of k-factors. The decomposition in Figure 9.11 is a 1-factorization of the cube and thus the cube is 1-factorable.

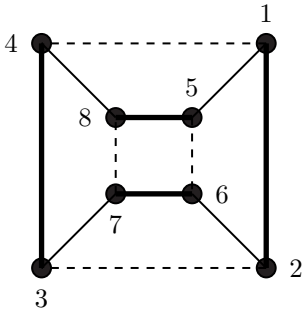


FIGURE 9.11
A 1-factorization of the cube

Lemma 9.6. *$K_{n,n}$ is 1-factorable.*

Proof. Let (X, Y) be the bipartition of $K_{n,n}$, where $X = \{x_0, x_1, \ldots, x_{n-1}\}$ and $Y = \{y_0, y_1, \ldots, y_{n-1}\}$. Define $M_0 = \{x_i y_i \mid i = 0, 1, \ldots, n-1\}$, $M_1 = \{x_i y_{i+1} \mid i = 0, 1, \ldots, n-1\}$, etc., where the addition is modulo n. In general $M_k = \{x_i y_{i+k} \mid$

$i = 0, 1, \ldots, n-1\}$. Clearly M_j and M_k have no edges in common, for any j and k, and together $M_0, M_1, \ldots, M_{n-1}$ contain all of $E(G)$. Thus we have a 1-factorization of $K_{n,n}$. □

Lemma 9.7. *K_{2n} is 1-factorable.*

Proof. Let $V(K_{2n}) = \{0, 1, 2, \ldots, 2n-2\} \cup \{\infty\}$. Draw K_{2n} with the vertices $0, 1, \ldots, 2n-2$ in a circle, placing ∞ in the center of the circle. This is illustrated for $n = 4$ in Figure 9.12. Take $M_0 = \{(0, \infty), (1, 2n-2), (2, 2n-3), \ldots, (n-1, n)\} = \{(0, \infty)\} \cup \{(i, -i) \mid i = 1, 2, \ldots, n-1\}$, where the addition is modulo $2n-1$. M_0 is illustrated by the thicker lines in Figure 9.12.

We can then "rotate" M_0 by adding one to each vertex, $M_1 = M_0 + 1 = \{(i+1, j+1) \mid (i, j) \in M_0\}$, where $\infty + 1 = \infty$, and addition is modulo $2n-1$. It is easy to see from the diagram that M_0 and M_1 have no edges in common. Continuing like this, we have

$$M_0, M_1, M_2, \ldots, M_{2n-2},$$

where $M_k = M_0 + k$. They form a 1-factorization of K_{2n}. □

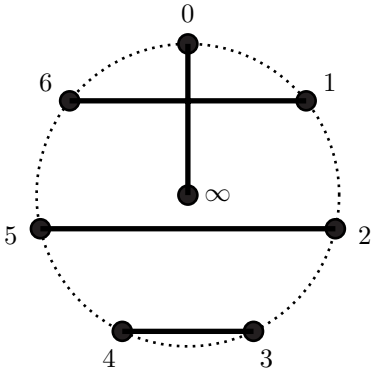


FIGURE 9.12
1-factorizing $\boldsymbol{K_{2n}}$, where $\boldsymbol{n = 4}$

We can use a similar technique to find a 2-factorization of K_{2n+1}.

Lemma 9.8. *K_{2n+1} is 2-factorable.*

Proof. Let $V(K_{2n+1}) = \{0, 1, 2, \ldots, 2n-1\} \cup \{\infty\}$. As in the previous lemma, draw the graph with the vertices in a circle, placing ∞ in the center. The first 2-factor is the cycle $H_0 = (0, 1, -1, 2, -2, \ldots, n-1, n+1, n, \infty)$, where the arithmetic is modulo $2n$. This is illustrated in Figure 9.13, with $n = 3$. We then rotate the cycle to get $H_1, H_2, \ldots, H_{n-1}$, giving a 2-factorization of K_{2n+1}. □

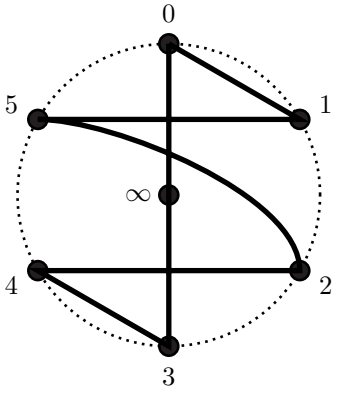


FIGURE 9.13
2-factorizing K_{2n+1}, where $n = 3$

Exercises

9.4.1 Find all perfect matchings of the cube. Find all of its 1-factorizations.

9.4.2 Find all perfect matchings and 1-factorizations of K_4 and K_6.

9.4.3 Prove that the Petersen graph has no 1-factorization.

9.4.4 Prove that for $k > 0$ every k-regular bipartite graph is 1-factorable.

9.4.5 Describe another 1-factorization of K_{2n}, when n is even, using the fact that $K_{n,n}$ is a subgraph of K_{2n}.

9.4.6 Let $M_1, M_2, \ldots, M_k$ and $M'_1, M'_2, \ldots, M'_k$ be two 1-factorizations of a k-regular graph G. The two factorizations are isomorphic if there is an automorphism θ of G such that for each i, $\theta(M_i) = M'_j$, for some j; that is, θ induces a mapping of $M_1, M_2, \ldots, M_k$ onto $M'_1, M'_2, \ldots, M'_k$. How many non-isomorphic 1-factorizations are there of K_4 and K_6?

9.4.7 How many non-isomorphic 1-factorizations are there of the cube?

9.5 The subgraph problem

Let G be a graph and let $f\colon V(G) \to \{0, 1, 2, \ldots\}$ be a function assigning a non-negative integer to each vertex of G. An *f-factor* of G is a subgraph H of G such that $\deg(u, H) = f(u)$, for each $u \in V(G)$. So a 1-factor is an f-factor in which each $f(u) = 1$.

Problem 9.1:	Subgraph Problem
Instance:	a graph G and a function $f\colon V(G) \to \{0, 1, 2, \ldots\}$.
Find:	an f-factor in G, if one exists.

There is an ingenious construction by TUTTE [174], that transforms the subgraph problem into the problem of finding a perfect matching in a larger graph G'. Construct G' as follows. For each edge $e = uv$ of G, G' has two vertices e_u and e_v, such that $e_u e_v \in E(G')$. For each vertex u of G, let $m(u) = \deg(u) - f(u)$. Corresponding to $u \in V(G)$, G' has $m(u)$ vertices $u_1, u_2, \ldots, u_{m(u)}$. For each edge $e = uv \in E(G)$, $u_1, u_2, \ldots, u_{m(u)}$ are all adjacent to $e_u \in V(G')$. This is illustrated in Figure 9.14, where $\deg(u) = 5$ and $f(u) = 3$.

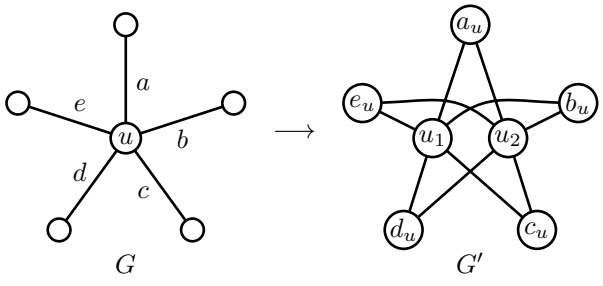


FIGURE 9.14
Tutte's transformation

Theorem 9.9. *G has an f-factor if and only if G' has a perfect matching.*

Proof. Suppose that G has an f-factor H. Form a perfect matching M in G' as follows. For each edge $uv \in H$, $e_u e_v \in M$. There are $m(u) = \deg(u) - f(u)$ remaining edges at vertex $u \in V(G)$. In G', these can be matched to the vertices $u_1, u_2, \ldots, u_{m(u)}$ in any order.

Conversely, given a perfect matching $M \subseteq G'$, the vertices $u_1, u_2, \ldots, u_{m(u)}$ will be matched to $m(u)$ vertices, leaving $f(u)$ adjacent vertices of the form e_u not matched to any u_i. They can therefore only be matched to vertices of the form e_v for some v. Thus $f(u)$ edges $e_u e_v$ are selected corresponding to each vertex u. This gives an f-factor of G. □

So finding an f-factor in G is equivalent to finding a perfect matching in G'. If G has n vertices and ε edges, then G' has

$$4\varepsilon - \sum f(u)$$

vertices and

$$\varepsilon + \sum (\deg^2(u) - \deg(u) f(u))$$

edges. Finding perfect matchings in non-bipartite graphs is considerably more complicated than in bipartite graphs, but is still very efficient. Edmonds' algorithm [47] will find a maximum matching in time $O(n^3)$. Thus, the subgraph problem can be solved using perfect matchings. However, it can be solved more efficiently by a direct algorithm than by constructing G' and then finding a maximum matching.

9.6 Coverings in bipartite graphs

A *covering* or *vertex cover* of a graph G is a subset $U \subseteq V(G)$ that covers every edge of G; that is, every edge has at least one endpoint in U.

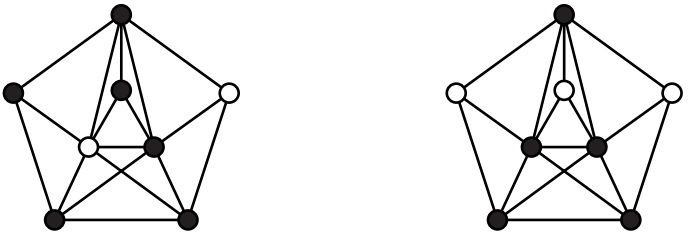


FIGURE 9.15
Coverings in a graph

In general, we want the smallest covering possible. This is called a *minimum covering*. Figure 9.15 shows two coverings, indicated by shaded vertices. The covering with six vertices is minimal; namely, it has no subset that is a smaller covering. The other is a minimum covering; namely, G has no smaller covering. This is because any covering must use at least three vertices of the outer 5-cycle, and at least two vertices of the inner triangle, giving a minimum of five vertices.

In bipartite graphs, there is a very close relation between minimum coverings and maximum matchings. In general, let M be a matching in a graph G, and let U be a covering. Then because U covers every edge of M, $|U| \geq |M|$. This is true even if U is minimum or if M is maximum. Therefore, we conclude that if $|U| = |M|$ for some M and U, then U is minimum and M is maximum. In bipartite graphs, equality can always be achieved.

Theorem 9.10. (König's theorem) *If G is bipartite, then the number of edges in a maximum matching equals the number of vertices in a minimum covering.*

Proof. Let M be a maximum matching, and let (X, Y) be a bipartition of G, where $|X| \leq |Y|$. Let $W \subseteq X$ be the set of all X-vertices not saturated by M. If $W = \emptyset$, then $U = X$ is a covering with $|U| = |M|$. Otherwise construct the set of all vertices reachable from W on alternating paths. Let S be the X-vertices reachable, and T the

Y-vertices reachable. Take $U = T \cup (X - S)$. Then U is a covering with $|U| = |M|$, as illustrated in Figure 9.16. □

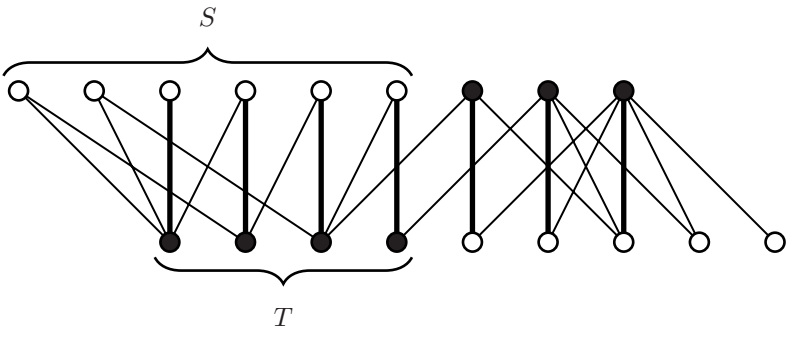


FIGURE 9.16
Minimum covering and maximum matching in a bipartite graph

9.7 Tutte's theorem

Tutte's theorem gives a necessary and sufficient condition for any graph to have a perfect matching.

Let $S \subseteq V(G)$. In general, $G - S$ may have several connected components. Write odd$(G - S)$ for the number of components with an odd number of vertices. The following proof of Tutte's theorem is due to LOVÁSZ [119].

Theorem 9.11. (Tutte's theorem) *A graph G has a perfect matching if and only if odd$(G - S) \leq |S|$, for every subset $S \subseteq V(G)$.*

Proof. Suppose that G has a perfect matching M and pick any $S \subseteq V(G)$. Let $G_1, G_2, \ldots, G_m$ be the odd components of $G - S$. Each G_i contains at least one vertex matched by M to a vertex of S. Therefore odd$(G - S) = m \leq |S|$. See Figure 9.17.

Conversely suppose that odd$(G - S) = m \leq |S|$, for every $S \subseteq V(G)$. Taking $S = \emptyset$ gives odd$(G) = 0$, so $n = |G|$ is even. The proof is by reverse induction on $\varepsilon(G)$, for any given n. If G is the complete graph, it is clear that G has a perfect matching, so the result holds when $\varepsilon = \binom{n}{2}$. Let G be a graph with the largest ε such that G has no perfect matching. If $uv \notin E(G)$, then because $G + uv$ has more edges than G, it must be that $G + uv$ has a perfect matching. Let S be the set of all vertices of G of degree $n - 1$, and let G' be any connected component of $G - S$. If G' is not a complete graph, then it contains three vertices x, y, z such that $x \longrightarrow y \longrightarrow z$, but

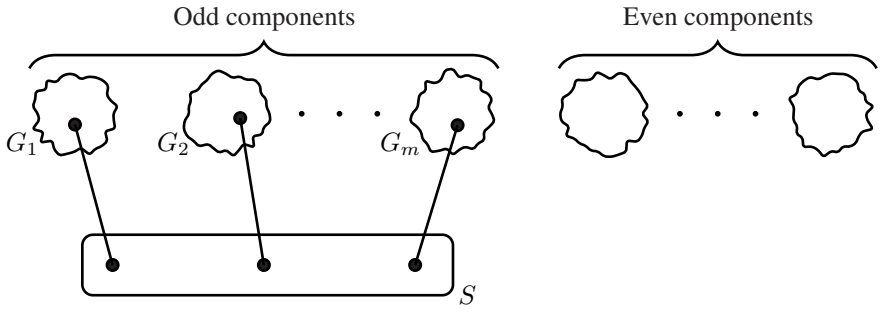


FIGURE 9.17
Odd and even components of $\boldsymbol{G - S}$

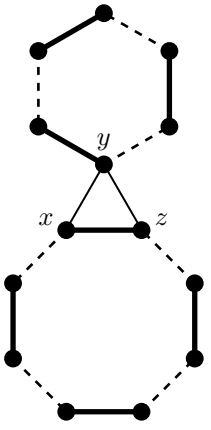


FIGURE 9.18
$\boldsymbol{H = M_1 \oplus M_2}$, case 1

$x \not\longrightarrow z$. Because $y \notin S$, $\deg(y) < n - 1$, so there is a vertex $w \not\longrightarrow y$. Let M_1 be a perfect matching of $G + xz$ and let M_2 be a perfect matching of $G + yw$, as shown in Figures 9.18 and 9.19. Then $xz \in M_1$ and $yw \in M_2$. Let $H = M_1 \oplus M_2$. H consists of one or more alternating cycles in G. Let C_{xz} be the cycle of H containing xz, and let C_{yw} be the cycle containing yw.

Case 1. $C_{xz} \neq C_{yw}$.

Form a new matching M by taking M_2-edges of C_{xz}, M_1-edges of C_{yw}, and M_1 edges elsewhere. Then M is a perfect matching of G, a contradiction.

Case 2. $C_{xz} = C_{yw} = C$.

C can be traversed in two possible directions. Beginning with the vertices y, w, we either come to x first or z first. Suppose it is z. Form a new matching M by

taking M_1-edges between w and z, M_2-edges between x and y, and the edge yz. Then take M_1 edges elsewhere. Again M is a perfect matching of G, a contradiction.

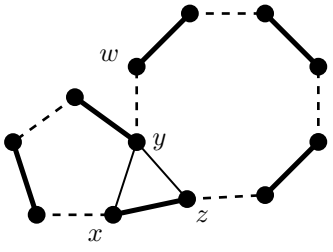


FIGURE 9.19
$\boldsymbol{H = M_1 \oplus M_2}$, case 2

We conclude that every component G' of $G - S$ must be a complete graph. But then we can easily construct a perfect matching of G as follows. Each even component of $G - S$ is a complete graph, so it has a perfect matching. Every odd component is also a complete graph, so is has a near perfect matching, namely, one vertex is not matched. This vertex can be matched to a vertex of S, because $\text{odd}(G - S) \leq |S|$. The remaining vertices of S form a complete subgraph, because they have degree $n - 1$, so they also have a perfect matching. It follows that every G satisfying the condition of the theorem has a perfect matching. □

Tutte's theorem is a powerful criterion for the existence of a perfect matching. For example, the following graph has no perfect matching, because $G - v$ has three odd components.

We can use Tutte's theorem to prove that every 3-regular graph G without cut-edges has a perfect matching. Let $S \subseteq V(G)$ be any subset of the vertices. Let $G_1, G_2, \ldots, G_k$ be the odd components of $G - S$. Let m_i be the number of edges connecting G_i to S. Then $m_i > 1$, because G has no cut-edge. Because $\sum_{v \in G_i} \text{DEG}(v) = 2\varepsilon(G_i) + m_i = 3|G_i| =$ an odd number, we conclude that m_i is odd. Therefore $m_i \geq 3$, for each i. But $\sum_{v \in S} \text{DEG}(v) = 3|S| \geq \sum_i m_i$, because all of the m_i edges have one endpoint in S. It follows that $3|S| \geq 3k$, or $|S| \geq \text{odd}(G - S)$, for all $S \subseteq V(G)$. Therefore G has a perfect matching M. G also has a 2-factor, because $G - M$ has degree two.

Exercises

9.7.1 For each integer $k > 1$, find a k-regular graph with no perfect matching.

9.7.2 A *near perfect matching* in a graph G is a matching which saturates all vertices of G but one. A *near 1-factorization* is a decomposition of $E(G)$ into near perfect matchings. Prove that K_{2n+1} has a near 1-factorization.

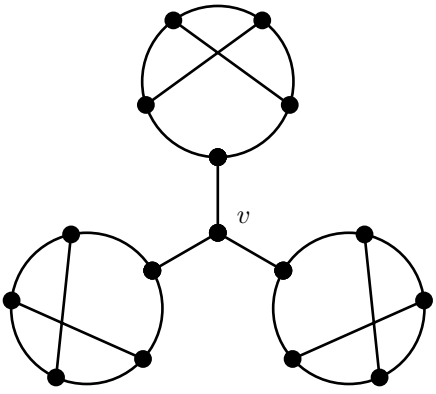


FIGURE 9.20
A 3-regular graph with no perfect matching

9.7.3 Find a condition similar to Tutte's theorem for a graph to have a near perfect matching.

9.8 Notes

An alternative description of Edmonds' algorithm appears in PAPADIMITRIOU and STEIGLITZ [134]. A good source book for the theory of matchings in graphs is LOVÁSZ and PLUMMER [120]. Exercise 7.1.2 is from BONDY and MURTY [23]. The proof of Tutte's theorem presented here is based on a proof by LOVÁSZ [119]. Tutte's transformation to reduce the subgraph problem to a perfect matching problem is from TUTTE [174]. His *Factor theorem*, TUTTE [175], is a solution to the subgraph problem. It is one of the great theorems of graph theory. The theory of 1-factorizations has important applications to the theory of combinatorial designs. A good reference is LINDNER and RODGER [116].

10

Network Flows

10.1 Introduction

A *network* is a directed graph used to model the distribution of goods, data, or commodities, etc., from their centers of production to their destinations. For example, Figure 10.1 shows a network in which goods are produced at node s, and shipped to node t. Each directed edge has a limited *capacity*, being the maximum number of goods that can be shipped through that channel per time period (e.g., 3 kilobytes per second or 3 truckloads per day). The diagram indicates the capacity as a positive integer associated with each edge. The actual number of goods shipped on each edge is shown in square brackets beside the capacity. This is called the *flow* on that edge. It is a non-negative integer less than or equal to the capacity. Goods cannot accumulate at any node; therefore, the total in-flow at each node must equal the out-flow at that node. The problem is to find the distribution of goods that maximizes the net flow from s to t.

This can be modeled mathematically as follows. When the edges of a graph have a direction, the graph is called a *directed graph* or *digraph*. A *network* N is a directed graph with two special nodes s and t; s is called the *source*, and t is called the *target*. All other vertices are called intermediate vertices. The edges of a directed graph are ordered pairs (u, v) of vertices, which we denote by $\overrightarrow{uv}$. We shall find it convenient to say that u is adjacent to v even when we do not know the direction of the edge. So the phrase u is adjacent to v means either $\overrightarrow{uv}$ or $\overrightarrow{vu}$ is an edge. Each edge $\overrightarrow{uv} \in E(N)$ has a *capacity* $\textsc{Cap}(\overrightarrow{uv})$, being a positive integer, and a flow $f(\overrightarrow{uv})$, a non-negative integer, such that $f(\overrightarrow{uv}) \leq \textsc{Cap}(\overrightarrow{uv})$. If v is any vertex of N, the *out-flow* at v is

$$f^+(v) = \sum_{u, v \longrightarrow u} f(\overrightarrow{vu})$$

where the sum is over all vertices u to which v is joined. The *in-flow* is the sum over all incoming edges at v

$$f^-(v) = \sum_{u, u \longrightarrow v} f(\overrightarrow{uv})$$

A valid flow f must satisfy two conditions.

1. Capacity constraint: $0 \leq f(\overrightarrow{uv}) \leq \textsc{Cap}(\overrightarrow{uv})$, for all $\overrightarrow{uv} \in E(N)$.

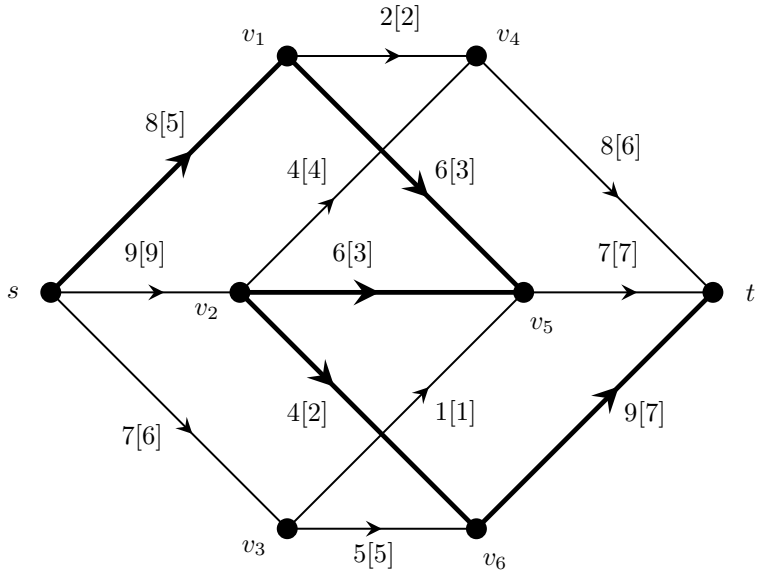


FIGURE 10.1
A network

2. Conservation condition: $f^+(v) = f^-(v)$, for all $v \neq s, t$.

Notice that in Figure 10.1 both these conditions are satisfied. The value of the flow is the net out-flow at s; in this case, $\text{VAL}(f) = 20$.

In general, there may be in-edges as well as out-edges at s. The net flow from s to t will then be the out-flow at the source minus the in-flow. This is called the *value* of the flow, $\text{VAL}(f) = f^+(s) - f^-(s)$. The max-flow problem is:

Problem 10.1:	Max-Flow
Instance:	a network N.
Find:	a flow f for N of maximum value.

Any flow f that has maximum value for the network N is called a *max-flow* of N. This problem was first formulated and solved by Ford and Fulkerson. In this chapter we shall present the Ford-Fulkerson algorithm, and study several applications of the max-flow-min-cut theorem.

It is possible that a network encountered in practice will have more than one source or target. If $s_1, s_2, \ldots, s_k$ are all sources in a network N, and $t_1, t_2, \ldots, t_m$ are all targets, we can replace N with a network N' with only one source and one target as follows. Add a vertex s to N, and join it to $s_1, s_2, \ldots, s_k$. Add a vertex t and join $t_1, t_2, \ldots, t_m$ to t. Assign a capacity $\text{CAP}(\overrightarrow{ss_i})$ being the sum of the capacities

of the out-edges at s_i, and a capacity $\textsc{Cap}(\overrightarrow{t_i t})$, being the sum of the capacities of all incoming edges to t_i. Call the resulting network N'. For every flow in N there is a corresponding flow in N' with equal value, and vice-versa. Henceforth we shall always assume that all networks have just one source and target. The model we are using assumes that edges are one-way channels and that goods can only be shipped in the direction of the arrow. If a two-way channel from u to v is desired, this can easily be accommodated by two directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$.

Let $S \subseteq V(N)$ be a subset of the vertices such that $s \in S$, $t \notin S$. Write $\overline{S} = V(N) - S$. Then $[S, \overline{S}]$ denotes the set of all edges of N directed from S to $\overline{S}$. See Figure 10.2. Consider the sum

$$\sum_{v \in S} (f^+(v) - f^-(v)). \tag{10.1}$$

Because $f^+(v) = f^-(v)$, if $v \neq s$, this sum equals $\textsc{Val}(f)$. On the other hand, $f^+(v)$ is the total out-flow at $v \in S$. Consider an out-edge $\overrightarrow{vu}$ at v. Its flow $f(\overrightarrow{vu})$ contributes to $f^+(v)$. It also contributes to $f^-(u)$. If $u \in S$, then $f(\overrightarrow{vu})$ will appear twice in the sum 10.1, once for $f^+(v)$ and once for $f^-(u)$, and will therefore cancel. See Figure 10.2, where S is the set of shaded vertices. If $u \notin S$, then $f(\overrightarrow{vu})$ will appear in the summation as part of $f^+(v)$, but will not be canceled by $f^-(u)$. A similar argument holds if $v \in \overline{S}$ and $u \in S$. Therefore

$$\begin{aligned} \textsc{Val}(f) &= \sum_{v \in S} (f^+(v) - f^-(v)) \\ &= \sum_{\overrightarrow{vu} \in [S, \overline{S}]} f(\overrightarrow{vu}) - \sum_{\overrightarrow{vu} \in [\overline{S}, S]} f(\overrightarrow{vu}) \end{aligned}$$

This says that the value of the flow can be measured across any edge-cut $[S, \overline{S}]$, such that $s \in S$ and $t \in \overline{S}$. If we write

$$f^+(S) = \sum_{v \in S} f^+(v)$$

and

$$f^-(S) = \sum_{v \in S} f^-(v)$$

then

$$\textsc{Val}(f) = f^+(S) - f^-(S).$$

If we write

$$f([S, \overline{S}]) = \sum_{\overrightarrow{vu} \in [S, \overline{S}]} f(\overrightarrow{vu})$$

and

$$f([\overline{S}, S]) = \sum_{\overrightarrow{vu} \in [\overline{S}, S]} f(\overrightarrow{vu})$$

then we can also express this as

$$\textsc{Val}(f) = f([S, \overline{S}]) - f([\overline{S}, S]).$$

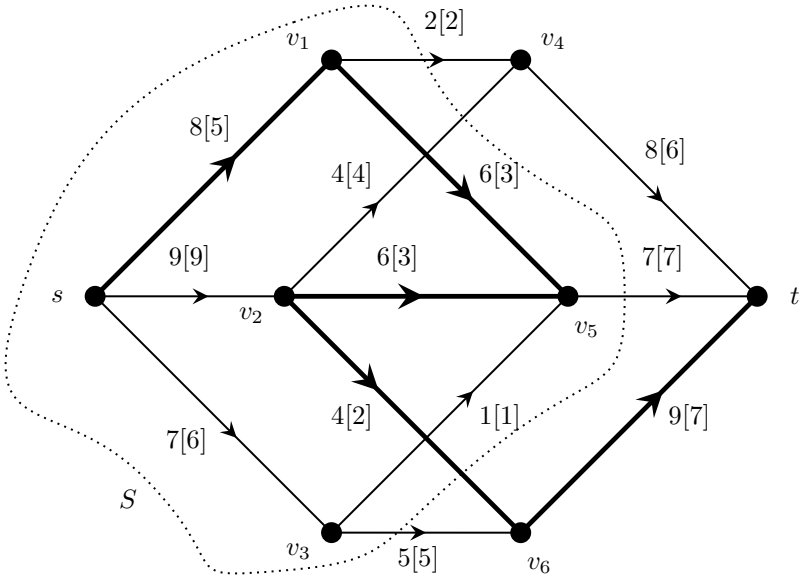


FIGURE 10.2
A set $\boldsymbol{S}$ where $\boldsymbol{s \in S, t \in \overline{S}}$

Let $K = [S, \overline{S}]$ be any edge-cut with $s \in S$ and $t \in \overline{S}$. The *capacity* of K is

$$\textsc{Cap}(K) = \sum_{\overrightarrow{uv} \in K} \textsc{Cap}(\overrightarrow{uv}).$$

This is the sum of the capacities of all edges out of S. The value of any flow in N is limited by the capacity of any edge-cut K. An edge-cut K is a *min-cut* if it has the minimum possible capacity of all edge-cuts in N.

Lemma 10.1. *Let $K = [S, \overline{S}]$ be an edge-cut in a network N with $s \in S$, $t \in \overline{S}$, and flow f. Then $\textsc{Val}(f) \leq \textsc{Cap}(K)$. If $\textsc{Val}(f) = \textsc{Cap}(K)$, then f is a max-flow and K is a min-cut.*

Proof. Clearly the maximum possible flow out of S is bounded by $\textsc{Cap}(K)$; that is, $f^+(S) \leq \textsc{Cap}(K)$. This holds even if K is a min-cut or f a max-flow. The flow into S is non-negative; that is, $f^-(S) \geq 0$. Therefore $\textsc{Val}(f) = f^+(S) - f^-(S) \leq \textsc{Cap}(K)$. If $\textsc{Val}(f) = \textsc{Cap}(K)$, then it must be that f is maximum, for the value of no flow can exceed the capacity of any cut. Similarly K must be a min-cut. Note

that in this situation $f^+(S) = \text{CAP}(K)$ and $f^-(S) = 0$. That is, every edge $\overrightarrow{uv}$ directed out of S satisfies $f(\overrightarrow{uv}) = \text{CAP}(\overrightarrow{uv})$. Every edge $\overrightarrow{uv}$ into S carries no flow, $f(\overrightarrow{uv}) = 0$. □

In the next section we shall prove the max-flow-min-cut theorem. This states that the value of a max-flow and the capacity of a min-cut are always equal, for any network N.

10.2 The Ford-Fulkerson algorithm

If we assign $f(\overrightarrow{uv}) = 0$, for all $\overrightarrow{uv} \in E(N)$, this defines a valid flow in N, the *zero flow*. The Ford-Fulkerson algorithm begins with the zero flow, and increments it through a number of iterations until a max-flow is obtained. The method uses augmenting paths. Consider the st-path $P = sv_1v_5v_2v_6t$ in Figure 10.1. (We ignore the direction of the edges when considering these paths.) Each edge of P carries a certain amount of flow. The traversal of P from s to t associates a direction with P. We can then distinguish two kinds of edges of P, *forward edges*, those like sv_1 whose direction is the same as that of P, and *backward edges*, those like v_5v_2 whose direction is opposite to that of P. Consider a forward edge $\overrightarrow{uv}$ in an st-path P. If $f(\overrightarrow{uv}) < \text{CAP}(\overrightarrow{uv})$, then $\overrightarrow{uv}$ can carry more flow. Define the *residual capacity* of $uv \in E(P)$ to be

$$\text{RESCAP}(uv) = \begin{cases} \text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv}), & \text{if } uv \text{ is a forward edge,} \\ f(\overrightarrow{vu}), & \text{if } uv \text{ is a backward edge.} \end{cases}$$

The residual capacity of a forward edge $uv \in E(P)$ is the maximum amount by which the flow on $\overrightarrow{uv}$ can be increased. The residual capacity of a backward edge $uv \in E(P)$ is the maximum amount by which the flow on $\overrightarrow{vu}$ can be decreased. For example, in the network of Figure 10.1, we increase the flow on all forward edges of P by 2, and decrease the flow on all backward edges of P also by 2. The result is shown in Figure 10.2. We have a new flow, with a larger value than in Figure 10.1.

In general, let P be an st-path in a network N with flow f. Define the *residual capacity* of P to be

$$\delta(P) = \text{MIN}\{\text{RESCAP}(\overrightarrow{uv}) : uv \in E(P)\}$$

Define a new flow f^* in N as follows:

$$f^*(\overrightarrow{uv}) = \begin{cases} f(\overrightarrow{uv}), & \text{if } uv \text{ is not an edge of } P, \\ f(\overrightarrow{uv}) + \delta(P), & \text{if } uv \text{ is a forward edge of } P, \text{ and} \\ f(\overrightarrow{uv}) - \delta(P), & \text{if } uv \text{ is a backward edge of } P. \end{cases}$$

Lemma 10.2. *f^* is a valid flow in N and* $\text{VAL}(f^*) = \text{VAL}(f) + \delta(P)$.

Proof. We must check that the capacity constraint and conservation conditions are both satisfied by f^*. It is clear that the capacity constraint is satisfied, because of the definition of the residual capacity of P as the *minimum* residual capacity of all edges in P. To verify the conservation condition, consider any intermediate vertex v of P. Let its adjacent vertices on P be u and w, so that uv and vw are consecutive edges of P. There are four cases, shown in Figure 10.3.

Case 1. uv and vw are both forward edges of P.

Because $f(\overrightarrow{uv})$ and $f(\overrightarrow{vw})$ both increase by $\delta(P)$ in f^*, it follows that $f^+(v)$ and $f^-(v)$ both increase $\delta(P)$. The net result on $f^+(v) - f^-(v)$ is zero.

Case 2. uv is a forward edge and vw is a backward edge.

In this case $f(\overrightarrow{uv})$ increases and $f(\overrightarrow{wv})$ decreases by $\delta(P)$ in f^*. It follows that $f^+(v)$ and $f^-(v)$ are both unchanged.

Case 3. uv is a backward edge and vw is a forward edge.

In this case $f(\overrightarrow{vu})$ decreases and $f(\overrightarrow{vw})$ increases by $\delta(P)$ in f^*. It follows that $f^+(v)$ and $f^-(v)$ are both unchanged.

Case 4. uv and vw are both backward edges of P.

Because $f(\overrightarrow{vu})$ and $f(\overrightarrow{wv})$ both decrease by $\delta(P)$ in f^*, it follows that $f^+(v)$ and $f^-(v)$ both decrease by $\delta(P)$. The net result on $f^+(v) - f^-(v)$ is zero.

The value of f^* is $f^{*+}(s) - f^{*-}(s)$. If the first edge of P is su, a forward edge, then it is clear that the value increases by $\delta(P)$, because $f(\overrightarrow{su})$ increases. If su is a backward edge, then $f(\overrightarrow{su})$ decreases, so that $f^-(s)$ also decreases, thereby increasing the value of the flow. Therefore $\text{VAL}(f^*) = \text{VAL}(f) + \delta(P)$. □

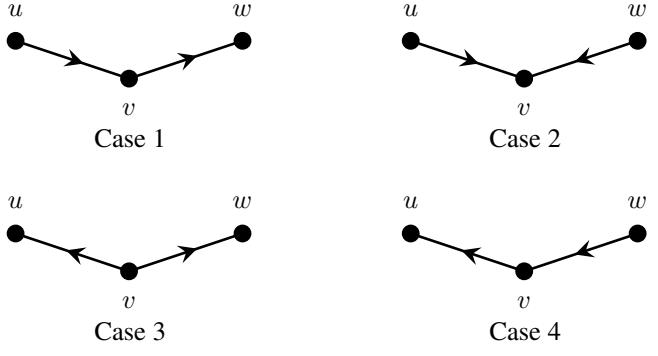


FIGURE 10.3
The four cases for edges uv and vw on path P

DEFINITION 10.1: An st-path P for which $\delta(P) > 0$ is called an *augmenting path*.

This method of altering the flow on the edges of P is called *augmenting the flow*. If $\delta(P) > 0$ it always results in a flow of larger value. We give an outline of the Ford-Fulkerson algorithm in Algorithm 10.2.1.

Algorithm 10.2.1: FF(N, s, t)

comment: $\begin{cases} N \text{ is a network with source } s \text{ and target } t. \\ f \text{ is the flow.} \\ P \text{ is a path.} \end{cases}$

$f \leftarrow$ the zero flow
search for an augmenting path P
while a path P was found
do $\begin{cases} \text{augment the flow on } P \\ \textsc{Val}(f) \leftarrow \textsc{Val}(f) + \delta(P) \\ \text{search for an augmenting path } P \end{cases}$
comment: the flow is now maximum

The algorithm stops when N does not contain an augmenting path. We show that in this situation the flow must be maximum. The outline given in Algorithm 10.2.1 does not specify how the augmenting paths are to be found. Among the possibilities are the breadth-first and depth-first searches. We shall see later that the breadth-first search is the better choice. As the algorithm searches for an augmenting path, it will construct paths from s to various intermediate vertices v. The paths must have positive residual capacity. An sv-path with positive residual capacity is said to be *unsaturated*. A vertex v is *s-reachable* if N contains an unsaturated sv-path. This means that v can be reached from s on an unsaturated path.

Theorem 10.3. *Let N be a network with a flow f. Then f is maximum if and only if N contains no augmenting path.*

Proof. Suppose that f is a max-flow. There can be no augmenting path in N, for this would imply a flow of larger value. Conversely, suppose that f is a flow for which there is no augmenting path. We show that f is maximum. Let S denote the set of all s-reachable vertices of N. Clearly $s \in S$. Because there is no augmenting path, the target is not s-reachable. Therefore $t \in \overline{S}$. Consider the edge-cut $K = [S, \overline{S}]$. If $\overrightarrow{uv} \in K$ is an edge out of S, then $\textsc{ResCap}(\overrightarrow{uv}) = 0$; for otherwise v would be s-reachable on the forward edge $\overrightarrow{uv}$ from $u \in S$. Therefore $f(\overrightarrow{uv}) = \textsc{Cap}(\overrightarrow{uv})$ for all $\overrightarrow{uv} \in K$ that are out edges of S. Thus $f^+(S) = \textsc{Cap}(K)$. If $\overrightarrow{uv} \in [\overline{S}, S]$ is any edge into S, then $f(\overrightarrow{uv}) = 0$; for otherwise u would be s-reachable on the backward edge $\overrightarrow{vu}$ from $v \in S$. Consequently $f^-(S) = 0$. It follows that $\textsc{Val}(f) = \textsc{Cap}(K)$, so that f is a max-flow and K a min-cut, by Lemma 10.1. □

This is illustrated in Figure 10.2, in which all edges out of S are saturated. In this example there are no edges into S. If there were, they would carry no flow. So the flow in Figure 10.2 is maximum. Notice that a consequence of this theorem is that when f is a max-flow, the set S of s-reachable vertices defines a min-cut $K = [S, \overline{S}]$. This is summarized as follows:

Theorem 10.4. (Max-flow-min-cut theorem) *In any network the value of a max-flow equals the capacity of a min-cut.*

We are now ready to present the Ford-Fulkerson algorithm as a breadth-first search for an augmenting path. The vertices will be stored on a queue, the *ScanQ*, an array of s-reachable vertices. *QSize* is the current number of vertices on the *ScanQ*. The unsaturated sv-paths will be stored by an array *PrevPt*$[\cdot]$, where *PrevPt*$[v]$ is the point previous to v on an sv-path P_v. The residual capacity of the paths will be stored by an array *ResCap*$[\cdot]$, where *ResCap*$[v]$ is the residual capacity $\delta(P_v)$ of P_v from s up to v. The algorithm is presented as a single procedure, but could be divided into smaller procedures for modularity and readability.

The procedure which augments the flow starts at t and follows *PrevPt*$[v]$ up to s. Given an edge $\overrightarrow{uv}$ on the augmenting path, where $u =$ *PrevPt*$[v]$, a means is needed of determining whether $\overrightarrow{uv}$ is a forward or backward edge. One way is to store *PrevPt*$[v] = u$ for forward edges and *PrevPt*$[v] = -u$ for backward edges. This is not indicated in Algorithm 10.2.2, but can easily be implemented.

In programming the max-flow algorithm, the network N should be stored in adjacency lists. This allows the loop

for all v adjacent to u **do**

to be programmed efficiently. The out-edges and in-edges at u should all be stored in the same list. We need to be able to distinguish whether $u \longrightarrow v$ or $v \longrightarrow u$. This can be flagged in the record representing edge $\overrightarrow{uv}$. If $\overrightarrow{uv}$ appears as an out-edge in the list for node u, it will appear as an in-edge in the list for vertex v. When the flow on edge $\overrightarrow{uv}$ is augmented, it must be augmented from both endpoints. One way to augment from both endpoints simultaneously is to store not the flow $f(\overrightarrow{uv})$ itself, but a pointer to it. Then it is not necessary to find the other endpoint of the edge. Thus a node x in the adjacency list for vertex u contains following four fields:

- *AdjPt*$\langle x \rangle$, a vertex v that is adjacent to or from u.
- *OutEdge*$\langle x \rangle$, a boolean variable set to **true** if $u \longrightarrow v$, and **false** if $v \longrightarrow u$.
- *Flow*$\langle x \rangle$, a *pointer* to the flow on $\overrightarrow{uv}$.
- *Next*$\langle x \rangle$, the next node in the adjacency list for u.

Algorithm 10.2.2: MAXFLOW(N, s, t)

$f \leftarrow$ the zero flow
for all vertices v **do** *PrevPt*$[v] \leftarrow 0$
while true "search for an augmenting path"
do
- *ScanQ*$[1] \leftarrow s$
- *QSize* $\leftarrow 1$
- $k \leftarrow 1$
- **repeat**
  - $u \leftarrow$ *ScanQ*$[k]$ "the k^{th} vertex on *ScanQ*"
  - **for all** v adjacent to u
    - **do if** $v \notin$ *ScanQ*
      - **if** $u \longrightarrow v$
        - **then**
          - **comment:** a forward edge
          - **if** CAP$(\overrightarrow{uv}) > f(\overrightarrow{uv})$ **then**
            - *QSize* $\leftarrow QSize + 1$
            - *ScanQ*$[$*QSize*$] \leftarrow v$
            - *PrevPt*$[v] \leftarrow u$
            - **if** *ResCap*$[u] <$ CAP$(\overrightarrow{uv}) - f(\overrightarrow{uv})$
              - **then** *ResCap*$[v] \leftarrow$ *ResCap*$[u]$
              - **else** *ResCap*$[v] \leftarrow$ CAP$(\overrightarrow{uv}) - f(\overrightarrow{uv})$
            - **if** $v = t$ **then go to** 1
        - **then**
          - **comment:** $v \longrightarrow u$, a backward edge
          - **if** $f(\overrightarrow{vu}) > 0$ **then**
            - *QSize* $\leftarrow QSize + 1$
            - *ScanQ*$[$*QSize*$] \leftarrow v$
            - *PrevPt*$[v] \leftarrow u$
            - **if** *ResCap*$[u] < f(\overrightarrow{vu})$
              - **then** *ResCap*$[v] \leftarrow$ *ResCap*$[u]$
              - **else** *ResCap*$[v] \leftarrow f(\overrightarrow{vu})$
            - **if** $v = t$ **then go to** 1
  - $k \leftarrow k + 1$ "advance *ScanQ*"
  - **until** $k >$ *QSize*
- **comment:** Flow is now maximum. *ScanQ* contains the s-reachable vertices.
- **output** (*ScanQ*, and the flow on each edge)
- **exit**
- 1 : **comment:** augmenting path found, re-initialize *ScanQ*
- AUGMENTFLOW(t)
- **for** $k \leftarrow 1$ **to** *QSize* **do** *PrevPt*$[$*ScanQ*$[k]] \leftarrow 0$

Algorithm 10.2.3: AUGMENTFLOW(t)

$v \leftarrow t$
$u \leftarrow$ *PrevPt*$[v]$
$\delta \leftarrow$ *ResCap*$[t]$
while $u \neq 0$
do $\begin{cases} \textbf{if } u \longrightarrow v \\ \quad \textbf{then } f(\overrightarrow{uv}) \leftarrow f(\overrightarrow{uv}) + \delta & \text{“a forward edge”} \\ \quad \textbf{else } f(\overrightarrow{uv}) \leftarrow f(\overrightarrow{uv}) - \delta & \text{“a backward edge”} \\ v \leftarrow u \\ u \leftarrow \textit{PrevPt}[v] \end{cases}$
VAL$(f) \leftarrow$ VAL$(f) + \delta$

This breadth-first search version of the Ford-Fulkerson algorithm is sometimes referred to as the “labeling” algorithm in some books. The values *ResCap*$[v]$ and *PrevPt*$[v]$ are considered the labels of vertex v.

The algorithm works by constructing all shortest unsaturated paths from s. If an augmenting path exists, it is sure to be found. This can easily be proved by induction on the length of the shortest augmenting path. The flow is then augmented and the algorithm exits from the inner repeat loop by branching to statement 1. If no augmenting path exists, then the inner repeat loop will terminate. The vertices on the *ScanQ* will contain the set S of all s-reachable vertices, such that $[S, \overline{S}]$ is a min-cut.

It is difficult to form an accurate estimate of the complexity of the BF-FF algorithm. We shall prove that it is polynomial. This depends on the fact that only *shortest* augmenting paths are used. If non-shortest paths are used, the FF algorithm is not always polynomial. Consider the network of Figure 10.4. We augment first on path $P = (s, a, b, t)$, which has residual capacity one. We then augment on path $Q = (s, b, a, t)$, also of residual capacity one, because ba is a backward edge, and $f(\overrightarrow{ab}) = 1$. Augmenting on Q makes $\delta(P) = 1$, so we again augment on P, and then augment again on Q, etc. After 2000 iterations a max-flow is achieved – *the number of iterations can depend on the value of the max-flow*. This is not polynomial in the parameters of the network. However, if shortest augmenting paths are used, this problem does not occur.

Consider an augmenting path P in a network N. $\delta(P)$ is the minimum residual capacity of all edges of P. Any edge $\overrightarrow{uv} \in P$ such that RESCAP$(\overrightarrow{uv}) = \delta(P)$ is called a *bottleneck*. Every augmenting path has at least one bottleneck and may have several. Suppose that a max-flow in N is reached in m iterations, and let P_j be the augmenting path on iteration j. Let $d_j(s, u)$ denote the length of a shortest unsaturated su-path in iteration j, for all vertices u.

Lemma 10.5. *$d_{j+1}(s, u) \geq d_j(s, u)$, for all $u \in V(N)$.*

Proof. Let $Q_j = Q_j(s, u)$ be a shortest unsaturated su-path at the beginning of iteration j, and let $Q_{j+1} = Q_{j+1}(s, u)$ be a shortest unsaturated su-path at the beginning of iteration $j + 1$. Then $\ell(Q_j) = d_j(s, u)$ and $\ell(Q_{j+1}) = d_{j+1}(s, u)$. If

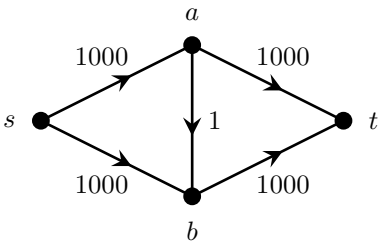


FIGURE 10.4
A max-flow in 2000 iterations

$\ell(Q_j) \leq \ell(Q_{j+1})$, the lemma holds for vertex u, so suppose that $\ell(Q_j) > \ell(Q_{j+1})$, for some u. Now Q_{j+1} is not unsaturated at the beginning of iteration j, so it must become unsaturated during iteration j. Therefore P_j and Q_{j+1} have at least one edge in common that becomes unsaturated during iteration j. The proof is by induction on the number of such edges. Suppose first that xy is the only such edge in common. Because xy becomes unsaturated during iteration j, it has opposite direction on P_j and Q_{j+1}. See Figure 10.5. Without loss of generality, $Q_{j+1}[s, x]$ and $Q_{j+1}[y, u]$ are unsaturated on iteration j. Because P_j is a shortest path, $\ell(P_j[s, x]) \leq \ell(Q_{j+1}[s, x])$. But then $P_j[s, y]Q_{j+1}[y, u]$ is an unsaturated su-path on iteration j, and has length less than $Q_{j+1}(s, u)$, which in turn has length less than $Q_j(s, u)$, a contradiction. If $P_j[s, y]$ intersects $Q_{j+1}[y, u]$ at a vertex z, then $P_j[s, z]Q_{j+1}[z, u]$ is an even shorter unsaturated path.

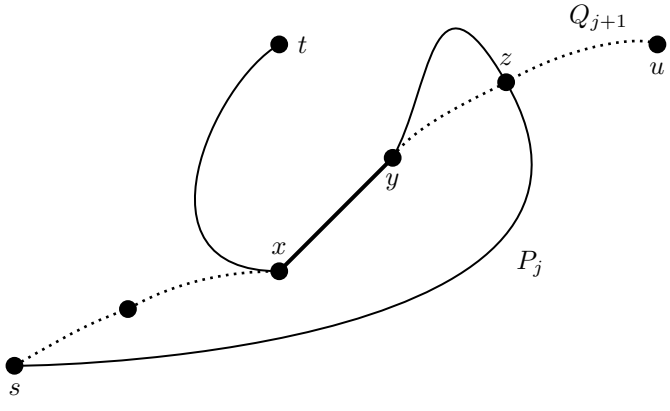


FIGURE 10.5
Paths $\boldsymbol{P_j}$ and $\boldsymbol{Q_{j+1}}$

Suppose now that P_j and Q_{j+1} have more than one edge in common that becomes unsaturated during iteration j. Let xy be the first such edge on Q_{j+1} traveling from s to u. Let z be the point on Q_{j+1} nearest to u that $P_j[s,x]$ contacts before reaching x (maybe $z = y$). Then $Q_{j+1}[s,x]$ and $P_j[s,z]$ are unsaturated at the beginning of iteration j. Because P_j is a shortest unsaturated path, $\ell(P_j[s,z]) < \ell(Q_{j+1}[s,x]) < \ell(Q_{j+1}[s,z])$. Now either $Q_{j+1}[z,u]$ is unsaturated on iteration j, or else it has another edge in common with P_j. If it is unsaturated, then $P_j[s,z]Q_{j+1}[z,u]$ is an unsaturated su-path that contradicts the assumption that $d_j(s,u) > d_{j+1}(s,u)$. If there is another edge in common with P_j, then we can repeat this argument. Let $x'y'$ be the first edge of $Q_{j+1}[z,u]$ in common with P_j. Let z' be the point on $Q_{j+1}[z,u]$ nearest to u that $P_j[s,x']$ contacts before reaching x', etc. Proceeding in this way we eventually obtain an su-path that is shorter than Q_j and unsaturated on iteration j, a contradiction. □

It follows that $d_{j+1}(s,t) \geq d_j(s,t)$, for every iteration j. By constructing unsaturated paths from t in a backward direction we can similarly prove that $d_{j+1}(u,t) \geq d_j(u,t)$, for all vertices u. If we can now prove that $d_{j+1}(s,t) > d_j(s,t)$, then we can bound the number of iterations, because the maximum possible distance from s to t is $n-1$, where n is the number of vertices of N.

Theorem 10.6. *The breadth-first Ford-Fulkerson algorithm requires at most* $\frac{1}{2}n\varepsilon+1$ *iterations.*

Proof. On each iteration some edge is a bottleneck. After $\varepsilon+1$ iterations, some edge has been a bottleneck twice, because there are only ε edges. Consider an edge $\overrightarrow{uv}$ which is a bottleneck on iteration i and then later a bottleneck on iteration j. Refer to Figure 10.6.

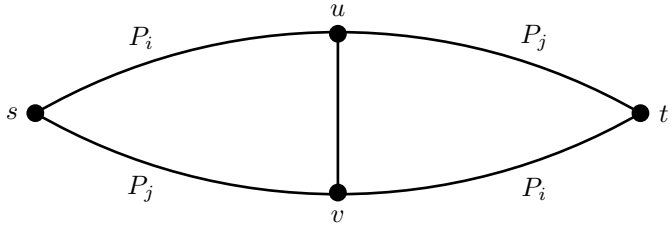


FIGURE 10.6
Paths P_i and P_j

Then $d_i(s,t) = d_i(s,u) + d_i(v,t) + 1$ and $d_j(s,t) = d_j(s,v) + d_j(u,t) + 1$. But $d_i(s,u) \leq d_j(s,u) = d_j(s,v) + 1$ and $d_i(v,t) \leq d_j(v,t) = d_j(u,t) + 1$. Therefore $d_i(s,u) + d_i(v,t) \leq d_j(s,v) + d_j(u,t) + 2$. It follows that $d_i(s,t) \leq d_j(s,t) + 2$. Each time an edge is repeated as a bottleneck, the distance from s to t increases by at least two. Originally $d_1(s,t) \geq 1$. After $\varepsilon + 1$ iterations, some edge has been a bottleneck twice. Therefore $d_{\varepsilon+1}(s,t) \geq 3$. Similarly $d_{2\varepsilon+1}(s,t) \geq 5$, and so on. In general $d_{k\varepsilon+1}(s,t) \geq 2k+1$. Because the maximum distance from s to t is $n-1$,

we have $2k + 1 \leq n - 1$, so that $k \leq n/2$. The maximum number of iterations is then $k\varepsilon + 1 \leq \frac{1}{2}n\varepsilon + 1$. □

Each iteration of the BF-FF algorithm is a breadth-first search for an augmenting path. A breadth-first search takes at most $O(\varepsilon)$ steps. Because the number of iterations is at most $\frac{1}{2}n\varepsilon + 1$, this gives a complexity of $O(n\varepsilon^2)$ for the breadth-first Ford-Fulkerson algorithm. This was first proved by EDMONDS and KARP [48].

Exercises

10.2.1 Find a max-flow in the network shown in Figure 10.7. Prove your flow is maximum by illustrating a min-cut K such that $\text{VAL}(f) = \text{CAP}(K)$.

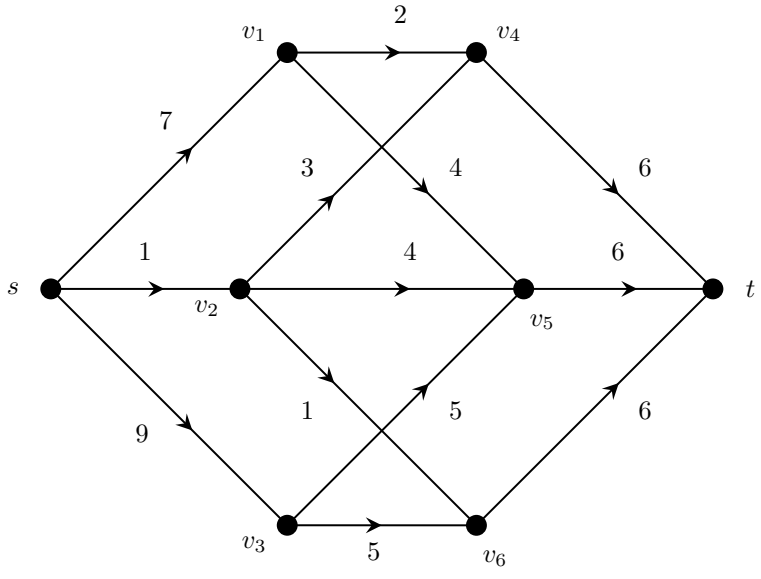


FIGURE 10.7
A network

10.2.2 Show that if there is no directed st-path in a network N, then the maximum flow in N has value zero. Can there be a flow whose value is negative? Explain.

10.2.3 Explain why $\sum_{v \in S} f^+(v)$ and $\sum_{\overrightarrow{vu} \in [S,\overline{S}]} f(\overrightarrow{vu})$ are in general, not equal.

10.2.4 Consider the network N of Figure 10.7 with flow f defined as follows: $f(sv_1) = 6$, $f(sv_2) = 0$, $f(sv_3) = 2$, $f(v_1v_4) = 2$, $f(v_1v_5) = 4$, $f(v_2v_4) = 0$, $f(v_2v_6) = 0$, $f(v_3v_5) = 2$, $f(v_3v_6) = 0$, $f(v_5v_2) = 0$, $f(v_4t) = 2$, $f(v_5t) = 6$, $f(v_6t) = 0$. A breadth-first search of N will construct the subnetwork of all shortest, unsaturated paths in N. This

subnetwork is called the *auxiliary network*, $\textsc{Aux}(N, f)$. A forward edge $\overrightarrow{uv}$ of N is replaced by a forward edge $\overrightarrow{uv}$ with capacity $\textsc{Cap}(\overrightarrow{uv}) - f(\overrightarrow{uv})$ in the auxiliary network. A backward edge $\overrightarrow{vu}$ of N is replaced by a forward edge $\overrightarrow{uv}$ with capacity $f(\overrightarrow{uv})$ in $\textsc{Aux}(N, f)$. Initially the flow in $\textsc{Aux}(N, f)$ is the zero flow. Construct the auxiliary network for the graph shown. Find a max-flow in $\textsc{Aux}(N, f)$, and modify f in N accordingly. Finally, construct the new auxiliary network for N.

10.2.5 Program the breadth-first Ford-Fulkerson algorithm. Test it on the networks of this chapter.

10.2.6 If $[S, \overline{S}]$ and $[T, \overline{T}]$ are min-cuts in a network N, show that $[S \cup T, \overline{S \cup T}]$ and $[S \cap T, \overline{S \cap T}]$ are also min-cuts. (*Hint*: Write $S = S_1 \cup (S \cap T)$ and $T = T_1 \cup (S \cap T)$ and use the fact that $[S, \overline{S}]$ and $[T, \overline{T}]$ are both min-cuts.)

10.2.7 Describe a maximum flow algorithm similar to the Ford-Fulkerson algorithm which begins at t and constructs unsaturated paths P until s is reached. Given that P is a ts-path, how should the residual capacity of an edge be defined in this case?

10.2.8 Describe an algorithm for finding an edge $\overrightarrow{uv}$ in a network N such that the value of a max-flow f in N can be increased if $\textsc{Cap}(\overrightarrow{uv})$ is increased. Prove that your algorithm is correct and find its complexity. Does there always exist such an edge $\overrightarrow{uv}$? Explain.

10.3 Matchings and flows

There is a marked similarity between matching theory and flow theory:

Matchings: A matching M in a graph G is maximum if and only if G contains no augmenting path.

Flows: A flow f in a network N is maximum if and only if N contains no augmenting path.

Hungarian algorithm: Construct alternating paths until an augmenting path is found.

Ford-Fulkerson algorithm: Construct unsaturated paths until an augmenting path is found.

The reason for this is that matching problems can be transformed into flow problems. Consider a bipartite graph G, with bipartition (X, Y) for which a max-matching is desired. Direct all the edges of G from X to Y, and assign them a capacity of one. Add a source s and an edge sx for all $x \in X$, with $\textsc{Cap}(\overrightarrow{sx}) = 1$. Add a

target t and an edge yt for all $y \in Y$, with $\text{CAP}(\overrightarrow{yt}) = 1$. Call the resulting network N. This is illustrated in Figure 10.8. Now find a max-flow in N. The flow-carrying edges of $[X, Y]$ will determine a max-matching in G. Because $\text{CAP}(\overrightarrow{sx}) = 1$, there will be at most one flow-carrying edge out of each $x \in X$. Because $\text{CAP}(\overrightarrow{yt}) = 1$, there will be at most one flow-carrying edge into y, for each $y \in Y$. The flow-carrying edges of N are called the *support of the flow*. An alternating path in G and an unsaturated path in N can be seen to be the same thing. If G is not bipartite there is no longer a direct correspondence between matchings and flows. However, it is possible to construct a special kind of *balanced network* such that a maximum balanced flow corresponds to a max-matching (see KOCAY and STONE [110] or FREMUTH-PAEGER and JUNGNICKEL [55]).

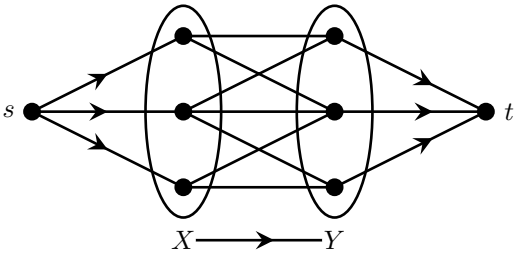


FIGURE 10.8
Matchings and flows

The basic BF-FF algorithm can be improved substantially. As it is presented here, it constructs a breadth-first network of all shortest unsaturated paths until t is reached. At this point, f is augmented, and the process is repeated. There may be many augmenting paths available at the point when t is reached, but only one augmenting path is used. The remaining unsaturated paths which have been built are discarded, and a new BFS is executed. In order to improve the BF-FF algorithm, one possibility is to construct the set of *all* shortest unsaturated paths. This is the auxiliary network of Exercise 10.2.4. We then augment on as many paths as possible in the auxiliary network before executing a new BFS. This has the effect of making $d_{j+1}(s,t) > d_j(s,t)$ so that the number of iterations is at most n. Several algorithms are based on this strategy. They improve the complexity of the algorithm markedly. See the book by PAPADIMITRIOU and STEIGLITZ [134] for further information.

Exercises

10.3.1 Let G be a bipartite graph with bipartition (X, Y). We want to find a subgraph H of G such that in H, $\text{DEG}(x) = b(x)$ and $\text{DEG}(y) = b(y)$, where $b(v)$ is a given non-negative integer, for all $v \in V(G)$, if there exists such an H. For example, if $b(v) = 1$ for all v, then H would be a perfect matching. If $b(v) = 2$ for all v, then H would be a 2-factor.

Show how to construct a network N such that a max-flow in N solves this problem.

10.3.2 Let N be a network such that every vertex $v \in N$ has a maximum *throughput* $t(v)$ defined. This is the maximum amount of flow that is allowed to pass through v; that is, $f^-(v) \leq t(v)$ must hold at all times. Show how to solve this problem by constructing a network N' such that a max-flow in N' defines a max-flow in N with maximum throughput as given.

10.4 Menger's theorems

Given any digraph, we can view it as a network N by assigning unit capacities to all edges. Given any two vertices $s, t \in V(N)$, we can compute a max-flow f from s to t. If $\text{VAL}(f) = 0$, then there are no directed paths from s to t, because a directed st-path would be an augmenting path. If $\text{VAL}(f) = 1$, then N contains a directed st-path P; however, there are no directed st-paths which are edge-disjoint from P, for such a path would be an augmenting path. In general, the value of a max-flow f in N is the maximum number of edge-disjoint directed st-paths in N. Suppose that $\text{VAL}(f) = k \geq 1$. The support of f defines a subgraph of N that contains at least one directed st-path P. Delete the edges of P to get N' and let f' be obtained from f by ignoring the edges of P. Then $\text{VAL}(f') = k - 1$, and this must be a max-flow in N', because f is a max-flow in N. By induction, the number of edge-disjoint directed st-paths in N' is $k - 1$, from which it follows that the number in N is k.

A min-cut in N can also be interpreted as a special subgraph of N. Let $K = [S, \overline{S}]$ be a min-cut in N, where $s \in S$ and $t \in \overline{S}$. If $\text{CAP}(K) = 0$, there are no edges out of S, so there are no directed st-paths in N. If $\text{CAP}(K) = 1$, there is only one edge out of S. The deletion of this edge will destroy all directed st-paths in N. We say that s is disconnected from t. In general, $\text{CAP}(K)$ equals the minimum number of edges whose deletion destroys all directed st-paths in N. Suppose that $\text{CAP}(K) = k \geq 1$. Delete the edges of K to get a network N'. Then in N', $\text{CAP}([S, \overline{S}]) = 0$, so that N' contains no directed st-paths. Thus the deletion of the edges of K from N destroys all directed st-paths. Because N contains k edge-disjoint such paths, it is not possible to delete fewer than k edges in order to disconnect s from t. The max-flow-min-cut theorem now gives the first of Menger's theorems.

Theorem 10.7. *Let s and t be vertices of a directed graph N. Then the maximum number of edge-disjoint directed st-paths equals the minimum number of edges whose deletion disconnects s from t.*

Recall that an undirected graph G is k-edge-connected if the deletion of fewer than k edges will not disconnect G. In Chapter 6 we showed that a graph is 2-edge-connected if and only if every pair of vertices is connected by at least two

edge-disjoint paths. We will use Theorem 10.7 to prove a similar result for k-edge-connected graphs. In order to convert Theorem 10.7 to undirected graphs, we can replace each edge $\overrightarrow{uv}$ of G by a "gadget", as shown in Figure 10.9, to get a directed graph N.

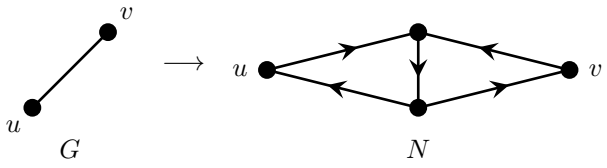


FIGURE 10.9
A gadget for edge-disjoint paths

The gadget contains a directed $\overrightarrow{uv}$-path and a directed $\overrightarrow{vu}$-path, but they both use the central edge of the gadget. Let $s, t \in V(G)$. Then edge-disjoint st-paths of G will define edge-disjoint directed st-paths in N. Conversely, edge-disjoint directed st-paths in N will define edge-disjoint st-paths in G. This gives another of Menger's theorems.

Theorem 10.8. *Let s and t be vertices of an undirected graph G. Then the maximum number of edge-disjoint st-paths equals the minimum number of edges whose deletion disconnects s from t.*

It follows that a graph G is k-edge-connected if and only if every pair s, t of vertices are connected by at least k edge-disjoint paths. This immediately gives an algorithm to compute $\kappa'(G)$, the edge-connectivity of G. Number the vertices of G from 1 to n. Let the corresponding vertices of N also be numbered from 1 to n. The algorithm computes the minimum max-flow over all pairs s, t of vertices. This is the minimum number of edges whose deletion will disconnect G. Exactly $\binom{n}{2}$ max-flows are computed, so the algorithm has polynomial complexity.

Algorithm 10.4.1: EDGE-CONNECTIVITY(G)

```
convert G to a directed graph N
κ' ← n
for s ← 1 to n − 1
do { for t ← s + 1 to n
     do { M ← MAXFLOW(N, s, t)
          if M < κ'
            then κ' ← M
return (κ')
```

Exercises

10.4.1 Let G be an undirected graph. Replace each edge uv of G with a pair of directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ to get a directed graph N. Let $s, t \in V(G)$. Show that the maximum number of edge-disjoint st-paths in G equals the maximum number of edge-disjoint directed st-paths in N.

10.4.2 Program the edge-connectivity algorithm, using the transformation of Exercise 10.4.1.

10.5 Disjoint paths and separating sets

Recall that paths in a graph G are internally disjoint if they can intersect only at their endpoints. A graph is k-connected if the deletion of fewer than k vertices will not disconnect it. We proved in Chapter 7 that G is 2-connected if and only if every pair of vertices is connected by at least two internally disjoint paths. We prove a similar result for k-connected graphs by utilizing a relation between internally disjoint paths in G and directed paths in a network N. We first make two copies u_1, u_2 of each vertex u of G. $V(N) = \{u_1, u_2 \mid u \in V(G)\}$. Let $\overrightarrow{uv}$ be an edge of G. N will contain the edges (u_1, u_2), (v_1, v_2), (u_2, v_1), and (v_2, u_1). This is illustrated in Figure 10.10.

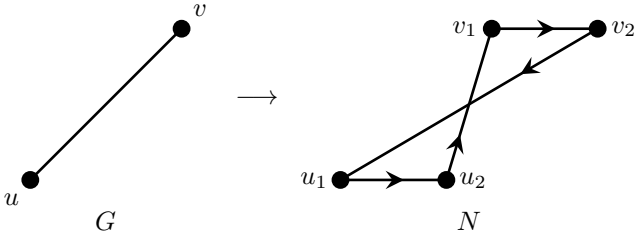


FIGURE 10.10
A gadget for internally disjoint paths

Let $u \in V(G)$. Notice the following observations:

1. The only out-edge at u_1 is u_1u_2.
2. The only in-edge at u_2 is u_1u_2.
3. The edge $\overrightarrow{uv} \in E(G)$ corresponds to u_2v_1 and v_2u_1 in N.

Consequently any st-path $suvw \dots t$ in G corresponds to an s_2t_1-path $s_2u_1u_2v_1v_2w_1w_2 \dots t_1$ in N. Internally disjoint st-paths in G give rise to internally disjoint s_2t_1-paths in N. On the other hand, edge-disjoint paths in N are in fact internally disjoint because of items 1 and 2. Therefore the maximum number of internally disjoint st-paths in G equals the maximum number of edge-disjoint directed

s_2t_1-paths in N. This in turn equals the minimum number of edges whose deletion will disconnect s_2 from t_1. If $s_2 \not\longrightarrow t_1$, then every s_2t_1-path in G will contain another vertex, say u_1 or u_2. By observations 1 and 2 , it must contain both u_1 and u_2. Deleting u_1u_2 will destroy this path. If $K = [S, \overline{S}]$ is a min-cut in N, then the edges out of S must be of the form u_1u_2, because u_2 can only be s_2-reachable if u_1 is, by observation 3 and Figure 10.10. Let $U = \{u \mid u_1u_2 \in K\}$. Then U is a set of vertices of G which separate s from t. This gives another of Menger's theorems.

Theorem 10.9. *Let G be a graph and let $s, t \in V(G)$, where $s \not\longrightarrow t$. Then the maximum number of internally disjoint st-paths in G equals the minimum number of vertices whose deletion separates s from t.*

Theorem 10.10. *A graph G is k-connected if and only if every pair of vertices is connected by at least k internally disjoint paths.*

Proof. Let $s, t \in V(G)$. If s and t are connected by at least k internally disjoint paths, then clearly G is k-connected; for at least k vertices must be deleted to disconnect s from t. Conversely suppose that G is k-connected. Then deleting fewer than k vertices will not disconnect G. If $s \not\longrightarrow t$, then by the Theorem 10.10, G must contain at least k internally disjoint st-paths. If $s \longrightarrow t$, then consider $G - st$. It is easy to see that $G - st$ is $(k-1)$-connected. Therefore in $G - st$, there are at least $k - 1$ internally disjoint st-paths. The edge st is another st-path, giving k paths in total. □

We can also use this theorem to devise an algorithm which computes $\kappa(G)$, the connectivity of G. We suppose that the vertices of G are numbered 1 to n, and that if $s \in V(G)$, then s_1 and s_2 are the corresponding vertices of N.

Algorithm 10.5.1: CONNECTIVITY(G)

```
convert G to a directed graph N
κ ← n − 1        "maximum possible connectivity"
s ← 0
while s < κ
  do { s ← s + 1        "vertex s"
       for t ← s + 1 to n
         do if s ↛ t
              then { M ← MAXFLOW(N, s2, t1)
                     if M < κ
                       then κ ← M
                     if s > κ
                       then return (κ) }
return (κ)
```

Theorem 10.11. *Algorithm 10.5.1 computes $\kappa(G)$*

Proof. Suppose first that $G = K_n$. Then $\kappa(G) = n - 1$. The algorithm will not call MAXFLOW() at all, because every s is adjacent to every t. The algorithm will terminate with $\kappa = n - 1$. Otherwise G is not complete, so there exists a subset $U \subseteq V(G)$ such that $G - U$ has at least two components, where $|U| = \kappa(G)$. The first $\kappa(G)$ choices of vertex s may all be in U. However, by the $(\kappa(G) + 1)^{\text{st}}$ choice of s we know that some $s \notin U$ has been selected. So s is in some component of $G - U$. The inner loop runs over all choices of t. One of these choices will be in a different component of U. For that particular t, the value of MAXFLOW(N, s_2, t_1) will equal $|U|$. After this, the value of κ in the algorithm will not decrease any more. Therefore we can conclude that some $s \notin U$ will be selected; that the value of κ after that point will equal $\kappa(G)$; and that after this point the algorithm can stop. This is exactly what the algorithm executes. □

The algorithm makes at most

$$\sum_{s=1}^{\kappa+1}(n - s)$$

calls to MAXFLOW(). Thus, it is a polynomial algorithm.

Exercises

10.5.1 Let G be k-connected. If $st \in E(G)$, prove that $G - st$ is $(k - 1)$-connected.

10.5.2 Program Algorithm 10.5.1, the CONNECTIVITY() algorithm.

10.5.3 Consider a network N where instead of specifying a capacity for each edge $\overrightarrow{uv}$, we specify a *lower bound* $b(\overrightarrow{uv}) \geq 0$ for the flow on edge $\overrightarrow{uv}$. Instead of the capacity constraint $f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$, we now have a lower bound constraint $f(\overrightarrow{uv}) \geq b(\overrightarrow{uv})$. The zero-flow is not a valid flow anymore. Show that there exists a valid flow in such a network if and only if for every edge $\overrightarrow{uv}$ such that $b(\overrightarrow{uv}) > 0$, $\overrightarrow{uv}$ is either: (i) on a directed st-path; or (ii) on a directed ts-path; or (iii) on a directed cycle. (*Hint*: If $\overrightarrow{uv}$ is not on such a path or cycle, follow directed paths forward from v and backward from u to get a contradiction.)

10.5.4 Consider the problem of finding a *minimum* flow in a network with lower bounds instead of capacities.

(a) How should an unsaturated path be defined?

(b) How should the capacity of an edge-cut be defined?

(c) Find a min-flow in the network of Figure 10.11, where the numbers are the lower bounds. Prove that your flow is minimum by illustrating an appropriate edge-cut.

(d) Is there a max-flow in the network given in Figure 10.11?

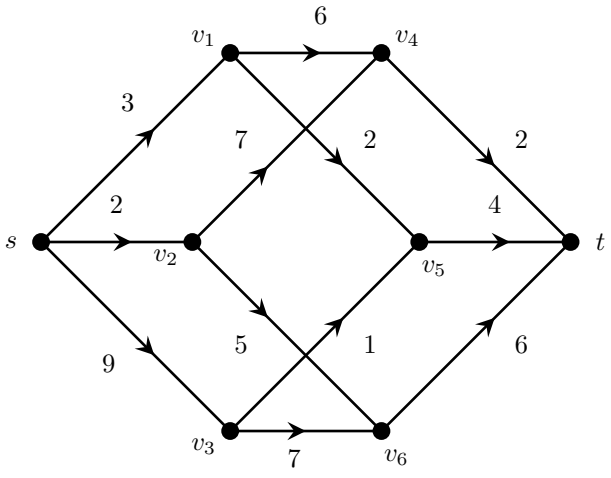


FIGURE 10.11
A network with lower bounds

10.5.5 Suppose that a network N has both lower bounds $b(\overrightarrow{uv})$ and capacities $\text{CAP}(\overrightarrow{uv})$ on its edges. We wish to find a max-flow f of N, where $b(\overrightarrow{uv}) \leq f(\overrightarrow{uv}) \leq \text{CAP}(\overrightarrow{uv})$. Notice that zero-flow may be no longer a valid flow. Before applying an augmenting path algorithm like the FF algorithm, we must first find a valid flow.

(a) Determine whether the networks N_1 and N_2 of Figure 10.12 have a valid flow.

(b) How should residual capacity be defined?

(c) How should the capacity of an edge-cut be defined?

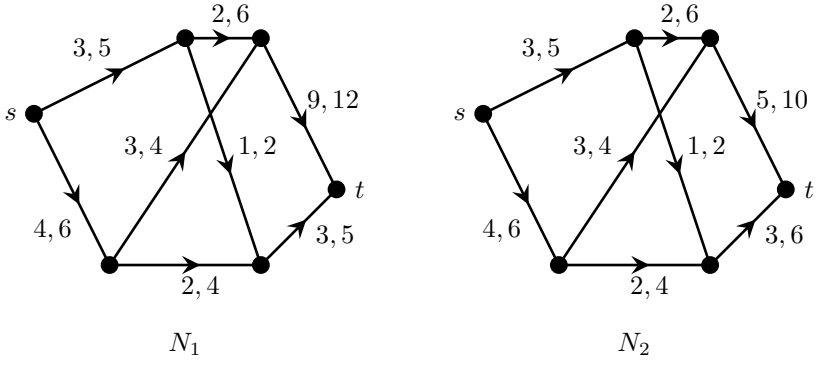


FIGURE 10.12
Networks with lower bounds and capacities

10.5.6 Let N be a network with lower bounds $b(\overrightarrow{uv})$ and capacities $\textsc{Cap}(\overrightarrow{uv})$ specified on its edges. Before finding a max-flow in N we need to find a valid flow. Construct a network N' as follows: Add a new source s' and target t'. Join s' to all vertices of N. Join every vertex of N to t'. Add edges $\overrightarrow{st}$ and $\overrightarrow{ts}$ to N'. The capacities in N' are defined as follows:

$$\textsc{Cap}'(\overrightarrow{s'u}) = \textstyle\sum_v b(\overrightarrow{vu}), \text{ (sum over in-edges at } u \in V(N)).$$
$$\textsc{Cap}'(\overrightarrow{ut'}) = \textstyle\sum_v b(\overrightarrow{uv}), \text{ (sum over out-edges at } u \in V(N)).$$
$$\textsc{Cap}'(\overrightarrow{uv}) = \textsc{Cap}(\overrightarrow{uv}) - b(\overrightarrow{uv}), (u, v \in V(N)).$$
$$\textsc{Cap}'(\overrightarrow{st}) = \textsc{Cap}'(\overrightarrow{ts}) = \infty.$$

Prove that there exists a valid flow in N if and only if there is a flow in N' that saturates all edges incident on s'.

10.5.7 Let N be a network such that there is a cost $c(\overrightarrow{uv})$ of using edge $\overrightarrow{uv}$, per unit of flow. Thus the cost of flow $f(\overrightarrow{uv})$ on edge $\overrightarrow{uv}$ is $f(\overrightarrow{uv})c(\overrightarrow{uv})$. Devise an algorithm to find a max-flow of min-cost in N.

10.5.8 The circulation of money in the economy closely resembles a flow in a network. Each node in the economy represents a person or organization that takes part in economic activity. The main differences are that there may be no limit to the capacities of the edges, and that flow may accumulate at a node if assets are growing. Any transfer of funds is represented by a flow on some edge. Various nodes of the economic network can be organized into groups, such as banks, insurance companies, wage earners, shareholders, government, employers, etc.

(a) Develop a simplified model of the economy along these lines.

(b) A bank charges interest on its loans. If there is a fixed amount of money in the economy, what does this imply? What can you conclude about the money supply in the economy?

(c) When a new business is created, a new node is added to the network. Where does the flow through this node come from?

(d) Consider the node represented by government. Where does its in-flow come from? Where is its out-flow directed? Consider how government savings bonds operate in the model.

(e) Where does inflation fit into this model?

(f) How do shareholders and the stock market fit into the model?

10.6 Notes

The max-flow algorithm is one of the most important algorithms in graph theory, with a great many applications to other graph theory problems (such as connectivity and Menger's theorems), and to problems in discrete optimization. The original algorithm is from FORD and FULKERSON [53]. See also FULKERSON [58]. EDMONDS and KARP [48] proved that the use of shortest augmenting paths results in a polynomial time complexity of the Ford-Fulkerson algorithm.

Balanced flows were introduced by KOCAY and STONE [110], and then developed greatly in a series of papers by Fremuth-Paeger and Jungnickel. An excellent summary with many references can be found in FREMUTH-PAEGER and JUNGNICKEL [55].

A great many techniques have been developed to improve the complexity of the basic augmenting path algorithm. See PAPADIMITRIOU and STEIGLITZ [134] for further information. The algorithms to find the connectivity and edge-connectivity of a graph in Sections 8.4 and 8.5 are from EVEN [50].

11

Hamilton Cycles

11.1 Introduction

A cycle that contains all vertices of a graph G is called a *hamilton cycle* (or *hamiltonian cycle*). G is *hamiltonian* if it contains a hamilton cycle. For example, Figure 11.1 shows a hamilton cycle in the graph called the *truncated tetrahedron*. It is easy to see that the graph of the *cube* is also hamiltonian (see Chapter 1).

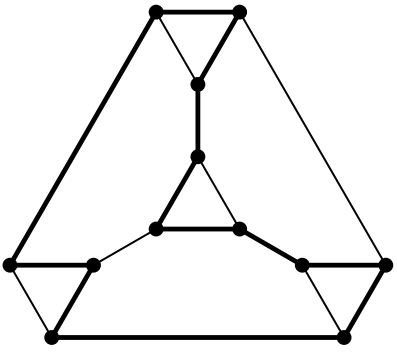


FIGURE 11.1
A hamiltonian graph

Figure 11.2 shows a non-hamiltonian graph H. It is easy to see that H is non-hamiltonian, because it is bipartite with an odd number of vertices. Clearly any bipartite graph that is hamiltonian must have an even number of vertices, because a hamilton cycle C must start and end on the same side of the bipartition. Although H is non-hamiltonian, it does have a *hamilton path*, that is, a path containing all its vertices.

The problem of deciding whether a given graph is hamiltonian is only partly solved.

Problem 11.1:	HamCycle
Instance:	a graph G
Question:	is G hamiltonian?

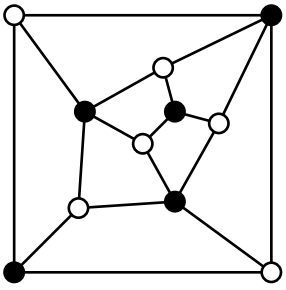


FIGURE 11.2
A non-hamiltonian graph

This is an example of an NP-*complete* problem. We will say more about NP-complete problems later. There is no known efficient algorithm for solving the HamCycle problem. Exhaustive search algorithms can take a very long time in general. Randomized algorithms can often find a cycle quickly if G is hamiltonian, but do not give a definite answer if no cycle is found.

The HamCycle problem is qualitatively different from most other problems in this book. For example, the questions "is G bipartite, Eulerian, 2-connected, planar?", and, "is a given flow f maximum?" can all be solved by efficient algorithms. In each case an algorithm and a theoretical solution are available. For the HamCycle problem, there is no efficient algorithm known, and only a partial theoretical solution. A great many graph theoretical problems are NP-complete.

A number of techniques do exist which can help to determine whether a given graph is hamiltonian. A graph with a cut-vertex v cannot possibly be hamiltonian, because a hamilton cycle C has no cut-vertex. This idea can be generalized into a helpful lemma.

Lemma 11.1. *If G is hamiltonian, and $S \subseteq V(G)$, then $\omega(G - S) \leq |S|$.*

This lemma says that if we delete $k = |S|$ vertices from G, the number of connected components remaining is at most k. Let C be a hamilton cycle in G. If we delete k vertices from C, the cycle C will be decomposed into at most k paths. Because C is a subgraph of G, it follows that $G - S$ will have at most k components.

For example, the graph of Figure 11.3 is non-hamiltonian, because the deletion of the three back vertices gives four components.

The Petersen graph is also non-hamiltonian, but this cannot be proved using Lemma 11.1. Instead we use an exhaustive search method called the *multi-path method*, see RUBIN [152]. Suppose that C were a hamilton cycle in the Petersen graph G, as shown in Figure 11.4. G is composed of an outer and inner pentagon, joined by a perfect matching. Because C uses exactly two edges at each vertex of G, it follows that C must use at least three edges of the outer pentagon, for otherwise some vertex on it would be missed by C. Consequently, C uses two adjacent edges of the outer pentagon. Without loss of generality, suppose that it uses the edges uv

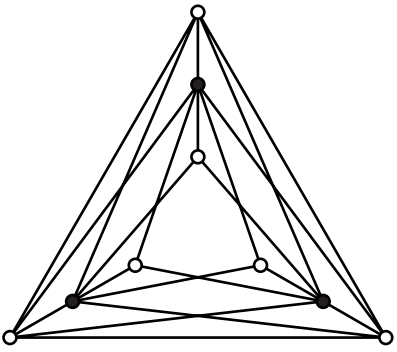


FIGURE 11.3
A non-hamiltonian graph

and vw. This means that C does not use the edge vy, so we can delete it from G. Deleting vy reduces the degree of y to two, so that now both remaining edges at y must be part of C. So the two paths (u, v, w) and (x, y, z) must be part of C, where a path is denoted by a sequence of vertices. This is illustrated in Figure 11.4.

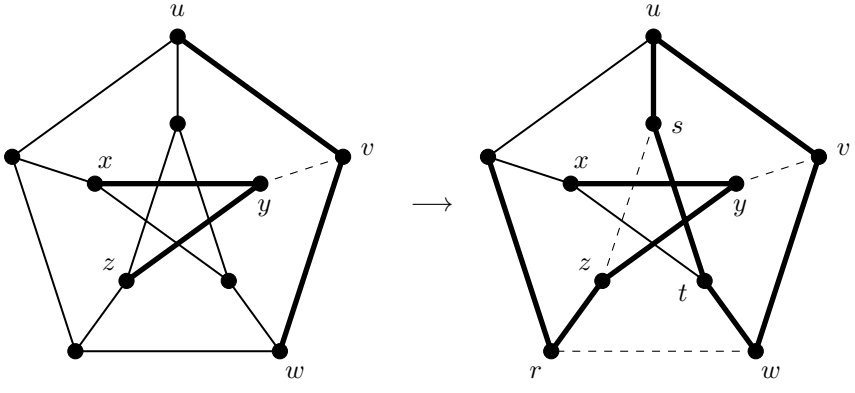


FIGURE 11.4
The multi-path method

C must use two edges at w, so there are two cases. Either $wt \in C$ or $wr \in C$. Suppose first that $wt \in C$. Then because $wr \notin C$, we can delete wr from G. This reduces the degree of r to two, so that the remaining edges at r must be in C. Therefore $rz \in C$. This uses up two edges at z, so we delete sz, which in turn reduces the degree of s to two. Consequently the edge $us \in C$. But this now creates a cycle (u, v, w, t, s) in C, which is not possible. It follows that the choice $wt \in C$

was wrong. If we now try $wr \in C$ instead, a contradiction is again reached, thereby proving that the Petersen graph is non-hamiltonian.

This is called the multi-path method, because the cycle C is gradually built from a number of paths which are forced by two operations: the deletion of edges which are known not to be in C; and the requirement that both edges at a vertex of degree two be in C. The multi-path method is very effective in testing whether 3-regular graphs are hamiltonian, because each time an edge is deleted, the degree of two vertices reduces to two, which then forces some of the structure of C. Graphs of degree four or more are not so readily tested by it. We will say more about the multi-path method later on.

Exercises

11.1.1 Decide whether or not the graphs in Figure 11.5 are hamiltonian.

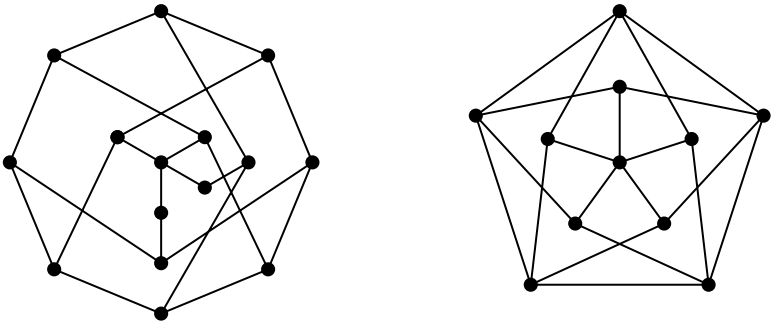


FIGURE 11.5
Are these hamiltonian?

11.1.2 Prove that Q_n, the n-cube, is hamiltonian for all $n \geq 2$.

11.1.3 Let P be a hamilton path in a graph G, with endpoints u and v. Show that $\omega(G - S) \leq |S| + 1$, for all $S \subseteq V(G)$, in two ways:

a) By counting the components of $G - S$.

b) By counting the components of $(G + uv) - S$, and using Lemma 9.1.

11.2 The crossover algorithm

Suppose that we want to find a hamilton cycle in a connected graph G. Because every vertex of G must be part of C, we select any vertex x. We then try to build a long path P starting from x. Initially $P = (x)$ is a path of length zero. Now execute the following steps:

$u \leftarrow x; \quad v \leftarrow x \quad$ "P is a uv-path"
while $\exists w \longrightarrow u$ such that $w \notin P$
do $\begin{cases} P \leftarrow P + uw \\ u \leftarrow w \end{cases}$

while $\exists w \longrightarrow v$ such that $w \notin P$
do $\begin{cases} P \leftarrow P + vw \\ v \leftarrow w \end{cases}$

The first loop extends P from u and the second loop extends P from v, until it cannot be extended anymore. At this point we have a uv-path

$$P = (u, \ldots, x, \ldots, v)$$

such that the endpoints u and v are adjacent only to vertices of P. The length of P is $\ell(P)$, the number of edges in P. The vertices of P are ordered from u to v. If $w \in P$, then w^+ indicates the vertex following w (if $w \neq v$). Similarly w^- indicates the vertex preceding w (if $w \neq u$).

If $u \longrightarrow v$, then we have a cycle $C = P + uv$. If C is a hamilton cycle, we are done. Otherwise, because G is connected, there is a vertex $w \in P$ such that $w \longrightarrow y$, where $y \notin P$. Hence there exists a longer path

$$P^* = P - ww^+ + wy.$$

This is illustrated in Figure 11.6.

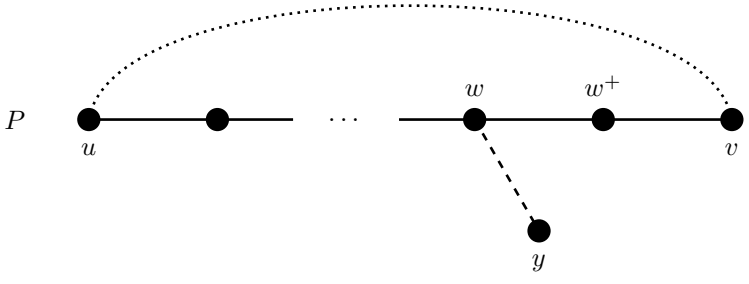


FIGURE 11.6
Finding a long path

If $u \not\longrightarrow v$, it may still be possible to find a cycle. Suppose that P contains a vertex w such that $v \longrightarrow w$ and $u \longrightarrow w^+$. This creates a pattern called a *crossover*, which is shown in Figure 11.7. When a crossover exists, there is a cycle

$$C = P + vw - ww^+ + uw^+$$

containing all the vertices of P.

Having converted the path P to a cycle C using the crossover, we are again in the situation where either C is a hamilton cycle, or else it contains a vertex $w \longrightarrow y \notin C$, which allows us to find a longer path P^*. We now extend P^* from both endpoints as far as possible, and then look for a crossover again, etc. The algorithm terminates either with a hamilton cycle, or with a long path that has no crossover. The crossover algorithm is summarized in Algorithm 11.2.1.

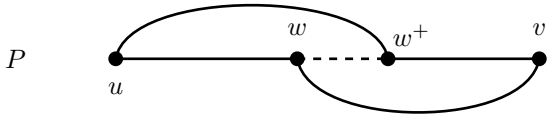


FIGURE 11.7
A crossover

Algorithm 11.2.1: LONGPATH(G, x)

comment: { Find a long path in G containing x, using crossovers.
P and C are linked lists.

$u \leftarrow x$; $v \leftarrow x$; $P \leftarrow (x)$ "a path of length 0"
repeat
comment: extend P from u
while $\exists w \longrightarrow u$ such that $w \notin P$
do { add w to P
$u \leftarrow w$
comment: extend P from v
while $\exists w \longrightarrow v$ such that $w \notin P$
do { add w to P
$v \leftarrow w$
comment: search for a crossover
for all $w \longrightarrow v$ **do if** $u \longrightarrow w^+$
then { **comment:** a crossover has been found
$C \leftarrow P + vw - ww^+ + uw^+$
if C is a hamilton cycle
then go to 1
find $z \in C$ such that $z \longrightarrow y \notin C$
convert $C + zy$ into a path P from y to z^+
$u \leftarrow y$; $v \leftarrow z^+$
until no crossover was found
1 : **comment:** P can be extended no more

11.2.1 Complexity

The main operations involved in the algorithm are extending P from u and v, converting P to a cycle C, and finding $z \in C$ such that $z \longrightarrow y \notin C$. We assume that the data structures are arranged so that the algorithm can check whether or not a vertex w is on P in constant time. This is easy to do with a boolean array. We also assume that the algorithm can test whether or not vertices v and w are adjacent in constant time.

- Extending P from u requires at most $\text{DEG}(u)$ steps, for each u. Because P can extend at most once for each u, the total number of steps taken to extend P is at most $\sum_u \text{DEG}(u) = 2\varepsilon$, taken over all iterations of the algorithm.
- Converting P to a cycle $C = P + vw - ww^+ + uw^+$ requires reversing a portion of P. This can take up to $\ell(P)$ steps. As $\ell(P)$ increases from 0 up to its maximum, this can require at most $O(n^2)$ steps, taken over all iterations.
- Checking whether $z \in C$ is adjacent to some $y \notin C$ requires $\text{DEG}(z)$ steps for each z. There are $\ell(C) = \ell(P) + 1$ vertices z to be considered. If at some point in the algorithm it is discovered that some z is not adjacent to any such y, we need never test that z again. We flag these vertices to avoid testing them twice. Thus the total number of steps spent looking for z and y is at most $O(n^2) + \sum_z \text{DEG}(z) = O(n^2 + \varepsilon)$.

So the total complexity of the algorithm is $O(n^2 + \varepsilon)$. More sophisticated data structures can reduce the $O(n^2)$ term, but there is likely no reason to do so, because the algorithm is already fast, and it is not guaranteed to find a hamilton cycle in any case.

The crossover algorithm works very well on graphs which have a large number of edges compared to the number of vertices. In some cases we can prove that it will always find a hamilton cycle. On sparse graphs (e.g., 3-regular graphs), it does not perform very well when the number of vertices is more than 30 or so.

Lemma 11.2. *Let G be a graph on n vertices such that $\text{DEG}(u) + \text{DEG}(v) \geq n$, for all non-adjacent vertices u and v. Then the crossover algorithm will always find a hamilton cycle in G.*

Proof. If the crossover algorithm does not find a hamilton cycle, let P be the last path found. Because P cannot be extended from its endpoints u and v, it follows that u and v are joined only to vertices of P. For each $w \longrightarrow v$, it must be that $u \not\longrightarrow w^+$, or a crossover would exist. Now v is joined to $\text{DEG}(v)$ vertices of P. There are thus $\text{DEG}(v)$ vertices that u is not joined to. Consequently u can be adjacent to at most $\ell(P) - \text{DEG}(v)$ vertices, where $\ell(P) \leq n - 1$ is the number of edges of P. So we have

$$\text{DEG}(u) + \text{DEG}(v) \leq \ell(P) \leq n - 1,$$

a contradiction, because we assumed that $\text{DEG}(u)+\text{DEG}(v) \geq n$ for all non-adjacent u and v. □

This lemma also shows that graphs which satisfy the condition $\text{DEG}(u) + \text{DEG}(v) \geq n$ are always hamiltonian. Such graphs have many edges, as we shall see. However, the crossover algorithm will often find hamilton cycles or hamilton paths, even when a graph does not satisfy this condition.

The crossover algorithm can be improved enormously by searching for crossovers of higher order. The crossover of Figure 11.7 can be defined to be the trail $Q = (u, w^+, w, v)$ which starts at u, intersects P in exactly one edge, and finishes at v. The cycle C is then given by $C = P \oplus Q$, where $\oplus$ indicates the operation of exclusive-OR, applied to the edges of P and Q. In general, higher order crossovers can be defined as follows.

DEFINITION 11.1: Let P be a uv-path. A *crossover* Q is a uv-trail such that $V(Q) \subseteq V(P)$ and $C = P \oplus Q$ is a cycle with $V(C) = V(P)$. The *order* of a crossover Q is the number $|P \cap Q|$ of edges common to P and Q. A *cross-edge* is any edge $xy \in E(Q) - E(P)$.

So a crossover of order 0 occurs when $u \longrightarrow v$. Then $Q = (u, v)$ and $C = P + uv$. There is only one kind of crossover of order one, which is shown in Figure 11.7. A crossover of order two is illustrated in Figure 11.8. There are five different kinds of crossovers of order two, as the reader can verify by constructing them. An algorithm employing crossovers of order higher than one requires a recursive search for crossovers up to a pre-selected maximum order M. It was found by KOCAY and LI [108] that choosing $M = 6$ still gives a fast algorithm, and that it improves the performance of the basic algorithm enormously. This algorithm requires sophisticated data structures for an efficient implementation.

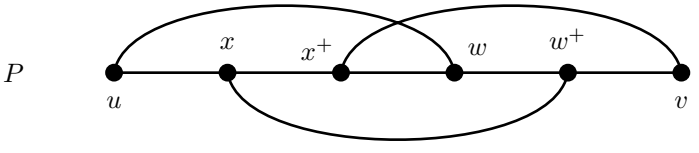


FIGURE 11.8
A crossover $\boldsymbol{Q = (u, w, w^+, x, x^+, v)}$ of order two

Suppose that a path P is the longest path found by the algorithm, and that it has no crossover. If there is a vertex $x \notin P$ such that $x \to w, w^+$, for some $w \in P$, then we can make a longer path by re-routing P through x: $P' = (\ldots, w, x, w^+, \ldots)$. Similarly, a configuration like Figure 11.9 can also be used to give a longer path. Once P has been re-routed to a longer path, we can again check for a crossover. When used in combination, crossovers and re-routings will very often find a hamilton cycle in G, if it is hamiltonian, even for sparse graphs G.

A re-routing is very much like a crossover. It is a closed trail Q whose endpoints are on the uv-path P, such that $P \oplus Q$ is a uv-path containing all vertices of P. It al-

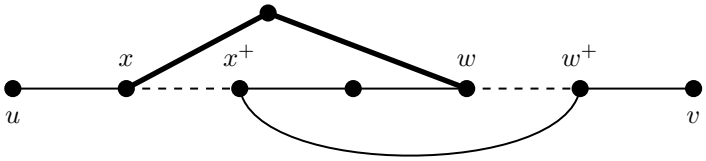


FIGURE 11.9
Re-routing P

ways results in a longer path. The algorithm that searches for higher order crossovers can be easily modified to search for re-routings as well.

Exercises

11.2.1 Show that if G is connected and $n > 2\delta$, where δ is the minimum degree of G, then G has a path of length at least 2δ. This is due to DIRAC [45]. (*Hint*: Consider a longest path.)

11.2.2 Program the crossover algorithm, and test it on the Petersen graph, on the graphs of Figure 11.5, and on the graph in Figure 11.10. Try it from several different starting vertices.

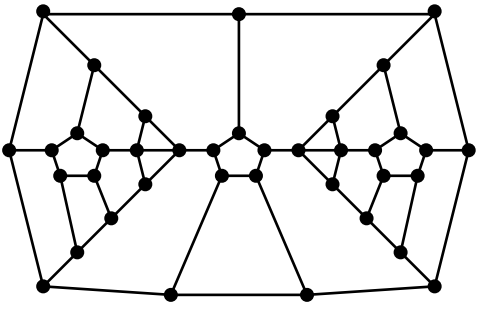


FIGURE 11.10
The Lederberg graph

11.2.3 Let G be a graph. Show how to create a graph G' from G by adding one vertex so that G has a hamilton path if and only if G' has a hamilton cycle.

11.2.4 Let G be a graph such that $\text{DEG}(u)+\text{DEG}(v) \geq n-1$, for all non-adjacent vertices u and v. Show that G has a hamilton path.

11.2.5 Construct all five kinds of crossovers of order two.

11.2.6 Construct the crossovers of order three.

11.3 The Hamilton closure

Suppose that $\text{DEG}(u) + \text{DEG}(v) \geq n$ in a graph G, where u and v are non-adjacent vertices. Let $G' = G + uv$. If G is hamiltonian, then so is G'. Conversely, if G' is hamiltonian, let C be a hamilton cycle in G'. If $uv \in C$, then $P = C - uv$ is a hamilton path in G. Because $\text{DEG}(u) + \text{DEG}(v) \geq n$, we know that P has a crossover, so that G has a hamilton cycle, too. Thus we have proved:

Lemma 11.3. *Let $\text{DEG}(u) + \text{DEG}(v) \geq n$ in a graph G, for non-adjacent vertices u and v. Let $G' = G + uv$. Then G is hamiltonian if and only if G' is.*

This lemma says that we can add all edges uv to G, where $\text{DEG}(u) + \text{DEG}(v) \geq n$, without changing the hamiltonicity of G. We do this successively, for all non-adjacent vertices u and v.

DEFINITION 11.2: The *hamilton closure* of G is $c_H(G)$, the graph obtained by successively adding all edges uv to G, whenever $\text{DEG}(u) + \text{DEG}(v) \geq n$, for non-adjacent vertices u and v.

For example, the hamilton closure of the graph of Figure 11.11 is the complete graph K_7. It must be verified that this definition is valid, namely, no matter in what order the edges uv are added to G, the resulting closure is the same. We leave this to the reader.

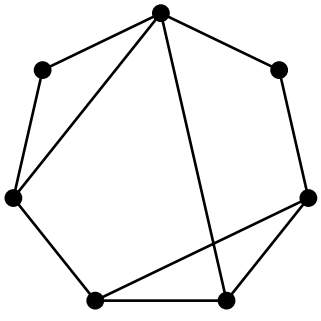


FIGURE 11.11
$\boldsymbol{c_H(G) = K_7}$

Lemma 11.3 tells us that $c_H(G)$ is hamiltonian if and only if G is. In particular, if $c_H(G)$ is a complete graph, then G is hamiltonian. The hamilton closure can be used to obtain a condition on the degree sequence of G which will force G to be hamiltonian.

Theorem 11.4. (Bondy-Chvátal theorem) *Let G be a simple graph with degree sequence $(d_1, d_2, \ldots, d_n)$, where $d_1 \leq d_2 \leq \ldots \leq d_n$. If there is no $m < n/2$ such that $d_m \leq m$ and $d_{n-m} < n - m$, then $c_H(G)$ is complete.*

Proof. Suppose that $c_H(G)$ is not complete. Let u and v be non-adjacent vertices such that $\text{DEG}(u) + \text{DEG}(v)$ is as large as possible, where the degree is computed in the closure $c_H(G)$. Then $\text{DEG}(u) + \text{DEG}(v) < n$ by definition of the closure. Let $m = \text{DEG}(u) \leq \text{DEG}(v)$. So u is joined to m vertices. There are $n - \text{DEG}(v) - 1$ vertices that v is not adjacent to (not counting v, because $v \not\longrightarrow v$). Each of these has degree $\leq m$. So the number of vertices with degree $\leq m$ is at least $n - \text{DEG}(v) - 1$. But $\text{DEG}(u) + \text{DEG}(v) < n$, so that $m = \text{DEG}(u) \leq n - \text{DEG}(v) - 1$. That is, the number of vertices of the closure with degree $\leq m$ is at least m. Because the degree sequence of $c_H(G)$ is at least as big as that of G, it follows that $d_m \leq m$.

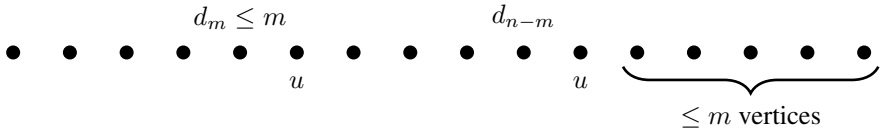


FIGURE 11.12
The degree sequence of $\boldsymbol{G}$

How many vertices have degree $> \text{DEG}(v)$? We know that u is adjacent to all of them. Therefore, the number of them is at most m, so that there are at most m vertices after v in the degree sequence. It follows that $\text{DEG}(v) \geq d_{n-m}$. But because $\text{DEG}(v) < n - m$, it follows that $d_{n-m} < n - m$. Thus, we have found a value m such that $d_m \leq m$ and $d_{n-m} < n - m$. Here $m = \text{DEG}(u) \leq \text{DEG}(v) < n - m$, so that $m < n/2$. This contradicts the assumptions of the theorem. Therefore $c_H(G)$ must be complete under these conditions. □

The degree sequence condition of the Bondy-Chvátal theorem is easy to apply. For example, any graph with the degree sequence $(2, 2, 3, 4, 5, 6, 6, 6)$ must be hamiltonian, because $d_1 = 2 > 1$, $d_2 = 2 \leq 2$, but $d_{8-2} = 6 \not< 6$, and $d_3 = 3 \leq 3$, but $d_{8-3} = 5 \not< 5$. Thus there is no $m < 8/2$ satisfying the condition that $d_m \leq m$ and $d_{n-m} < n - m$.

This is the strongest degree sequence condition possible which forces an arbitrary graph G to be hamiltonian. Any stronger condition would have to place non-degree sequence restrictions on G. To see that this is so, let G be any non-hamiltonian graph. Let its degree sequence be $(d_1, d_2, \ldots, d_n)$, where $d_1 \leq d_2 \leq \ldots \leq d_n$. Because G is not hamiltonian, there is a value m such that $d_m \leq m$ and $d_{n-m} < n - m$. Construct a new degree sequence by increasing each d_i until the sequence $(m, \ldots, m, n-m-1, \ldots, n-m-1, n-1, \ldots, n-1)$ is obtained, where the first m degrees are m, the last m degrees are $n-1$, and the middle $n-2m$ degrees are $n-m-1$. We construct a non-hamiltonian graph $C(m, n)$ with this degree sequence.

$C(m, n)$ is composed of three parts, a complete graph K_m, a complete graph K_{n-2m}, and an empty graph $\overline{K}_m$. Every vertex of $\overline{K}_m$ is joined to every vertex of K_m, and every vertex of K_m is joined to every vertex of K_{n-2m}. This is illustrated in Figure 11.13. The vertices of $\overline{K}_m$ have degree m, those of K_{n-2m} have degree $n-m-1$, while those of K_m have degree $n-1$. $C(3, 9)$ is illustrated in Figure 11.3.

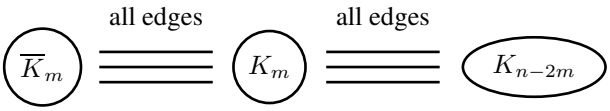


FIGURE 11.13
$\boldsymbol{C(m,n)}$

It is easy to see that $C(m,n)$ is always non-hamiltonian, because the deletion of the vertices of K_m leaves $m+1$ components. By Lemma 11.1, we conclude that $C(m,n)$ is non-hamiltonian. Yet for every non-hamiltonian graph G on n vertices, there is some $C(m,n)$ whose degree sequence is at least as large as that of G, in the lexicographic order.

Exercises

11.3.1 Prove that $c_H(G)$ is well-defined; that is, the order in which edges uv are added to G does not affect the result.

11.3.2 Prove that the crossover algorithm will find a hamilton cycle in G if $c_H(G)$ is complete or find a counterexample.

11.3.3 Use the Bondy-Chvátal theorem to show that any graph with the degree sequence $(2,3,3,4,5,6,6,6,7)$ is hamiltonian. What about $(3,3,4,4,4,4,4,4)$?

11.3.4 Define the hamilton-path closure to be $c'_H(G)$, obtained by adding all edges uv whenever $\text{DEG}(u) + \text{DEG}(v) \geq n-1$. Prove that G has a hamilton path if and only if $c'_H(G)$ does.

11.3.5 Obtain a condition like the Bondy-Chvátal theorem which will force $c'_H(G)$ to be complete.

11.3.6 Construct the graphs $C(2,8)$ and $C(4,12)$.

11.3.7 Work out $\varepsilon(C(m,n))$. Show that ε has its smallest value when

$$m = \frac{n}{3} - \frac{1}{6},$$

for which

$$\varepsilon = \frac{2}{3}\binom{n}{2} - \frac{1}{24}.$$

11.3.8 Show that if G is a graph on $n \geq 4$ vertices with $\varepsilon \geq \binom{n-1}{2} + 1$, then G is hamiltonian.

Notice that according to Exercise 11.3.7, $C(m,n)$ has approximately two-thirds of the number of edges of the complete graph K_n, at the minimum. This means that degree sequence conditions are not very strong. They apply only to graphs with very many edges.

11.4 The extended multi-path algorithm

The multi-path algorithm tries to build a hamilton cycle C using a recursive exhaustive search. At any stage of the algorithm, a number of disjoint paths $S_1, S_2, \ldots, S_k$ in G are given, which are to become part of C. Call them *segments* of C. Initially, we can take $k = 1$, and the single segment S_1 can consist of the starting vertex, that is, a path of length zero. On each iteration a vertex u is selected, an endpoint of some segment $P = S_i$. Every $w \longrightarrow u$ is taken in turn, and P is extended to $P' = P + uw$. Vertex u may now have degree two in S_i. In this case, the remaining edges ux of G are deleted. This reduces each $\textsc{Deg}(x)$ by one. When $\textsc{Deg}(x) = 2$, both remaining edges at x must become part of C. A new segment is created containing x. Thus, the choice of uw can force certain edges to be a part of C. It can also happen that when edges are forced in this way, that an edge connecting the endpoints of two segments is forced, and the two segments must be merged into one. This in turn forces other edges to be deleted, etc. The forcing of edges can be performed using a queue. There are three possible outcomes of this operation:

1. An updated set of segments can be produced.
2. A hamilton cycle can be forced.
3. A small cycle can be forced.

By a small cycle, we mean any cycle smaller than a hamilton cycle. If a small cycle is forced, we know that the extension of P to $P + uw$ does not lead to a hamilton cycle. If a hamilton cycle is forced, the algorithm can quit. If a new set of segments is produced, the algorithm proceeds recursively. This can be summarized as follows. We assume a global graph G, and a global boolean variable *IsHamiltonian*, which is initially **false**, but is changed to **true** when a hamilton cycle is discovered.

Suppose that the multi-path algorithm were applied to a disconnected graph G. Although we know that G is not hamiltonian, the algorithm could still take a very long time to discover this, for example, the connected components of G could be complete graphs. More generally, it is quite possible for the operation of forcing edges to delete enough edges so as to disconnect G. Thus the algorithm really is obliged to check that G is still connected before making a recursive call. This takes $O(\varepsilon)$ steps. Now we know that a graph with a cut-vertex also has no hamilton cycle, and we can test for a cut-vertex at the same time as checking that G is connected. A depth-first search (DFS) can do both in $O(\varepsilon)$ steps. Thus we add a DFS to the multi-path algorithm before the recursive call is made. But we can make a still greater improvement.

Algorithm 11.4.1: MULTIPATH(S)

comment: Search for a ham cycle containing all segments of S

choose a vertex u, an endpoint of some path $P \in S$
for all $w \longrightarrow u$
do
- extend path P to $P + uw$
- **comment:** extending P to $P + uw$ may force some edges
- FORCEEDGES(uw)
- **if** a hamilton cycle was forced
  - **then**
    - *IsHamiltonian* $\leftarrow$ **true**
    - **return**
- **if** a small cycle was not forced
  - **then**
    - **comment:** the segments S have been updated
    - MULTIPATH(S)
    - **if** *IsHamiltonian*
      - **then return**
- restore G and S to their state before uw was chosen

comment: otherwise no hamilton cycle was found

Suppose that the multi-path algorithm were applied to the graph of Figure 11.14. This graph is non-hamiltonian because the deletion of the two shaded vertices leaves three components. In certain cases the algorithm is able to detect this, using the DFS that tests for cut-vertices. Suppose that the segments of G are the bold edges. Notice that one of the segments contains the shaded vertex u. When the non-segment edges incident on u are deleted, v becomes a cut-vertex in the resulting graph. The DFS will detect that v is a cut-vertex, and the algorithm will report that adding the edge uw to the segment does not extend to a hamilton cycle.

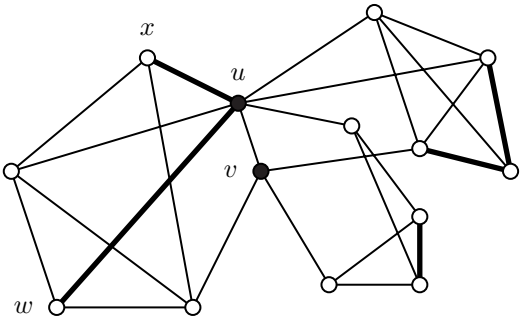


FIGURE 11.14
A non-hamiltonian graph

Normally the algorithm would then try the next edge incident on u, etc. But it

can do more. When the cut-vertex v is discovered, the DFS can *count the number of components* of $G - v$. This will be one plus the number of descendants of v in the DF-tree. It requires almost no extra work for the DFS to calculate this. For vertex v in Figure 11.14, the count will be three components. But because this is the result of deleting only two vertices, namely, u and v, the algorithm can determine that the original G is non-hamiltonian, and stop the search at that point. More generally, a non-hamiltonian graph like Figure 11.14 can arise at some stage during the algorithm as a result of deleting edges, even though the original G is hamiltonian. The algorithm must be able to detect which graph in the search tree is found to be non-hamiltonian by this method. We leave it to the reader to work out the details.

It is helpful to view vertices like u in Figure 11.14 which have degree two in some segment as having been deleted from G. Each segment is then replaced by an equivalent single edge connecting its endpoints. The set of segments then becomes a matching in G, which is changing dynamically. For example, when the segments of Figure 11.14 are replaced by matching edges, the resulting graph appears as in Figure 11.15. The procedure which forces edges can keep a count of how many vertices internal to segments have been deleted in this way, at each level in the recursion. When the DFS discovers a cut-vertex, this count is used to find the size of a separating set in G. In cases like this, large portions of the search tree can be avoided.

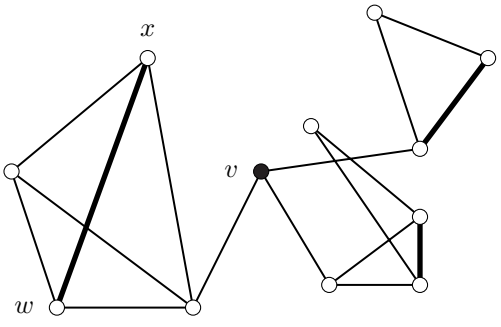


FIGURE 11.15
Segments viewed as a matching

A bipartite graph like the Herschel graph of Figure 11.2 is also non-hamiltonian, but the algorithm is not likely to delete enough vertices to notice that it has a large separating set. In general, suppose that at some stage in the algorithm $G - E(S)$ is found to be bipartite, with bipartition (X, Y), where S is viewed as a matching in G. If there is a hamilton cycle C in G using the matching edges S, it must look something like Figure 11.16, where the bipartition of $G - E(S)$ is shown by the shading of the nodes. There are now three kinds of segments: those contained within X, those contained within Y, and those connecting X to Y. Suppose that there are ε_X of the first type, and ε_Y of the second type. The vertices of C must alternate between X and Y, except for the ε_X and ε_Y edges, which must have endpoints

of the same color. If we contract each of these edges to a single node, we obtain perfect alternation around the cycle. Therefore $|X| - \varepsilon_X = |Y| - \varepsilon_Y$ if G has a hamilton cycle. If this condition is not satisfied, we know that G is non-hamiltonian, and can break off the search. We again employ the DFS that tests for cut-vertices to simultaneously check whether $G - E(S)$ is bipartite, and to keep a count of the numbers $|X| - \varepsilon_X$ and $|Y| - \varepsilon_Y$. This requires very little extra work, and is still $O(\varepsilon)$. In this way, non-hamiltonian graphs derived from bipartite graphs or near-bipartite graphs can often be quickly found to be non-hamiltonian.

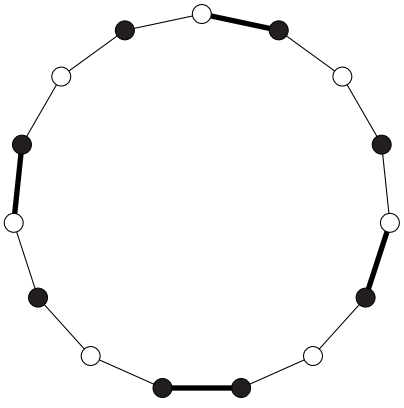


FIGURE 11.16
$\boldsymbol{G - E(S)}$ is bipartite

In summary, the extended multi-path algorithm adds a DFS before the recursive call. The DFS computes several things:

- Whether G is connected.
- $\omega(G - v)$, for each cut-vertex v.
- Whether $G - E(S)$ is bipartite.
- $|X| - \varepsilon_X$ and $|Y| - \varepsilon_Y$, if $G - E(S)$ is bipartite.

It may be possible to add other conditions to detect situations when G is non-hamiltonian. For example, every hamiltonian graph G with an even number of vertices n has two disjoint perfect matchings. If n is odd, every $G - v$ has a perfect matching.

11.4.1 Data structures for the segments

The extended multi-path algorithm still has exponential worst-case running time. Operations on the segments must be made as fast as possible. The operations that must

be performed using segments are, given any vertex v, to determine which segment contains v, and to find its endpoints; and to merge two segments when their endpoints are joined. One way to do this is with the merge-find data structure. An array $\mathit{Segment}[v]$ is stored, which is an integer, pointing to the representative of the segment containing v. Each segment has two endpoints, which we arbitrarily designate as the right and left endpoints. The right endpoint x is the segment representative. It is indicated by a negative value of $\mathit{Segment}[x]$. Its value is $-y$, where y is the left endpoint. Thus we find the segment representative by following the pointers, using path compression (see Chapter 2). Segments are merged by adjusting the pointers of their endpoints.

Exercises

11.4.1 Program the multi-path algorithm. Use a DFS to test for the conditions mentioned above.

11.5 Decision problems, NP-completeness

The theory of NP-*completeness* is phrased in terms of *decision problems*, that is, problems with a yes or no answer, (e.g., "is G hamiltonian?"). This is so that an algorithm can be modeled as a Turing machine, a theoretical model of computation. Although Turing machines are very simple, they can be constructed to execute all the operations that characterize modern random access computers. Turing machines do not usually produce output, except for yes or no. Thus, a Turing machine can be constructed to read in a graph, and perform an exhaustive search for a hamilton cycle. If a cycle exists, it will be found, and the algorithm will report a yes answer. However, the exhaustive search will tend to take an exponential amount of time in general.

The class of all decision problems contains an important subclass called P, all those which can be solved in *polynomial* time; that is, the complexity of a problem is bounded by some polynomial in its parameters. For a graph, the parameters will be n and ε, the number of vertices and edges.

There is another class of decision problems for which polynomial algorithms are not always known, but which have an additional important property. Namely, if the answer to a problem is yes, then it is possible to write down a solution which can be verified in polynomial time. The HamCycle problem is one of these. If a graph G has a hamilton cycle C, and the order of vertices on the cycle is written down, it is easy to check in n steps that C is indeed a hamilton cycle. So if we are able to guess a solution, we can verify it in polynomial time. We say that we can write a *certificate* for the problem, if the answer is yes. A great many decision problems have this property that a certificate can be written for them if the answer is yes, and it can be checked in polynomial time. This forms the class NP of *non-deterministic*

polynomial problems. The certificate can be checked in polynomial time, but we do not necessarily have a deterministic way of finding a certificate.

Now it is easy to see that P $\subseteq$ NP, because every problem which can be solved in polynomial time has a certificate – we need only write down the steps which the algorithm executed in solving it. It is generally believed that HamCycle is in NP but not in P. There is further evidence to support this conjecture beyond the fact that no one has been able to construct a polynomial-time algorithm to solve HamCycle; namely, it can be shown that the HamCycle problem is one of the NP-complete problems.

To understand what NP-completeness means we need the concept of polynomial transformations. Suppose Π_1 and Π_2 are both decision problems. A *polynomial transformation* from Π_1 to Π_2 is a polynomial-time algorithm which when given any instance I_1 of problem Π_1 will generate an instance I_2 of problem Π_2, satisfying:

$$I_1 \textit{ is a yes instance of } \Pi_1 \textit{ if and only if } I_2 \textit{ is a yes instance of } \Pi_2$$

We use the notation $\Pi_1 \propto \Pi_2$ to indicate that there is a polynomial transformation from Π_1 to Π_2. We say that Π_1 *reduces* to Π_2. This is because if we can find a polynomial algorithm A to solve Π_2, then we can transform Π_1 into Π_2, and then use A to solve Π_2, thereby giving a solution to Π_1.

DEFINITION 11.3: A decision problem is Π is NP-*complete*, if

1. Π is in NP.
2. For any problem $\Pi' \in$ NP, $\Pi' \propto \Pi$.

It was Cook who first demonstrated the existence of NP-complete problems. He showed that Problem 11.2, *satisfiability of boolean expressions* (Sat) is NP-complete. Let U be a set of n boolean variables $u_1, u_2, \ldots, u_n$ with their complements $\overline{u}_1, \overline{u}_2, \ldots, \overline{u}_n$. These variables can only take on the values **true** and **false**, such that u_i is **true** if and only if $\overline{u}_i$ is **false**, and vice versa. If $x, y \in U$, then we denote by $x + y$ the boolean **or** of x and y by xy the boolean **and** of x and y. A *clause* over U is a sum of variables in U. For example, $(u_1 + \overline{u}_3 + \overline{u}_4 + u_6)$ is a clause. A boolean expression is a product of clauses. For example $(u_1 + \overline{u}_3 + \overline{u}_4 + u_6)(u_2 + u_5)(\overline{u}_7)$ is a boolean expression. A truth assignment t is an assignment of values **true** and **false** to the variables in U. If B is a boolean expression, then $t(B)$ is the evaluation of B with truth assignment t. For example if

$$B = (u_1 + \overline{u}_3 + \overline{u}_4 + u_6)(u_2 + u_5)(\overline{u}_7)$$

and

$$t = \begin{pmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 \\ \textbf{true} & \textbf{false} & \textbf{false} & \textbf{true} & \textbf{true} & \textbf{false} & \textbf{false} \end{pmatrix},$$

then

$$t(B) = (\ \textbf{true} + \textbf{true} + \textbf{false} + \textbf{false}\ )(\ \textbf{false} + \textbf{true}\ )(\ \textbf{true}\ ) = \textbf{true}.$$

Not every boolean expression B has a truth assignment t such that $t(B) =$ **true**. For example there is no way to assign **true** and **false** to the variables in the expression $(\overline{u}_1 + \overline{u}_2)(u_1)(u_2)$ so that it is **true**. If there is a truth assignment t such that $t(B) =$ **true**, we say that B is *satisfiable*. The *satisfiability of boolean expressions* problem is

Problem 11.2:	Sat
Instance:	a set of boolean variables U and boolean expression B over U.
Question:	is B satisfiable?

and was shown by COOK [36] to be NP-complete. See also KARP [96]. The proof of this is beyond the scope of this book; however, a very readable proof can be found in the book by PAPADIMITRIOU and STEIGLITZ [134]. Many problems have subsequently been proved NP-complete, by reducing them either to satisfiability, or to other problems already proved NP-complete.

The importance of the NP-complete problems is that, if a polynomial algorithm for any NP-complete problem Π were discovered, then every problem in NP would have a polynomial algorithm; that is, P = NP would hold. Many people have come to the conclusion that this is not very likely, on account of the large number of NP-complete problems known, all of which are extremely difficult. We will now show that

$$\text{Sat} \propto \text{3-Sat} \propto \text{Vertex Cover} \propto \text{HamCycle}$$

and thus the HamCycle problem (as well as 3-Sat and Vertex Cover) is an NP-complete problem. Thus if P $\neq$ NP, then a polynomial algorithm for the HamCycle problem would not exist. This is why we say that the HamCycle problem is qualitatively different from most other problems in this book.

Among the most useful problems for establishing the NP-completeness of other problems is 3-Sat.

Problem 11.3:	3-Sat
Instance:	a set of boolean variables U and boolean expression B over U, in which each clause contains exactly three variables.
Question:	is B satisfiable?

Theorem 11.5. *3-Sat is* NP*-complete.*

Proof. It is easy to see that 3-Sat is in NP. Any truth assignment satisfying the boolean expression B can be checked in polynomial time by assigning the variables and then evaluating the expression.

We reduce Sat to 3-Sat as follows. Let U be a set of boolean variables and $B = C_1 C_2 \cdots C_m$ be an arbitrary boolean expression, so that U and B is an instance of Sat. We will extend the variable set U to a set U' and replace each clause C_i in B by a boolean expression B_i, such that

(a) B_i is a product of clauses that use exactly three variables of U'.

(b) B_i is satisfiable if and only if C_i is.

Then $B' = B_1 B_2 \cdots B_m$ will be an instance of 3-Sat that is satisfiable over U' if and only if B is satisfiable over U. Let $C_i = (x_1 + x_2 + x_3 + \cdots + x_k)$. There are three cases.

Case 1: $k = 1$.

In this case we introduce new variables y_i and z_i and replace C_i with

$$B_i = (x_1 + y_i + z_i)(x_1 + y_i + \overline{z}_i)(x_1 + \overline{y}_i + z_i)(x_1 + \overline{y}_i + \overline{z}_i).$$

Case 2: $k = 2$.

In this case we introduce a new variable y_i and replace C_i with

$$B_i = (x_1 + x_2 + y_i)(x_1 + x_2 + \overline{y}_i).$$

Case 3: $k = 3$.

In this case we replace C_i with $B_i = C_i$. Thus we make no change.

Case 4: $k > 3$.

In this case we introduce new variables $y_{i_1}, y_{i_2}, \ldots, y_{i_{k-3}}$ and replace C_i with

$$B_i = (x_1 + x_2 + y_{i_1})(\overline{y}_{i_1} + x_3 + y_{i_2})(\overline{y}_{i_2} + x_4 + y_{i_3}) \cdots (\overline{y}_{i_{k-3}} + x_{k-1} + x_k).$$

It is routine to verify for each of Cases 1, 2, 3, and 4, that B_i satisfies (a) and (b), see Exercise 11.5.2. We still must show that

$$B' = B_1 B_2 B_3 \cdots B_m$$

can be constructed with a polynomial time algorithm. If $C_i = (x_1 + x_2 + x_3 + \cdots + x_k)$, then B_i contains at most $4k$ clauses of three variables and at most $k + 1$ new variables were introduced. Because $k \leq n$, we conclude that to construct B', at most $4mn$ new clauses of three variables are needed, and at most $(n + 1)m$ new variables are introduced. Both are polynomial in the size of the instance of Sat. Consequently we can construct B' in polynomial time. □

Given a graph G, a k-element subset $K \subseteq V(G)$ of vertices is a called a *vertex cover* of size k if each edge of G has at least one end in K. The Vertex Cover decision problem is:

Problem 11.4:	Vertex Cover
Instance:	a graph G and positive integer k.
Question:	does G have a vertex cover of size at most k?

Theorem 11.6. *Vertex Cover is* NP*-complete.*

Proof. It is easy to see that Vertex Cover is in NP, for if K is a purported vertex cover of the graph G of size k, then we simply check each edge of G to see that at least one endpoint is in K. There are ε edges to check so this takes time $O(\varepsilon)$ and we can check in time $|K| \leq n$ whether or not $|K| \leq k$.

We now give a polynomial transformation from 3-Sat to Vertex Cover. Let $B = C_1C_2 \cdots C_m$ be a boolean expression over $U = \{u_1, u_2, \ldots, u_n\}$ in which each clause is a sum of exactly three variables. Thus for $i = 1, 2, \ldots, n$, $C_i = (x_i + y_i + z_i)$ for some $x_i, y_i, z_i \in U \cup \overline{U}$, where $\overline{U} = \{\overline{u}_1, \overline{u}_2, \ldots, \overline{u}_n\}$. We construct a graph G on the vertex set

$$V = U \cup \overline{U} \cup W,$$

where $W = \cup_{i=1}^{m} \{a_i, b_i, c_i\}$. The edge set of G is the union of the edges of m subgraphs H_i, $i = 1, 2, \ldots, m$, where H_i is the subgraph shown in Figure 11.17. It consists of a triangle (a_i, b_i, c_i), edges from a_i, b_i, c_i to the variables contained in the clause, and edges connecting the variables to their complements. G has $2n + 3m$ vertices and $n + 6m$ edges and hence can be built in polynomial time. Choose $k = n + 2m$ to obtain an instance of the Vertex Cover problem for the graph G constructed.

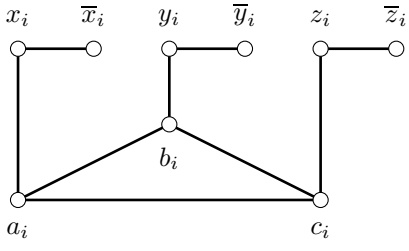


FIGURE 11.17
Subgraph $\boldsymbol{H_i}$ corresponding to clause $\boldsymbol{(x_i + y_i + z_i)}$

We show that B has a satisfying truth assignment if and only if G has a vertex cover K of size $k = n + 2m$. If t is a truth assignment, such that

$$t(B) = \textbf{true},$$

then t must assign at least one variable x_i, y_i, or z_i to be **true** in clause C_i. Assume it is x_i. As x_i is adjacent to exactly one vertex, a_i, in the triangle $\{a_i, b_i, c_i\}$, it follows that $\{x_i, b_i, c_i\}$ is a vertex cover of H_i, and hence

$$K = \cup_{i=1}^{m} \{x_i, b_i, c_i\}$$

is a vertex cover of size k for G. An example is given in Figure 11.18.

Conversely suppose that K is a vertex cover of size $k = n + 2m$ of G. Then K must include at least one end of each of the n edges $\{u_i, \overline{u}_i\}$, $i = 1, 2, \ldots, n$, accounting for at least n vertices in K. Also K must cover the edges of each

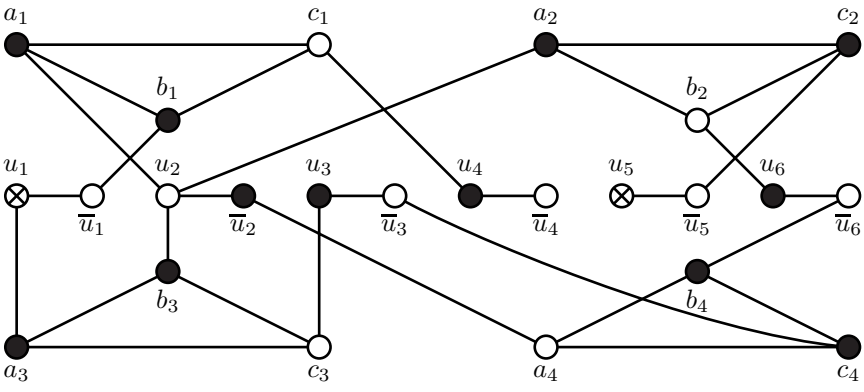


FIGURE 11.18
Graph $\boldsymbol{G}$ corresponding to the boolean expression $\boldsymbol{B = (u_2 + \overline{u}_1 + u_4)(u_2 + u_6 + \overline{u}_5)(u_1 + u_2 + u_3)(\overline{u}_2 + \overline{u}_6 + \overline{u}_3)}$. A vertex cover is $\boldsymbol{K = \{u_4, a_1, b_1, u_6, a_2, c_2, u_3, a_3, b_3, \overline{u}_2, b_4, c_4 i, u_1, u_5\}}$ and $\boldsymbol{u_3 = u_4 = u_6 =}$ **true**, $\boldsymbol{u_2 =}$ **false** , $\boldsymbol{u_1, u_5}$, assigned arbitrarily is a truth assignment satisfying $\boldsymbol{B}$.

triangle (a_j, b_j, c_j), and thus must contain at least two of $\{a_j, b_j, c_j\}$, for each $j = 1, 2, \ldots, m$. This accounts for $2m$ more vertices, for a total of $n + 2m = k$ vertices. Hence K must contain exactly one of the endpoints of each edge $\{u_i, \overline{u}_i\}$, for $i = 1, 2, \ldots, n$, and exactly two of a_j, b_j, c_j, for each $j = 1, 2, \ldots, m$, corresponding to clause C_j. For each clause C_j, there is exactly one vertex a_j, b_j, or c_j of the triangle which is not in K. Call it d_j. Choose the unique variable of $U \cup \overline{U}$ adjacent to d_i, and assign it **true**. Then at least one variable in each clause C_j has been assigned the value **true**, giving a truth assignment that satisfies B. Any remaining unassigned variables in U can be assigned **true** or **false** arbitrarily. □

Theorem 11.7. *HamCycle is* NP*-complete.*

Proof. Let G be a graph. Given an ordering $v_1, v_2, \ldots, v_n$ of vertices of G we can check whether $(v_1, v_2, v_3, \ldots, v_n)$ is a hamilton cycle in polynomial time. Thus HamCycle is in NP. To show that HamCycle is NP-complete we transform from Vertex Cover.

Let G and k be an instance of Vertex Cover, where k is a positive integer. We will construct a graph G' such that G' has a hamilton cycle if and only if G has a vertex cover $K = \{x_1, x_2, \ldots, x_k\}$ of size k. The graph G' will have $k + 12m$ vertices

$$V(G') = K \cup \{(u, e, i) : u \in V(G) \text{ is incident to } e \in E(G) \text{ and } i = 1, 2, \ldots, 6\},$$

where $m = |E(G)|$. The edges of G' are of three types.

Type 1 edges of G'

The type 1 edges are the $14m$ edges among the subgraphs H_e, $e \in E(G)$. We display H_e, where $e = uv$ in Figure 11.19.

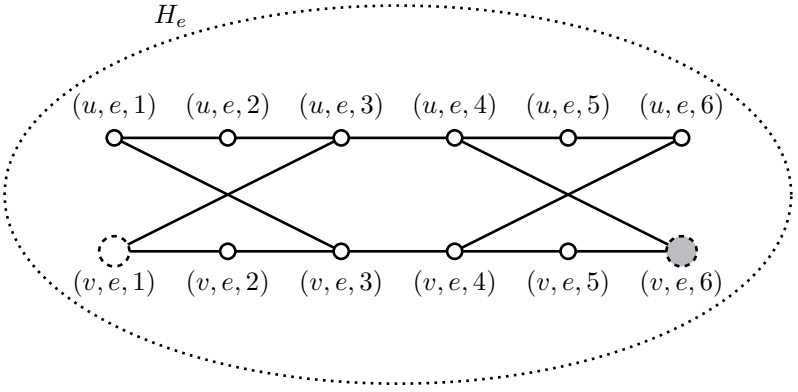


FIGURE 11.19
The subgraph $\boldsymbol{H_e}$, where $\boldsymbol{e = uv}$

Type 2 edges of G'

For each vertex v of G choose a fixed but arbitrary ordering $e_{v_1}, e_{v_2}, \ldots, e_{v_3}$ of the $d = \text{DEG}(v)$ edges incident to v. The type 2 edges of G' corresponding to v are:

$$\left\{\{(v, e_{v_i}, 6), (v, e_{v_{i+1}}, 1)\} : i = 1, 2, \ldots, d-1\right\}$$

Type 3 edges of G'

The type 3 edges of G' are:

$$\{\{x_i, (v, e_{v_1}, j)\} : x_i \in K, v \in V(G), j \in \{1, 6\}\}$$

The subgraph of G' corresponding to the edges incident to a vertex v in G is illustrated in Figure 11.20. Before proving that G has a vertex cover of size k if and only if G' has a hamilton cycle $C = v_1, v_2, \ldots, v_n$, we make five observations.

1. C must enter and exit the subgraph H_e, $e = uv$ from the four corners $(u, e, 1), (u, e, 6), (v, e, 1), (v, e, 6)$.

2. If C enters H_e at $(u, e, 1)$, it must exit at $(u, e, 6)$ and either pass through all the vertices of H_e or only those vertices with first coordinate u. (In the first case as we shall see, u will be in the vertex cover of G, and in the latter case both u and v will be in the vertex cover of G.)

3. If C enters H_e at $(v, e, 1)$, it must exit at $(v, e, 6)$ and either pass through all the vertices of H_e or only those vertices with first coordinate v. (In the first case as we shall see, v will be in the vertex cover of G, and in the latter case both u and v will be in the vertex cover of G.)

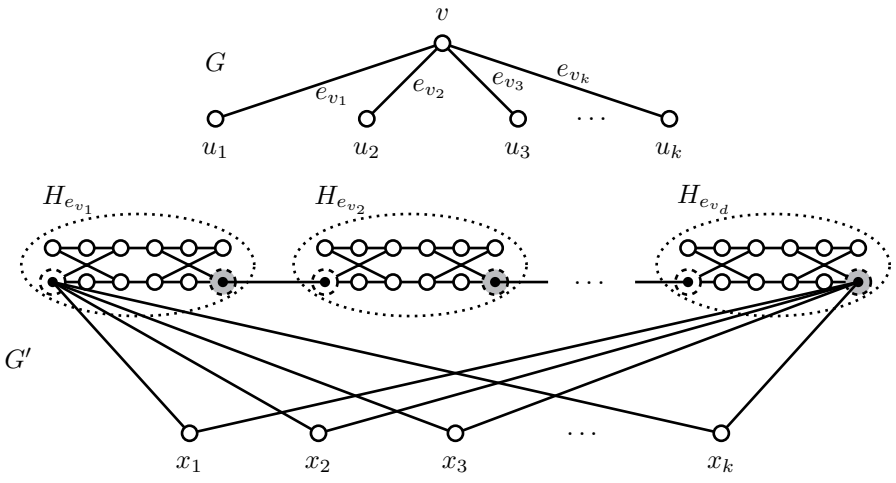


FIGURE 11.20
Subgraph of G' corresponding to the edges incident to v in G

4. The vertices $\{x_1, x_2, \ldots, x_k\}$ divide C into paths. Thus we may assume, relabeling the vertices $x_1, x_2, \ldots, x_k$ if necessary, that $C = P_1 P_2 \cdots P_k$ where P_i is an x_i to x_{i+1} path, where $x_{k+1} = x_k$.

5. Let v_i be such that x_i is adjacent to (v_i, e, j) in P_i where $j = 1$ or 6. Then P_i contains every vertex (v_i, e', h) where e is incident to v.

We claim that the k vertices $v_1, v_2, \ldots, v_k$ selected in observation 5 are a vertex cover of G. This is because the hamilton cycle C must contain all vertices of each of the subgraphs H_e for each $e \in G$; and when H_e is traversed by C, it is traversed by some P_i in C and that P_i selects an endpoint v_i of e.

Conversely, suppose $K = \{v_1, v_2, \ldots, v_k\} \subseteq V(G)$ is a vertex cover of G, of size k. To construct a hamilton cycle C of G', choose for each edge $e \in E(G)$ the edges of H_e specified in Figure 11.21 (a), (b), or (c) depending on whether $\{u, v\} \cup K$ equals $\{u\}$, $\{u, v\}$, or $\{v\}$, respectively. (One of these must occur, because K is a vertex cover.) Also include the edges

$$\{(v_i, e_{v_i}, 6), (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k,$$

the edges

$$\{x_i, (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k,$$

and the edges

$$\{x_{i+1}, (v_i, e_{v_i}, 1)\}, i = 1, 2, \ldots, k, \text{ where } v_{k+1} = v_1.$$

It is an easy exercise to verify that the included edges form a hamilton cycle in G'; see Exercise 11.5.5. □

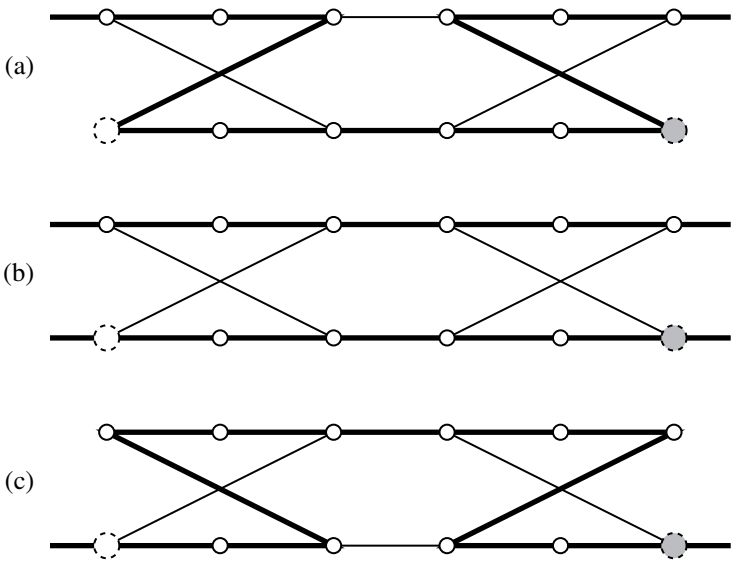


FIGURE 11.21
The three possible ways that a Hamilton cycle can traverse the subgraph $\boldsymbol{H_e}$, corresponding to the cases for $\boldsymbol{e = \{u, v\}}$ in which (a) $\boldsymbol{e \cap K = \{u\}}$, (b) $\boldsymbol{e \cap K = \{u, v\}}$, and (c) $\boldsymbol{e \cap K = \{v\}}$.

Exercises

11.5.1 Consider the boolean expression

$$B = (x_1 + \overline{x}_2 + x_3 + \overline{x}_6)(x_2 + \overline{x}_4)(x_5)(x_2 + \overline{x}_4 + x_5)$$

Find a boolean expression equivalent to B in which each clause uses only three variables.

11.5.2 Show for each Case 1, 2, 3, and 4 in Theorem 11.5 that the pair B_i, C_i satisfies

(a) B_i is a product of clauses that use at most three variables in U'.
(b) B_i is satisfiable if and only if C_i is.

11.5.3 Consider the boolean expression

$$\begin{aligned} B \quad = \quad & (x + y + z)(x + y + \overline{z})(w + \overline{x} + z)(w + \overline{x} + \overline{z})(\overline{w} + \overline{x} + z) \\ & (\overline{w} + \overline{x} + \overline{z})(w + \overline{y} + z)(w + \overline{y} + \overline{z})(\overline{w} + \overline{y} + z)(\overline{w} + \overline{y} + \overline{z}) \end{aligned}$$

(a) Show that there is no truth assignment that satisfies B.
(b) Construct the graph G in Theorem 11.6 that corresponds to B.
(c) Show that G does not have a vertex cover of size 25.

11.5.4 Verify the five observations in Theorem 11.7.

11.5.5 Verify that the included edges in the converse part of Theorem 11.7, do indeed form a hamilton cycle.

11.6 The traveling salesman problem

The traveling salesman problem (TSP) is very closely related to the HamCycle problem. A salesman is to visit n cities $v_1, v_2, \ldots, v_n$. The cost of traveling from v_i to v_j is $W(v_i v_j)$. Find the cheapest tour which brings him back to his starting point. Figure 11.22 shows an instance of the TSP problem. It is a complete graph K_n with positive integral weights on the edges. The problem asks for a hamilton cycle of minimum cost.

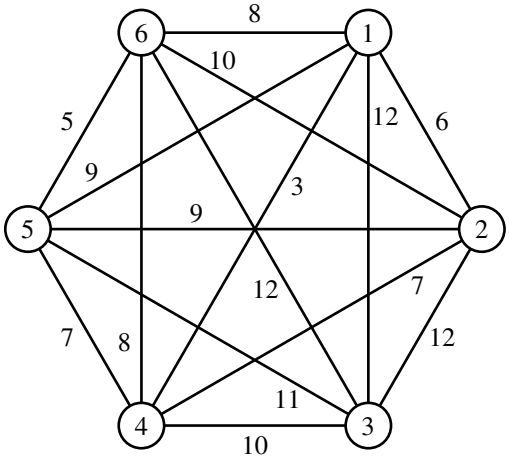


FIGURE 11.22
An instance of the TSP problem.

It is easy to show that the HamCycle problem can be reduced to the TSP problem. In order to do this, we first must phrase it as a decision problem.

Problem 11.5:	TSP Decision
Instance:	a weighted complete graph K_n, and an integer M,
Question:	does K_n have a hamilton cycle of cost $\leq M$?

We can then find the actual minimum by doing a binary search on the range of

values $n \leq M \leq nW_{\max}$, where $W_{\max}$ is the maximum edge-weight. Suppose that we had an efficient algorithm for the TSP Decision problem. Let G be any graph on n vertices which we want to test for hamiltonicity. Embed G in a compete graph K_n, giving the edges of G weight 1, and the edges of $\overline{G}$ weight 2. Now ask whether G has a TSP tour of cost $\leq n$. If the answer is yes, then G is hamiltonian. Otherwise G is non-hamiltonian.

Because HamCycle is NP-complete, we conclude that the TSP Decision problem is at least as hard as an NP-complete problem. In a certain sense, it is harder than the NP-complete problems, because the edge weights $W(v_iv_j)$ are not bounded in size. So it may take many steps just to add two of the weights. However, if we limit the size of the weights to the range of numbers available on a computer with a fixed word length, then the TSP Decision problem is also NP-complete. It is easy to see that TSP Decision $\in$ NP, because we can write down the sequence of vertices on a cycle C of cost $\leq M$ and verify it in n steps.

One way to approximate a solution is similar to the crossover technique. Choose a hamilton cycle C in K_n arbitrarily. For each edge $uv \in C$, search for an edge $wx \in C$ such that $W(uv)+W(wx) > W(uw)+W(vx)$. If such an edge exists, re-route C as shown in Figure 11.23. Repeat until no improvement can be made. Do this for several randomly chosen starting cycles, and take the best as an approximation to the optimum.

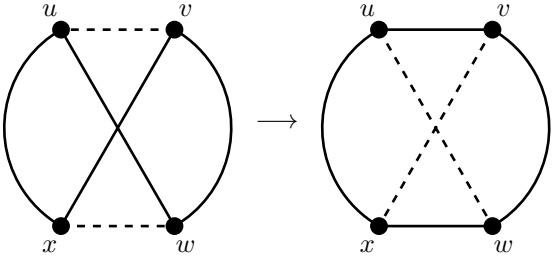


FIGURE 11.23
Re-routing a TSP tour

The cycle $Q = (u, v, x, w)$ is similar to a crossover. In general, if Q is any cycle such that $C \oplus Q$ is a hamilton cycle, and $W(C \cap Q) > W(Q - C)$, then $C \oplus Q$ will be a TSP tour of smaller cost than C. We can search for crossovers Q containing up to M edges, for some fixed value M, and this will provide a tour which may be close to the optimum. How close does it come to the optimum?

It is possible to obtain a rough estimate of how good a tour C is, by using a minimum spanning tree algorithm. Let C^* be an optimum TSP tour. For any vertex v, $C^* - v$ is a spanning tree of $K_n - v$. Let T_v be a minimum spanning tree of $K_n - v$. Then $W(C^* - v) \geq W(T_v)$. Given the path $C^* - v$, we must add back two edges incident on v to get C^*. If we add two edges incident on v to T_v, of minimum possible weight, we will get a graph T_v^*, such that $W(C^*) \geq W(T_v^*)$. For example,

Figure 11.24 shows a minimum spanning tree T_3, of $K_n - 3$ for the instance of TSP shown in Figure 11.22.

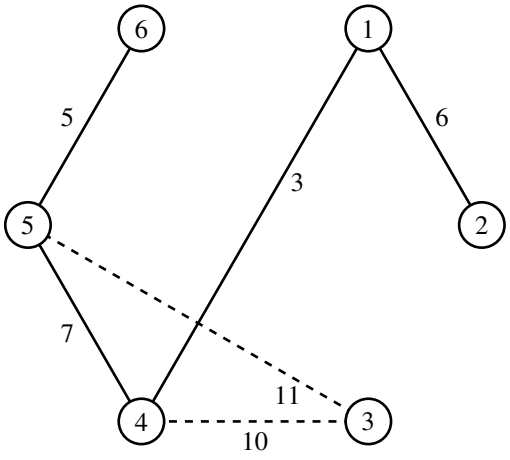


FIGURE 11.24
A minimum spanning tree T_3, plus two edges

The two edges incident on vertex 3 that we add to T_3 have weights 10 and 11 in this case. We thus obtain a bound $W(C^*) \geq W(T_3^*) = 42$. We do this for each vertex v, and choose the *maximum* of the bounds obtained. This is called the *spanning tree bound* for the TSP:

$$W(C^*) \geq \text{MAX}_v W(T_v^*).$$

Exercises

11.6.1 Work out the spanning tree bound for the TSP instance of Figure 11.22.

11.6.2 Find a TSP tour C in the graph of Figure 11.22 by re-routing any starting cycle, until no more improvement is possible. Compare the weight of C with the result of Exercise 11.6.1.

11.6.3 Construct all possible re-routing patterns (crossovers) containing three or four edges of C.

11.7 The ΔTSP

Distances measured on the earth satisfy the *triangle inequality*, namely, for any three points X, Y, and Z, $\text{DIST}(X, Y) + \text{DIST}(Y, Z) \geq \text{DIST}(X, Z)$. The *triangle traveling salesman problem*, denoted ΔTSP, refers to instances of the TSP satisfying this

inequality. When the triangle inequality is known to hold, additional methods are possible.

Theorem 11.8. *Let K_n be an instance of the ΔTSP, and let G be any Eulerian spanning subgraph of K_n. If C^* is an optimum TSP tour, then $W(C^*) \leq W(G)$.*

Proof. Consider an Euler tour H in G starting at any vertex. The sequence of vertices traversed by H is $v_{i_0}, v_{i_1}, v_{i_2}, v_{i_3}, \ldots$. If G is a cycle, then H is a hamilton cycle, so that $W(C^*) \leq W(G)$, and we are done. Otherwise, H repeats one or more vertices. Construct a cycle C from H by taking the vertices in the order that they appear in H, simply ignoring repeated vertices. Because G is a spanning subgraph of K_n, all vertices will be included in C. For example, if G is the graph of Figure 11.25, and H is

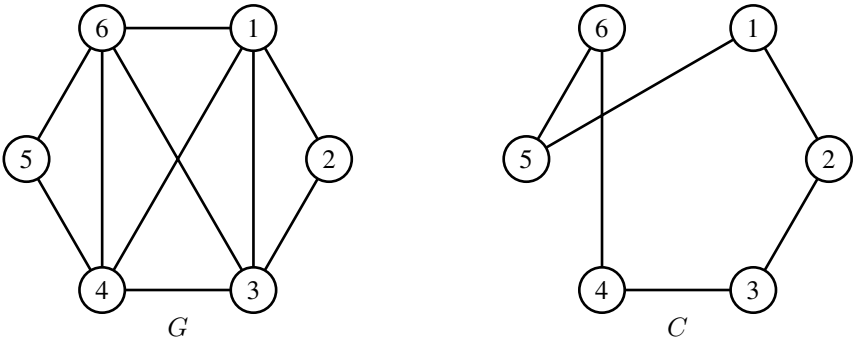


FIGURE 11.25
An Eulerian graph $\boldsymbol{G}$ and TSP tour $\boldsymbol{C}$

the Euler tour $(1, 2, 3, 4, 6, 1, 3, 6, 5, 4)$, then the cycle C obtained is $(1, 2, 3, 4, 6, 5)$. Because of the triangle inequality, it will turn out that $W(C) \leq W(G)$. Let the cycle obtained be $C = (u_1, u_2, \ldots, u_n)$, and suppose that the Euler tour H contains one or more vertices between u_k and u_{k+1}. Without loss of generality, suppose that there are just three vertices x, y, z between u_k and u_{k+1}. See Figure 11.26. Then because

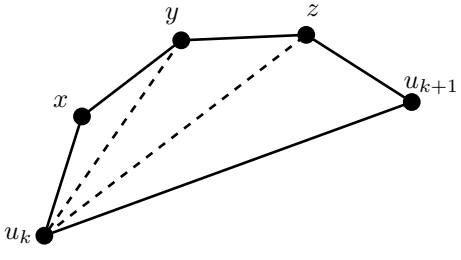


FIGURE 11.26
Applying the triangle inequality

of the triangle inequality, we can write

$$W(u_k x) + W(xy) \geq W(u_k y),$$

$$W(u_k y) + W(yz) \geq W(u_k z),$$

and

$$W(u_k z) + W(z u_{k+1}) \geq W(u_k u_{k+1}).$$

Thus

$$W(u_k u_{k+1}) \leq W(u_k x) + W(xy) + W(yz) + W(z u_{k+1}).$$

The left side of the inequality contributes to $W(C)$. The right side contributes to $W(H)$. It follows that $W(C) \leq W(G)$, for any Eulerian G. □

Notice that the particular cycle C obtained from G depends on the Euler tour H chosen, so that the graph G will give rise to a number of different hamilton cycles C. In particular, we could construct G from a minimum spanning tree T, by simply doubling each edge. This gives an Eulerian multigraph G. The method used in the theorem will also work with multigraphs, so we conclude that $W(C^*) \leq 2W(T)$. This is called the *tree algorithm* for the TSP.

Lemma 11.9. *The tree algorithm produces a cycle of cost at most twice the optimum.*

Proof. Let C be the cycle obtained by the tree algorithm, let C^* be an optimum cycle, and let T be a minimum spanning tree of the instance for ΔTSP. Because C^* is a spanning subgraph of K_n, we conclude that $W(C^*) > W(T)$. But we know that $W(C) \leq 2W(T) < 2W(C^*)$. □

11.8 Christofides' algorithm

Christofides found a way to construct an Eulerian subgraph of smaller weight than $2W(T)$. Let K_n be an instance of the ΔTSP, and let T be a minimum spanning tree. Let $X \subseteq V(K_n)$ be the vertices of T of odd degree. X contains an even number of vertices. The subgraph of K_n induced by X is a complete subgraph. Let M be a perfect matching in X of minimum weight. For example, Figure 11.27 shows a minimum spanning tree for the graph of Figure 11.22, together with a minimum-weight matching M, shown as dashed lines. This gives a graph $G = T + M$ which is Eulerian. It is quite possible that G is a multigraph. We now find an Euler tour in G and use it to construct a TSP tour C of cost at most $W(T) + W(M)$. This is called *Christofides' algorithm*.

Theorem 11.10. *Let C be the TSP tour produced by Christofides' algorithm and let C^* be an optimum tour. Then*

$$W(C) \leq \frac{3}{2} W(C^*).$$

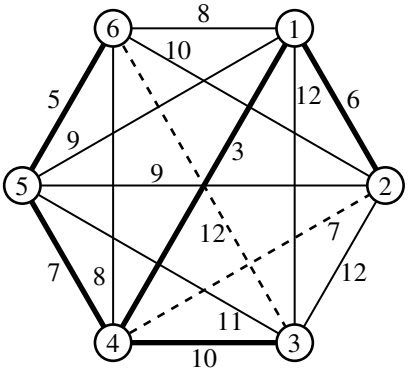


FIGURE 11.27
Christofides' algorithm

Proof. Let $u_1, u_2, \ldots, u_{2k}$ be the vertices of odd degree, and suppose that they appear on C^* in that order. This defines two matchings,

$$M_1 = \{u_1u_2, u_3u_4, \ldots\}$$

and

$$M_2 = \{u_2u_3, u_4u_5, \ldots, u_{2k}u_1\}.$$

See Figure 11.28. If M is the minimum weight matching, we conclude that $W(M_1), W(M_2) \geq W(M)$. The portion of C^* between u_i and u_{i+1} satisfies

$$W(C^*[u_i, u_{i+1}]) \geq W(u_iu_{i+1}),$$

by the triangle inequality. Therefore

$$W(C^*) \geq W(M_1) + W(M_2) \geq 2W(M),$$

or

$$W(M) \leq \frac{1}{2}W(C^*).$$

The cycle C found by Christofides's algorithm satisfies

$$W(C) \leq W(T) + W(M) < W(C^*) + \frac{1}{2}W(C^*),$$

because $W(T) < W(C^*)$. It follows that

$$W(C) \leq \frac{3}{2}W(C^*).$$

□

Thus, Christofides' algorithm always comes within 50% of the optimum.

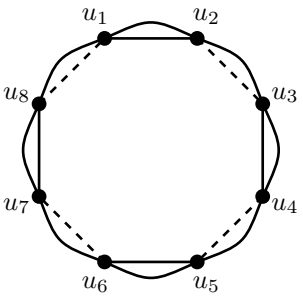


FIGURE 11.28
Two matchings M_1 and M_2

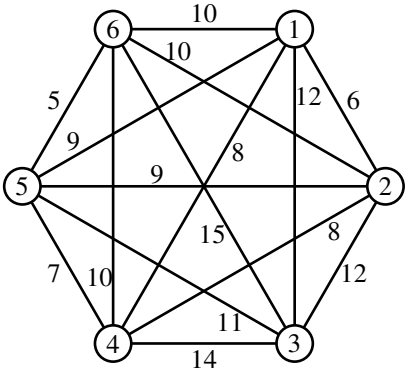


FIGURE 11.29
An instance of ΔTSP

Exercises

11.8.1 Use the tree algorithm to find a TSP tour for the graph of Figure 11.22.

11.8.2 Solve the same TSP instance using Christofides' algorithm. Compare the values found for $W(C)$, $W(T)$, and $W(T + M)$.

11.8.3 Solve the ΔTSP instance of Figure 11.29, using Christofides' algorithm. Compute the spanning tree bound as well.

11.9 Notes

An excellent survey of hamiltonian graphs appears in BERMOND [16]. The hamilton closure and the Bondy-Chvátal theorem are from BONDY and MURTY [23]. The extended multi-path algorithm is from KOCAY [105]. A classic book on the theory of NP-completeness is the text by GAREY and JOHNSON [64]. A very readable proof of Cook's theorem, that Satisfiability is NP-complete, appears in PAPADIMITRIOU and STEIGLITZ [134], which also contains an excellent section on Christofides' algorithm. The book by CHRISTOFIDES [34] has an extended chapter on the traveling salesman problem. The book LAWLER, LENSTRA, RINNOOY KAN, and SHMOYS [114] is a collection of articles on the traveling salesman problem.

12

Digraphs

12.1 Introduction

Directed graphs have already been introduced in the Chapter 1. If G is a digraph and $u, v \in V(G)$, we write $u \longrightarrow v$ to indicate that the edge uv is directed from u to v. The *in-edges* at v are the edges of the form (u, v). The *in-degree* of v is $d^-(v)$, the number of in-edges. Similarly the *out-edges* at u are all edges of the form (u, v) and the *out-degree* $d^+(u)$ is the number of out-edges at u. The degree of u is

$$\text{DEG}(u) = d^+(u) + d^-(u).$$

Given any undirected graph G, we can assign a direction to each of its edges, giving a digraph called an *oriented graph*. A digraph is *simple* if it is an orientation of a simple graph. A digraph is *strict* if it has no loops, and no two directed edges have the same endpoints. A strict digraph can have edges (u, v) and (v, u), whereas an oriented graph cannot.

Digraphs have extremely wide application, for the social sciences, economics, business management, operations research, operating systems, compiler design, scheduling problems, combinatorial problems, solving systems of linear equations, and many other areas. We shall describe only a few fundamental concepts in this chapter.

12.2 Activity graphs, critical paths

Suppose that a large project is broken down into smaller tasks. For example, building a house can be subdivided into many smaller tasks: dig the basement, install the sewer pipes, water pipes, electricity, pour the basement concrete, build the frame, floor, roof, cover the roof and walls, install the wiring, plumbing, heating, finish the walls, etc. Some of these tasks must be done in a certain order – the basement must be dug before the concrete can be poured, the wiring must be installed before the walls can be finished, etc. Other tasks can take place at the same time, (e.g., the wiring and plumbing can be installed simultaneously). We can construct a directed graph, called an *activity graph*, to represent such projects. It has a starting node s,

where the project begins, and a completion node t, where it is finished. The subtasks are represented by directed edges. The nodes represent the beginning and end of tasks (the synchronization points between tasks). Figure 12.1 shows an example of an activity graph. Each task takes a certain estimated time to complete, and this is represented by assigning each edge uv a weight $\text{WT}(uv)$, being the amount of time required for that task.

What is the *minimum amount of time* required for the entire project? It will be the length of the *longest* directed path from start to completion. Any longest directed path from s to t is called a *critical path*. Figure 12.1 shows a critical path in an activity graph.

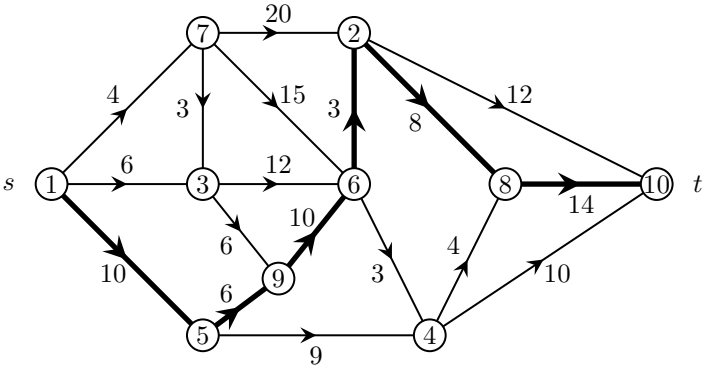


FIGURE 12.1
An activity graph

Notice that an activity graph must have no directed cycles. For if a directed cycle existed, it would be impossible to complete the project according to the constraints. Thus, activity graphs are *acyclic* digraphs. Activity graphs are applicable to any large project, such as building construction, business projects, or factory assembly lines.

The *critical path method* (CPM) is a technique for analyzing a project acccording to the longest paths in its activity graph. In order to find a longest path from s to t, we proceed very much as in Dijkstra's algorithm (Chapter 2), which builds a spanning tree, rooted at s, of shortest paths. To find longest paths instead, the algorithm builds an out-directed spanning tree, rooted at s, of longest directed paths from s to each vertex v. We store a value $T[v]$ for each v, being the earliest time at which tasks starting from v can begin. $T[v]$ is the length of a longest sv-path. When the algorithm completes, the critical path is the unique path in the spanning tree to vertex t. Notice that in Figure 12.1, the edge from vertex 1 to 3 has length 6, but that the path $(1, 7, 3)$ has the longer length of 7. In order for the algorithm to correctly choose the longer path, it must be sure to assign $T(7)$ before $T(3)$. Thus, the vertices must be taken in a certain order. For every edge (u, v), $T(u)$ must be computed before $T(v)$.

12.3 Topological order

A *topological ordering* of an acyclic digraph G is a permutation o of

$$V(G) = \{1, 2, \ldots, n\}$$

such that $o(u) < o(v)$ whenever $u \longrightarrow v$. Thus all edges are directed from smaller to higher vertex numbers. Notice that only acyclic digraphs have topological orderings, since a directed cycle cannot be ordered in this way. Topological orderings are easy to find. We present both a breadth-first and a depth-first algorithm.

Algorithm 12.3.1: BFTOPSORT(G, n)

comment: Breadth-first topological sort of G
InDegree$[v]$ is the in-degree of vertex v, an array
ScanQ$[k]$ is the k^{th} vertex on a queue, an array
Qsize is the number of points on *ScanQ*

Qsize $\leftarrow 0$
$k \leftarrow 1$
for $v \leftarrow 1$ **to** n
do compute *InDegree*$[v]$
if *InDegree*$[v] = 0$
then *Qsize* $\leftarrow$ *Qsize* $+ 1$
ScanQ[*Qsize*] $\leftarrow v$
while $k \leq$ *Qsize*
do $u \leftarrow$ *ScanQ*$[k]$
for each v such that $u \longrightarrow v$
do *InDegree*$[v] \leftarrow$ *InDegree*$[v] - 1$
if *InDegree*$[v] = 0$
then *Qsize* $\leftarrow$ *Qsize* $+ 1$
ScanQ[*Qsize*] $\leftarrow v$
if *Qsize* $= n$
then go to 1
$k \leftarrow k + 1$
1 : **if** *Qsize* $< n$ **then** G contains a directed cycle

Algorithm 12.3.1 is the the breadth-first algorithm. The topological order is built on the queue. Algorithm 12.3.1 begins by placing all vertices with in-degree 0 on the queue. These are first in the topological order. *InDegree*$[v]$ is then adjusted so that it counts the in-degree of v only from vertices not yet on the queue. This is done by decrementing *InDegree*$[v]$ according to its in-edges from the queue. When *InDegree*$[v] = 0$, v has no more in-edges, so it too, is added to the queue. When all n vertices are on the queue, the vertices are in topological order. Notice that if G has

a directed cycle, none of the vertices of the cycle will ever be placed on the queue. In that case, the algorithm will terminate with fewer than n vertices on the queue. This is easy to detect. Computing *InDegree*$[v]$ takes $\sum_v d^-(v) = \varepsilon$ steps. Each vertex is placed on the queue exactly once, and its $d^+(u)$ out-edges are taken in turn, taking $\sum_u d^+(u) = \varepsilon$ steps. Thus the complexity of the algorithm is $O(n + \varepsilon)$.

Algorithm 12.3.2 is the depth-first topological sort algorithm and is easier to program, but somewhat subtler. It calls the recursive Procedure DFS(), and we assume that Procedure DFS() has access to the variables of Algorithm DFTOPSORT() as globals.

When Procedure DFS(u) is called from DFTOPSORT(), it builds a rooted tree, directed outward from the root u. *DFNum*$[v]$ gives the order in which the vertices are visited. The depth-first search does a traversal of this tree, using a recursive call to visit all descendants of v before v itself is assigned a number *NUM*$[v]$, its rank in the topological order. Thus, if G is acyclic, all vertices that can be reached on directed paths out of v will be ranked before v itself is ranked. Thus, for every edge (u, v), the numbering will satisfy *NUM*$[u] <$ *NUM*$[v]$. The first vertex numbered is assigned a *Rank* of n. The variable *Rank* is then decremented. So the vertices are numbered 1 to n, in topological order. It is obvious that the complexity of the algorithm is $O(n+\varepsilon)$.

Algorithm 12.3.2: DFTOPSORT(G, n)

comment:
- Depth-first topological sort of G
- *DFNum*$[v]$ is the DF-numbering assigned to the vertex v
- *NUM*$[v]$ is the topological numbering of the vertex v
- *DFCount*, *Rank* are counters

procedure DFS(v)
comment: extend the depth-first search to vertex v

DFCount $\leftarrow$ *DFCount* $+ 1$
DFNum$[v]$ $\leftarrow$ *DFCount*
for each w such that $v \longrightarrow w$
  do if *DFNum*$[w] = 0$
  then DFS(w)
NUM$[v]$ $\leftarrow$ *Rank*
Rank $\leftarrow$ *Rank* $- 1$

main
for $u \leftarrow 1$ **to** n
  do
- *NUM*$[u] \leftarrow 0$
- *DFNum*$[u] \leftarrow 0$

DFCount $\leftarrow 0$
Rank $\leftarrow n$
for $u \leftarrow 1$ **to** n
  do if *DFNum*$[u] = 0$
  then DFS(u)

The depth-first topological sort does not provide the vertices on a queue in sorted order. Instead it assigns a number to each vertex giving its rank in the topological order. If we need the vertices on a queue, as we likely will, we can construct one from the array *NUM* by executing a single loop.

for $v \leftarrow 1$ **to** n **do** $\mathit{ScanQ}[\mathit{NUM}[v]] \leftarrow v$

This works because *NUM* is a permutation of the numbers 1 to n, and the loop computes the inverse of the permutation. Another method is to compute the inverse array during the DFS simultaneously with the *NUM* array.

What happens if the depth-first topological sort is given a digraph that is not acyclic? It will still produce a numbering, but it will not be a topological ordering. We will have more to say about this in Section 12.4. Notice that $\mathrm{DFS}(u)$ may be called several times from Algorithm 12.3.2. Each time it is called, a rooted tree directed outward from the root is constructed. With undirected graphs, a DFS constructs a single rooted spanning tree of G (see Chapter 7). For directed graphs, a single out-directed tree may not be enough to span all of G. A spanning forest of rooted, out-directed trees is constructed.

We return now to the critical path method. Let G be an activity graph with $V(G) = \{1, 2, \ldots, n\}$, and suppose that the vertices have been numbered in topological order; that is, $u < v$ whenever $u \longrightarrow v$. The start vertex is $s = 1$. We set $T(1) \leftarrow 0$. We know that vertex 2 has an in-edge only from s, so $T(2)$ is assigned the cost of the edge $(1, 2)$. In general, v can have in-edges only from vertices $1, \ldots, v-1$, and we can take $T(v)$ to be

$$T(v) \leftarrow \textsc{Max}\{T(u) + \textsc{Wt}(uv) : u \longrightarrow v\}.$$

We also store an array *PrevPt*, where $\mathit{PrevPt}[v]$ is the point previous to v on a directed sv-path. If $T(v)$ is computed to be $T(u) + \textsc{Wt}(uv)$ for some u, we simultaneously assign $\mathit{PrevPt}[v] \leftarrow u$. When the algorithm completes, we can find the critical path by executing $w \leftarrow \mathit{PrevPt}[w]$ until $w = 0$, starting with $w = t\ (= n)$. The number of steps required to compute the longest paths once the topological sort has been completed is proportional to $n + \sum_v d^-(v) = O(n + \varepsilon)$.

The minimum time required to complete the project is $T(n)$. This can be achieved only if all tasks along the critical path begin and end on time. These tasks are critical. There may be some slack elsewhere in the system, though, which can be used to advantage. The earliest time at which a node v in the activity graph can be reached is $T(v)$, the length of the longest sv-path. We could also compute the latest time at which node v must be reached if the project is to finish on time. This is $T(n)$ minus the length of the longest directed path from v to t. Let $T'(v)$ be the length of the longest directed path from v to t. We can compute this in the same way that $T(v)$ is computed, but beginning with t instead of s, and working backward. Thus for each node v, we can find the two values $T(v)$ and $T(n) - T'(v)$, being the earliest and latest times at which node v can be reached. This slack time can create some flexibility in project management.

Exercises

12.3.1 Find a topological ordering of the activity graphs of Figures 12.1 and 12.2. Apply the critical path method to find the longest sv-paths and vt-paths, for each v. Work out the earliest and latest times for each node v.

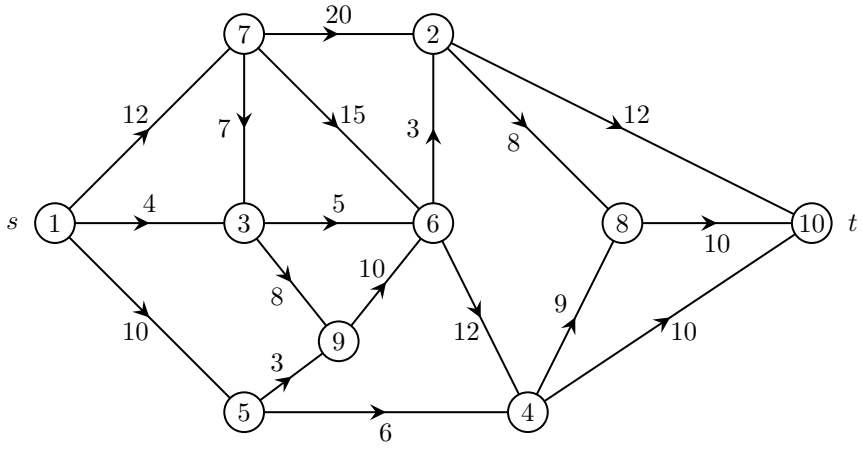


FIGURE 12.2
An activity graph

12.3.2 Program the breadth-first and depth-first topological sort algorithms. Test them on the graph of Figure 12.1.

12.3.3 Consider the recursive procedure *DFS*(v) defined in Algorithm 12.4.1, applied to a directed graph G. Suppose that *DFS*(v) has just been called, and that $A(v)$ is the set of all vertices which are ancestors of v (the path from v to the root contains the ancestors of v). Suppose that $G - A(v)$ contains a directed path from v to w. Prove that w will be visited before *DFS*(v) returns. Use induction on the length of the path from v to w.

12.4 Strong components

A digraph G is connected if every pair of vertices u and v is connected by a path. This need not be a directed path. The digraph G is *strongly connected* if every pair of vertices is connected by a *directed* path. Thus, if G is strongly connected, G contains both a uv-path and a vu-path, for every u and v. It follows that every vertex of G is contained in a directed cycle. A digraph which is strongly connected is said to be *strong*. Notice that a strong digraph does not have to be 2-connected. It may contain one or more cut-vertices.

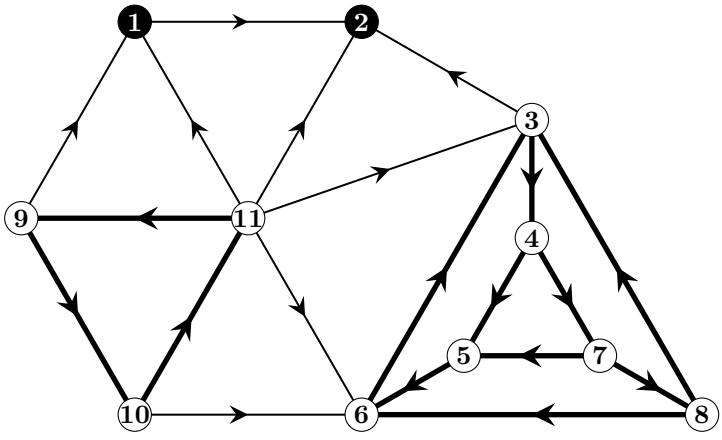


FIGURE 12.3
Strong components

By default, the complete digraph K_1 is strong, since it does not have a pair of vertices. If G is acyclic, then the only strong subgraphs of G are the individual nodes. But if G contains any directed cycle, then G will contain one or more non-trivial strongly connected subgraphs. A subgraph H is a *strong component* of G if it is a maximal strongly connected subgraph; that is, H is strong, and G has no larger subgraph containing H which is also strong. Figure 12.3 shows a digraph G with four strong components. The edges of the strong components are indicated by thicker lines. Two of the strong components are single vertices, which are shaded black.

Notice that every vertex of G is contained in exactly one strong component, but that some edges of G need not be contained in any strong component. Exercise 12.4.1. shows that this definition of strong components is well-defined.

If $G_1, G_2, \ldots, G_m$ are the strong components of G, we can construct a new digraph by contracting each strong component into a single vertex.

DEFINITION 12.1: Let $G_1, G_2, \ldots, G_m$ be the strong components of G. The *condensation* of G is the digraph whose vertices are $G_1, G_2, \ldots, G_m$, and whose edges are all ordered pairs (G_i, G_j) such that G has at least one edge directed from a vertex of G_i to a vertex of G_j.

It is proved in Exercise 12.4.3 that the condensation is an acyclic digraph.

Exercises

12.4.1 Suppose that H is a strong subgraph of G such that H is contained in two larger strong subgraphs: $H \leq H_1$ and $H \leq H_2$, where H_1 and H_2 are both strong. Show that $H_1 \cup H_2$ is strong. Conclude that the strong components of G are well-defined.

12.4.2 Show that an edge (u, v) is contained in a strong component if and only if (u, v) is contained in a directed cycle.

12.4.3 Find the condensation of the digraph of Figure 12.3. Prove that the condensation of a digraph is always acyclic.

12.4.4 The *converse* of a digraph is obtained by reversing the direction of each edge. A digraph is *self-converse* if it is isomorphic to its converse. Find all self-converse simple digraphs on one, two, three, and four vertices.

12.4.5 Show that the condensation of the converse is the converse of the condensation.

12.4.6 Let G be a self-converse simple digraph, and let G' be the converse of G. Let θ be an isomorphism of G with G', so that θ is a permutation of $V(G) = V(G')$. Prove that θ has at most one cycle of odd length. Find the possible cycle structures of θ when G has at most five vertices. Use this to find all the self-converse digraphs on five vertices.

In this section, we present an algorithm to find the strong components of a digraph G. It is based on a depth-first search. It is very similar to the DFS used to find the blocks of a graph in Chapter 7, and to the DFS used in Algorithm 12.4.1 to find a topological ordering in an acyclic digraph. When finding a topological ordering, we saw that in a digraph G, Algorithm 12.3.2 constructs a spanning forest of out-directed, rooted trees. Each time DFS(u) is called, a DF-tree rooted at u is built. The edges of G can be classified as either tree-edges or fronds. For example, a spanning forest for the graph of Figure 12.3 is shown in Figure 12.4 below. The fronds are shown as dashed edges. Not all the fronds are shown, as can be seen by comparing Figures 12.3 and 12.4. The numbering of the nodes is the DF-numbering.

Let the components of the spanning forest constructed by a depth-first search in a graph G be denoted $T_1, T_2, \ldots, T_k$, where the T_i were constructed in that order. Figure 12.4 has $k = 3$. Each T_i is an out-directed, rooted tree. Notice that each strong component of G is contained within some T_i, and that each T_i may contain more than one strong component. Fronds can be directed from a tree T_i to a previous tree T_j, where $j < i$, but not to a later tree, by nature of the depth-first search.

Given any vertex v, v is contained in some T_i. The set of ancestors of v is $A(v)$, all vertices (except v) contained in the path in T_i from v to the root of T_i. When DFS(v) is called, it will in turn call DFS(w) for several vertices w. The *branch* of T_i at v containing w is the sub-tree built by the recursive call DFS(w). For example, in Figure 12.4, there are two branches at vertex 4, constructed by the recursive calls DFS(5) and DFS(7). If x is any vertex for which $v \in A(x)$, we write $B_v(x)$ for the branch at v containing x. In Figure 12.4, we have $B_4(7) = B_4(8)$ and $B_4(5) = B_4(6)$.

Lemma 12.1. *Suppose that a depth-first search in G is visiting vertex v, and that $G - A(v)$ contains a directed path from v to w. Then vertex w will be visited before the algorithm returns from visiting v.*

Proof. Exercise 12.4.1. □

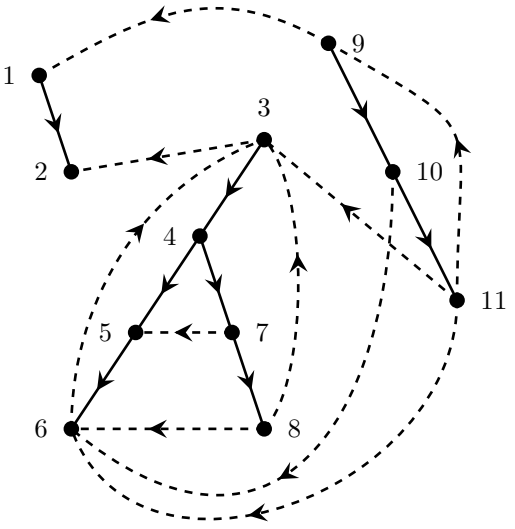


FIGURE 12.4
A depth-first, rooted, spanning forest

This lemma allows us to classify the fronds of G with respect to a depth-first forest.

Theorem 12.2. *Let $T_1, T_2, \ldots, T_k$ be the components of a depth-first spanning forest of G, where the T_i were constructed in that order. Let (x, y) be a frond, where $x \in T_i$. Then there are three possible cases:*

1. *$y \in T_j$, where $j < i$.*
2. *$y \in T_i$, and one of x and y is an ancestor of the other.*
3. *$y \in T_i$, and x and y are in different branches of a common ancestor v, where $B_v(y)$ was searched before $B_v(x)$.*

Proof. Let (x, y) be a frond, where $x \in T_i$. If $y \in T_j$, where $j \neq i$, then we must have $j < i$, for otherwise Lemma 12.1 tells us that y would be visited before DFS(x) returns, so that x would be an ancestor of y. Otherwise $x, y \in T_i$. If x is visited before y, then Lemma 12.1 again tells us that x would be an ancestor of y. This gives the second case. Otherwise y is visited before x. If G contains a directed yx-path, then we have y an ancestor of x, again the second case. Otherwise there is no directed yx-path. The paths in T_i from x and y to the root of T_i first meet in some vertex v. Then $B_v(y)$ was searched before $B_v(x)$, giving Case 3. □

We call a frond (x, y) type 1, 2, or 3 according to whether it falls in Case 1, 2, or 3 in Theorem 12.2. Fronds of type 1 cannot be part of any directed cycle since there are no edges from T_j to T_i when $j < i$. Therefore these fronds are not in any strong component. Consequently each strong component is contained within some T_i. A frond of type 2 creates a directed cycle, so that all edges of T_i on the path connecting x to y are in the same strong component. The low-point technique used to find blocks in Chapter 7 will work to find these cycles. A frond (x, y) of type 3 may or may not be part of a directed cycle. Consider the frond $(7, 5)$ in Figure 12.4. Vertices 7 and 5 are in different branches at vertex 4. Since 4 is an ancestor of 7, we have a directed path $(4, 7, 5)$. If we were to compute low-points, we would know that the low-point of 5 is vertex 3, an ancestor of 4. This would imply the existence of a directed cycle containing 3, 4, 7, and 5, namely, $(3, 4, 7, 5, 6)$. So we find that 7 is in the same strong component as 5 and that the low-point of 7 is also 3.

We can build the strong components of G on a stack. Define the low-point of a vertex v to be

$$\textit{LowPt}[v] \quad = \quad \text{the smallest } \textit{DFNum}[w],$$

where either $w = v$ or $w \in A(v)$ and G contains a directed path from v to w.

The main component of Algorithm 12.4.1 to compute the strong componets just initiallizes the variables and calls Proceedure DFS() to build each rooted tree of the spanning forest and to compute the low-points. We assume that Proceedure DFS() has access to the variables of the calling program as globals. The algorithm stores the vertices of each strong component on a stack, stored as an array. As before, we have the *DFNum*$[\cdot]$ and *LowPt*$[\cdot]$ arrays. We also store the *Stack*$[\cdot]$ as an array of vertices. *OnStack*$[v]$ is **true** if v is on the stack. *DFCount* is a global counter. *Top* is a global variable giving the index of the current top of the stack.

The Procedure DFS() computes the low-points and builds the stack. The algorithm begins by stacking each vertex that it visits. The vertices on the stack will form the current strong component being constructed. *LowPt*$[v]$ is initiallized to *DFNum*$[v]$. Each w such that $w \longrightarrow v$ is taken in turn. The statements at point (1) extend the DFS from vertex v to w. Upon returning from the recursive call, *LowPt*$[v]$ is updated. Since $v \longrightarrow w$ and G contains a directed path from w to *LowPt*$[w]$, we update *LowPt*$[v]$ if *LowPt*$[w]$ is smaller.

The statements at point (2) are executed if vw is a frond. If *DFNum*$[w] >$ *DFNum*$[v]$, it means that v is an ancestor of w. These fronds are ignored. Otherwise w was visited before v. If w is the parent of v, vw is a tree edge rather than a frond, and is ignored. If w is no longer on the stack, it means that w is in a strong component previously constructed. The edge vw cannot be part of a strong component in that case, so it is also ignored. If each of these tests is passed, G contains a directed path from v to *LowPt*$[w]$, which is in the same strong component as w. Therefore v and w are in the same strong component. If this value is smaller than *LowPt*$[v]$, then *LowPt*$[v]$ is updated. Statement (3) is reached after all w adjacent to v have been considered. At this point the value of *LowPt*$[v]$ is known. If *LowPt*$[v] =$ *DFNum*$[v]$, it means that there is no directed path from v to any ancestor of v. Every vertex of

the strong component containing v has been visited, and so is on the stack. These vertices are then popped off the stack before returning.

Algorithm 12.4.1: STRONGCOMPONENTS(G, n)

comment: Find the strong components using a depth-first search.

procedure DFS(v)
comment: extend the depth-first search to vertex v
DFCount $\leftarrow DFCount + 1$
DFNum$[v] \leftarrow$ *DFCount*
LowPt$[v] \leftarrow$ *DFCount* "initial value"
Top $\leftarrow$ *Top* $+ 1$
Stack[*Top*] $\leftarrow v$ "push v on *Stack*"
OnStack$[v] \leftarrow$ **true**
for each w such that $v \longrightarrow w$
do
- **if** *DFNum*$[w] = 0$ (1)
  - **then**
    - DFS(w)
    - **if** *LowPt*$[w] <$ *LowPt*$[v]$
      - **then** *LowPt*$[v] \leftarrow$ *LowPt*$[w]$
  - **else**
    - **if** *DFNum*$[w] <$ *DFNum*$[v]$ (2)
      - **then if** $w \neq$ parent of v
      - **then if** *OnStack*$[w]$
      - **then if** *LowPt*$[w] <$ *LowPt*$[v]$
      - **then** *LowPt*$[v] \leftarrow$ *LowPt*$[w]$

if *LowPt*$[v] =$ *DFNum*$[v]$ (3)
then
- **comment:** the points on the stack up to v form a strong component – pop them off
- **repeat**
  - $w \leftarrow$ *Stack*[*Top*]
  - *Top* $\leftarrow$ *Top* $- 1$
  - *OnStack*$[w] \leftarrow$ **false**
- **until** $w = v$

main
for $u \leftarrow 1$ **to** n
do
- *DFNum*$[u] \leftarrow 0$
- *OnStack*$[v] \leftarrow$ **false**

DFCount $\leftarrow 0$
Top $\leftarrow 0$
for $u \leftarrow 1$ **to** n
do if *DFNum*$[u] = 0$
then $DFS(u)$

The complexity of Algorithm 12.4.1 is easily seen to be $O(n+\varepsilon)$. For each vertex v, all out-edges vw are considered, giving

$$\sum_v d^+(v) = \varepsilon$$

steps. Some arrays of length n are maintained. Each node is stacked once, and removed once from the stack.

We finish this section with the following theorem.

Theorem 12.3. (Robbins' Theorem)
Every 2-connected graph has a strong orientation.

Proof. Let G be a 2-connected graph. Then G contains a cycle C, which has a strong orientation. Let H be a subgraph of G with the largest possible number of vertices, such that H has a strong orientation. If $u \notin H$, then since G is 2-connected, we can find two internally disjoint paths P and Q connecting u to H. Orient P from u to H, and Q from H to u. This gives a strong orientation of a larger subgraph than H, a contradiction. □

12.4.1 An application to fabrics

A fabric consists of two sets of strands at right angles to each other, called the *warp and weft*, woven together. The pattern in which the strands are woven can be represented by a rectangular matrix. Let the horizontal strands be $h_1, h_2, \ldots, h_m$ and let the vertical strands be $v_1, v_2, \ldots, v_n$. The matrix shown below contains an X wherever h_i passes under v_j, and a blank otherwise. The pattern can be repeated as often as desired.

	v_1	v_2	v_3	v_4	v_5	v_6
h_1	X		X	X		X
h_2		X		X	X	
h_3	X		X			X

Suppose that the strand h_1 were lifted. Since it passes under v_1, v_3, v_4, and v_6, these vertical strands would also be lifted. But since v_1 passes under h_2, this would in turn lift h_2. Similarly lifting h_2 would cause v_2 to be lifted, which in turn causes h_3 to be lifted. So the fabric hangs together if any strand is lifted.

For some pattern matrices, it is quite possible that the fabric defined does not hang together. For example, in the simplest case, a strand h_i could lie under or over every v_j, allowing it to be lifted off the fabric, or the fabric could fall apart into two or more pieces. In general, we can form a bipartite directed graph whose vertices are the set of all strands. The edges are

$$\{(u, w) : \text{ strand } u \text{ lies under strand } w\}.$$

Call this the *fabric graph*. It is an oriented complete bipartite graph.

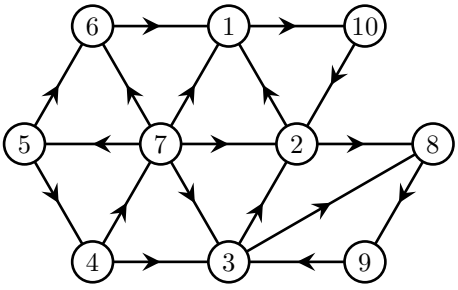


FIGURE 12.5
Find the strong components

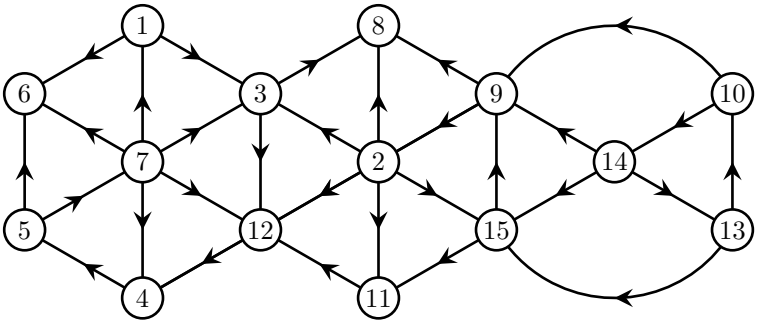


FIGURE 12.6
Find the strong components

If the fabric graph is strongly connected, then it hangs together, since there is a directed path from any strand to another. If the fabric graph is not strongly connected, then it can be separated into its strong components. Some strong component will lie completely over another strong component, and be capable of being lifted off.

Exercises

12.4.1 Prove Lemma 12.1.

12.4.2 Program the algorithm for strong components, and test it on the digraphs of Figures 12.1, 12.2, 12.3, 12.5, and 12.6.

12.4.3 Find all digraphs which can be obtained by orienting a cycle of length 5 or 6.

12.4.4 Determine whether the fabric defined by the pattern matrix in Figure 12.7 hangs together.

			X		X		
X						X	
X	X		X	X	X	X	
X			X		X		
X		X	X	X	X	X	X
			X			X	
X	X		X		X	X	
X		X	X		X	X	X

FIGURE 12.7
A pattern matrix

12.5 Tournaments

In a round-robin tournament with n teams, each team plays every other team. Assuming that ties are not allowed, we can represent a win for team u over team v by a directed edge (u, v). When all games have been played we have a directed complete graph. We say that a *tournament* is any oriented complete graph. It is easy to see that there are exactly two possible tournaments on three vertices, as shown in Figure 12.8.

FIGURE 12.8
The tournaments on three vertices

The second of these tournaments has the property that if $u \longrightarrow v$ and $v \longrightarrow w$, then $u \longrightarrow w$. Any tournament which has this property for all vertices u, v, w, is called a *transitive tournament*. It is easy to see that there is a unique transitive tournament T_n for each $n \geq 1$. For if T_n is a transitive tournament, there must be a unique node u that is undefeated. If we delete it, we are left with a transitive tournament on $n-1$ vertices. We use induction to claim that this is the unique T_{n-1}. When u is restored, we have the uniqueness of T_n.

If G is any tournament on n vertices, it will have a number of strong components. Let G^* denote its condensation. Then since G^* is acyclic, it is a transitive tournament on $m \leq n$ vertices. We can find a topological ordering of $V(G^*)$, and this will define an ordering of the strong components of G. We can then make a list of the possible sizes of the strong components of G, ordered according to the topological ordering of G^*. We illustrate this for $n = 4$. The possible sizes of the strong components are $(1, 1, 1, 1)$, $(1, 3)$, $(3, 1)$, and (4), since a simple digraph cannot have a strong

component with only two vertices. The first ordering corresponds to the transitive tournament T_4. The orderings $(1,3)$ and $(3,1)$ correspond to the first two tournaments of Figure 12.9. It is easy to see that they are unique, since there is only one strong tournament on three vertices, namely the directed cycle.

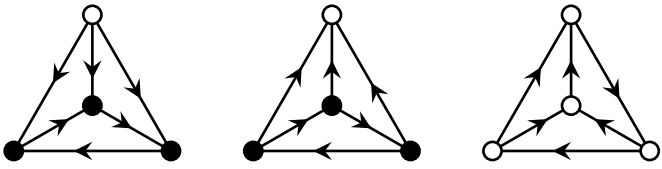


FIGURE 12.9
Non-transitive tournaments on 4 vertices

The third tournament is strongly connected. We leave it to the reader to verify that it is the only strong tournament on four vertices. The following theorem will be helpful. A digraph is said to be hamiltonian if it contains a directed hamilton cycle.

Theorem 12.4. *Every strong tournament on $n \geq 3$ vertices is hamiltonian.*

Proof. Let G be a strong tournament, and let $C = (u_1, u_2, \ldots, u_k)$ be the longest directed cycle in G. If G is non-hamiltonian then $\ell(C) < n$. Pick any $v \notin C$. Because G is a tournament, either $v \longrightarrow u_1$ or else $u_1 \longrightarrow v$. Without loss, suppose that $u_1 \longrightarrow v$. If $v \longrightarrow u_2$, then we can produce a longer cycle by inserting v between u_1 and u_2. Therefore $u_2 \longrightarrow v$. If $v \longrightarrow u_3$, then we can produce a longer cycle by inserting v between u_2 and u_3. Therefore $u_3 \longrightarrow v$, etc. Eventually we have $v \longrightarrow u_i$ for all u_i. This is impossible, since G is strong. □

12.5.1 Modules

Modules for undirected graphs were described in section 7.5. They provide a helpful decomposition of tournaments.

DEFINITION 12.2: Let T be a tournament and let $U \subseteq V(T)$ be a set of vertices of T. Then U is a *module* or *interval* of T if it has the property: if $u \in V(G) - U$ then either $u \longrightarrow w$, for all $w \in U$, or else $w \longrightarrow u$, for all $w \in U$.

Clearly all singleton sets $\{u\}$ are modules, as are $\emptyset$ and $V(T)$. These are called *trivial* modules.

DEFINITION 12.3: A tournament is *indecomposable* if all its modules are trivial. Otherwise it is *decomposable*. A *modular partition* of T is a partition of $V(T)$ into modules.

Given a modular partition $\{U_1, U_2, \ldots, U_m\}$ of a decomposable tournament T, a quotient tournament can be constructed, whose vertices are $\{U_1, U_2, \ldots, U_m\}$, and the module $U_i \longrightarrow U_j$ if and only if some $u \in U_i$ and $v \in U_j$ satisfy $u \longrightarrow v$. This

provides a means of reducing T to a smaller tournament, useful in algorithms. An algorithm to find the modules of a tournament appears in [26].

Gallai discovered a modular partition of tournaments with special properties. It is described in [62]. Schmerl and Trotter [155] develop the theory of critically indecomposable tournaments and other structures.

Exercises

12.5.1 Show that there is a unique strong tournament on four vertices.

12.5.2 Find all the tournaments on five vertices. Show that there are exactly 12 tournaments, of which 6 are strong.

12.5.3 Show that every tournament has a hamilton path.

12.5.4 Show that if an odd number of teams play in a round robin tournament, it is possible for all teams to tie for first place. Show that if an even number of teams play, it is not possible for all teams to tie for first place.

12.5.5 Prove the following theorem. Let G be a digraph on n vertices such that $d^+(u) + d^-(v) \geq n$ whenever $u \not\longrightarrow v$. Then G is strong.

12.5.6 Describe a $O(n+\varepsilon)$ algorithm to find a strong orientation of a 2-connected graph.

12.5.7 Show that every connected graph has an acyclic orientation.

12.5.8 Show that a strongly connected tournament is indecomposable.

12.5.9 Show that a tournament that is not strongly connected has a modular partition with just two vertex classes.

12.5.10 Prove the following properties of modules:

if X and Y are modules of T, then so is $X \cap Y$.

if X and Y are modules of T, and $X \cap Y \neq \emptyset$, then $X \cup Y$ is a module of T.

12.6 2-Satisfiability

In Chapter 11 we saw that 3-Sat is NP-complete. The related problem 2-Sat $\in$ P. It has a number of practical applications.

Problem 12.1:	2-Sat
Instance:	a set of boolean variables U and boolean expression B over U, in which each clause contains exactly two variables.
Question:	is B satisfiable?

Consider the following instance of 2-Sat:

$$(u_1 + u_2)(u_1 + \overline{u}_2)(u_2 + u_3)(\overline{u}_1 + \overline{u}_3) \tag{12.1}$$

We want a truth assignment satisfying this expression. If $u_1 =$ **false**, the first clause tells us that $u_2 =$ **true**. We could write this implication as $\overline{u}_1 \rightarrow u_2$. The second clause tells us that if $u_1 =$ **false**, then $u_2 =$ **false**. We could write this implication as $\overline{u}_1 \rightarrow \overline{u}_2$. As this gives a contradiction, we conclude that $u_1 =$ **true** is necessary. Continuing in this line of reasoning quickly gives the solution.

This example shows that a clause $(x + y)$ of an instance of 2-Sat, where $x, y \in U \cup \overline{U}$, corresponds to two implications $\overline{x} \rightarrow y$ and $\overline{y} \rightarrow x$. We can construct a digraph with edges based on these implications.

Given an instance of 2-Sat with variables U and boolean expression B, construct a digraph G whose vertex set is $U \cup \overline{U}$. The edges of G consist of all ordered pairs $(\overline{x}, y)$ and $(\overline{y}, x)$, where $(x + y)$ is a clause of B. G is called the *implication digraph* of B. The implication digraph corresponding to instance (12.1) is shown in Figure 12.10. A sequence of implications $\overline{x} \rightarrow y \rightarrow z$ corresponds to a directed path in G. Thus, directed paths in G are important. If any variable in a directed path is assigned the value **true**, then all subsequent variables in the path must also be **true**. Similarly, if any variable in a directed path is assigned the value **false**, then all previous variables must also be **false**. If G contains a directed cycle C, then all variables of C must be **true**, or all must be **false**. We are talking about the strong components of G.

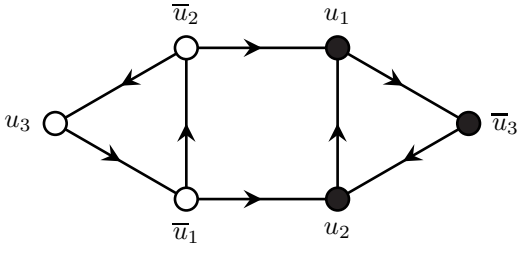


FIGURE 12.10
The implication digraph corresponding to an instance of 2-Sat

The graph G has an *antisymmetry* – if there is an edge (x, y), then there is also an edge $(\overline{y}, \overline{x})$, as can be seen from the definition. Therefore the mapping of $V(G)$ that interchanges every u_i and $\overline{u}_i$ reverses the orientation of every edge.

Let $G_1, G_2, \ldots, G_m$ be the strong components of G. The variables in any strong component are either all true, or all false. If any strong component contains both u_i and $\overline{u}_i$, for any i, then the expression B is not satisfiable; for u_i and $\overline{u}_i$ cannot be both true, or both false. So if B is satisfiable, u_i and $\overline{u}_i$ are in different strong components.

The antisymmetry maps each strong component G_j to another strong component G'_j, the *complementary strong component*, such that x is in G_j if and only if $\overline{x}$ is

in G'_j. If all variables of G_j are true, then all variables of G'_j must be false, and conversely. This gives the following algorithm for 2-Sat:

Algorithm 12.6.1: 2SAT(B, U)

comment: { Given an instance B of 2-Sat with variables U, construct a solution, if there is one.

construct the implication digraph G corresponding to B
construct the strong components of G
for each $u \in U$
  do if u and $\overline{u}$ are in the same strong component
  then return (*NonSatisfiable*)
construct the *condensation* of G, and find a topological ordering of it
let $G_1, G_2, \ldots, G_m$ be the topological ordering of the strong components
for $i \leftarrow m$ **downto** 1
do { **if** the variables of G_i have not been assigned
  then { assign all variables of G_i to be **true**
  assign all variables of G'_i to be **false**

In the graph of Figure 12.10 there are two strong components – the shaded vertices and the unshaded vertices. The condensation is a digraph with one edge, directed from left to right. The algorithm will assign **true** to all variables in the shaded strong component, and **false** to all variables in the unshaded one. This is the unique solution for this instance of 2-Sat.

Theorem 12.5. *Given an instance B of 2-Sat with variables U. Algorithm* 2SAT*(B,U) finds a solution if and only if a solution exists.*

Proof. Every clause $(x+y)$ of B corresponds to two implications $\overline{x} \rightarrow y$ and $\overline{y} \rightarrow x$. The implication digraph G contains all these implications. Any assignment of truth values to the variables that satisfies all implications satisfies B. If some strong component of G contains both u_i and $\overline{u}_i$, for some variable u_i, then there is no solution. The algorithm will detect this. Otherwise, u_i and $\overline{u}_i$ are always in complementary strong components. The algorithm assigns values to the variables such that complementary strong components always have opposite truth values. Therefore, for every u_i and $\overline{u}_i$, exactly one will be true, and one will be false. Consider a variable $x \in U \cup \overline{U}$. Suppose that x is in a strong component G_j. Its complement $\overline{x}$ is in G'_j. Without loss, suppose that G'_j precedes G_j in the topological order. Then x will be assigned **true** and $\overline{x}$ will be assigned **false**. All clauses $(x + y)$ containing x are thereby satisfied. All clauses $(\overline{x} + z)$ containing $\overline{x}$ correspond to implications $x \rightarrow z$ and $\overline{z} \rightarrow \overline{x}$. It follows that z is either in the same strong component as x, or else in a strong component following G_j, and $\overline{z}$ is in a strong component preceding G'_j. In either case, the algorithm has already assigned $z \leftarrow$ **true**, so that the clause $(\overline{x} + z)$ is also satisfied. We conclude that the truth assignment constructed by the algorithm satisfies B. □

This also gives the following theorem.

Theorem 12.6. *Given an instance B of 2-Sat with variables U. B is satisfiable if and only if, for every $u_i \in U$, u_i and $\overline{u}_i$ are contained in different strong components of the implication digraph.*

If U has n variables, and B has k clauses, the implication graph G will have $2n$ vertices and $2k$ edges. It takes $O(n+k)$ steps to construct G, and $O(n+k)$ steps to find its strong components, and to construct a topological order of them. It then takes $O(n)$ steps to assign the truth values. Thus, we have a linear algorithm that solves 2-Sat.

Exercises

12.6.1 Construct the implication digraph for the following instance of 2-Sat.

$$(u_1+u_2)(\overline{u}_1+u_3)(\overline{u}_3+\overline{u}_4)(u_1+u_4)(\overline{u}_2+\overline{u}_5)(u_5+\overline{u}_6)(u_2+u_6)(\overline{u}_3+u_4)$$

12.6.2 Solve the previous instance of 2-Sat.

12.6.3 Given an instance of 2-Sat with the additional requirement that $u_1 =$ **true**. Show how to convert this into an instance of 2-Sat and solve it. Show also how to solve it if u_1 is required to be **false**.

12.6.4 Consider an instance of Sat in which each clause has exactly two variables, except that one clause has three or more variables. Describe an algorithm to solve it in polynomial time.

12.7 Notes

An excellent reference for digraphs is BANG-JENSEN and GUTIN [9]. The algorithms for strong components is from AHO, HOPCROFT, and ULLMAN [1]. Strong components and 2-satisfiability are further examples of the importance and efficiency of the depth-first search. The subject of tournaments is a vast area. A survey can be found in REID and BEINEKE [146]. A good monograph on tournaments is MOON [127]. STOCKMEYER [164] discovered an infinite family of non-reconstructible tournaments. THOMASSEN [170] discovered that the structure of a tournament is largely determined by its directed 4-cycles. LOPEZ [118] showed that if two digraphs G and G' with $V(G) = V(G')$, have the property that the induced digraphs $G[U]$ and $G'[U]$ are isomorphic for all subsets $U \subseteq V(G)$, where $|U| \leq 6$, then $G \cong G'$.

13

Graph Colorings

13.1 Introduction

A *coloring* of the vertices of a graph G is an assignment of colors to the vertices. A coloring is *proper* if adjacent vertices always have different colors. We shall usually be interested only in proper colorings. It is clear that the complete graph K_n requires n distinct colors for a proper coloring. Any bipartite graph can be colored in just two colors. More formally,

DEFINITION 13.1: An *m-coloring* of G is a mapping from $V(G)$ onto the set $\{1, 2, \ldots, m\}$ of m "colors". The *chromatic number* of G is $\chi(G)$, the minimum value m such that G has a proper m-coloring. If $\chi(B) = m$, G is then said to be *m-chromatic*.

If G is bipartite, we know that $\chi(G) = 2$. Moreover, there is an $O(\varepsilon)$ algorithm to determine whether an arbitrary G is bipartite, and to construct a 2-coloring of it. When $\chi(G) \geq 3$, the problem becomes NP-complete. We show this in Section 13.8. (See Problem 13.1.) A consequence of this is that there is no complete theoretical characterization of colorability. As with the Hamilton cycle problem, there are many interesting techniques, but most problems only have partial solutions. We begin with a simple algorithm for coloring a graph, the *sequential algorithm*. We will indicate various colorings of G by the notation $\chi_1, \chi_2, \ldots$, etc. While $\chi(G)$ represents the chromatic number of G, there may be many colorings of G that use this many colors. If χ_1 is a coloring, then $\chi_1(v)$ represents the color assigned to vertex v under the coloring χ_1.

Algorithm 13.1.1: SEQUENTIALCOLORING(G, n)

comment: Construct a coloring χ_1 of a graph G on n vertices

mark all vertices "uncolored"
order $V(G)$ in some sequence $v_1, v_2, \ldots, v_n$
for $i \leftarrow n$ **downto** 1
  do $\chi_1(v_i) \leftarrow$ the first color available for v_i

To calculate the first available color for v_i, we consider each $u \rightarrow v_i$. If u is already colored, we mark its color "in use". We then take the first color not in use

as $\chi_1(v_i)$, and then reset the color flags before the next iteration. Thus, iteration i of the for-loop takes $O(\text{DEG}(v_i))$ steps. Algorithm 13.1.1 is then easily seen to be $O(\varepsilon)$. The number of colors that it uses usually depends on the sequence in which the vertices are taken.

If we knew a proper $\chi(G)$-coloring before beginning the algorithm, we could order the vertices by color: all vertices of color 1 last, then color 2, etc. Algorithm 13.1.1 would then color G in exactly $\chi(G)$ colors. Because we do not know $\chi(G)$ beforehand, we investigate various orderings of $V(G)$.

Spanning trees often give useful orderings of the vertices. We choose a vertex v_1 as a root vertex, and build a spanning tree from it. Two immediate possibilities are a breadth-first or depth-first tree. The ordering of the vertices would then be the order in which they are numbered in the spanning tree. This immediately gives the following lemma.

Lemma 13.1. *If G is a connected, non-regular graph, then $\chi(G) \leq \Delta(G)$. If G is regular, then $\chi(G) \leq \Delta(G) + 1$.*

Proof. Assume first that G is non-regular, and choose a vertex v_1 of degree $< \Delta(G)$ as the root of a spanning tree. Apply Algorithm 13.1.1 to construct a coloring χ_1. When each vertex $v_i \neq v_1$ comes to be colored, the parent of v_i has not yet been colored. Therefore at most $\text{DEG}(v_i) - 1$ adjacent vertices have already been colored. Hence $\chi_1(v_i) \leq \Delta(G)$. When v_1 comes to be colored, all adjacent vertices have already been colored. Since $\text{DEG}(v_1) < \Delta(G)$, we conclude that $\chi_1(v_1) \leq \Delta(G)$. Hence $\chi(G) \leq \Delta(G)$.

If G is regular, then the proof proceeds as above, except that $\chi_1(v_1) \leq \Delta(G)+1$. The conclusion follows. Notice that if G is regular, only one vertex needs to use color $\Delta(G) + 1$. □

If Algorithm 13.1.1, using a breadth-first or depth-first spanning tree ordering, is applied to a complete graph K_n, it is easy to see that it will use exactly n colors. If the algorithm is applied to a cycle C_n, it will always use two colors if n is even, and three colors, if n is odd. Complete graphs and odd cycles are special cases of Brooks' theorem.

Theorem 13.2. (Brooks' theorem) *Let G be a connected graph that is not a complete graph, and not an odd cycle. Then $\chi(G) \leq \Delta(G)$.*

Proof. Suppose first that G is 3-connected. If G is non-regular, we know the result to be true. Hence we assume that G is a regular graph. Since G is not complete, we can choose vertices u, v, w such that $u \to v, w$, but $v \not\to w$. Construct a spanning tree ordering of $G - \{v, w\}$, with u as root, so that $v_1 = u$. Then set $v_{n-1} = v$, $v_n = w$, and apply Algorithm 13.1.1 to construct a coloring χ_1 of G. Vertices v and w will both be assigned color 1. Each $v_i \neq v_1$ will have $\chi_1(v_i) \leq \Delta(G)$. When v_1 comes to be colored, the two adjacent vertices v and w will have the same color. Hence $\chi_1(v_1) \leq \Delta(G)$.

Otherwise G is not 3-connected. Suppose that G is 2-connected. Choose a pair of vertices $\{u, v\}$, such that $G - \{u, v\}$ is disconnected. If H is any connected component of $G - \{u, v\}$, let H_{uv} be the subgraph of $G + uv$ induced by $V(H) \cup \{u, v\}$. Now G is regular, of degree at least three (or G would be a cycle). Therefore we can choose u and v so that H_{uv} is a non-regular graph. Therefore $\chi(H_{uv}) \leq \Delta(H_{uv})$. But $\Delta(H_{uv}) \leq \Delta(G)$. Color H_{uv} in at most $\Delta(G)$ colors. Notice that u and v have different colors, because $uv \in E(H_{uv})$. We can do the same for every subgraph K_{uv} so constructed from each connected component of $G - \{u, v\}$. Furthermore, we can require that u and v are colored identically in each K_{uv}, by permuting colors if necessary. These subgraph colorings determine a coloring of G with at most $\Delta(G)$ colors.

If G is not 2-connected, but has a cut-vertex u, we use an identical argument, deleting only u in place of $\{u, v\}$. □

13.1.1 Intersecting lines in the plane

An interesting example of the use of Algorithm 13.1.1 is given by intersecting lines in the plane. Suppose that we are given a collection of m straight lines in the plane, with no three concurrent. Construct a graph G whose vertices are the points of intersection of the lines, and whose edges are the line segments connecting the vertices. An example is shown in Figure 13.1. We can use Algorithm 13.1.1 to show that $\chi(G) \leq 3$. Notice first that because at most two lines are concurrent at a point that we have $\Delta(G) \leq 4$. Also note that G can be enclosed by a disc in the plane. Choose a disc in the plane containing G, and choose a line ℓ in the plane that is outside the disc, such that ℓ is not parallel to any of the original m lines. The dotted line in Figure 13.1 represents ℓ. Assign an orientation to the edges of G by directing each edge toward ℓ. This converts G to an acyclic digraph. Notice that each vertex has at most two incident edges oriented outward, and at most two oriented inward. Let $v_1, v_2, \ldots, v_n$ be a topological ordering of $V(G)$. Apply Algorithm 13.1.1. The vertices of G have degree two, three, or four. When each v_i comes to be colored, at most two adjacent vertices have already been colored – those incident on out-edges from v_i. Therefore $\chi_1(v_i) \leq 3$, for each v_i. It follows that $\chi(G) \leq 3$.

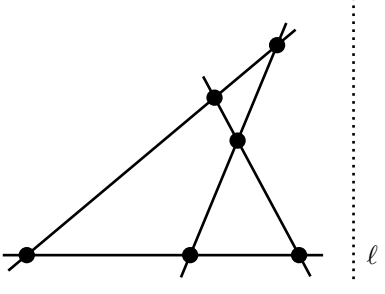


FIGURE 13.1
Intersecting lines in the plane

Exercises

13.1.1 Write computer programs to apply the sequential algorithm using breadth-first and depth-first searches, and compare the colorings obtained.

13.1.2 Show that if a breadth-first or depth-first sequential algorithm is applied to a bipartite graph, that exactly two colors will be used.

13.1.3 Construct a complete bipartite graph $K_{n,n}$ with bipartition $X = \{x_1, x_2, \ldots, x_n\}$, $Y = \{y_1, y_2, \ldots, y_n\}$, and remove the matching $M = \{x_1y_1, \ldots, x_ny_n\}$, to get $G = K_{n,n} - M$. Order the vertices $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$. Show that Algorithm 13.1.1 with this ordering will use n colors.

13.1.4 Construct a complete tripartite graph $K_{n,n,n}$ with tripartition $X = \{x_1, x_2, \ldots, x_n\}$, $Y = \{y_1, y_2, \ldots, y_n\}, Z = \{z_1, z_2, \ldots, z_n\}$, and remove the triangles $T = \{x_1y_1z_1, \ldots, x_ny_nz_n\}$, to get $G = K_{n,n,n} - T$. Order the vertices $x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n$. Show that Algorithm 13.1.1 with this ordering will use n colors. How many colors will be used by the breadth-first sequential algorithm?

13.1.5 Show that in the intersecting lines problem, if three or more lines are allowed to be concurrent, that $\chi(G)$ can be greater than three.

13.1.6 Let G_1 and G_2 be graphs with m-colorings χ_1 and χ_2, respectively. We say that G_1 and G_2 are *color-isomorphic* if there is an isomorphism θ from G_1 to G_2 that induces a permutation of the colors. More formally, if $u_1, u_2 \in V(G_1)$ are mapped by θ to $v_1, v_2 \in V(G_2)$, respectively, then $\chi_1(u_1) = \chi_1(u_2)$ if and only if $\chi_2(v_1) = \chi_2(v_2)$. Show how to construct graphs G_1' and G_2' such that G_1 and G_2 are color-isomorphic if and only if G_1' and G_2' are isomorphic.

13.1.7 Determine whether the graphs of Figure 13.2 are color-isomorphic, where the colors are indicated by the numbers.

13.1.8 Let the vertices of G be listed in the sequence $v_1, v_2, \ldots, v_n$ and apply Algorithm 13.1.1. Show that $\chi_1(v_i)$ is at most $\min\{n - i + 1, \text{DEG}(v_i) + 1\}$. Conclude that $\chi(G) \leq \max_{i=1}^{n} \min\{n - i + 1, \text{DEG}(v_i) + 1\}$.

13.2 Cliques

Let G be a graph with a proper coloring. The subset of vertices of color i is said to be the *color class* with color i. These vertices induce a subgraph with no edges. In $\overline{G}$ they induce a complete subgraph.

DEFINITION 13.2: A *clique* in G is an induced complete subgraph. Thus a clique is a subset of the vertices that are pairwise adjacent. An *independent set* is a subset

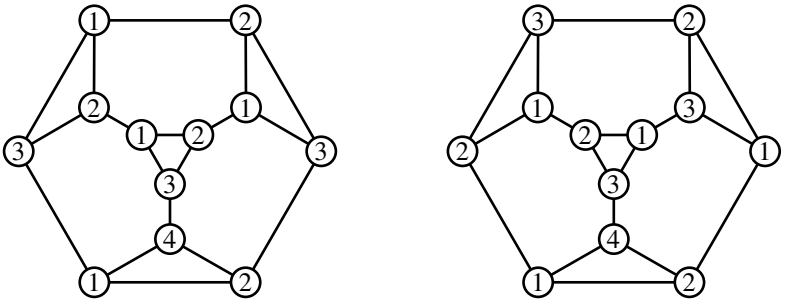


FIGURE 13.2
Are these color-isomorphic?

of $V(G)$ which induces a subgraph with no edges. A clique is a *maximum clique* if G does not contain any larger clique. Similarly, an independent set is a *maximum independent set* if G does not contain any larger independent set. An independent set is also called a *stable set*.

The problem of finding a maximum clique or maximum independent set in G is NP-complete, as shown in Section 13.8.

Write $\alpha(G)$ for the number of vertices in a maximum independent set in G, and $\overline{\alpha}(G)$ for the number of vertices in a maximum clique. It is clear that $\chi(G) \geq \overline{\alpha}(G)$, since a clique in G of m vertices requires at least m colors. If we are given a coloring χ_1 of G and χ_2 of $\overline{G}$, let $V_{\max}$ be a largest color class in G, and let $\overline{V}_{\max}$ be a largest color class in $\overline{G}$. Since $\overline{V}_{\max}$ induces a clique in G, we have $\overline{\alpha}(G) \geq |\overline{V}_{\max}|$. Since each color class is an independent set, $\alpha(G) \geq |V_{\max}|$. This gives the bounds

$$|\overline{V}_{\max}| \leq \overline{\alpha}(G) \leq \chi(G)$$

and

$$|V_{\max}| \leq \alpha(G) \leq \chi(\overline{G}).$$

Although we do not know $\chi(G)$ and $\chi(\overline{G})$, we can use the sequential Algorithm 13.1.1 to construct colorings χ_1 and χ_2 of G and $\overline{G}$, so as to obtain bounds on the clique number and independent set number of G and $\overline{G}$. We write $\chi_1(G)$ to denote the number of colors used by the coloring χ_1.

Lemma 13.3. *If χ_1 and χ_2 satisfy $|V_{\max}| = \chi_2(\overline{G})$, then $\alpha(G) = \chi(\overline{G}) = \chi_2(\overline{G})$. If χ_1 and χ_2 satisfy $|\overline{V}_{\max}| = \chi_1(G)$, then $\overline{\alpha}(G) = \chi(G) = \chi_1(G)$.*

Proof. The inequalities above hold for any proper colorings χ_1 and χ_2. If $|V_{\max}| = \chi_2(\overline{G})$, then the second inequity determines $\alpha(G)$. Therefore $\chi(\overline{G})$ is at least as big as this number. But since we have a coloring in $\chi_2(\overline{G})$ colors, this determines $\chi(\overline{G})$. □

Thus, by using Algorithm 13.1.1 to color both G and $\overline{G}$, we can obtain bounds on $\alpha(G), \overline{\alpha}(G), \chi(G)$, and $\chi(\overline{G})$. Sometimes this will give us exact values for some of these parameters. When it does not give exact values, it gives colorings χ_1 and χ_2, as well as a clique $\overline{V}_{\max}$ and independent set $V_{\max}$ in G. In general, Algorithm 13.1.1 does not construct colorings that are optimal, or even nearly optimal. There are many variations of Algorithm 13.1.1. Improvements to Algorithm 13.1.1 will give improvements in the above bounds, by decreasing the number of colors used, and increasing the size of the maximum color classes found. One modification that is found to work well in practice is the *degree saturation method* of Brelaz [25], which orders the vertices by the "saturation" degree.

Consider a graph G for which a coloring χ_1 is being constructed. Initially all vertices are marked uncolored. On each iteration of the algorithm, another vertex is colored. For each vertex, let $c(v)$ denote the number of distinct colors adjacent to v. $c(v)$ is called the *saturation degree* of v. Initially $c(v) = 0$. At any stage of the algorithm the vertices are partially ordered. A vertex u such that the pair $(c(u), \text{DEG}(u))$ is largest is chosen as the next vertex to color; that is, vertices are compared first by $c(u)$, then by $\text{DEG}(u)$.

Algorithm 13.2.1: DEGREESATURATION(G, n)

comment: Construct a coloring χ_1 of a graph G on n vertices

mark all vertices "uncolored"
initialize $c(v) \leftarrow 0$, for all v
for $i \leftarrow 1$ **to** n
do { select u as a vertex with largest $(c(u), \text{DEG}(u))$
$\chi_1(u) \leftarrow$ the first color available for u
for each v⟶u
do if v is uncolored, adjust $c(v)$

Algorithm 13.2.1 requires a priority queue in order to efficiently select the next vertex to color. In practice, it is found that it uses significantly fewer colors than Algorithm 13.1.1 with a fixed ordering of the vertices. However, algorithms based on the sequential algorithm are limited in their efficacy. JOHNSON [95] discusses the limitations of many coloring algorithms based on the sequential algorithm.

The first vertex, u_1, that Algorithm 13.2.1 colors will be one of maximum degree. The second vertex, u_2, will be a vertex adjacent to u_1 of maximum possible degree. The third vertex, u_3, will be adjacent to u_1 and u_2, if there is such a vertex, and so on. Thus, Algorithm 13.2.1 begins by constructing a clique of vertices of large degree. The algorithm could save this information, and use it as a lower bound on $\overline{\alpha}(G)$.

A description and comparison of various coloring algorithms can be found in the book by Lewis [115], in particular *DSatur* (Degree Saturation), *TabuCol, PartialCol, AntCol*, *Hill Climbing*, the *Hybrid Evolutionary* algorithm, and also a backtracking *DSatur* algorithm. This book also contains an extensive list of graph coloring references.

A recursive search algorithm MAXCLIQUE() to find a maximum clique can be based on a similar strategy. Algorithm 13.2.2 that follows constructs cliques C'. The maximum clique found is stored in a variable C. C and C' are stored as global arrays. It also uses an array S of vertices eligible to be added to C'. The set of all neighbors of v is denoted by $N(v)$.

Algorithm 13.2.2: MAXCLIQUE(G, n)

comment: Construct a maximum clique C in G

procedure EXTENDCLIQUE(S, v)
comment: v has just been added to C' – adjust S and extend the clique
$S' \leftarrow S \cap N(v)$
if $|S'| = 0$
then { **if** $|C'| > |C|$ **then** $C \leftarrow C'$; **return**
while $|S'| > 0$ **and** $|C'| + |S'| > |C|$
do { select $u \in S'$; $C' \leftarrow C' \cup u$; $S' \leftarrow S' - u$; EXTENDCLIQUE(S', u) ; $C' \leftarrow C' - u$

main
choose an ordering $v_1, v_2, \ldots, v_n$ of the vertices
$C \leftarrow \emptyset$ "largest clique found so far"
$C' \leftarrow \emptyset$ "clique currently being constructed"
$S \leftarrow V(G)$ "vertices eligible for addition to C'"
$i \leftarrow 1$
while $|S| > |C|$
do { $C' \leftarrow \{v_i\}$; $S \leftarrow S - v_i$; EXTENDCLIQUE(S, v_i) ; $i \leftarrow i + 1$
comment: C is now a maximum clique of size $\overline{\alpha}$
$\overline{\alpha} \leftarrow |C|$

This algorithm builds a clique C' and tries to extend it from a set S' of eligible vertices. When $|C'| + |S'|$ is smaller than the largest clique found so far, it backtracks. The performance will depend on the ordering of the vertices used. As with Algorithm 13.2.1, the ordering need not be chosen in advance, but can be constructed as the algorithm progresses. For example, v_i might be selected as a vertex of largest degree in the subgraph induced by the vertices not yet considered. If a maximum clique is found early in the algorithm, the remaining calls to EXTENDCLIQUE() will finish more quickly. We can estimate the complexity by noticing that the loop in

MAXCLIQUE() runs at most n times. After each choice of v_i, the set S will contain at most $\Delta(G)$ vertices. There are $2^{\Delta(G)}$ subsets of a set of size $\Delta(G)$. The algorithm might construct each subset at most once. Therefore the complexity is at most $O(n \cdot 2^{\Delta(G)})$, an exponential value.

13.3 Mycielski's construction

A graph G which contains a 4-clique necessarily has $\chi(G) \geq 4$. However, it is possible for a graph with no 4-clique to have $\chi(G) \geq 4$. Mycielski found a way to construct triangle-free graphs with arbitrarily large chromatic number.

We start with a triangle-free graph G with $\chi(G) \geq 3$. Any odd cycle with five or more vertices will do (e.g., $G = C_5$). We now extend G to a graph G' as follows. For each $v \in V(G)$, we add a vertex v' to G' adjacent to the same vertices of G that v is adjacent to. We now add one more vertex v_0 adjacent to each v'. Thus, if G has n vertices, G' will have $2n + 1$ vertices. Refer to Figure 13.3.

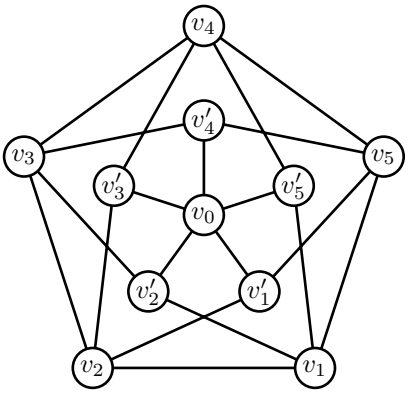


FIGURE 13.3
Mycielski's construction

Lemma 13.4. $\chi(G') = \chi(G) + 1$. *Furthermore, G' has no triangles if G has none.*

Proof. Consider a coloring χ_1 of G'. Since it induces a coloring of G, we conclude that $\chi(G') \geq \chi(G)$. Let $m = \chi(G)$. Some vertex of G with color number m is adjacent to $m - 1$ other colors in G; for otherwise each vertex of color m could be recolored with a smaller color number. It follows that the vertices v' must be colored with at least $\chi(G)$ colors. In fact, we could assign each $\chi_1(v') = \chi_1(v)$. The vertex v_0 is adjacent to $\chi(G)$ colors, so that $\chi_1(v_0) = m + 1$. It follows that $\chi(G') = \chi(G) + 1$. It is easy to see that G' has no triangles, because G has none. □

By iterating Mycielski's construction, we can construct triangle-free graphs with large chromatic numbers. In fact, if we begin the construction with $G = K_2$, the result is C_5. Or we could start Mycielski's construction with a graph which contains triangles, but is K_4-free, and construct a sequence of K_4-free graphs with increasing chromatic numbers, and so forth.

13.4 Critical graphs

Let G be a graph with $\chi(G) = m$. If we remove an edge uv from G, there are two possibilities, either $\chi(G - uv) = m$ or $\chi(G - uv) = m - 1$. In the latter case, we say that edge uv is *critical*.

DEFINITION 13.3: A graph G is *critical* if $\chi(G - uv) = \chi(G) - 1$ for all edges $uv \in E(G)$. If $\chi(G) = m$, we say that G is *m-critical*.

It is easy to see that every graph contains a critical subgraph. If $\chi(G - uv) = \chi(G)$ for some edge uv, we can remove uv. Continue deleting edges like this until every edge is critical. The result is a critical subgaph. Critical graphs have some special properties.

Lemma 13.5. *If G is m-critical, then $\delta(G) \geq m - 1$.*

Proof. If $\text{DEG}(u) < m-1$, choose an edge uv, and color $G-uv$ with $m-1$ colors. Since $\text{DEG}(u) < m - 1$, there are at most $m - 2$ adjacent colors to u in G. So there is always a color in $\{1, 2, \ldots, m-1\}$ with which u can be colored to obtain an $(m - 1)$-coloring of G, a contradiction. □

If G is an m-critical graph, then G has at least m vertices, and each has degree at least $m - 1$. Therefore every graph with $\chi(G) = m$ has at least m vertices of degree $\geq m - 1$.

Lemma 13.6. *Every critical graph with at least three vertices is 2-connected.*

Proof. Suppose that G is an m-critical graph with a cut-vertex v. Let H be a connected component of $G - v$, and let H_v be the subgraph induced by $V(H) \cup \{v\}$. Color H_v with $\leq m-1$ colors. Do the same for every such subgraph H_v. Ensure that v has the same color in each subgraph, by permuting colors if necessary. The result is a coloring of G in $\leq m - 1$ colors, a contradiction. □

The ideas of the Lemma 13.6 can be extended to separating sets in general.

Lemma 13.7. *Let S be a separating set in an m-critical graph G. Then S does not induce a clique.*

Proof. If S is a separating set, let H be a component of $G - S$, and consider the subgraph H_S induced by $V(H) \cup S$. It can be colored in $m-1$ colors. The vertices of S are colored with $|S|$ distinct colors, which can be permuted in any desired way. Do the same for every component of $G - S$. The result is a coloring of G in $m-1$ colors. □

It follows from this lemma that if $\{u, v\}$ is a separating set in an m-critical graph, that $uv \notin E(G)$. Suppose that $\{u, v\}$ is a separating set. Let H be a component of $G - \{u, v\}$, and let K be the remaining components. Construct H_{uv} induced by $V(H) \cup \{u, v\}$, and K_{uv} induced by $V(K) \cup \{u, v\}$. H_{uv} and K_{uv} can both be colored in $m-1$ colors. If H_{uv} and K_{uv} both have $(m-1)$-colorings in which u and v have different colors, then we can use these colorings to construct an $(m-1)$-coloring of G. Similarly, if H_{uv} and K_{uv} both have $(m-1)$-colorings in which u and v have the same color, we can again construct an $(m-1)$-coloring of G. We conclude that in one of them, say H_{uv}, u and v have the same color in every $(m-1)$-coloring; and that in K_{uv}, u and v have different colors in every $(m-1)$-coloring.

Now consider the graph $H' = H_{uv} + uv$. It cannot be colored in $m-1$ colors, however $H' - uv$ can be. Let xy be any other edge of H'. Then $G - xy$ can be colored in $m-1$ colors. Now K_{uv} is a subgraph of $G - xy$. Therefore u and v have different colors in this coloring. It follows that $H' - xy$ can be colored in $m-1$ colors. Hence, $H' = H_{uv} + uv$ is an m-critical graph.

Now consider the graph $K' = (K_{uv} + uv) \cdot uv$. It cannot be colored in $m-1$ colors, as this would determine an $(m-1)$-coloring of K_{uv} in which u and v have the same color. Let xy be any edge of K'. It corresponds to an edge $x'y'$ of G. $G - x'y'$ can be colored in $(m-1)$ colors. Since H_{uv} is a subgraph of G, it follows that u and v have the same color. This then determines a coloring of $K' - xy$ in $(m-1)$ colors. Hence $K' = (K_{uv} + uv) \cdot uv$ is also an m-critical graph.

Exercises

13.4.1 Program Algorithm 13.2.1, and compare its performance with a breadth-first or depth-first sequential algorithm.

13.4.2 Program the MAXCLIQUE() algorithm.

13.4.3 Let G be an m-critical graph, and let $v \in V(G)$. Show that G has an m-coloring in which v is the only vertex of color number m.

13.4.4 Let G be an m-critical graph. Apply Mycielski's construction to obtain a graph G'. Either prove that G' is $(m+1)$-critical, or find a counterexample.

13.5 Chromatic polynomials

Suppose that we wish to properly color the complete graph K_n in at most λ colors, where $\lambda \geq n$. Choose any ordering of $V(K_n)$. The first vertex can be colored in λ choices. The next vertex in $\lambda - 1$ choices, and so on. Thus, the number of ways to color K_n is $\lambda(\lambda-1)(\lambda-2)\dots(\lambda-n+1)$. This is a polynomial of degree n in λ.

DEFINITION 13.4: The *chromatic polynomial* of a graph G is $\pi(G,\lambda)$, the number of ways to color G in $\leq \lambda$ colors.

In order to show that $\pi(G,\lambda)$ is in fact a polynomial in λ, we use a method that is familiar from counting spanning trees. We first find $\pi(T,\lambda)$ for any tree T, and then give a recurrence for any graph G.

Lemma 13.8. *Let T be a tree on n vertices. Then $\pi(T,\lambda) = \lambda(\lambda-1)^{n-1}$.*

Proof. By induction on n. It is certainly true if $n = 1$ or $n = 2$. Choose a leaf v of T, and let $T' = T - v$. T' is a tree on $n-1$ vertices. Therefore $\pi(T',\lambda) = \lambda(\lambda-1)^{n-2}$. In any coloring of T', the vertex adjacent to v in T has some color. There are $\lambda - 1$ colors available for v. Every coloring of T arises in this way. Therefore $\pi(T,\lambda) = \lambda(\lambda-1)^{n-1}$. □

Suppose now that G is any graph. Let $uv \in E(G)$. In practice we will want to choose uv so that it is an edge on a cycle.

Theorem 13.9. $\pi(G,\lambda) = \pi(G-uv,\lambda) - \pi(G\cdot uv,\lambda)$.

Proof. In each coloring of $G-uv$ in $\leq \lambda$ colors, either u and v have different colors, or they have the same color. The number of colorings of $G-uv$ is the sum of these two. If u and v have different colors, then we have a coloring of G in $\leq \lambda$ colors. Conversely, every coloring of G in $\leq \lambda$ colors gives a coloring of $G-uv$ in which u and v have different colors. If u and v have the same color in $G-uv$, then this gives a coloring of $G\cdot uv$. Any coloring of $G\cdot uv$ in $\leq \lambda$ colors determines a coloring of $G-uv$ in which u and v have the same color. We conclude that $\pi(G-uv,\lambda) = \pi(G,\lambda) + \pi(G\cdot uv,\lambda)$. □

One consequence of this theorem is that $\pi(G,\lambda)$ is in fact a polynomial of degree n in λ. Now if $n > 0$, then $\lambda \mid \pi(G,\lambda)$, since G cannot be colored in $\lambda = 0$ colors. Similarly, if $\varepsilon(G) \neq 0$, we conclude that $\lambda(\lambda-1) \mid \pi(G,\lambda)$, since G cannot be colored in $\lambda = 1$ color. If G is not bipartite, then it cannot be colored in $\lambda = 2$ colors. In this case $\lambda(\lambda-1)(\lambda-2) \mid \pi(G,\lambda)$. In general:

Lemma 13.10. *If $\chi(G) = m$, then $\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1) \mid \pi(G,\lambda)$.*

Proof. G cannot be colored in fewer than m colors. Therefore $\pi(G,\lambda) = 0$, for $\lambda = 1, 2, \dots, m-1$. □

Notice that if G contains an m-clique S, then $\chi(G) \geq m$, so that $\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1) \mid \pi(G,\lambda)$. There are $\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1)$ ways to color S in $\leq \lambda$ colors. The number of ways to complete a coloring of G, given a coloring of S, is therefore $\pi(G,\lambda)/\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1)$.

Suppose that G has a cut-vertex v. Let H be a connected component of $G-v$, and let H_v be the subgraph induced by $V(H)\cup\{v\}$. Let K_v be the subgraph $G-V(H)$. Every coloring of G induces colorings of H_v and K_v, such that v has the same color in both. Every coloring of H_v and K_v occurs in this way. Given any coloring of H_v, there are $\pi(K_v,\lambda)/\lambda$ ways to complete the coloring of K_v. It follows that $\pi(G,\lambda) = \pi(H_v,\lambda)\pi(K_v,\lambda)/\lambda$.

More generally, suppose that S is a separating set of G which induces an m-clique. Let H be a component of $G-S$, and let H_S be the subgraph induced by $V(H)\cup S$. Let K_S be the subgraph $G-V(H)$. We have:

Lemma 13.11. *Let S be a separating set which induces an m-clique in G. Let H_S and K_S be defined as above. Then $\pi(G,\lambda) = \pi(H_S,\lambda)\pi(K_S,\lambda)/\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1)$.*

Proof. Every coloring of G induces a coloring of H_S and K_S. There are $\pi(H_S,\lambda)$ ways to color H_S. There are $\pi(K_S,\lambda)/\lambda(\lambda-1)(\lambda-2)\dots(\lambda-m+1)$ ways to complete a coloring of S to a coloring of K_S. This gives all colorings of G. □

There are no efficient means known of computing chromatic polynomials. This is due to the fact that most coloring problems are NP-complete. If $\pi(G,\lambda)$ could be efficiently computed, we would only need to evaluate it for $\lambda = 3$ to determine if G can be 3-colored.

Exercises

13.5.1 Find $\pi(C_{2n},\lambda)$ and $\pi(C_{2n+1},\lambda)$.

13.5.2 Find $\pi(G,\lambda)$, where G is the graph of the cube and the graph of the octahedron.

13.5.3 Let G' be constructed from G by adding a new vertex joined to every vertex of G. Determine $\pi(G',\lambda)$ in terms of $\pi(G,\lambda)$.

13.5.4 The *wheel* W_n is obtained from the cycle C_n by adding a new vertex joined to every vertex of C_n. Find $\pi(W_n,\lambda)$.

13.5.5 A *unicyclic* graph G is a graph formed from a tree T by adding a single edge connecting two vertices of T. G has exactly one cycle. Let G be a unicyclic graph on n vertices, such that the unique cycle of G has length m. Find $\pi(G,\lambda)$.

13.5.6 Let G' be constructed from G by adding two new adjacent vertices joined to every vertex of G. Determine $\pi(G',\lambda)$ in terms of $\pi(G,\lambda)$.

13.5.7 Let G' be constructed from G by adding k new mutually adjacent vertices joined to every vertex of G. Determine $\pi(G',\lambda)$ in terms of $\pi(G,\lambda)$.

13.5.8 Let G' be constructed from G by adding two new non-adjacent vertices joined to every vertex of G. Determine $\pi(G', \lambda)$ in terms of $\pi(G, \lambda)$.

13.5.9 Find $\pi(K_{m,m}, \lambda)$.

13.5.10 Let G be a graph, and suppose that $uv \notin E(G)$. Show that $\pi(G, \lambda) = \pi(G + uv, \lambda) + \pi((G + uv) \cdot uv, \lambda)$. When G has many edges, this is a faster way to compute $\pi(G, \lambda)$ than the method of Theorem 13.9.

13.5.11 Calculate $\pi(K_n - uv, \lambda)$, $\pi(K_n - uv - vw, \lambda)$, and $\pi(K_n - uv - wx, \lambda)$, where u, v, w, x are distinct vertices of K_n.

13.5.12 If G is a connected graph on n vertices, show that $\pi(K_n, \lambda) \leq \pi(G, \lambda) \leq \lambda(\lambda - 1)^{n-1}$, for all $\lambda \geq 0$.

13.5.13 Prove that the coefficient of λ^n in $\pi(G, \lambda)$ is 1, and that the coefficients alternate in sign.

13.6 Edge colorings

A coloring of the edges of a graph G is an assignment of colors to the edges. More formally, an *m-edge-coloring* is a mapping from $E(G)$ onto a set of m colors $\{1, 2, \ldots, m\}$. The coloring is *proper* if adjacent edges always have different colors. The *edge-chromatic number* or *chromatic index* of G is $\chi'(G)$, the minimum value of m such that G has a proper m-edge coloring. Notice that in any proper edge-coloring of G, the edges of each color define a matching in G. Thus, $\chi'(G)$ can be viewed as the minimum number of matchings into which $E(G)$ can be partitioned.

When the edges of a multi-graph are colored, all edges with the same endpoints must have different colors.

Consider a vertex v of maximum degree in G. There must be $\text{DEG}(v)$ colors incident on v. Therefore $\chi'(G) \geq \Delta(G)$. There is a remarkable theorem by Vizing (Theorem 13.14) that states $\chi'(G) \leq \Delta(G) + 1$ for simple graphs.

Before we come to the proof of Vizing's theorem, first consider the case when G is bipartite. A matching M_i saturating all vertices of degree $\Delta(G)$ can be found with Algorithm 9.3.1 (the Hungarian algorithm). Alternating paths can be used to ensure that each vertex of degree $\Delta(G)$ is saturated. Thus we know that the bipartite graph G has a maximum matching saturating every vertex of degree $\Delta(G)$. This gives an algorithm for edge-coloring a bipartite graph.

Algorithm 13.6.1: BIPARTITECOLORING(G)

comment: Edge-color a graph G on n vertices

find the degrees of all vertices
$i \leftarrow 1$
repeat
  find a maximum matching M_i in G
    saturating every vertex of degree $\Delta(G)$
  assign color i to the edges of M_i
  $G \leftarrow G - M_i$
  $i \leftarrow i + 1$
until $\Delta(G) = 0$

It follows from this algorithm, that when G is bipartite, $\chi'(G) = \Delta(G)$.

Suppose that G is a k-regular graph, edge-colored in k colors. Then every color occurs at every vertex of G. If i and j are any two colors, then the (i, j)-subgraph is the subgraph of G that contains only the edges of color i and j. Because the (i, j)-subgraph is the union of two matchings, it is the disjoint union of alternating cycles. Let $U \subseteq V(G)$. Let n_i and n_j denote the number of edges of colors i and j, respectively, in the edge-cut $[U, V - U]$. Each cycle of the (i, j)-subgraph intersects $[U, V - U]$ in an even number of edges. Thus we conclude that $n_i + n_j$ is even. Therefore $n_i \equiv n_j \pmod 2$. This gives the following parity lemma:

Lemma 13.12. (Parity lemma) *Let G be a k-regular graph, edge-colored in colors $\{1, 2, \ldots, k\}$. Let $U \subseteq V(G)$. Let n_i denote the number of edges of color i in $[U, V - U]$. Then $n_1 \equiv n_2 \equiv \ldots \equiv n_k$ (mod 2).*

Vizing's theorem (Theorem 13.14) is based on an algorithm to edge-color a graph in $\leq \Delta(G) + 1$ colors. We present two proofs of Vizing's theorem, the first is based on that of FOURNIER [52]. The second proof follows from an algorithm to color the edges. Fournier's proof begins with an arbitrary coloring of G in $\Delta(G)+1$ colors, and then gradually improves it until it becomes a proper coloring. Given a coloring, let $c(v)$ denote the number of colors occurring at vertex $v \in V(G)$. If $c(v) = \text{DEG}(v)$, for all v, then the coloring is proper. Otherwise $c(v) < \text{DEG}(v)$, for some v. The sum $\Sigma_v c(v)$ is an indication of how close an arbitrary coloring is to being a proper coloring.

Suppose first that G is arbitrarily colored in two colors.

Lemma 13.13. *If G is a graph that is not an odd cycle, then G has a 2-edge-coloring in which $c(v) \geq 2$, for all vertices v, with $\text{DEG}(v) \geq 2$.*

Proof. If G is Eulerian, choose an Euler tour, and color the edges alternately blue and red along the tour. If G is not Eulerian, add a new vertex v_0 adjacent to every odd degree vertex of G. The result is an Eulerian graph. Color it in the same way. □

We can use Lemma 13.13 on subgraphs of G. Given a proper coloring of G, the edges of colors i and j each define a matching in G. Consider the (i, j)-*subgraph*. Each connected component is a path or an even cycle whose colors alternate. If, however, we begin with an arbitrary coloring of G, then we want to maximize $\Sigma_v c(v)$. If some component of the (i, j)-subgraph is not an odd cycle then by Lemma 13.13, it can be 2-colored so that $c(v) \geq 2$, for all vertices v, with $\text{DEG}(v) \geq 2$.

Theorem 13.14. (Vizing's theorem) *If G is simple, then $\chi'(G) \leq \Delta(G) + 1$.*

Proof. We begin by arbitrarily coloring the edges in $\Delta(G) + 1$ colors. We show that when $\Sigma_v c(v)$ is as large as possible, the coloring must be proper. Suppose that the coloring is not proper, and choose a vertex u with $c(u) < \text{DEG}(u)$. Some color i_0 is missing at u, and some color i_1 occurs at least twice. Let edges uv_0 and uv_1 have color i_1. If color i_0 is missing at either v_0 or v_1, we can recolor one of these edges with color i_0, thereby increasing $\Sigma_v c(v)$. Hence, we can assume that color i_0 occurs at both v_0 and v_1. Some color is missing at v_1; call it i_2. If i_2 is also missing at u, we can recolor uv_1 with color i_2, thereby increasing $\Sigma_v c(v)$. Hence, let uv_2 be an edge of color i_2. Some color is missing at v_2; call it i_3. If i_3 is also missing at u, we can recolor uv_1 with color i_2, and uv_2 with color i_3, thereby increasing $\Sigma_v c(v)$. It follows that $i_3 \neq i_0$, so that i_0 occurs at v_2. We continue in this way, constructing a sequence of edges $uv_1, uv_2, \ldots, uv_k$ of distinct colors $i_1, i_2, \ldots, i_k$, such that color i_0 occurs at each of $v_1, \ldots, v_k$, and color i_{j+1} does not occur at v_j. Refer to Figure 13.4. We continue in this fashion generating a sequence $i_0, i_1, \ldots$, of distinct colors until

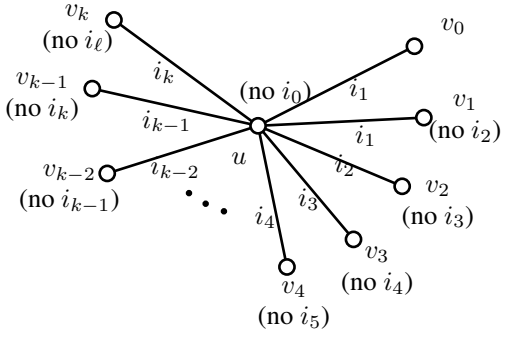


FIGURE 13.4
Vizing's theorem

we find a color i_{k+1} that is not missing at v. Thus i_{k+1} has previously occurred in the sequence. Suppose that $i_{k+1} = i_\ell$, where $1 \leq \ell \leq k$. Recolor the edges $uv_1, uv_2, \ldots, uv_{\ell-1}$ with the colors $i_2, i_3, \ldots, i_\ell$, respectively. This does not change $\Sigma_v c(v)$, because each $c(v_j)$ is unchanged. Notice that $uv_{\ell-1}$ and uv_ℓ are now both colored i_ℓ. Consider the (i_0, i_ℓ)-subgraph containing u. It contains $v_{\ell-1}$ and v_ℓ. If it is not an odd cycle, it can be recolored so that colors i_0 and i_ℓ both occur at each of the vertices that have degree exceeding one. This would increase $c(u)$ and therefore

$\Sigma_v c(v)$. Hence this subgraph can only be an odd cycle, so that G contains an (i_0, i_ℓ)-path from $v_{\ell-1}$ to v_ℓ.

Now recolor the edges $uv_\ell, uv_{\ell+1}, \ldots, uv_k$ with the colors $i_{\ell+1}, \ldots, i_k, i_{k+1} = i_\ell$, respectively. Once again $\Sigma_v c(v)$ is unchanged. The (i_0, i_ℓ)-subgraph containing u now contains $v_{\ell-1}$ and v_k, and must be an odd cycle. Hence G contains an (i_0, i_ℓ)-path from $v_{\ell-1}$ to v_k. This contradicts the previous path found. It follows that the coloring can only be proper, and that $\chi' \leq \Delta(G) + 1$. □

This proof of Vizing's theorem makes use of a *color rotation* at vertex u, namely, given a sequence of incident edges $uv_1, uv_2, \ldots, uv_k$ of colors $i_1, i_2, \ldots, i_k$, respectively, such that v_j is missing color i_{j+1}, for $j = 1, 2, \ldots, k-1$, and v_k is missing color i_ℓ where $\ell \in \{1, 2, \ldots, k-1\}$. We then recolor $uv_1, uv_2, \ldots, uv_{\ell-1}$ with colors $i_2, i_3, \ldots, i_\ell$, respectively. This idea will be used in the coloring algorithm.

The second proof of Vizing's theorem is based on an edge-coloring algorithm. Given a graph G, let the vertices be numbered $1, 2, \ldots, n$. Initially all edges are assigned color 0, indicating that they are uncolored. The algorithm takes the vertices in the order $u = 1, 2, \ldots, n$, and attempts to color the edges incident on vertex u. When vertex 1 is selected, it is sufficient to take the first available color for each edge, as no other edges have been colored yet. When vertex $u > 1$ is selected, it may be adjacent to some vertices $v < u$ (whose edges have already been colored), and to some vertices $w > u$. For the edges uw where $w > u$, the algorithm will take the first available color. This may create a situation where some $w > u$ has two or more incident edges of the same color. This situation will be resolved during the iteration when $u = w$.

At the beginning of iteration u, the colors of the incident edges are counted. The color of edge uv is denoted *Clr*$[uv]$. An array *Color*$[k]$, which is initially 0, will store the number of occurrences of color k at vertex u:

$$\textbf{for each } v \longrightarrow u \quad \textbf{do } \mathit{Color}[\mathit{Clr}[uv]] \leftarrow \mathit{Color}[\mathit{Clr}[uv]] + 1$$

If every color k has *Color*$[k] \leq 1$, then every color occurs at most once at u. The coloring at u is then completed by assigning the first available color to each uncolored edge. An uncolored edge uv necessarily has $v > u$. The first available color is found with a simple loop.

$clr \leftarrow 1$
while *Color*$[k] > 0$ **do** $clr \leftarrow clr + 1$

If there are no duplicate colors at u, the number of steps required to color the edges incident on u with these loops is $O(\text{DEG}(u))$.

Otherwise there are two or more edges incident on u with the same color k. This is detected by *Color*$[k] > 1$. There are $\Delta + 1$ colors available, but the number of colors used is $< \text{DEG}(u) < \Delta$. Therefore there are at least two colors missing at vertex u. Let j and i be colors missing at u. They have *Color*$[j] = 0$ and *Color*$[i] = 0$. The color numbers j, i, and k are found, and the algorithm does a breadth-first search, called *ColorBFS*, to resolve the repeated color.

Algorithm 13.6.2: EDGECOLOR(G, n)

comment: Initially the edges all have color 0.
Construct an edge-coloring in
at most $\Delta(G) + 1$ colors.

for $u \leftarrow 1$ **to** n
do
- count the color frequencies at vertex u
- $k \leftarrow$ first repeated color at u
- **while** $k > 0$
  - **do**
    - $j \leftarrow$ first color missing at u
    - **if** COLORBFS(u, k, j)
      then go to $L1$ "success"
    - $i \leftarrow$ second color missing at u
    - **if** COLORBFS(u, k, i)
      then go to $L1$ "success"
    - **comment:** otherwise a color rotation is needed
    - let uv be an edge of color k
    - $\ell \leftarrow$ first color missing at v
    - $clr[uv] \leftarrow \ell$
    - $L1$:
    - $k \leftarrow$ first repeated color at u
- **comment:** there are now no repeated colors at vertex u
- **for each** $v \longrightarrow u$
  do if $clr[uv] = 0$
  then color uv with the first available color

The procedure *ColorBFS*(u, k, j) builds alternating paths from vertex u, of colors k and j, in a breadth-first manner. It initially places u and all vertices incident on an edge of color k on an array, called the *ScanQ*. There are at least two incident edges of color k. The paths are then alternately extended by edges of color j and k. If at any point, the *ColorBFS* encounters a vertex $w > u$, it recolors the path from w to u, by interchanging colors j and k. As color j was initially not used at u, and color k appeared twice, this improves the coloring at u. *ColorBFS* then returns the value *true*. This will usually be the outcome near the beginning of the algorithm.

Algorithm 13.6.3: COLORBFS(u, k, j)

comment: color k is repeated at vertex u, color j is missing at u

comment: initialize the *ScanQ* with u and all adjacent vertices of color k

$QSize \leftarrow 1$
$ScanQ[1] \leftarrow u$
for each $v \longrightarrow u$ **do if** $clr[uv] = k$
  then { **comment:** add v to *ScanQ*
  $QSize \leftarrow QSize + 1$
  $ScanQ[QSize] \leftarrow v$
$m \leftarrow 1$
while $m \leq QSize$
  do {
    $v \leftarrow ScanQ[m]$
    if v was added on color k
      then {
        if (color j is missing at v) **or** $(v > u)$
          then { re-color the path from v to u
          return ***true***
        comment: otherwise add v to *ScanQ*
        $QSize \leftarrow QSize + 1$
        $ScanQ[QSize] \leftarrow v$
      else {
        comment: v was added on color j
        if (color k is missing at v) **or** $(v > u)$
          then { re-color the path from v to u
          return ***true***
        comment: otherwise add v to *ScanQ*
        $QSize \leftarrow QSize + 1$
        $ScanQ[QSize] \leftarrow v$
    $m \leftarrow m + 1$

Otherwise *ColorBFS* encounters only vertices $v < u$. This will usually be the case near the end of the algorithm. When vertex $v < u$ is discovered by *ColorBFS*, it is either by following an edge of color j or of color k. Now v has *at most one edge* of each color. If v has an edge of color j, but no edge of color k, then the path from v to u is recolored, by interchanging colors. This improves the coloring at u, and does not affect the coloring at the other vertices on the alternating path. Similarly if v has an edge of color k, but no edge of color j, the path is recolored. The *only* time it does not succeed in recoloring a path to vertex u, is if the edges of colors k and j induce one or more odd cycles, in which each odd cycle has two edges of color k incident on vertex u. *ColorBFS* then returns the value *false*.

If the call to *ColorBFS*(u, k, j) does not succeed in Algorithm *EdgeColor*, the algorithm now finds the other missing color i, and tries again, using *ColorBFS*(u, k, i). The reason for using both colors j and i is that they are easy to find, and if one does not work, the other frequently will. If this does not succeed, it is time for a color rotation. A vertex v is chosen, such that *clr*$[uv] = k$. A missing color ℓ at vertex v is found, and the edge uv is recolored with color ℓ. Vertex u now has two incident edges of color ℓ, and no incident edges of colors j or i. The loop continues. *ColorBFS*(u, ℓ, j) and *ColorBFS*(u, ℓ, i) are used to search for alternating paths. If this does not succeed, a vertex w is chosen such that *clr*$[uw] = \ell$, and the edge is re-colored as before. The vertex w is chosen so that it is not the same as the vertex v chosen on the previous iteration. In this way a sequence of edges at u is followed which must eventually result in a successful re-coloring. For when a missing color is repeated, eg., color k, then the odd cycle of colors k and j will be a different odd cycle than previously found, which is impossible. It follows that the algorithm works, so that it will always succeed in edge-coloring a graph using at most $\Delta + 1$ colors. Vizing's theorem follows.

It is difficult to estimate the complexity of this algorithm accurately. It is reasonable to expect that iteration u will take approximately $O(\text{DEG}(u))$ steps. For even when it is necessary to call *ColorBFS*, the search visits very few vertices. Thus, the algorithm is likely to be of order $O(\varepsilon)$ for most graphs.

Graphs G for which $\chi'(G) = \Delta(G)$ are said to be *Class I graphs*. If $\chi'(G) = \Delta(G)+1$, then G is a *Class II graph*. See Wallis [186] for a discussion of Class I and Class II graphs. Although there is an efficient algorithm to color a graph in at most $\Delta(G) + 1$ colors, it is an NP-complete problem to determine whether an arbitrary graph is of Class I or II. A proof of this remarkable result is presented at the end of the chapter.

If G is a multigraph with no loops, then the general form of Vizing's theorem states that $\chi'(G) \leq \Delta(G) + \mu(G)$, where $\mu(G)$ is the maximum edge-multiplicity. It can often happen that $\mu(G)$ is not known. *Shannon's theorem* gives an alternative bound $\chi'(G) \leq \lfloor 3\Delta(G)/2 \rfloor$. The proof presented is due to ORE [133]. It requires two lemmas. Given a proper edge-coloring of G, we write $C(u)$ for the set of colors present at u.

Lemma 13.15. (Uncolored edge lemma) *Let G be a multigraph without loops. Let uv be any edge of G, and let $G - uv$ be edge-colored with k colors, and suppose that $\chi'(G) = k + 1$. Then:*

$$|C(u) \cup C(v)| = k$$

$$|C(u) \cap C(v)| = \text{DEG}(u) + \text{DEG}(v) - k + 2$$

$$|C(u) - C(v)| = k - \text{DEG}(v) + 1$$

$$|C(v) - C(u)| = k - \text{DEG}(u) + 1$$

Proof. Every color missing at u is present at v, or there would be a color available

for uv, thereby making $\chi'(G) = k$. Therefore all colors are present at one of u or v. This gives the first equation. The colors present at u can be counted as

$$|C(u) - C(v)| + |C(u) \cap C(v)| = \text{DEG}(u) - 1.$$

Similarly, those present at v are given by

$$|C(v) - C(u)| + |C(u) \cap C(v)| = \text{DEG}(v) - 1.$$

Now

$$|C(u) \cup C(v)| = |C(u) - C(v)| + |C(v) - C(u)| + |C(u) \cap C(v)|,$$

so that we can solve these equations for $|C(u) \cap C(v)|$, giving the second equation. The third and fourth equations then result from combining this with the previous two equations. □

Lemma 13.16. (Ore's lemma) *Let G be a multigraph without loops. Then:*

$$\chi'(G) \leq \text{MAX}\{\Delta(G), \frac{1}{2}\text{MAX}_{\{u,v,w\}}\{\text{DEG}(u) + \text{DEG}(v) + \text{DEG}(w)\}\},$$

where the second maximum is over all triples of vertices u, v, w such that $v \longrightarrow u \longrightarrow w$.

Proof. The proof is by induction on $\varepsilon(G)$. It is clearly true if $\varepsilon(G) \leq 3$. Suppose it holds for all graphs with $\varepsilon(G) \leq m$ and consider G with $\varepsilon(G) = m+1$. Let uv be any edge. Delete uv to obtain $G - uv$, for which the result holds. Let $\chi'(G - uv) = k$, and consider a proper k-edge coloring of $G - uv$. If there is a color available for uv, then $\chi'(G) = k$, and the result holds. Otherwise the uncolored edge lemma (Lemma 13.15) applies to G.

Pick a color $i \in C(u) - C(v)$ and an edge uw of color i. We first show that $C(v) - C(u) \subseteq C(w)$. Let $j \in C(v) - C(u)$. If color j is missing at w, then since j is also missing at u, we can recolor edge uw with color j, and assign color i to uv. This results in a k-edge-coloring of G, a contradiction. Therefore $j \in C(w)$, so that $C(v) - C(u) \subseteq C(w)$.

We also show that $C(u) - C(v) \subseteq C(w)$. We know that $i \in C(u) - C(v)$. If there is no other color in $C(u) - C(v)$, we are done. Otherwise, pick also $\ell \in C(u) - C(v)$. If $\ell \notin C(w)$, consider the (ℓ, j)-subgraph H containing u. If H does not contain v, we can interchange colors in H, and assign color ℓ to uv, a contradiction. Therefore H consists of an alternating path from u to v. Interchange colors in H, recolor edge uw with color ℓ, and assign color i to edge uv, again a contradiction. We conclude that $C(u) - C(v) \subseteq C(w)$.

We have $|C(u) - C(v)| + |C(u) - C(v)| \leq |C(w)|$. By the uncolored edge lemma, this means that $\text{DEG}(w) \geq 2k - \text{DEG}(u) - \text{DEG}(v) + 2$. Therefore $\text{DEG}(u) + \text{DEG}(v) + \text{DEG}(w) \geq 2k + 2$, so that $k + 1 \leq \frac{1}{2}(\text{DEG}(u) + \text{DEG}(v) + \text{DEG}(w))$. The result then holds for G, as required. □

Theorem 13.17. (Shannon's theorem) *Let G be a multigraph without loops. Then* $\chi'(G) \leq \lfloor 3\Delta(G)/2 \rfloor$.

Proof. By Ore's lemma:

$$\chi'(G) \leq \text{MAX}\{\Delta(G), \frac{1}{2}\{\Delta(G) + \Delta(G) + \Delta(G)\}\} = 3\Delta(G)/2.$$

□

Exercises

13.6.1 Describe an algorithm using alternating paths in a bipartite graph which finds a maximum matching saturating all vertices of degree $\Delta(G)$.

13.6.2 Work out the complexity of the bipartite coloring algorithm.

13.6.3 Program the bipartite edge-coloring algorithm.

13.6.4 Program the *EdgeColor* algorithm.

13.6.5 Determine whether the line graph of K_5 is of Class I or II.

13.6.6 Show that an edge coloring of G gives a vertex coloring of $L(G)$.

13.6.7 Determine χ' for the Petersen graph.

13.6.8 Show that when the inverter shown in Figure 13.12 is edge-colored in three colors, one of the two pairs of edges $\{a, b\}, \{c, d\}$ has the same color, and the other pair has different colors.

13.7 Graph homomorphisms

Colorings of the vertices of a graph are closely related to graph homomophisms. Consider the 3-coloring of the graph G in Figure 13.5. The vertices of G can be mapped to the vertices of a triangle, such that the vertices of each color are mapped to separate vertices of K_3. This determines a coloring of K_3. Here vertices $1, 4, 7$ are all mapped to u in the K_3; vertices $2, 5, 8$ are mapped to w; and vertices $3, 6$ are mapped to v. The colors assigned to K_3 are determined by the colors of G. Notice that every edge of G is mapped to an edge of K_3. This defines a graph *homomorphism*.

DEFINITION 13.5: A *graph homomorphism* from graph G to H is a mapping $\phi : V(G) \to V(H)$ such that if $uv \in E(G)$, then $\phi(u)\phi(v) \in E(H)$. We write $\phi : G \to H$.

If $\phi : G \to H$ is a homomorphism, and $u \in V(H)$, then $\phi^{-1}(u)$ is a *subset* of $V(G)$ that is an independent set in G. We immediately see that a homomorphism of G onto K_m is equivalent to an m-coloring of G:

Lemma 13.18. *G is m-colorable if and only if there is a homomorphism from G onto K_m.*

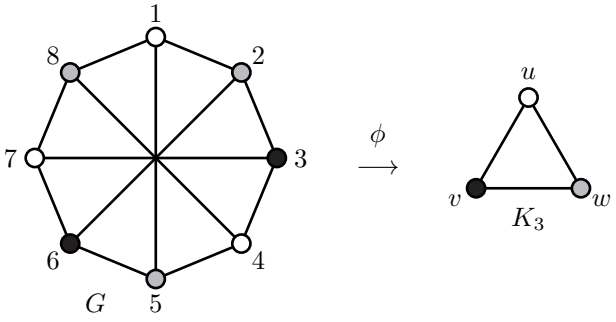


FIGURE 13.5
A graph homomorphism

Proof. Let the colors of $V(G)$ be $\{1, 2, \ldots, m\}$. Map the vertices of color i to vertex i of K_m. □

It follows that every bipartite graph has a homomorphism onto K_2. We see that graph coloring can be viewed as a special case of homomorphisms. If $\phi : G \to H$ is a homomorphism, then edges of G must map only to edges of H, but non-edges of G can map to non-edges or edges of H. An m-clique in G must map to an m-clique in H, but an independent set in G can map to various subgraphs of H. In general, it is a difficult problem to determine the homomorphisms of a graph G into H.

An even cycle is bipartite, and so has a homomorphism onto K_2. Consider an odd cycle C_m. In Figure 13.6 we see a homomorphism of C_m onto C_{m-2}. Notice how a vertex of degree two can be "folded" over and removed. This gives:

Lemma 13.19. *The cycle C_m, where $m \geq 4$, has a homomorphism onto C_{m-2}.*

When m is odd, C_m can be reduced as far as C_3. An odd cycle C_m has homomorphisms only onto other odd cycles — for a homomorphism onto an even cycle would mean that C_m can be colored in two colors.

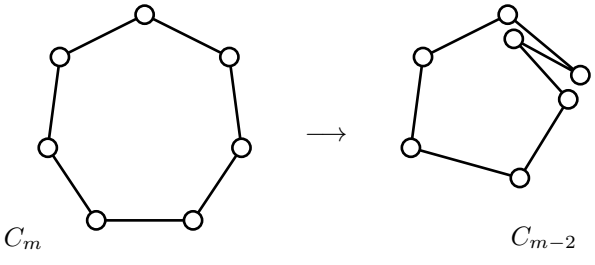


FIGURE 13.6
A homomorphism of a cycle

If $G = \overline{K}_n$, then there is a homomorphism from G into any graph H, because G has no edges. A homomorphism ϕ is said to be *faithful* if $\phi(G)$ is an induced subgraph of H. That is, if $uv \in E(H)$, then G has at least one edge between $\phi^{-1}(u)$ and $\phi^{-1}(v)$. A faithful homomorphism $\phi : G \to H$ that is one-to-one must be an isomorphism of G with H. If $G = H$, then ϕ is an automorphism of G. Now the automorphisms of G form a group, however this is not the case with homomorphisms.

DEFINITION 13.6: A homomorphism $\phi : G \to G$ is called an *endomorphism*.

The automorphism group $\text{AUT}(G)$ is a subset of the set of all endomorphisms of G. Homomorphisms can be composed. If $\phi : G \to G$ is an endomorphism, then $\phi^2, \phi^3, \ldots$, are also endomorphisms. If $\phi(G)$ is smaller than G, then a decreasing sequence of subgraphs is produced. In general, when ϕ is an endomorphism, $\phi(G)$ is called a *retract* of G. Consider the graph of the octahedron, shown in Figure 13.7, together with two retracts, deriving from ϕ which maps $\{4, 5, 6\}$ in a cycle $(4, 5, 6)$, and also maps 3 to 1 to 5, and 2 to 6. The first application of ϕ results in the first retract shown. The second application produces a triangle as a retract. Any further applications of ϕ just rotate the triangle. More generally, any graph isomorphic to a retract $\phi(G)$ is also called a retract of G.

DEFINITION 13.7: A *retract* of G is any graph isomorphic to some $\phi(G)$, where ϕ is an endomorphism of G.

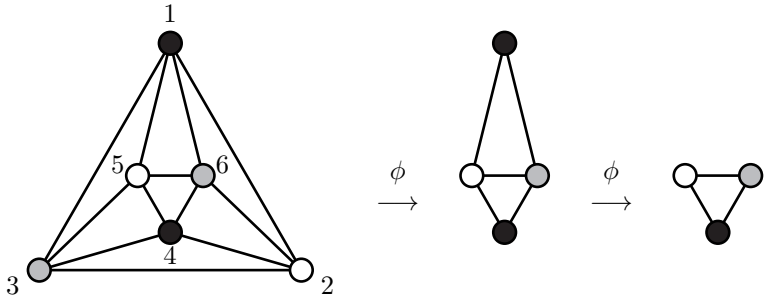


FIGURE 13.7
Retracts of the octahedron

DEFINITION 13.8: A *core* of G is a retract K of G such that no proper subgraph of K is also a retract of G.

It is clear that if K is a retract of G, then a retract of K is also a retract of G. Therefore we can also say that a core is *any* graph that has no proper retract. For example, K_n is a core, for all n. K_2 is the only bipartite graph that is a core. If K is a core then any endomorphism of K is necessarily an automorphism. Figure 13.7 shows that K_3 is a core of the octahedron.

Suppose that $W = (u_0, u_1, \ldots, u_m)$ is a walk of length m in a graph G. If there is a u_i such that $u_{i+2} = u_i$, then W can be reduced by replacing the sub-sequence

(u_i, u_{i+1}, u_{i+2}) by the single vertex u_i. This reduction can be iterated until no further reductions are possible, resulting in a *reduced walk*. A path is always a reduced walk. Some walks may reduce to a single vertex as a result.

Lemma 13.20. *Let $W = (u_0, u_1, \ldots, u_m)$ be a reduced walk of length m in a graph G, and let $\phi : G \to G$ be an endomorphism. Then $\phi(W)$ is a walk, which has a corresponding reduced walk of length $k \leq m$, such that $k \equiv m$ (mod 2).*

Proof. Each edge $u_i u_{i+1}$ of W is mapped by ϕ to an edge. Therefore $\phi(W)$ is a walk in G containing m edges. Each reduction in $\phi(W)$ that replaces $(\phi(u_i), \phi(u_{i+1}), \phi(u_{i+2}))$ by $\phi(u_i)$ when $\phi(u_i) = \phi(u_{i+2})$ removes two edges from the walk. □

Lemma 13.21. *Every odd cycle is a core.*

Proof. Let $\phi : C_m \to C_m$ be an endomorphism of an odd cycle. C_m is a reduced closed walk of length m. By Lemma 13.20, $\phi(C_m)$ contains a reduced closed walk of odd length $k \leq m$. But the shortest odd length reduced closed walk of C_m has length m. It follows that every endomorphism of C_m is an automorphism, so that C_m is a core. □

Notice that if the proof of Lemma 13.21 is applied to an even cycle, we find that $\phi(C_m)$ contains an even length reduced closed walk. It is easy to see that the length can be zero for certain endomorphisms ϕ.

Theorem 13.22. *Let G be a graph. All cores of G are isomorphic.*

Proof. Suppose that H_1 and H_2 are two cores of G. Let ϕ_1 and ϕ_2 be endomorphisms such that $\phi_1(G) = H_1$, and $\phi_2(G) = H_2$. Now ϕ_1 maps $V(H_2)$ into $V(H_1)$, so that $\phi(H_2)$ is a subgraph of H_1. Since $\phi_2(G) = H_2$, we have $\phi_1(\phi_2(G)) = \phi_1(H_2)$ is a retract of G. But a core of G has no proper subgraph that is a retract of G. Therefore $\phi_1(H_2) = H_1$. □

Theorem 13.23. *Let G be a vertex-transitive graph. All cores of G are vertex-transitive.*

Proof. Let K be a core of G, and let ϕ be an endomorphism such that $\phi(G) = K$. Then $\phi(K) = K$. Let v and w be any two vertices of K. Choose a vertex u of G such that $\phi(u) = w$. Now G is vertex transitive, so that there is $\theta \in \text{AUT}(G)$ such that $\theta(v) = u$. We have $\theta(G) = G$, so that $\phi(\theta(G)) = K$. Hence $\phi(\theta(K))$ is a subgraph of K. But K is a core, and $\theta(K) \cong K$, so that $\phi(\theta(K)) = K$. That is, the mapping "first θ, then ϕ" maps K to K, and is therefore an autmorphism of K. But $\phi(\theta(v)) = \phi(u) = w$, so that this automorphism maps v to w. It follows that K is vertex-transitive. □

Consider the graph G of Figure 13.5 consisting of a cycle C_8 with main diagonals. The shortest cycle of G has length five. If ϕ is any endomorphism of G, then each C_5 in G must map to a closed walk of length five, which must also be a cycle of G. It follows that ϕ must be one-to-one, so that G is a core. In general, it can be

difficult to determine whether a given graph is a core. This example shows that a shortest closed walk W of odd length is important — for any endomorphism ϕ must map W to a closed walk containing a reduced closed walk, also of odd length, which must therefore also be a shortest closed walk of odd length.

Consider the problem of finding an endomorphism ϕ of G, such that $\phi \notin \text{AUT}(G)$. We sketch a brief outline of how to construct such an algorithm, although a general algorithm is beyond the scope of this book. We first consider vertices of degree $n-1$, where $n = |G|$.

Lemma 13.24. *Let $U \neq \emptyset$ be the vertices of G of degree $n-1$. Then U is contained in every core of G.*

Proof. Let ϕ be an endomorphism of G, and let K be a core of G such that $\phi(G) = K$. Then $\phi(K) = K$. We can iterate ϕ sufficient times, say k times, until every vertex $u \notin V(K)$ satisfies $\phi^k(u) \in V(K)$. We have $\phi^k(G) = K$. So without loss of generality, we can replace ϕ by ϕ^k, or equivalently, assume that $k = 1$. Let $u \in U$. Suppose that $u \notin V(K)$, but that $v = \phi(u) \in V(K)$. Let $w \in V(K)$ satisfy $\phi(w) = v$, which must exist, because ϕ is one-to-one acting on $V(K)$. Then $uw \in E(G)$, but $\phi(uw) = v$, which is impossible. It follows that $U \subseteq V(K)$. □

Corollary 13.25. *Let G be a core. Construct a graph G^+ by attaching a new vertex v, adjacent to all vertices of G. Then G^+ is a core.*

Proof. By Lemma 13.24 the new vertex v is contained in every core of G^+. Let ϕ be an endomorphism of G^+. Without loss of generality, we can assume that $\phi(v) = v$. Let K be a core of G^+. If $K = G^+$, the result is true, so suppose that $V(K)$ contains a non-empty subset of $V(G)$. As in the proof of Lemma 13.24 we can assume that every $u \notin V(K)$ has $\phi(u) \in V(K)$. Now $uv \in E(G)$, because v is adjacent to all of G. Hence $\phi(u) \neq v$. It follows that $\phi(G)$ is a retract of G, which is not possible, because G is a core. □

We have seen previously that the odd cycles C_m are cores. By adding one or more new vertices to C_m, adjacent to all vertices, a sequence of cores can be constructed.

Vertices of degree zero are irrelevant to homomorphisms. Therefore the algorithm to find an endomorphism can initially delete all vertices of degree zero. When a graph is disconnected it is necessary to search for a homomorphism between any two connected components. Therefore we assume a connected graph G to simplify the problem, although the techniques used will also be applicable to disconnected graphs as well.

If there is a vertex u with $\text{DEG}(u) = 1$, let u be adjacent to v. Then any vertex $w \longrightarrow v$, where $w \neq u$, can be chosen as $\phi(u)$. Thus vertices of degree one can be successively eliminated.

It follows from Lemma 13.24 that every vertex u of degree $|G|-1$ is in every core K, and furthermore, we can assume that each such u is fixed by an endomorphism ϕ mapping G to K. Therefore the algorithm can assign $\phi(v) = v$, for all vertices of degree $|G| - 1$.

If K is a core of G with $\phi(G) = K$, then $\phi(K) = K$. Similar to Lemma 13.24,

we can iterate ϕ sufficient times until every vertex of K is fixed, so that there is no loss in generality in assuming that ϕ fixes every vertex of a core K. And since $\phi \notin \text{AUT}(G)$, we can assume that there is some vertex $v \in V(K)$ and some $u \in V(G)$ such that $\phi(u) = \phi(v) = v$. Clearly $uv \notin E(G)$. Therefore once $\phi(v) = v$ has been assigned, the algorithm partitions $V(G)$ into vertices adjacent to v, and non-adjacent.

The algorithm will iteratively take all vertices v in turn, and then successively take all suitable non-adjacent vertices u, and assign $\phi(u) = \phi(v) = v$, and then recursively attempt to extend ϕ to all of $V(G)$. Once $\phi(u) = v$ has been assigned, the remaining vertices adjacent to u must be mapped to vertices adjacent to v. A key element in the search is the length of a shortest closed odd walk through any vertex.

If W is a closed walk of odd length in G, then $\phi(W)$ is a closed walk of odd length $\leq \ell(W)$. Therefore vertices contained in a *shortest* closed odd walk can only be mapped to vertices also contained in a shortest odd closed walk. Therefore the algorithm finds the length of the shortest closed odd walk through each vertex, and partitions the vertices according to the length. A vertex u with a shortest closed odd walk of length ℓ can only be mapped to vertices with a shortest closed odd walk of length $\leq \ell$. For many graphs, this is a sufficiently strong constraint to enable a computer search to find an endomorphism ϕ fairly quickly, for graphs of moderate size.

We have seen that the question of whether there exists a homomorphism $\phi : G \to H$ reduces to graph coloring, when H is a clique. Therefore such a homomorphism is also known as an *H-coloring*.

DEFINITION 13.9: Given graphs G and H, the problem: "*is there a homomorphism $\phi : G \to H$*" is known as the *H-coloring problem*.

In general, H-coloring problems tend to be NP-complete. In the next section it is proved that C_5-coloring is NP-complete.

Exercises

13.7.1 Show that the Petersen graph is a core.

13.7.2 The *odd girth* of G is γ_{odd}, the length of a shortest odd cycle in G, if one exists. Construct an algorithm to find the odd girth of G, and to find a cycle of length γ_{odd}.

13.7.3 Let H denote a path of length m. Describe an algorithm to determine whether there is a homomorphism from a graph G onto H.

13.7.4 Describe an algorithm to determine whether there is a homomorphism from G onto $K_{3,1}$.

13.7.5 Show that the graph of Figure 13.8 is a core. What is its chromatic number?

13.7.6 Let W be a shortest closed walk of odd length containing vertex u, in a graph G. Show that W consists of two shortest paths P_{uv} and P_{uw} from u to v, and from u to w, of the same length, plus the edge vw.

13.7.7 Construct an algorithm to find the length of a shortest closed odd walk through each vertex of G.

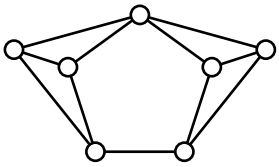


FIGURE 13.8
A possible core

13.8 NP-completeness

In this section we show that several coloring-related problems are NP-complete.

Problem 13.1:	3-Colorability
Instance:	a graph G.
Question:	is $\chi(G) \leq 3$?

We transform from 3-Sat. Consider an instance of 3-Sat containing variables $u_1, u_2, \ldots, u_n$, with complements $\overline{u}_1, \overline{u}_2, \ldots, \overline{u}_n$. Suppose that there are m clauses, denoted $c_1, c_2, \ldots, c_m$. We construct a graph G which has a 3-coloring if and only if the instance of 3-Sat has a solution. The construction is based on the graph T shown in Figure 13.9. The key property of this graph is that in any 3-coloring, the two vertices of degree two, denoted a and b, must have the same color.

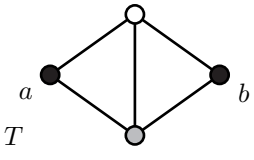


FIGURE 13.9
$\boldsymbol{T}$ consists of two triangles

Three copies of this graph are now combined to produce the graph $G(u_i, \overline{u}_i)$ of Figure 13.10, which is used to represent the variable u_i and its complement $\overline{u}_i$.

The vertices colored black must always have the same color in any 3-coloring, because they are part of a chain of graphs isomorphic to T. Associated with u_i are two vertices, shaded gray in Figure 13.10. And associated with $\overline{u}_i$ are two vertices, one white, and one shaded gray in Figure 13.10. We call these the u_i vertices, and $\overline{u}_i$ vertices, respectively. Two useful properties of this graph are:

Lemma 13.26. *In every 3-coloring of $G(u_i, \overline{u}_i)$, either the u_i vertices have the same color, and the $\overline{u}_i$ vertices have different colors, or vice-versa. Both situations are possible. Furthermore $|\text{AUT}(G(u_i, \overline{u}_i))| = 1$.*

Proof. The vertices shaded black must always have the same color, because they are part of a chain of T subgraphs. It is easy to verify that every 3-coloring has the required property, and that there is only the identity automorphism. □

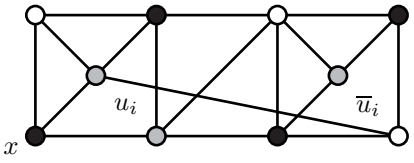


FIGURE 13.10
The graph $G(u_i, \overline{u}_i)$

The graph $G(u_i, \overline{u}_i)$ is used to relate a 3-coloring to the values of the Boolean variables u_i and $\overline{u}_i$. If the two vertices corresponding to u_i have the same color, then u_i is taken to be *true*. If they have different colors, then u_i is taken to be *false*. Lemma 13.26 ensures that if u_i is true, then $\overline{u}_i$ is false, and vice-versa. Because $|\text{AUT}(G(u_i, \overline{u}_i))| = 1$, it is always possible to distinguish the u_i vertices from the $\overline{u}_i$ vertices.

Each variable u_i has a graph $G(u_i, \overline{u}_i)$ constructed for it. Given a clause like $c_1 = (u_1 + \overline{u}_2 + \overline{u}_3)$, a subgraph C_1 is constructed to represent it, as in Figure 13.11. In general, C_j consists of a triangle with three "pendant" vertices, i.e., vertices of degree one. In any 3-coloring, the vertices of the triangle must use all three colors. The pendant vertices each represent one of the variables contained in the clause C_j, in this case $u_1, \overline{u}_2$, and $\overline{u}_3$. The vertex corresponding to a variable u_i or $\overline{u}_i$ will eventually have edges connecting it to the two vertices of $G(u_i, \overline{u}_i)$ corresponding to that variable.

Lemma 13.27. *Let C_j be colored with three colors white, gray, and black. Then there are colorings with one, two, or three pendant vertices colored white or gray. There are no colorings with all pendant vertices colored black.*

Proof. In Figure 13.11 it is obvious that u_1 cannot be colored *black*. It is easy to verify that there are 3-colorings in which $\overline{u}_2$ and/or $\overline{u}_3$ are also not colored *black*. □

We now construct a graph G combining these pieces as follows.

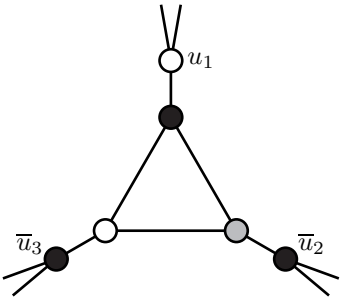


FIGURE 13.11
A subgraph C_j representing a clause $c_j = (u_1 + \overline{u}_2 + \overline{u}_3)$

- For each variable u_i, a graph $G(u_i, \overline{u}_i)$ is constructed.
- For each clause c_j, a graph C_j is constructed.
- The vertices corresponding to variable u_i or $\overline{u}_i$ in each C_j are connected to the u_i-vertices or $\overline{u}_i$-vertices, respectively, of $G(u_i, \overline{u}_i)$.
- Given graphs $G(u_i, \overline{u}_i)$ and $G(u_{i+1}, \overline{u}_{i+1})$, where u_{n+1} represents u_1, a T graph is used to connect the vertices labeled x of $G(u_i, \overline{u}_i)$ and $G(u_{i+1}, \overline{u}_{i+1})$ (see Figure 13.10), by identifying a and b of T with the x-vertex of $G(u_i, \overline{u}_i)$ and of $G(u_{i+1}, \overline{u}_{i+1})$, respectively.

The purpose of the last step is to ensure that the vertices of $G(u_i, \overline{u}_i)$ colored black in Figure 13.10 have the same color in every $G(u_i, \overline{u}_i)$.

Theorem 13.28. *G has a 3-coloring if and only if the instance of 3-Sat has a solution.*

Proof. Suppose that G has a 3-coloring using *white*, *gray*, and *black*. Without loss of generality, suppose that the black vertices of every $G(u_i, \overline{u}_i)$ as in Figure 13.10 are all colored *black*. Then all u_i and $\overline{u}_i$ vertices are colored *white* or *gray*. Consider any clause c_j with corresponding graph C_j. By Lemma 13.27, either one, two, or three of the pendant vertices of C_j is colored *white* or *gray*. Say the vertex corresponding to variable u_i is colored *white*. Then the u_i-vertices of $G(u_i, \overline{u}_i)$ must both be colored *gray*. Assign the value *true* to this u_i. Notice that u_i and $\overline{u}_i$ cannot both be assigned *true*, by Lemma 13.26. Then each c_j has at least one variable that is *true*, giving a solution.

Conversely, suppose that a solution is given to the instance of 3-Sat. One of each $u_i, \overline{u}_i$ is *true*, and the other is *false*. Choose a variable u_i or $\overline{u}_i$ that is *true*. Without loss of generality, suppose that $u_i = u_1$ and color the vertices of $G(u_1, \overline{u}_1)$ as follows. The black vertices of Figure 13.10 are colored *black*. The u_1 vertices are colored *gray*, and the $\overline{u}_1$ vertices are colored *white* and *gray* in the unique way.

Now for each clause c_j containing u_1, proceed as follows. Without loss of generality, take $c_j = u_1 + \overline{u}_2 + \overline{u}_3$ as an example, and consider C_j, as in Figure 13.11. The vertex of C_j corresponding to u_1 is colored *white*, and its adjacent triangle vertex is colored *black*. The other two triangle vertices are colored *white* and *gray*, with the adjacent pendant vertex to each colored *black*. If $\overline{u}_2$ is *true*, then color the $\overline{u}_2$-vertices of $G(u_2, \overline{u}_2)$ both *gray*, and extend the coloring uniquely to the u_2-vertices. If $\overline{u}_2$ is *false*, then color the $\overline{u}_2$-vertices *white* and *gray* and extend the coloring. Do the same for the $G(u_3, \overline{u}_3)$ graph.

Now every $G(u_i, \overline{u}_i)$ has been colored, where u_i or $\overline{u}_i$ appears in some clause together with u_1. Let U be this set of variables. Consider any clause c_j, all of whose variables are in U. At least one of the variables in c_j is *true*, say u_i, and its vertices in $G(u_i, \overline{u}_i)$ have been colored. Color C_j by coloring its pedant vertex adjacent to u_i in the unique way, and color its adjacent triangle vertex *black*. The coloring extends to all of C_j by coloring the other pendant vertices *black*. This produces a coloring of all clauses whose variables are all in U.

If there are more clauses c_j containing at least one variable of U, then for each one, find a variable it contains with value *true* and color the associated C_j. Continue like this until all of G is colored. □

We conclude that a polynomial algorithm which solves 3-Colorability could also be used to solve 3-Sat. Since 3-Sat is NP-complete, so is 3-Colorability.

Problem 13.2:	Clique
Instance:	a graph G and an integer k.
Question:	does G have a clique with $\geq k$ vertices (i.e., is $\overline{\alpha}(G) \geq k$)?

We transform from Problem 11.5.11.4 Vertex Cover. Recall that a vertex cover in a graph G is a subset $U \subseteq V(G)$ such that every edge has at least one endpoint in U. Given an integer k the Vertex Cover problem asks: does G have a vertex cover with $\leq k$ vertices?

If U is a vertex cover, the set $\overline{U} = V(G) - U$ is an independent set in G. Hence $\overline{U}$ induces a clique in $\overline{G}$. If G has n vertices, and U has $\geq m$ vertices, then U is a vertex cover with $\leq n - m$ vertices. Thus, given an instance of the Vertex Cover problem, we construct $\overline{G}$, and ask whether it has a clique with at least $n - m$ vertices. If the anwer is yes, then G has a vertex cover with at most m vertices. We conclude that a polynomial algorithm which solves Clique could also be used to solve Vertex Cover. It follows that Clique is NP-complete.

Problem 13.3:	Chromatic Index
Instance:	a graph G.
Question:	is $\chi'(G) = \Delta(G)$?

There is an ingenious construction of Holyer proving that it is NP-complete to

determine whether a 3-regular graph is of Class I or II. Holyer's construction is based on the graph shown in Figure 13.12, which he calls an *inverting component*, or *inverter*. The inverter was originally discovered by Loupekine. It consists of two 5-cycles sharing two common consecutive edges. Edges a, b, c, d, e are attached to five of the vertices.

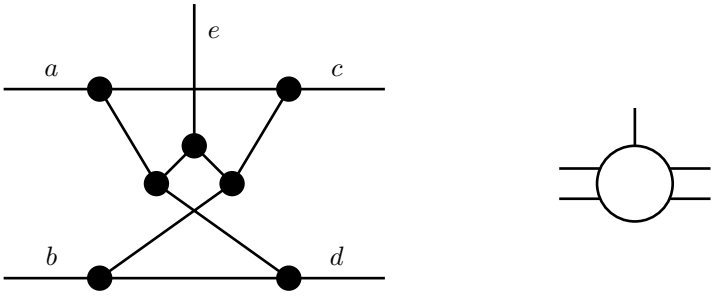


FIGURE 13.12
The inverter and its schematic representation
(Used with permission of the SIAM Journal of Computing)

A 5-cycle requires at least three colors. Consider any 3-edge-coloring of the inverter. Apply the parity condition (Lemma 11.13) to the set of seven vertices shown in the diagram. We determine that in any 3-edge coloring, that three of the five edges a, b, c, d, e must have the same color, and that the other two have distinct colors, since 5 can only be written as $3 + 1 + 1$, as a sum of three odd numbers. We then find that a, e, and c cannot all have the same color, so that in any 3-edge-coloring, either a and b have the same color, and c, d, e have three distinct colors; or by symmetry, c and d have the same color, and a, b, e have three distinct colors. The inverter is represented schematically in Figure 13.12, as a circle with two pairs of inputs (or outputs) and a fifth input (or output).

Holyer transforms from 3-Sat to Chromatic Index. Consider an instance of 3-Sat with clauses $c_1, c_2, \ldots, c_m$ involving variables $u_1, u_2, \ldots, u_n$ and their complements $\overline{u}_i$. Each variable is either true or false. A value of true is represented in the edge coloring of an inverter, as an input pair of the same color. A value of false is represented as an input pair of different colors. In every 3-edge-coloring of the inverter, a value of true is inverted to a value of false, or vice versa.

A clause of three variables, for example, $(u_i + u_j + \overline{u}_k)$, is represented by three inverters tied together by a 7-cycle, as shown in Figure 13.13. Note the edges marked a, b, x, y. It is easy to see that a and b have the same color if and only if x and y have the same color. Because of the 7-cycle, at least one of the three inverters must have two outside inputs the same color. This will be used to indicate that at least one of the variables u_i, u_j, and $\overline{u}_k$ in the above clause must have a value of true (i.e., every clause will be satisfied).

Now a given variable u_i, and/or its complement $\overline{u}_i$, will appear in several clauses. It must have the same value in all clauses. In order to ensure this, the 7-cycles cor-

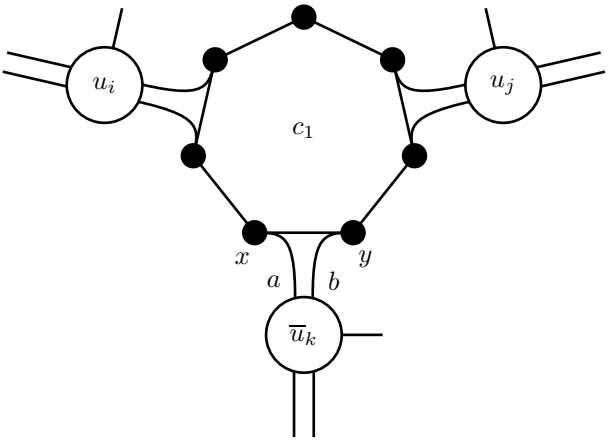


FIGURE 13.13
The representation of a clause $\boldsymbol{c_1 = (u_i + u_j + \overline{u}_k)}$ of 3-Sat
(Used with permission of the SIAM Journal of Computing)

responding to clauses containing u_i or $\overline{u}_i$ are also tied together. Holyer constructs a cycle of inverters, with two inverters for each occurrence of u_i or $\overline{u}_i$ in the clauses. For example, if u_i and $\overline{u}_i$ occur a total of four time in clauses c_1, c_2, c_3, c_4, the cycle of Figure 13.14 is constructed. Each pair of inverters corresponds to a clause containing u_i or $\overline{u}_i$. Notice that there are a total of six inputs connecting a pair of inverters to the rest of the graph. By the parity lemma, if this graph is 3-edge-colored, the possible color combinations for the six inputs are $6+0+0$, $4+2+0$, or $2+2+2$. If one pair of external inputs represents true, then by the properties of an inverter, the opposite pair also represents true. Then the parity lemma implies that the remaining two inputs also represent true. It follows that all pairs of inverters in the diagram have all external input pairs representing true. Consequently, if any one pair represents false, they all represent false. This mechanism is used to guarantee that u_i has the same value in all clauses.

We now put these ideas together. We are given an instance of 3-Sat. For each clause c_j, three inverters are tied together using a 7-cycle, as in Figure 13.13. For each variable u_i, a cycle of pairs of inverters is constructed as in Figure 13.14. The input corresponding to c_j in this structure is connected to the input corresponding to u_i in the 7-cycle component corresponding to c_j. If the clause contains $\overline{u}_i$ rather than u_i, then another inverter is placed between the two before connecting. The result is a graph containing a subgraph for each u_i and for each c_j, tied together through their inputs. There are still a number of inputs not connected to anything. In order to complete the construction and have a 3-regular graph, a second copy of this graph is constructed. The corresponding unused inputs of the two copies are connected together. The result is a 3-regular graph G. In any 3-edge-coloring of G, every clause is guaranteed to have at least one variable representing true. All occurrences of each

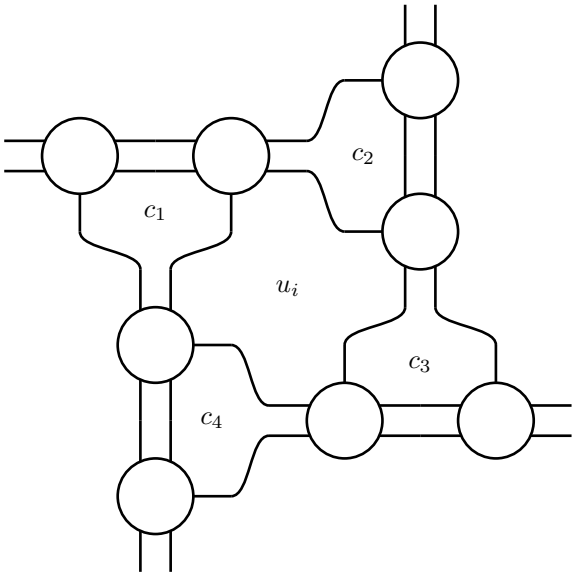


FIGURE 13.14
A pair of inverters for each clause containing u_i or $\overline{u}_i$
(Used with permission of the SIAM Journal of Computing)

variable are guaranteed to have the same value. The result is an assignment of values to the variables solving the 3-Sat instance. We conclude that if we could determine whether $\chi'(G) = 3$, we could solve 3-Sat. Since 3-Sat is NP-complete, we conclude that Chromatic Index is also NP-complete.

Problem 13.4:	C_5-Coloring.
Instance:	a graph G.
Question:	is there a homomorphism $\phi : G \to C_5$?

We transform from 5-Colorability, which is known to be NP-complete. We show that if C_5-Coloring can be solved in polytime, then a C_5 coloring of G can be used to construct a 5-coloring of G. Let the vertices of C_5 be $\{v_1, v_2, v_3, v_4, v_5\}$, with edges $v_i v_{i+1}$, where v_6 is taken to mean v_1. Let G be a graph to be 5-colored. Transform G into a new graph G^* by subdividing each edge of G with two new vertices, so that each edge of G becomes a path of length three in G^*. An example is shown in Figure 13.15, using $G = K_4$, where the vertices of K_4 are shaded. Suppose that a homomorphism $\phi : G^* \to C_5$ has been found. Let V_i denote the vertices of G^* such that $\phi(V_i) = v_i$. Color the vertices of $V(G) \cap V_i$ with color i. We show that this is a proper 5-coloring of G. Let uv be an edge of G. In G^*, the shortest uv-path has

length three. It must be mapped by ϕ to a walk of odd length ≤ 3 in C_5. But C_5 has girth five. Therefore $\phi(u) \neq \phi(v)$. It follows that this is a proper 5-coloring of G.

Conversely, given any proper 5-coloring of G, we construct a homomorphism $\phi : G^* \to C_5$ mapping the vertices of color i in G to v_i. Let uv be an edge of G, and suppose that u has color 1. The same argument will apply to all other colors. If v has color 2, then the uv-path in G^* can be mapped to the walk $v_1v_2v_1v_2$ in C_5. If v has color 3, then the uv-path in G^* can be mapped to the walk $v_1v_5v_4v_3$ in C_5. A similar argument holds for the other cases, since C_5 has a path of length three connecting any two non-adjacent vertices. Every vertex of G^* is mapped to a vertex of C_5. Therefore the homomorphism ϕ exists if and only if G can be 5-colored.

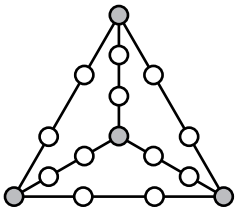


FIGURE 13.15
A transformation of K_4

13.9 Notes

The DEGREESATURATION algorithm is from BRELAZ [25]. A very good paper on the limitations of the sequential algorithm is JOHNSON [95]. A very readable survey of chromatic polynomials appears in READ and TUTTE [145]. See also TUTTE [176], where the word *chromial* is coined for chromatic polynomial. The proofs of the uncolored edge lemma and Ore's lemma (Lemmas 13.15 and 13.16) are based on those of BERGE [14]. A very efficient edge-coloring algorithm based on Vizing's theorem was developed by ARJOMANDI [8]. The edge-coloring algorithm presented here is based on Vizing's original algorithm VIZING [184]. A study of Class I and Class II graphs can be found in WALLIS [186]. A great source of information on graph homomorphisms is the book by HELL and NEŠETŘIL [82]. The proof of the NP-completeness of C_5-Coloring is based on their proof. In a classic paper [96], KARP presents 21 basic problems, and proves they are NP-complete. The proof of the NP-completeness of Chromatic Index presented here is based on HOLYER [86]. Figures 13.12, 13.13, and 13.14 are modified diagrams based on those appearing in Holyer's paper. They are used with the permission of the SIAM Journal of Computing.

14

Planar Graphs

14.1 Introduction

A graph G is *planar* if it can be drawn in the plane such that no two edges intersect, except at a common endpoint. The vertices of G are represented as points of the plane, and each edge of G is drawn as a continuous curve connecting the endpoints of the edge. For example, Figure 14.1 shows a planar graph (the cube), and a planar drawing of the same graph. Although the cube is planar, the drawing on the left is not a planar drawing. These drawings both have straight lines representing the edges, but any continuous curve can be used to represent the edges. We shall often find it more convenient to represent the vertices in planar drawings as circles rather than points, but this is just a drawing convenience.

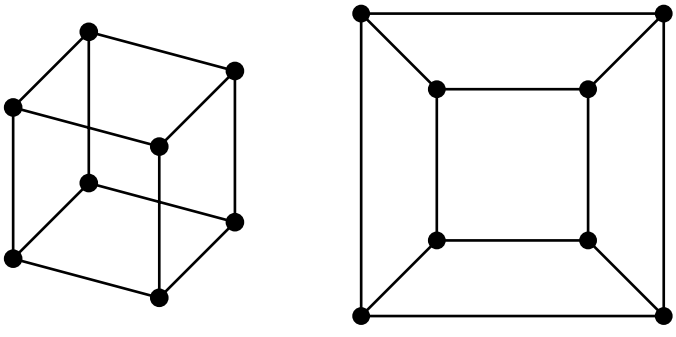


FIGURE 14.1
Two drawings of the cube

14.2 Jordan curves

Any closed, continuous, non-self-intersecting curve C drawn in the plane divides the plane into three regions: the points inside C, the points outside C, and the points of C itself. This is illustrated in Figure 14.2. We are relying on an intuitive understanding of the words "plane", "curve", "continuous", "region", etc. Exact mathematical definitions of these ideas would require a lengthy excursion into topology. An intuitive understanding should suffice for this chapter. Notice that the interior of C, denoted $\text{INT}(C)$, is bounded, because it is enclosed by C, and that the exterior, denoted $\text{EXT}(C)$, is unbounded, because the plane is unbounded. If u is any point in $\text{INT}(C)$, and $v \in \text{EXT}(C)$, then *any continuous curve with endpoints u and v must intersect C in some point.* This fact is known as the *Jordan curve theorem*. It is fundamental to an understanding of planarity.

DEFINITION 14.1: A closed, continuous, non-self-intersecting curve C in a surface is called a *Jordan curve*.

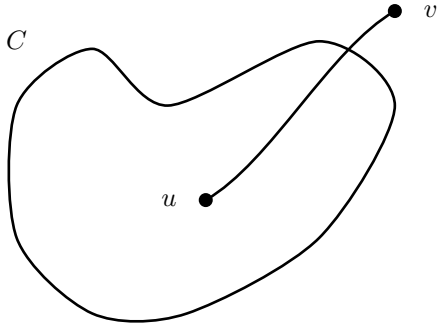


FIGURE 14.2
The Jordan curve theorem

Let G be any graph. We would like to construct a planar embedding of G, if possible. That is, we want to map the vertices of G into distinct points in the plane, and the edges of G into continuous curves that intersect only at their endpoints. Let ψ denote such a mapping. We write G^ψ to indicate the image of G under the mapping ψ. Let C be any cycle in G. If ψ is a planar embedding of G, then ψ maps C onto C^ψ, a Jordan curve in the plane. For example, consider $G = K_5$. Let $V(K_5) = \{v, w, x, y, z\}$. If K_5 were planar, the cycle $C = (x, y, z)$ must embed as a Jordan curve C^ψ in the plane. Vertex u is either in $\text{INT}(C^\psi)$ or $\text{EXT}(C^\psi)$. Without loss of generality, we can place u in $\text{INT}(C^\psi)$, as in Figure 14.3. The paths ux, uy, and uz then divide $\text{INT}(C^\psi)$ into three smaller regions, each bounded by a Jordan curve. We cannot place v in $\text{EXT}(C^\psi)$, as we then cannot embed the path uv without crossing C^ψ. We cannot place v in any of the smaller regions in $\text{INT}(C^\psi)$, for in

each case there is a vertex outside the Jordan curve bounding the region that cannot be reached. We conclude that K_5 cannot be embedded in the plane. K_5 is a *non-planar* graph. We state this as a lemma.

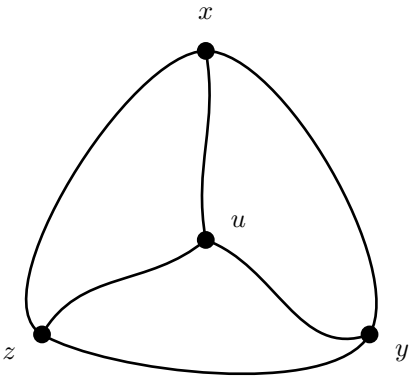


FIGURE 14.3
Embedding $\boldsymbol{K_5}$

Lemma 14.1. *K_5 and $K_{3,3}$ are non-planar graphs.*

Proof. The proof for K_5 appears above. The proof for $K_{3,3}$ is in Exercise 12.3.1. □

14.3 Graph minors, subdivisions

The graphs K_5 and $K_{3,3}$ are special graphs for planarity. If we construct a graph from K_5 by replacing one or more edges with a path of length ≥ 2, we obtain a *subdivision* of K_5. We say that the edges of K_5 have been *subdivided*.

DEFINITION 14.2: Given a graph G, a *subdivision* of G is any graph obtained from G by replacing one or more edges by paths of length two or more.

It is clear that any subdivision of K_5 or $K_{3,3}$ is non-planar, because K_5 and $K_{3,3}$ are non-planar. It is apparent that vertices of degree two do not affect the planarity of a graph. The inverse operation to subdividing an edge is to *contract* an edge with an endpoint of degree two.

DEFINITION 14.3: Graphs G_1 and G_2 are *topologically equivalent* or *homeomorphic*, if G_1 can be transformed into G_2 by the operations of subdividing edges and/or contracting edges with an endpoint of degree two.

We will denote by TK_5 any graph that is topologically equivalent to K_5. Similarly, $TK_{3,3}$ denotes any graph that is topologically equivalent to $K_{3,3}$. In general, TK denotes a graph topologically equivalent to K, for any graph K. If G is a graph containing a subgraph TK_5 or $TK_{3,3}$, then G must be non-planar. Kuratowski's theorem states that this is a necessary and sufficient condition for a graph to be non-planar. We will come to it later.

If G is a planar graph, and we delete any vertex v from G, then $G - v$ is still planar. Similarly, if we delete any edge uv, then $G - uv$ is still planar. Also, if we contract any edge uv of G, then $G \cdot uv$ is still planar. Contracting an edge can create parallel edges or loops. Because parallel edges and loops do not affect the planarity of a graph, loops can be deleted, and parallel edges can be replaced by a single edge, if desired.

DEFINITION 14.4: Let H be a graph obtained from G by any sequence of deleting vertices and/or edges, and/or contracting edges. H is said to be a *minor* of G.

Notice that if G contains a subgraph TK_5, K_5 is a minor of G, even though K_5 need not be a subgraph of G. For we can delete all vertices and edges which do not belong to the subgraph TK_5, and then contract edges to obtain K_5. Similarly, if G has a subgraph $TK_{3,3}$, then $K_{3,3}$ is a minor of G, but need not be a subgraph. Any graph having K_5 or $K_{3,3}$ as a minor is non-planar. A special case of minors is when a graph K is subdivided to obtain G.

DEFINITION 14.5: Let G contain a subgraph that is a subdivision of a graph K, where $\delta(K) \geq 3$. Then K is said to be a *topological minor* of G.

Lemma 14.2. *If H is a minor of K, and K is a minor of G, then H is a minor of G.*

Proof. This follows from the definition. □

A consequence of this lemma is that the relation of being a graph minor is a partial order on the set of all graphs.

The inverse operation to contracting an edge whose endpoints have degree three or more is *splitting* a vertex.

DEFINITION 14.6: Let G be any graph with a vertex v of degree at least three. Let v be adjacent to vertices $\{u_1, u_2, \ldots, u_k\}$. Construct a graph G_v^+ by *splitting* vertex v: replace v with two new vertices v_1 and v_2. Join v_1 to $\ell_1 \geq 2$ of $\{u_1, u_2, \ldots, u_k\}$, and join v_2 to $\ell_2 \geq 2$ of them, such that together, v_1 and v_2 are adjacent to all of these vertices. Then join v_1 to v_2.

In any graph G_v^+ resulting from splitting vertex v, v_1 and v_2 both have degree at least three, and $G_v^+ \cdot v_1v_2 = G$, that is, splitting vertex v is an inverse operation to contracting the edge v_1v_2. Notice that G is a minor of G_v^+. Splitting a vertex is illustrated for $G = K_5$ in Figure 14.4. The following lemma shows that K_5 and $K_{3,3}$ are very closely related graphs.

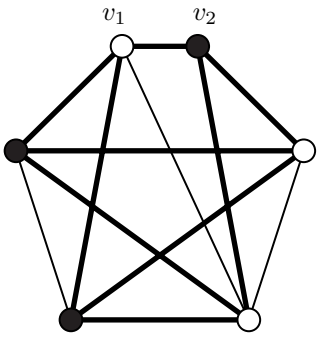


FIGURE 14.4
Splitting a vertex of $\boldsymbol{K_5}$

Lemma 14.3. *Let G be any graph obtained by splitting a vertex of K_5. Then G contains a subgraph $TK_{3,3}$.*

Proof. Let v_1 and v_2 be the two vertices resulting from splitting a vertex of K_5. Each has at least degree three. Consider v_1. It is joined to v_2. Together, v_1 and v_2 are joined to the remaining four vertices of G, and each is joined to at least two of these vertices. Therefore we can choose a partition of these four vertices into x, y and w, z such that $v_1 \longrightarrow x, y$ and $v_2 \longrightarrow w, z$. Then G contains a $K_{3,3}$ with bipartition v_1, w, z and v_2, x, y, as illustrated in Figure 14.4. □

In the previous example, it was convenient to form a minor K_5 of G by first deleting a subset of vertices and/or edges, and then contracting a sequence of edges to obtain K_5. All minors of G can be obtained in this way, as shown by the following lemma:

Lemma 14.4. *Suppose that G has a minor H. Then H can be obtained by first deleting a subset of vertices and/or edges of G, and then contracting a sequence of edges.*

Proof. Let $G_0, G_1, \ldots, G_k$ be a sequence of graphs obtained from G, where $G_0 = G$ and $G_k = H$, such that each G_i, where $i \geq 1$, is obtained from G_{i-1} by deleting a vertex, deleting an edge, or contracting an edge. If all deletions occur before contractions, there is nothing to prove. So let G_i be the first graph obtained from G_{i-1} by the deletion of an edge or vertex. Without loss of generality we can assume that $i \geq 2$, and that $G_1, \ldots, G_{i-1}$ were obtained by contracting edges $e_1, \ldots, e_{i-1}$, where e_j is an edge of G_{j-1}. Let $e_{i-1} = v_1v_2$.

Suppose first that $G_i = G_{i-1} - v$, for some vertex v. If v is the result of identifying v_1 and v_2 when e_{i-1} is contracted, then we can replace G_{i-1}, G_i in the sequence of graphs by G'_{i-1}, G'_i, where G'_{i-1} and G'_i are obtained by deleting v_1, and then v_2 from G_{i-2}. If v is not the result of identifying v_1 and v_2, we can interchange the order of G_i and G_{i-1} by deleting v before contracting e_{i-1}. In each case, we obtain an

equivalent sequence of graphs with only $i-1$ edge contractions preceding a deletion. The number of edges contracted does not increase, and the final result is still H.

Suppose now that $G_i = G_{i-1} - uv$, for some edge uv. We know that $G_{i-1} = G_{i-2} \cdot e_{i-1}$. We can reverse the order of the two operations, and delete uv before contracting e_{i-1}, thereby replacing G_{i-1}, G_i with graphs G'_{i-1}, G'_i. Again we obtain an equivalent sequence of graphs with only $i-1$ edge contractions preceding a deletion. We repeat this as many times as required until all deletions precede all contractions. □

It follows that when constructing a minor H of graph G, we can start with a subgraph of G and apply a sequence of edge-contractions only to obtain H. Often this is used as the definition of graph minor.

Consider the situation when a vertex v of degree three in G is split into v_1v_2. If v is adjacent to vertices x, y, z in G, then in G_v^+, v_1 is adjacent to at least two of x, y, z, and via v_2 there is always a path from v_1 to the third vertex. This is used in the following theorem, and also in Exercise 12.3.5.

Theorem 14.5. *If G has a minor $K_{3,3}$, then G contains a subgraph $TK_{3,3}$. If G has a minor K_5, then G contains a subgraph TK_5 or $TK_{3,3}$.*

Proof. Suppose that G has a minor K_5 or $K_{3,3}$. If no edges were contracted to obtain this minor, then it is also a subgraph of G. Otherwise let $G_0, G_1, \ldots, G_k$ be a sequence of graphs obtained from G, where G_0 is a subgraph of G, edge e_i of G_{i-1} is contracted to obtain G_i, and G_k is either K_5 or $K_{3,3}$.

If each e_i has an endpoint of degree two, then we can reverse the contractions by subdividing edges, resulting in a TK_5 or $TK_{3,3}$ in G, as required. Otherwise let e_i be the edge with largest i, with both endpoints of at least degree three. All edges contracted subsequent to G_i have an endpoint of degree two, so that G_i has a subgraph TK_5 or $TK_{3,3}$. G_{i-1} can be obtained by splitting a vertex v of G_i. If v is a vertex of TK_5, then by Lemma 14.3, G_{i-1} contains $TK_{3,3}$. If v is a vertex of $TK_{3,3}$, then G_{i-1} also contains $TK_{3,3}$. If v is a vertex of neither TK_5 nor $TK_{3,3}$, then G_{i-1} still contains TK_5 or $TK_{3,3}$. In each case we find that G_0 must have a subgraph TK_5 or $TK_{3,3}$. □

DEFINITION 14.7: Given a subgraph TK of G, equal to TK_5 or $TK_{3,3}$. The vertices of TK which correspond to vertices of K_5 or $K_{3,3}$ are called the *corners* of TK. The other vertices of TK are called *inner vertices* of TK.

Suppose that G is a non-planar graph with a subgraph TK_5. Let $v_1, v_2, \ldots, v_5$ be the corners of TK_5. Each v_i has degree four in TK_5; the inner vertices of TK_5 have degree two. Let P_{ij} be the path in TK_5 connecting v_i to v_j. Consider the situation where G contains a path P from $x \in P_{ij}$ to $y \in P_{k\ell}$, where x and y are inner vertices. This is illustrated in Figure 14.5, where a vertex $x \in P_{45}$ is connected by a path to $y \in P_{23}$. We see that in this case, G contains a $TK_{3,3}$.

Theorem 14.6. *Let G contain a subgraph TK_5, with corners $v_1, v_2, \ldots, v_5$ connected by paths P_{ij}. If G has vertices x and y such that x is an inner vertex of P_{ij}, and $y \in P_{k\ell}$ but $y \notin P_{ij}$, where $P_{ij} \neq P_{k\ell}$, then G contains a $TK_{3,3}$.*

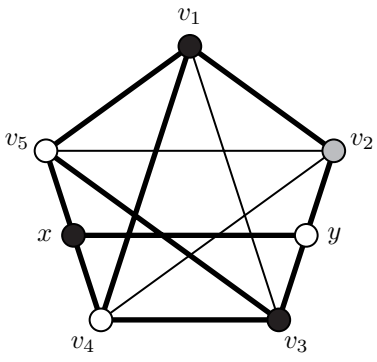


FIGURE 14.5
$\boldsymbol{TK_5}$ and $\boldsymbol{TK_{3,3}}$

Proof. One case of the proof is illustrated in Figure 14.5. The remaining cases are done in Exercise 12.3.2. □

A consequence of Theorem 14.6 is that nearly any graph that has a subgraph TK_5 also has a subgraph $TK_{3,3}$. This theorem will be useful in embedding algorithms for non-planar graphs in Chapter 15. It also permits a recursive characterization of graphs which contain TK_5 but not $TK_{3,3}$.

Exercises

14.3.1 Show that $K_{3,3}$ is non-planar, using the Jordan curve theorem.

14.3.2 Complete the proof of Theorem 14.6.

14.3.3 Characterize the class of 2-connected graphs which contain TK_5 but not $TK_{3,3}$.

14.3.4 Construct a $O(\varepsilon)$ algorithm which accepts as input a graph G and a subgraph TK_5 with corners $v_1, v_2, \ldots, v_5$, and finds a $TK_{3,3}$ containing $v_1, v_2, \ldots, v_5$ if one exists.

14.3.5 Let K be a graph such that $\Delta(K) \leq 3$. Show that K is a minor of G if and only if G has a subgraph TK.

14.4 Euler's formula

Let G be a connected planar graph with n vertices and ε edges, embedded in the plane by a mapping ψ. If we remove all the points of the image G^ψ from the plane, the plane falls apart into several connected regions. This is equivalent to *cutting* the

plane along the edges of G^ψ. For example, if we cut the plane along the edges of the planar embedding of the cube in Figure 14.1, there are six regions, one of which is unbounded.

DEFINITION 14.8: The *faces* of an embedding G^ψ are the connected regions that remain when the plane is cut along the edges of G^ψ. The unbounded region is called the *outer face*.

Notice that if uv is an edge of G contained in a cycle C, then $(uv)^\psi$ is a portion of a Jordan curve C^ψ. The face on one side of $(uv)^\psi$ is in $\text{INT}(C^\psi)$ and the face on the other side is in $\text{EXT}(C^\psi)$. These faces are therefore distinct. But if uv is an edge not contained in any cycle, then it is a cut-edge, and the same face appears on each side of $(uv)^\psi$. For example, in a tree, every edge is a cut-edge, and there is only one face, the outer face.

We view the plane as an oriented surface, which can be viewed from "above" or "below". Given an embedding G^ψ in the plane, we will view it consistently from one side, which we can assume to be "above" the plane. If we then view an embedding G^ψ from "below" the surface, it will appear to have been reversed. Therefore we choose one orientation ("above") for all embeddings.

The *boundary* of a face F of an embedding G^ψ is a closed curve in the plane. It is the image under ψ of a closed walk C in G. We can walk along C^ψ so that the interior of C^ψ is to our right-hand side. We will call this a *clockwise* direction and thereby assign an *orientation* to C. *We shall always choose a clockwise orientation for traversing the boundaries of faces*, so that the face F will be to our right-hand side.

DEFINITION 14.9: An oriented closed walk C in G bounding a face of G^ψ is called a *facial walk* of G^ψ (or *facial cycle* if C is a cycle).

Notice that if C contains a cut-edge uv, then uv will appear twice on C. The two occurrences of uv on C will have opposite orientations. All other edges appear at most once on C. As we will mostly be interested in 2-connected graphs G, facial walks will almost always be cycles.

DEFINITION 14.10: The *degree* of a face F is $\text{DEG}(F)$, the length of its facial walk.

Notice that a cut-edge appearing on the facial walk of F will contribute two to its degree.

Theorem 14.7. (Euler's formula) *Let G be a connected planar graph with n vertices and ε edges. Let G^ψ have f faces, where ψ is a planar embedding of G. Then*

$$n + f - \varepsilon = 2$$

Proof. The proof is by induction on $\varepsilon - n$. Every connected graph has a spanning tree. If $\varepsilon - n = -1$, then G is a tree. It is clear that every tree has a planar embedding. Because a tree has no cycles, there is only one face, the outer face, so $f = 1$. Euler's formula is then seen to hold for all embeddings of G.

Now suppose that $\varepsilon - n = k \geq 0$. Choose any cycle C in G, and any edge $uv \in C$. Let the faces on the two sides of $(uv)^\psi$ be F_1 and F_2. Consider $G' = G - uv$, with n', ε' and f' vertices, edges, and faces, respectively. Clearly $n' = n$ and $\varepsilon' = \varepsilon - 1$. G' is connected and ψ is a planar embedding of it. One of the faces of G'^ψ is $F_1 \cup F_2$. The other faces of G'^ψ are those of G^ψ. Therefore $f' = f - 1$. Euler's formula follows by induction. □

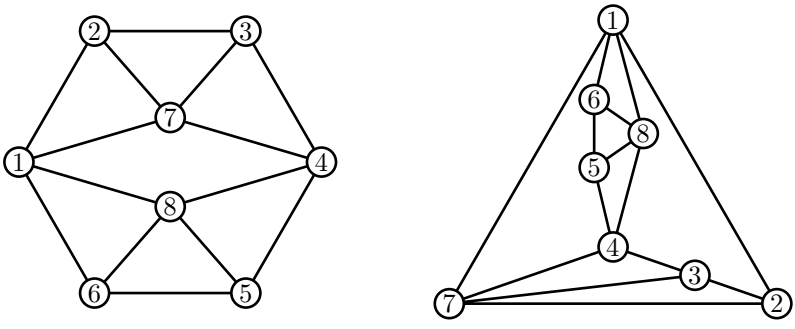


FIGURE 14.6
Two embeddings of a graph

It follows from Euler's formula that all planar embeddings of a connected graph G have the same number of faces. Hence we will refer to $f(G)$ as the number of faces of G, without specifying an embedding. In Figure 14.6 there is an example of a graph with two distinct embeddings. The embeddings have the same number of faces, but the actual faces and their boundaries are different.

14.5 Rotation systems

Once we have an embedding G^ψ, we can choose any vertex $v \in V(G)$, and walk around v^ψ in a small, clockwise circle. We encounter the incident edges in a certain cyclic order. For example, in the embedding on the left of Figure 14.6, the edges incident on vertex 1 have the clockwise cyclic order $(12, 17, 18, 16)$. Those incident on vertex 2 have the order $(23, 27, 21)$. Those incident vertex 3 have the order $(34, 37, 32)$, etc.

In Figure 14.6, the edges are drawn as straight lines. It is conceivable that the embedding ψ could assign wildly behaved functions, like $\sin(1/x)$ to the curves representing the edges of G. Each edge may then be encountered many times when walking around v^ψ in a small circle, no matter how small the circle is chosen. We will assume that this does not occur, and that ψ assigns well behaved functions (like straight lines or gradual curves) to the edges. In fact, for graphs embedded on the

plane, we shall see that it is always possible to draw the edges as straight lines. For a more complete treatment, the reader is referred to the books of GROSS and TUCKER [74] or MOHAR and THOMASSEN [126].

DEFINITION 14.11: Let G^ψbe an embedding in the plane of a loopless connected graph G. A *rotation system* p for G is a mapping from $V(G)$ to the set of permutations of $E(G)$, such that for each $v \in V(G)$, $p(v)$ is the cyclic permutation of edges incident on v, obtained by walking around v^ψ in a clockwise direction.

Notice that if G has a loop vv, then as we walk around v^ψ, we will cross the loop twice. Therefore $p(v)$ will contain vv twice. In order to extend the definition to be correct for graphs with loops, we must ensure that for each loop vv, that $p(v)$ contains two corresponding "loops" $(vv)_1$ and $(vv)_2$.

Suppose we are given the rotation system p determined by an embedding G^ψ. We can then easily find the facial cycles of G^ψ. The following fundamental algorithm shows how to do this. Let $u \in V(G)$, and let e be any edge incident on u. We are assuming that given an edge $e' = uv$ in $p(u)$, we can find the corresponding edge $e'' = vu$ in $p(v)$. A data structure of linked lists can do this easily in constant time. If G is a simple graph, then the rotation system is completely specified by the cyclic adjacency lists. Given a planar embedding G^ψ, we will assume that the adjacency lists are always given in cyclic order, so that the rotation system p corresponding to ψ is available.

Algorithm 14.5.1: FACIALCYCLE(G^ψ, u, e)

comment: Given an embedding G^ψ with corresponding rotation system p and vertex u with incident edge e, find the facial cycle containing e.

$e' \leftarrow e$
repeat
  comment: e' currently equals uv, for some v
  $v \leftarrow$ other end of e'
  $e'' \leftarrow$ edge of $p(v)$ corresponding to e'
  comment: e'' currently equals vu
  $e' \leftarrow$ edge preceding e'' in $p(v)$
  $u \leftarrow v$
until $e' = e$

Lemma 14.8. *The sequence of edges traversed by* FACIALCYCLE(G^ψ, u, e) *forms the facial cycle of the face to the right of* e^ψ.

Proof. Let F be the face to the right of e^ψ. Let $e = uv$. As we walk along e^ψ from u^ψ to v^ψ, the face F is to our right-hand side. When we reach v^ψ, we are on the image of an edge vu in $p(v)$. Because $p(v)$ has a clockwise cyclic order, the next edge in the facial cycle is the one preceding vu in $p(v)$. This is the one chosen by the

algorithm. The algorithm repeats this process until it arrives back at the starting edge e. □

Algorithm FACIALCYCLE() is very simple, but it is tremendously important. It is used in nearly all algorithms dealing with graph embeddings. Notice that its running time is $O(\varepsilon)$ and that it can be used to find all the facial cycles of G^ψ in $O(\varepsilon)$ time.

Corollary 14.9. *The facial cycles of an embedding G^ψ are completely determined by its rotation system.*

Proof. All facial cycles can be determined by executing Algorithm FACIALCYCLE(G^ψ, u, e), such that each edge e is visited at most twice, once for the face on each side of e. The rotation system is the only information about ψ that is needed. □

Thus, it turns out that planar embeddings are essentially combinatorial, as the facial cycles of an embedding are completely determined by the set of cyclic permutations of incident edges of the vertices. Later we will see that for 3-connected planar graphs, the rotation system is unique, up to orientation. If p is the rotation system corresponding to an embedding ψ, we will often write G^p instead of G^ψ. We call G^ψ a *topological embedding*, and G^p a *combinatorial embedding*. The combinatorial embedding determines the facial cycles of the embedding, but it does not give an actual drawing of G in the plane.

DEFINITION 14.12: A *plane map* is a combinatorial embedding G^p, where p is a rotation system for an embedding of G in the plane.

14.6 Dual graphs

Consider an embedding G^ψ, illustrated by the cube in Figure 14.7. Let its faces be listed as $\{F_1, F_2, \ldots, F_f\}$. Two faces F_i and F_j are adjacent if they share a common edge $(uv)^\psi$ on their boundaries. We can construct a planar graph $G^{\psi*}$ by placing a new vertex f_i in each region F_i, for $i = 1, 2, \ldots, f$. Whenever two faces F_i and F_j share an edge $(uv)^\psi$ on their boundaries, we draw a continuous curve from f_i to f_j, passing through $(uv)^\psi$ in exactly one interior point, and intersecting G^ψ in only this point. This is illustrated for an embedding of the cube in Figure 14.7. We call $G^{\psi*}$ a *planar dual* of G^ψ.

Lemma 14.10. *Let G^ψ be a planar embedding with a planar dual $G^{\psi*}$. Let $G^{\psi**}$ be any planar dual of $G^{\psi*}$. Then $G^{\psi**} \cong G^\psi$.*

Proof. Let the faces of G^ψ be $F_1, F_2, \ldots, F_f$, and let f_i be the vertex of $G^{\psi*}$ corresponding to F_i. Consider any vertex u of G and its cyclic permutation $p(u) = (uv_1, uv_2, \ldots, uv_k)$ of incident edges. Each edge $(uv_\ell)^\psi$ separates two faces F_i and

F_j, and so is crossed by a curve connecting f_i to f_j. As we successively take the edges uv_ℓ of $p(u)$, we traverse these curves, thereby constructing a facial boundary of $G^{\psi *}$. Vertex u^ψ is contained in the region interior to this facial boundary. We conclude that each face of $G^{\psi *}$ contains exactly one u^ψ, and that the edges $(uv_\ell)^\psi$ are curves connecting the vertices u^ψ and v_ℓ^ψ located inside the faces of $G^{\psi *}$. That is, the planar dual construction applied to $G^{\psi *}$ gives back G^ψ. Equivalently, $G^{\psi **} \cong G^\psi$. □

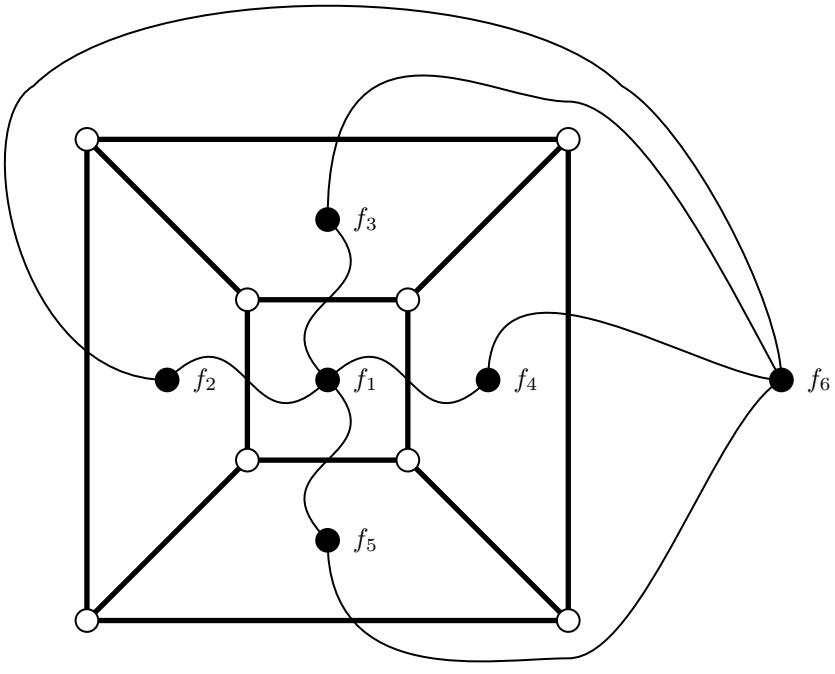


FIGURE 14.7
Constructing a dual graph

Now the adjacencies of the planar dual are determined completely by common edges of the facial cycles of G^ψ, and these are determined by the rotation system p of G^ψ. Therefore we define the combinatorial planar dual in terms of the rotation system.

DEFINITION 14.13: Let p be the rotation system for G corresponding to a planar embedding ψ. Let the facial cycles of G^p be $F = \{F_1, F_2, \ldots, F_f\}$. The *combinatorial planar dual* of G^p is denoted G^{p*}. The vertex set of G^{p*} is $\{F_1, F_2, \ldots, F_f\}$. The edges of G^{p*} are defined by a rotation system, also denoted p, and given as follows. Consider a facial cycle

$$F_i = (v_1, v_2, \ldots, v_k),$$

traversed in a clockwise direction. Each edge $v_\ell v_{\ell+1}$ is contained in exactly two facial cycles, which are adjacent in G^{p*} . As we walk along the facial cycle, the face corresponding to F_i appears on the right-hand side of $(v_\ell v_{\ell+1})^\psi$. On the left-hand side is the face corresponding to $F_{\ell'}$, where $F_{\ell'}$ is the unique facial cycle containing edge $v_{\ell+1} v_\ell$. We then take

$$p(F_i) = (F_{1'}, F_{2'}, \ldots, F_{k'}).$$

It is easy to see that $F_i F_j$ occurs in $p(F_i)$ if and only if $F_j F_i$ occurs in $p(F_j)$. Thus the definition is valid. If F_i contains a cut-edge uv, then the same face occurs on both sides of $(uv)^\psi$. G^{p*} will then contain a loop $F_i F_i$. Because uv appears twice on the facial cycle, $F_i F_i$ will occur twice in $p(F_i)$.

The graph G^{p*} constructed by this definition is always isomorphic to the planar dual $G^{\psi*}$ constructed above, because of the correspondence between faces and facial cycles. Therefore the rotation system constructed for G^{p*} always corresponds to a planar embedding $G^{\psi*}$. It follows from the Lemma 14.10 that $G^{p**} \cong G^p$.

Now let G^p denote a combinatorial planar embedding of G, and let $\{F_1, F_2, \ldots, F_f\}$ be the facial cycles of G^p. The degree of F_i is $\textsc{Deg}(F_i)$, the length of the walk. Let G^{p*} be the dual of G^p, and write n^*, ε^*, and f^* for the numbers of vertices, edges, and faces, respectively, of G^{p*}.

Lemma 14.11. $\displaystyle\sum_{i=1}^{f} \textsc{Deg}(F_i) = 2\varepsilon(G).$

Proof. Each edge uv of G is incident on two faces of G^p. □

Lemma 14.12. *$n^* = f$, $f^* = n$, and $\varepsilon^* = \varepsilon$.*

Proof. $n^* = f$ follows from the definition of G^{p*}. Because $G^{p**} \cong G^p$, we have $f^* = n$. Each edge of G^{p*} corresponds to exactly one edge of G^p, and every edge of G^p corresponds to exactly one edge of G^{p*}. Therefore $\varepsilon^* = \varepsilon$. □

Algorithm 14.6.1: CONSTRUCTDUAL(G^p)

comment: Given a graph G with a planar rotation system p, construct the combinatorial dual G^{p*}.

nFaces $\leftarrow 0$
for all edges uv
  do *FaceNumber*$\langle uv \rangle \leftarrow 0$
for all vertices u
do **for each** uv in $p(u)$
  do **if** *FaceNumber*$\langle uv \rangle = 0$
    then *nFaces* $\leftarrow$ *nFaces* $+ 1$
      traverse the facial cycle F to the right of uv
      using FACIALCYCLE(G^p, u, uv) and store
      nFaces in the *FaceNumber* of each edge of F

comment: we have numbered all the faces, now construct the edges of the dual

for all vertices u
do **for each** uv in $p(u)$
  do $i \leftarrow$ *FaceNumber*$\langle uv \rangle$
    if face i has not been traversed yet
    then traverse the facial cycle F to the right of uv
      using FACIALCYCLE(G^p, u, uv),
      and for each edge xy of F
      let $j =$ *FaceNumber*$\langle yx \rangle$ in $p(y)$
      append ij to $p(i)$ in G^{p*}

Algorithm 14.6.1 is a simple algorithm to construct the dual in $O(\varepsilon)$ time. We assume that the faces of G^p are numbered $1, 2, \ldots, f$; that the rotation system p is represented as cyclic linked lists; and that the linked list node corresponding to uv in $p(u)$ contains a field called the *FaceNumber*, used to indicate which face of G^p is on the right-hand side of uv as it is traversed from u to v. We will denote this by *FaceNumber*$\langle uv \rangle$, although it is not stored as an array. We will also use a variable *nFaces* to count the faces of G^p.

Algorithm 14.6.1 uses FACIALCYCLE() (Algorithm 14.5.1) to walk around the facial cycles of G^p and number them. Notice that FACIALCYCLE() only requires the rotation system p rather than the topological embedding ψ. Each edge of G is traversed exactly twice, taking a total of $O(\varepsilon)$ steps. It then traverses the facial cycles again, constructing the rotation system of G^{p*}, using the face numbers which were previously stored. This again takes $O(\varepsilon)$ steps. Thus, the dual graph is completely determined by the combinatorial embedding G^p.

14.7 Platonic solids, polyhedra

The cube is a 3-regular planar graph whose faces all have degree four. So its dual is 4-regular. It can be seen from Figure 14.7 that the dual of the cube is the octahedron. Let G be a connected k-regular planar graph whose dual is ℓ-regular, where $k, \ell \geq 3$. These graphs are called graphs of the *Platonic solids*. Then $kn = 2\varepsilon$ and $\ell f = 2\varepsilon$. Substituting this into Euler's formula and dividing by ε gives

$$\frac{1}{k} + \frac{1}{\ell} = \frac{1}{2} + \frac{2}{\varepsilon}$$

If we consider graphs with $\varepsilon \geq 4$ edges, we have

$$\frac{1}{2} < \frac{1}{k} + \frac{1}{\ell} \leq 1$$

As the number of integers satisfying this inequality is limited, this can be used to find all such graphs. They are the graphs of the regular polyhedra – the tetrahedron, cube, octahedron, dodecahedron, and icosahedron.

In general, a *polyhedron* is a geometric solid whose faces are *polygons*, that is, regions of a plane bounded by a finite sequence of line segments. Each edge of a polyhedron is common to exactly two polygonal faces. So if we consider an edge uv_1 incident on a vertex u, there are two polygons, P_1 and P_2 incident on uv_1. Now P_2 has two edges incident on u. Let the other be uv_2. But this edge is also incident on two polygons, P_2 and P_3. Because P_3 has two edges incident on u, we obtain another edge uv_3, etc. Continuing in this way, we get a sequence $v_1, v_2, \ldots$ of vertices adjacent to u, until we return to P_1.

DEFINITION 14.14: A *polyhedron* is a connected collection of polygons such that

1. Each edge is contained in exactly two polygons.
2. Polygons do not intersect, except on a common edge.
3. Any two polygons intersect in at most one edge.
4. The polygons incident on a vertex form a single cycle.

The fourth condition is to prevent a polyhedron created by identifying two vertices of otherwise disjoint polyhedra.

DEFINITION 14.15: The *skeleton* of a polyhedron is the graph whose vertices are the vertices of the polyhedron, such that vertices u and v are adjacent in the graph if and only if uv is an edge of the polyhedron.

DEFINITION 14.16: A *regular polygon*, denoted $\{p\}$, is a planar polygon with p sides all of equal length. A *regular polyhedron*, denoted $\{p, q\}$, is a polyhedron

whose faces are polygons $\{p\}$, such that exactly q polygons are incident on each vertex.

The symbols $\{p\}$ and $\{p, q\}$ are called the *Schläfli symbols* for the polygon and polyhedron, respectively. Notice that given a vertex u of a regular polyhedron $\{p, q\}$, the midpoints of the edges incident on u form a regular polygon $\{q\}$. This polygon corresponding to vertex u is called the *vertex figure* of u.

A polyhedron is *convex* if its interior is a convex region; that is, given any two points P and Q in the interior, the line segment connecting P to Q is completely contained inside the polyhedron. There is a remarkable theorem of Steinitz characterizing the skeletons of convex polyhedra.

Theorem 14.13. (Steinitz's theorem) *A graph G is the skeleton of a convex polyhedron if and only if G is planar and 3-connected.*

A proof of Steinitz's theorem can be found in the book by GRÜNBAUM [76] or ZIEGLER [196]. It is too lengthy to include here.

Exercises

14.7.1 Find the planar dual of the line graph of K_4. Find the line graph of the cube, and find its planar dual.

14.7.2 Find all k-regular planar graphs whose duals are ℓ-regular, for all possible values of k and ℓ.

14.7.3 Find the dual of the multigraph constructed from K_4 by doubling each edge. Refer to Figure 14.8.

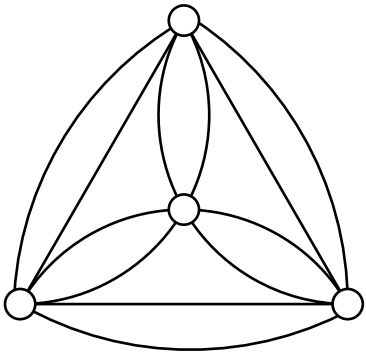


FIGURE 14.8
Find the dual graph

14.7.4 A planar graph G is *self-dual* if it is isomorphic to its planar dual. Find a self-dual planar graph on n vertices, for all $n \geq 4$.

14.7.5 Program the algorithm CONSTRUCTDUAL(), and test it on the graphs of Exercises 14.7.1 and 14.7.2.

14.7.6 Show that a planar graph is bipartite if and only if its dual is Eulerian.

14.7.7 Find the Schläfli symbols of the tetrahedron, cube, octahedron, dodecahedron, and icosahedron.

14.7.8 Given a Schläfli symbol $\{p, q\}$ for a polyhedron, find a formula for the number of vertices, edges, and faces of the polyhedron in terms of p and q.

14.8 Triangulations

A planar embedding G^p whose faces all have degree three is called a *triangulation*. If G^p is a triangulation, then $3f = 2\varepsilon$. Substituting into Euler's formula gives:

Lemma 14.14. *A triangulation G^p satisfies $\varepsilon = 3n - 6$ and $f = 2n - 4$.*

If G^p is not a triangulation, and has no multiple edges or loops, then every face has at least degree three. We can convert G^p to a triangulation by adding some diagonal edges to faces of degree four or more. This gives the following:

Lemma 14.15. *A simple planar graph G has $\varepsilon \leq 3n - 6$.*

For example, because K_5 has $\varepsilon = 10 > 3n - 6$, we can conclude that K_5 is non-planar.

One consequence of Lemma 14.15 is that $O(\varepsilon)$ algorithms on planar graphs are also $O(n)$ algorithms. For example, a DFS or BFS in a planar graph takes $O(n)$ steps. This will be useful in algorithms for testing planarity, or for drawing or coloring a planar graph, or constructing dual graphs.

Given a graph G, we can subdivide any edges of G without affecting the planarity of G. Therefore, we will assume that G has no vertices of degree two. We will also take G to be 2-edge-connected, so that there are no vertices of degree one. Let G be a simple planar graph, and let n_i be the number of vertices of degree i, for $i = 3, 4, \ldots$. Counting the edges of G gives

$$3n_3 + 4n_4 + 5n_5 + \ldots = 2\varepsilon \leq 6n - 12$$

Counting the vertices of G gives

$$n_3 + n_4 + n_5 + \ldots = n$$

Multiply the second equation by 6 and subtract the two equations to obtain:

Lemma 14.16. *A simple planar graph with no vertices of degree one or two satisfies*

$$3n_3 + 2n_4 + n_5 \geq 12 + n_7 + 2n_8 + 3n_9 + \ldots$$

Corollary 14.17. *A simple planar graph with no vertices of degree one or two has a vertex of degree three, four, or five.*

Proof. The values n_i are non-negative integers. □

The Corollary 14.17 results in a technique for reducing a planar triangulation on n vertices to one on $n-1$ vertices that is fundamental for understanding the structure of planar graphs, and for handling them algorithmically.

Algorithm 14.8.1: REDUCEGRAPH(G^p)

comment: Given a simple planar triangulation G on $n > 4$ vertices with rotation system p. Construct a planar triangulation G' on $n-1$ vertices.

if there is a vertex u with $\text{DEG}(u) = 3$
then let $p(u) = (ux, uy, uz)$
$G' \leftarrow G - u$
return (G')

if there is a vertex u with $\text{DEG}(u) = 4$
then let $p(u) = (uw, ux, uy, uz)$
if $w \not\longrightarrow y$
then $G' \leftarrow G - u + wy$
wy replaces wu in $p(w)$ and yw replaces yu in $p(y)$
else $G' \leftarrow G - u + xz$
xz replaces xu in $p(x)$ and zx replaces zu in $p(z)$
return (G')

comment: otherwise, $\text{DEG}(u) = 5$
let $p(u) = (uv, uw, ux, uy, uz)$
if $v \not\longrightarrow x$ **and** $v \not\longrightarrow y$
then $G' \leftarrow G - u + vx + vy$
vx, vy replace vu in $p(v)$
xv replaces xu in $p(x)$ and yv replaces yu in $p(y)$
else if $v \longrightarrow x$
then $G' \leftarrow G - u + wy + wz$
wy, wz replace yu in $p(w)$
yw replaces yu in $p(y)$ and zw replaces zu in $p(z)$
else if $v \longrightarrow y$
then $G' \leftarrow G - u + zw + zx$
zw, zx replace zu in $p(z)$
wz replaces wu in $p(w)$ and zw replaces zu in $p(z)$
return (G')

Let G_n be a triangulation on $n \geq 4$ vertices. If $n = 4$, then K_4 is the only possibility. So we assume that $n > 4$. We know that G_n always has a vertex of degree three, four, or five.

Theorem 14.18. *Given a simple planar triangulation G_n on $n > 4$ vertices, Algorithm* REDUCEGRAPH*() constructs a simple planar triangulation G_{n-1} on $n - 1$ vertices.*

Proof. We know that G_n has a vertex of degree three, four, or five. If there is a vertex u of degree three, let $p(u) = (ux, uy, uz)$, as illustrated in Figure 14.9. Because G_n is a triangulation, we know that xy, yz, and zx are edges of G_n. Consequently $G_n - u$ is also a triangulation.

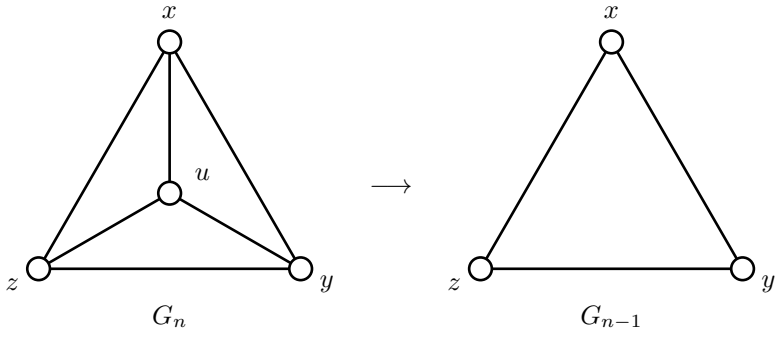


FIGURE 14.9
DEG$(u) = 3$

If there is a vertex u of degree four, let $p(u) = (uw, ux, uy, uz)$, as illustrated in Figure 14.10. Because G_n is a triangulation, we know that wx, xy, yz, and zw are edges of G_n. When u is deleted, one face of $G_n - u$ is a quadrilateral. If $w \not\longrightarrow y$, we can add the edge wy to get a planar triangulation $G_n - u + wy$. If $w \longrightarrow y$, then edge wy is exterior to the quadrilateral face. Consequently $x \not\longrightarrow z$, so that we can add the edge xz to get a planar triangulation $G_n - u + xz$. In either case we get a planar triangulation G_{n-1}.

If there is a vertex u of degree five, let $p(u) = (uv, uw, ux, uy, uz)$, as illustrated in Figure 14.11. Because G_n is a triangulation, we know that vw, wx, xy, yz, and zv are edges of G_n. When u is deleted, one face of $G_n - u$ is a pentagon. If $v \not\longrightarrow x$ and $v \not\longrightarrow y$, we can add the edges vx and vy to get a planar triangulation $G_n - u + vx + vy$. Otherwise, if $v \longrightarrow x$, then edge vx is exterior to the pentagonal face. Consequently, $w \not\longrightarrow y$ and $w \not\longrightarrow z$, so that we can add the edges wy and wz to get a planar triangulation $G_n - u + wy + wz$. Otherwise, if $v \longrightarrow y$, then edge vy is exterior to the pentagonal face, and we can proceed as above to get a triangulation $G_n - u + zw + zx$. The proof is complete. □

Note that Algorithm REDUCEGRAPH() requires the degrees of the vertices. These can be computed in $O(n)$ time and stored in an array. Once the degrees are known, the reduction from G_n to G_{n-1} takes constant time. Usually, this algorithm will be applied recursively to reduce a planar triangulation G_n to G_4, which must

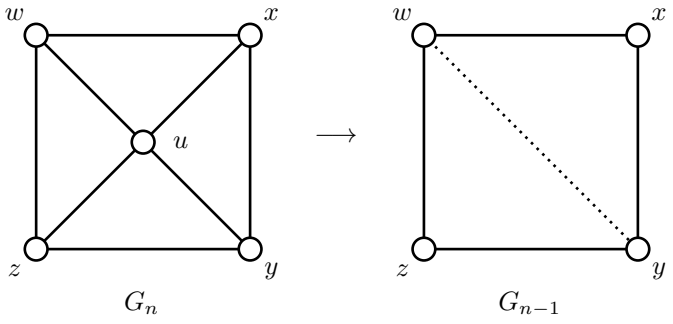


FIGURE 14.10
$\text{DEG}(u) = 4$

equal K_4. If the algorithm is being used to find a planar embedding of G_n, or to color it, the graph will then be rebuilt in reverse order.

14.9 The sphere

The plane can be mapped onto the surface of the sphere by a simple transformation called *stereographic projection*. Place a sphere on the surface of the plane, so that it is tangent at the south pole. See Figure 14.12. Now from any point P on the plane, construct a straight-line L to the north pole of the sphere. L will intersect the surface of the sphere in some point. Call it P'. This transformation maps any point P in the plane to a point P' on the surface of the sphere. The mapping is clearly invertible and continuous. The only point on the sphere to which no point of the plane is mapped is the north pole.

If G^ψ is an embedding of a graph on the plane, then stereographic projection will map the points of G^ψ onto an embedding of G on the surface of the sphere. Conversely, if we are given an embedding of G on the surface of the sphere, we can roll the sphere to ensure that the north pole is not a point of the embedding. Then use stereographic projection to map the surface of the sphere onto the plane, thereby obtaining an embedding of G on the plane. Consequently, embedding graphs on the plane is equivalent to embedding them on the sphere.

When a graph is embedded on the surface of the sphere, the faces are the regions that remain when the sphere is cut along the edges of G. There is no outer face. Every face is bounded. However, the face that contains the north pole will become the outer face when the embedding is projected onto the plane. By rolling the sphere to place any desired face at the top, we can make any face the outer face. We state this as a lemma.

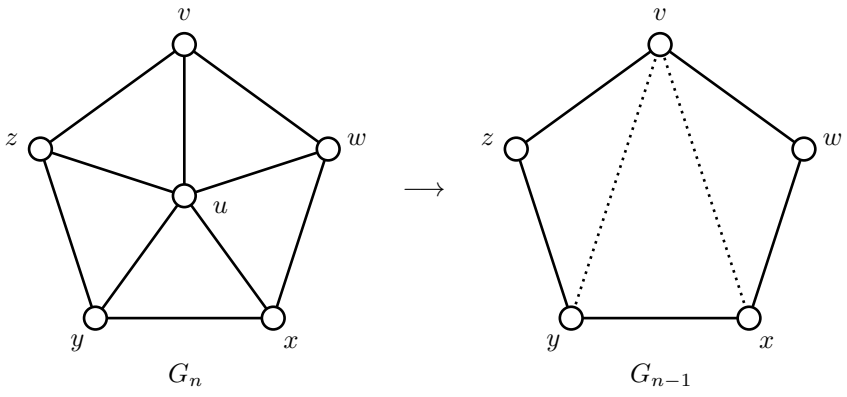


FIGURE 14.11
$\text{DEG}(\boldsymbol{u}) = \mathbf{5}$

Lemma 14.19. *A planar graph can be drawn so that any facial cycle, any edge, or any vertex appears on the boundary of the outer face.*

Exercises

14.9.1 Given a sphere of radius one tangent to the (x, y)-plane at the origin, with north pole at $(0, 0, 1)$. Show that a point (u, v) in the plane is projected onto $(u/D, v/D, (u^2 + v^2)/D)$ on the sphere, where $D = u^2 + v^2 + 1$.

14.9.2 Show that a point (x, y, z) on the sphere is projected onto $(\frac{x}{1-z}, \frac{y}{1-z})$ on the plane.

14.9.3 Denote a point (u, v) in the plane by the complex number $w = u + iv$. Express the formulas of question 1 in terms of w.

14.9.4 Sometimes stereographic projection is done with a sphere of radius one with its *center* at the origin. How do the formulas of questions 1 and 2 change? Is there a qualitative change? What are the images of the southern and northern hemispheres under the projection?

14.10 Whitney's theorem

The plane and sphere are oriented surfaces. Consider a graph G embedded on the plane as G^p, where p gives the clockwise orientation of the edges incident on each vertex. We are assuming that the plane is viewed from above. If we now view the plane from below, the clockwise orientation of each $p(u)$ will appear counter clock-

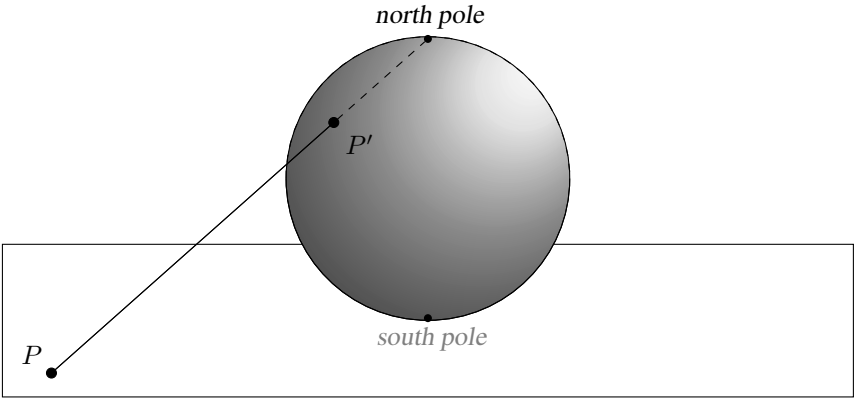


FIGURE 14.12
Mapping the plane to the sphere

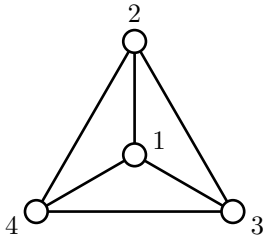
wise. If an embedding G^p is projected onto the sphere, then each $p(u)$ will appear clockwise if viewed from inside the sphere, but counter clockwise if viewed from outside the sphere. Given a rotation system p, we write $\overline{p}$ for the rotation system obtained by reversing the cyclic order of each $p(u)$. So $\overline{p}(u) = p(u)^{-1}$. The embeddings G^p and $G^{\overline{p}}$ are usually considered equivalent.

DEFINITION 14.17: Let G^{p_1} and G^{p_2} be two plane embeddings of a graph G, with rotation systems p_1 and p_2, respectively. G^{p_1} and G^{p_2} are *isomorphic embeddings* if there is an automorphism of G which transforms p_1 into p_2. G^{p_1} and G^{p_2} are *equivalent embeddings* if there is an automorphism of G which transforms p_1 into either p_2 or $\overline{p}_2$.

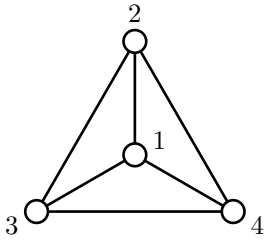
An automorphism of G will permute the vertices of G, and consequently alter the edges in the cyclic permutations of a rotation system. If θ is an automorphism of G, then $p_1(u) = (e_1, e_2, \ldots, e_k)$ is transformed by θ into $\theta(p_1(u)) = (\theta(e_1), \theta(e_2), \ldots, \theta(e_k))$. If this equals $p_2(\theta(u))$, for all vertices u, then G^{p_1} and G^{p_2} are isomorphic. Isomorphic rotation systems are equivalent.

Two isomorphic rotation systems for K_4 are illustrated in Figure 14.13. Here $p_2 = \overline{p_1}$. It is easy to see that if we take $\theta = (3, 4)$, then $\theta(p_1(u)) = p_2(\theta(u))$, for all $u = 1, 2, 3, 4$.

A triangulation G on 7 points is shown in Figure 14.14. Two rotation systems for it, p_1 and p_2, are also given below. Here we also have $p_2 = \overline{p_1}$. However, there is no automorphism of G that will transform p_1 into p_2. This can be verified using the degrees of the vertices. Vertex 2 is the only vertex of degree six. Hence any automorphism θ must fix 2. So $\theta(p_1(2))$ must equal $p_2(2)$. The only vertices of degree four are vertices 1 and 5. Therefore either $\theta(1) = 1$ or $\theta(1) = 5$. If $\theta(1) = 1$, then from $p_2(2)$ we see that $\theta(3) = 6$. This is impossible as vertices 3 and 6 have different



$$p_1(1) = (12, 13, 14)$$
$$p_1(2) = (21, 24, 23)$$
$$p_1(3) = (31, 32, 34)$$
$$p_1(4) = (41, 43, 42)$$



$$p_2(1) = (12, 14, 13)$$
$$p_2(2) = (21, 23, 24)$$
$$p_2(3) = (31, 34, 32)$$
$$p_2(4) = (41, 42, 43)$$

FIGURE 14.13
Two isomorphic rotation systems for $\boldsymbol{K_4}$

degrees. If $\theta(1) = 5$, then from $p_2(2)$ we have $\theta(3) = 4$, which is also impossible, as vertices 3 and 4 have different degrees.

So G^{p_1} and G^{p_2} are equivalent embeddings that are non-isomorphic. This can only occur if there is an automorphism of G mapping p_1 to $\overline{p}_2$, but no automorphism mapping p_1 to p_2; that is, G^{p_2} is obtained by "flipping" G^{p_1} upside down. The example of Figure 14.13 shows that this is not possible with K_4, but is possible with the triangulation of Figure 14.14.

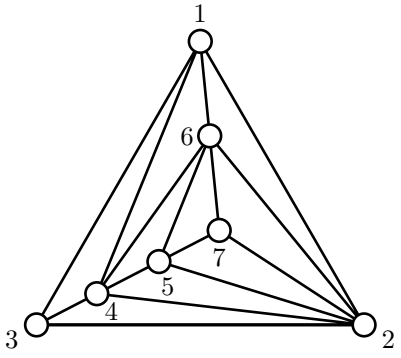
DEFINITION 14.18: A planar map G^p is *orientable* if $G^p \ncong G^{\overline{p}}$. Otherwise G^p is *non-orientable*.

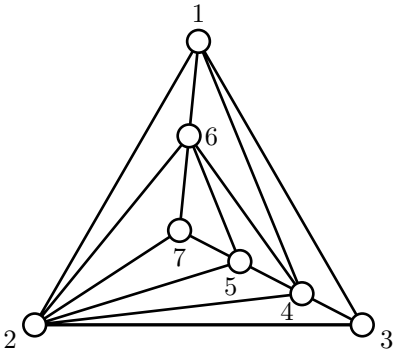
So the embedding of K_4 is non-orientable, but the embeddings of Figure 14.14 are orientable. An example of a 2-connected graph with two inequivalent planar embeddings is shown in Figure 14.6. Whitney's theorem states that if G is 3-connected, this cannot happen. Let C be a cycle in a connected graph G. C is a *separating cycle* if $G - V(C)$ is disconnected.

Theorem 14.20. (Whitney's theorem) *Let G be a 3-connected planar graph. Let p be any planar rotation system for G. The facial cycles of G^p are the induced, non-separating cycles of G.*

Proof. Let C be an induced, non-separating cycle of G. In any planar embedding of G, C corresponds to a Jordan curve in the plane. There can be vertices of G in the interior or exterior of the Jordan curve, but not both, because C is non-separating. Without loss of generality, assume that any vertices of $G - C$ are in the exterior of the Jordan curve. It follows that the interior of the Jordan curve is a face, so that C is a facial cycle of G.

Conversely, let C be a facial cycle. Without loss of generality, we can assume that C corresponds to a Jordan curve whose interior is a face. If G contains an edge uv which is a chord of C, then u and v divide C into two paths $C[u, v]$ and $C[v, u]$.





$p_1(1) = (12, 16, 14, 13)$
$p_1(2) = (21, 23, 24, 25, 27, 26)$
$p_1(3) = (31, 34, 32)$
$p_1(4) = (41, 46, 45, 42, 43)$
$p_1(5) = (52, 54, 56, 57)$
$p_1(6) = (61, 62, 67, 65, 64)$
$p_1(7) = (72, 75, 76)$

$p_2(1) = (12, 13, 14, 16)$
$p_2(2) = (21, 26, 27, 25, 24, 23)$
$p_2(3) = (31, 32, 34)$
$p_2(4) = (41, 43, 42, 45, 46)$
$p_2(5) = (52, 57, 56, 54)$
$p_2(6) = (61, 64, 65, 67, 62)$
$p_2(7) = (72, 76, 75)$

FIGURE 14.14
Two equivalent, non-isomorphic rotation systems

Because uv must be embedded exterior to C, there can be no path from an interior vertex of $C[u, v]$ to an interior vertex of $C[v, u]$. Therefore $G - \{u, v\}$ is disconnected, a contradiction, as G is 3-connected. Consequently, C is an induced cycle. Let x be any vertex of $G - C$. If x is the only vertex of $G - C$, then $G - C = x$, so that C is a non-separating cycle, and we are done. Otherwise let y be another vertex of $G - C$. Because G is 3-connected, G contains at least three internally disjoint xy-paths. At most two of these paths can intersect C. See Figure 14.15. Therefore $G - C$ contains an xy-path, for all x, y. It follows that C is a non-separating cycle of G. □

One consequence of Whitney's theorem is that, if G is a 3-connected planar graph, an embedding can be found purely from the cycles of G. If we can identify an induced, non-separating cycle C, we can then assign an orientation to C. Each edge of C will be contained in another induced, non-separating cycle, so that the orientation of adjacent cycles will thereby also be determined. Continuing in this way, a complete rotation system for G can be constructed. This rotation system can then be used to construct a dual graph.

DEFINITION 14.19: Let G be a 3-connected planar graph. The *abstract dual* is G^*, a dual graph determined by the induced non-separating cycles of G.

The abstract dual is the same as G^{p*}, where p is a rotation system determined by

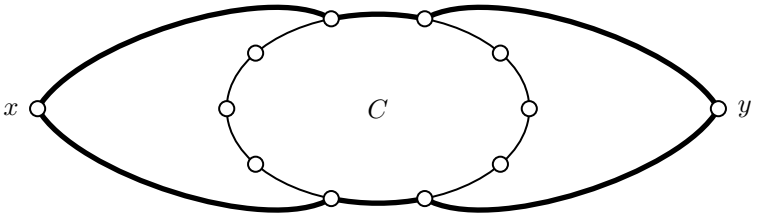


FIGURE 14.15
Whitney's theorem

the induced, non-separating cycles, but it can be constructed without reference to a rotation system.

Up to equivalence of embeddings, a 3-connected planar graph has just one rotation system, so that the abstract dual is uniquely defined. Orientable 3-connected planar graphs have just two rotation systems (which are inverses of each other). Non-orientable 3-connected planar graphs have just one rotation system.

Whitney's theorem does not provide an algorithm for determining whether a graph is planar, as the characterization of planarity in terms of induced, non-separating cycles does not lead to an efficient algorithm. There are too many cycles in a graph to effectively find them all and determine whether they are induced, non-separating cycles.

14.11 Medial digraphs

Let G be a loopless graph with a planar rotation system p. Note that G is allowed to be a multigraph. There are several possible ways of constructing a digraph representing G^p. One of them is the following.

DEFINITION 14.20: The *medial digraph* $M(G^p)$ of a planar map G^p is obtained from G^p by subdividing every edge uv of G with a vertex x_{uv}. The edges of $M(G^p)$ consist of all arcs of the form (u, x_{uv}), (x_{uv}, u), and (x_{uv}, x_{uw}), where uv and uw are consecutive edges in $p(u)$, with uw following uv.

An example of a medial digraph is shown in Figure 14.16. Because G is planar, $M(G^p)$ will also be planar.

Lemma 14.21. *Let G^{p_1} and G^{p_2} be planar embeddings of a graph G. Then G^{p_1} and G^{p_2} are isomorphic embeddings if and only if $M(G^{p_1})$ and $M(G^{p_2})$ are isomorphic as digraphs.*

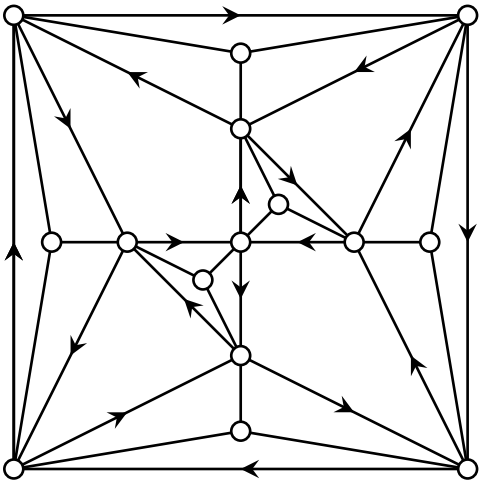


FIGURE 14.16
A medial digraph

Proof. Suppose that G^{p_1} and G^{p_2} are isomorphic embeddings. Let θ be an automorphism of G that maps p_1 to p_2. Given a vertex u, let θ map u to v. Then θ maps $p_1(u)$ to $p_2(v)$. Consequently θ can be extended to map the directed edges $\{x_{uw}\}$ of $M(G^{p_1})$ based on $p_1(u)$ to the directed edges $\{x_{vw}\}$ of $M(G^{p_2})$ based on $p_2(v)$. Thus $M(G^{p_1}) \cong M(G^{p_2})$.

Conversely, if $M(G^{p_1})$ and $M(G^{p_2})$ are isomorphic, let θ be an isomorphism from $M(G^{p_1})$ to $M(G^{p_2})$. Then θ must map $V(G)$ to $V(G)$, and the set of subdividing vertices $\{x_{uw}\}$ to itself. Suppose that θ maps $u \in V(G)$ to v. The directed edges ensure that θ maps $p_1(u)$ to $p_2(v)$. Hence G^{p_1} and G^{p_2} are isomorphic. □

Note that if p_2 is taken to be $\overline{p}_1$, then G^{p_1} and G^{p_2} will be isomorphic when $M(G^{p_1})$ and $M(G^{\overline{p}_1})$ are isomorphic. Here $M(G^{\overline{p}_1})$ is the digraph converse of $M(G^{p_1})$, obtained by reversing all directed edges. There are two kinds of automorphisms of G^p — those that preserve the orientation, and those that reverse the orientation.

DEFINITION 14.21: Let G^p be a planar embedding. An *orientation-preserving automorphism* of G^p is an automorphism of G that maps G^p to G^p. An *orientation-reversing automorphism* of G^p is an automorphism of G that maps maps G^p to $G^{\overline{p}}$.

The automorphism group of G^p is denoted $\text{AUT}(G^p)$:

DEFINITION 14.22: The *orientation-preserving automorphism group* of G^p is $\text{AUT}(G^p)$, those automorphisms of G induced by $\text{AUT}(M(G^p))$. The *full automorphism group* of G^p is $\text{AUT}^+(G^p)$ consisting of $\text{AUT}(G^p)$, plus those automorphisms that map G^p to $G^{\overline{p}}$.

If G^p is orientable, then $\text{AUT}(G^p) = \text{AUT}^+(G^p)$, because every automorphism preserves the orientation. But if G^p is non-orientable, then $\text{AUT}(G^p)$ is a subgroup of index two in $\text{AUT}^+(G^p)$. If θ is any orientation-reversing automorphism, then the coset $\text{AUT}(G^p)\theta$ contains all the orientation-reversing automorphisms.

Theorem 14.22. *Let G^p be a planar embedding of a 3-connected graph G. Then* $\text{AUT}^+(G^p) = \text{AUT}(G)$.

Proof. By Whitney's theorem, the facial cycles of G^p are the induced non-separating cycles. Because every automorphism $\theta \in \text{AUT}(G)$ must map an induced non-separating cycle to another, we see that $\theta \in \text{AUT}^+(G^p)$. □

A consequence of this theorem is that $|\text{AUT}(G)| = 2|\text{AUT}(G^p)|$ when G is a 3-connected non-orientable planar graph. When G is not 3-connected, this does not hold. An example is $G = K_{2,6}$. A non-orientable planar embedding is shown in Figure 14.17, where $|\text{AUT}(G^p)| = 12$, but $|\text{AUT}(G)| = 1440$.

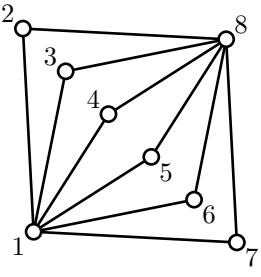


FIGURE 14.17
A planar embedding of $\boldsymbol{K_{2,6}}$

Exercises

14.11.1 Find all triangulations on $4, 5, 6$, and 7 vertices.

14.11.2 Determine the group of orientation-preserving mappings of an embedding of K_4 on the plane. Compare it with $\text{AUT}(K_4)$.

14.11.3 Let G be K_5, with an edge removed. Determine the group of orientation-preserving mappings of an embedding of G on the plane. Compare it with $\text{AUT}(G)$.

14.11.4 Verify that the embedding G^p of Figure 14.17 is non-orientable, that $|\text{AUT}(G^p)| = 12$, and that $|\text{AUT}(G)| = 1440$.

14.11.5 Determine whether the platonic solids are orientable.

14.11.6 Prove that a planar embedding G^p is orientable if and only if G^{p*} is orientable.

14.11.7 Determine which of the triangulations on $5, 6$, and 7 vertices are orientable.

14.11.8 Determine the graph G for which $M(G)$ is shown in Figure 14.16.

14.12 The 4-color problem

Given a geographic map drawn in the plane, how many colors are needed such that the map can be colored so that any two regions sharing a common border have different colors? In 1852, it was conjectured by Francis Guthrie that four colors suffice. This simple problem turned out to be very difficult to solve. Several flawed "proofs" were presented. Much of the development of graph theory originated in attempts to solve this conjecture. See AIGNER [2] for a development of graph theory based on the 4-color problem. In 1976, Appel and Haken announced a proof of the conjecture. Their proof was based on the results of a computer program that had to be guaranteed bug-free. A second computer proof by ALLAIRE [3] appeared in 1977. Each of these approaches relied on showing that any planar graph contains one of a number of configurations, and that for each configuration, a proper coloring of a smaller (reduced) graph can be extended to a proper coloring of the initial graph. The computer programs generated all irreducible configurations, and colored them. In the Appel-Haken proof, there were approximately 1800 irreducible configurations. The uncertainty was whether all irreducible configurations had indeed been correctly generated. In 1995, ROBERTSON, SANDERS, SEYMOUR, and THOMAS [149] presented another proof, also based on a computer program, but considerably simpler than the original, requiring only 633 irreducible configurations.

In this section, we present the main ideas of Kempe's 1879 "proof" of the 4-color theorem.

Given a geographic map drawn in the plane, one can construct a dual graph, by placing a vertex in the interior of each region, and joining vertices by edges if they correspond to adjacent regions. Coloring the regions of the map is then equivalent to coloring the vertices of the dual, so that adjacent vertices are of different colors. Consequently, we shall be concerned with coloring the vertices of a planar graph.

Theorem 14.23. (4-Color theorem) *Every planar graph can be properly colored with four colors.*

If G is any simple planar graph, then it is always possible to extend G to a simple triangulation, by adding diagonal edges in non-triangular faces. Therefore, if we can prove that all simple planar triangulations are 4-colorable, the result will be true for all planar graphs. Hence we assume that we are given a planar triangulation G_n on n vertices. We attempt to prove the 4-color theorem (Theorem 14.23) by induction on n.

The colors can be chosen as the numbers $\{1, 2, 3, 4\}$. Given a coloring of G, then the subgraph induced by any two colors i and j is bipartite. We denote it by K^{ij}.

DEFINITION 14.23: Given any 4-coloring of a planar graph G, each connected component of K^{ij} is called a *Kempe* component. The component containing a vertex x is denoted $K^{ij}(x)$. A path in K^{ij} between vertices u and v is called a *Kempe chain*.

Notice that if we interchange the colors i and j in any Kempe component, we obtain another coloring of G.

Now let G_n be a simple triangulation on n vertices. If $n = 4$, then $G_n = K_4$. It is clear that Theorem 14.23 is true in this case. Assume that $n > 4$. By Corollary 14.17, we know that G_n has a vertex of degree three, four, or five. Let u be such a vertex. Using Algorithm 14.8.1, we reduce G_n to a simple planar triangulation G_{n-1} by deleting u and adding up to two diagonals in the resulting face. We assume as an induction hypothesis, that G_{n-1} has a 4-coloring. There are three cases.

Case 1. $\text{DEG}(u) = 3$.

Let the three adjacent vertices to u be (x, y, z). They all have different colors. Therefore there is a fourth color available for v, giving a coloring of G_n.

Case 2. $\text{DEG}(u) = 4$.

Let the four vertices adjacent to u in G_n be (w, x, y, z), with a diagonal wy in G_{n-1}. It is clear that w, x, and y have different colors. If x and z have the same color, then a fourth color is available for u. Otherwise, let w, x, y, z be colored $1, 2, 3, 4$, respectively. There may be a Kempe chain from x to z. If there is no Kempe chain, interchange colors in the Kempe component $K^{24}(x)$, so that x and z now both have color 4. If there is a Kempe chain from x to z, there can be no Kempe chain from w to y, for it would have to intersect the xz-Kempe chain. Interchange colors in $K^{13}(w)$, so that w and z now both have color 3. In each case there is a fourth color available for u.

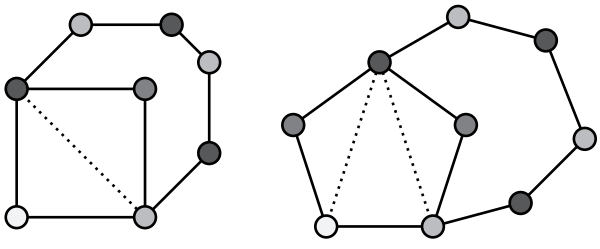


FIGURE 14.18
Kempe chains

Case 3. $\text{DEG}(u) = 5$.

Let the five vertices adjacent to u in G_n be (v, w, x, y, z), with diagonals vx and vy in G_{n-1}. It is clear that v, x, and y have different colors. Because we have a 4-coloring of G_{n-1}, the pentagon (v, w, x, y, z) is colored in either

3 or 4 colors. If it is colored in three colors, there is a fourth color available for u. If it is colored in four colors, then without loss of generality, we can take these colors to be $(1, 2, 3, 4, 2)$, respectively. If $K^{13}(v)$ contains no vx-Kempe chain, then we can interchange colors in $K^{13}(v)$, so that v and x are now both colored 3. Color 1 is then available for u. If $K^{14}(v)$ contains no vy-Kempe chain, then we can interchange colors in $K^{14}(v)$, so that v and y are now both colored 4. Color 1 is again available for u. Otherwise there is a Kempe chain P_{vx} connecting v to x and a Kempe chain P_{vy} connecting v to y. It follows that $K^{24}(w)$ contains no wy-Kempe chain, as it would have to intersect P_{vx} in $K^{13}(v)$. Similarly, $K^{23}(z)$ contains no vz-Kempe chain, as it would have to intersect P_{vy} in $K^{14}(v)$. If P_{vx} and P_{vy} intersect only in vertex v, then we can interchange colors in both $K^{24}(w)$ and $K^{23}(z)$, thereby giving w color 4 and z color 3. This makes color 2 available for u. The difficulty is that P_{vx} and P_{vy} can intersect in several vertices. Interchanging colors in $K^{24}(w)$ can affect the other Kempe chains, as shown in Figure 14.19, where the pentagon (v, w, x, y, z) is drawn as the outer face.

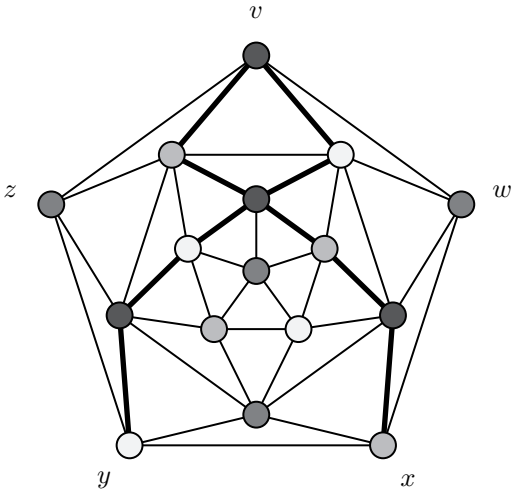


FIGURE 14.19
Intersecting Kempe chains

Although this attempted proof of Theorem 14.23 fails at this point, we can use these same ideas to prove the following.

Theorem 14.24. (5-Color theorem) *Any planar graph can be colored in five colors.*

Proof. See Exercise 14.12.1. □

Appel and Haken's proof of the 4-color theorem is based on the important concept of *reducibility*. Given a graph G, a *reducible configuration* H is a subgraph of

G with the property that H can be reduced to a smaller subgraph H', such that a 4-coloring of H' can be extended to all of H and G. If every planar graph contained a reducible configuration, then every planar graph could be 4-colored. Appel and Haken's proof was essentially a computer program to construct all irreducible configurations, and to show that they could be 4-colored. The difficulty with this approach is being certain that the computer program is correctly constructing all irreducible configurations. The reader is referred to SAATY and KAINEN [153] or WOODALL and WILSON [194] for more information on reducibility.

14.13 Nowhere-zero flows

Consider a plane map G^p with no cut-edge, whose faces have been colored using colors $\{1, 2, 3, 4\}$. Each edge uv lies on the boundary of two distinct faces. We consider the plane to be an oriented surface, viewed from "above", so that in traversing uv, there is a well-defined face to the *right* of uv, and a well-defined face to the *left*. Assign an orientation and flow to the edges of G to create a digraph, also denoted G, as follows. Consider an edge uv, traversed from u to v, with face F_1 to the right of uv, and face F_2 to the left. Let c_1 be the color of F_1, and c_2 the color of F_2. We have $c_1 \neq c_2$, because the coloring is a proper coloring. If $c_1 > c_2$, then uv is oriented from u to v, with flow $\varphi(uv) = c_1 - c_2$. Otherwise uv is oriented from v to u, and the flow $\varphi(uv) = c_2 - c_1$. Then $\varphi(uv) \neq 0$, so that $\varphi(uv) \in \{1, 2, 3\}$. The function φ is called a *nowhere-zero flow* on G^p.

Using this φ, let vertex u have rotation $p(u) = (v_1, v_2, \ldots, v_m)$, and let the face with edges uv_i and uv_{i+1} on its boundary be F_i, with color c_i, where v_{m+1} is the same as v_1. If edge uv_i is directed out of u, then its flow is $\varphi(uv_i) = c_i - c_{i-1}$. If it is directed into u, its flow is $\varphi(uv_i) = c_{i-1} - c_i$. Taking the sum of flows on all edges directed out of u, minus the sum of flows into u, we obtain

$$(c_m - c_1) + (c_1 - c_2) + \ldots + (c_{m-1} - c_m) = 0$$

Thus, φ satisfies the conservation condition of a network flow.

We have used a coloring of the dual of the planar embedding to find this flow. However, the idea of a k-flow can be applied to any 2-edge-connected graph. It does not have to be planar.

DEFINITION 14.24: Let G be a 2-edge-connected graph. A *nowhere-zero k-flow* on G is an orientation of $E(G)$ and a function $\varphi : E(G) \to \{1, 2, \ldots, k-1\}$ which satisfies the flow conservation condition at each vertex.

An example of a nowhere-zero 4-flow is shown in Figure 14.20. Here the color number is shown inside each face, and the arrows indicate the orientation of each edge according to the face colors.

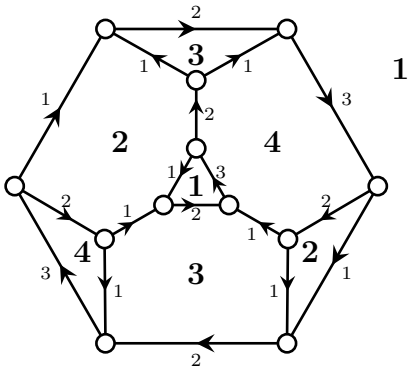


FIGURE 14.20
A nowhere-zero 4-flow.

So a face-coloring of a planar map G^p determines a nowhere-zero 4-flow in G. The converse is also true, as stated in the following theorem. A proof can be found in Bondy and Murty [24].

Theorem 14.25. *A 2-edge-connected planar graph G has a nowhere-zero 4-flow if and only if it has a planar dual that can be colored in 4 colors.*

So the existence of a nowhere-zero 4-flow in a planar graph is equivalent to the 4-color theorem.

Consider a nowhere-zero 2-flow in 2-edge-connected graph G. This corresponds to an orientation of the edges of G such that each edge uv has $\varphi(uv) = 1$. The conservation condition then says that the number of edges directed out of every vertex u equals the number of edges directed into u. So G must be Eulerian. Conversely, an Euler tour in G determines a nowhere-zero 2-flow.

Given a nowhere-zero k-flow in a graph G, the orientation of any edge uv can be reversed, with the flow $\varphi(uv)$ changed to $-\varphi(uv)$. This can be done for any set of edges. The resulting flow still satisfies the conservation condition at each vertex, but some values are now negative. Thus, it is convenient to relax the definition of nowhere-zero flow as follows.

DEFINITION 14.25: Let G be a 2-edge-connected graph. A nowhere-zero k-*flow* or k-*circulation* on G is an orientation of $E(G)$ and a function $\varphi : E(G) \to \{\pm 1, \pm 2, \ldots, \pm(k-1)\}$ which satisfies the flow conservation condition at each vertex.

Tutte [179, 180, 181], has made three fundamental conjectures on k-flows.

Conjecture. *Every 2-edge-connected graph has a 5-flow.*

Seymour [159] has proved that every 2-edge-connected graph has a 6-flow. An algorithm to find a nowhere-zero 6-flow appears in Younger [195].

Conjecture. *Every 2-edge-connected graph that does not have the Petersen graph as a minor has a 4-flow.*

Conjecture. *Every 4-edge-connected graph has a 3-flow.*

Exercises

14.13.1 Prove Theorem 14.24, the 5-color theorem.

14.13.2 Let G be a planar triangulation with a separating 3-cycle (u, v, w). Let H and K be the two connected subgraphs of G that intersect in exactly (u, v, w), such that $G = H \cup K$. Show how to construct a 4-coloring of G from 4-colorings of H and K.

14.13.3 Let G be a planar triangulation with a separating 4-cycle (u, v, w, x). Let H and K be the two connected subgraphs of G that intersect in exactly (u, v, w, x), such that $G = H \cup K$. Show how to construct a 4-coloring of G from 4-colorings of the triangulations $H + uw$ and $K + uw$. *Hint*: u, v, and w can be assumed to have the same colors in H and K. If x is colored differently in H and K, look for an xv-Kempe chain, try interchanging colors in $K^{ij}(x)$, or try coloring $H + vx$ and $K + vx$.

14.13.4 All lakes are blue. Usually all bodies of water are colored blue on a map. Construct a planar graph with two non-adjacent vertices that must be blue, such that the graph cannot be colored in four colors subject to this requirement.

14.13.5 Construct a nowhere-zero 4-flow for the graph of the dodecahedron.

14.13.6 Construct a nowhere-zero 5-flow for the Petersen graph.

14.13.7 Let G have a k-flow for some k. Given a cycle C in G, show how to change $\varphi(uv)$ for all edges uv on C to get a new flow.

14.14 Straight-line drawings

Every simple planar graph can be drawn in the plane with no edges crossing, so that each edge is a straight line. Read's Algorithm is a linear-time algorithm for doing this. It is based on the triangulation reduction.

Suppose that G_n is a triangulation on n vertices that has been reduced to a triangulation G_{n-1} on $n - 1$ vertices, by deleting a vertex u as in Algorithm 14.8.1, and adding up to two edges e and e'. Suppose that a straight-line embedding of G_{n-1} has already been computed. If $\text{DEG}(u) = 3$, let x, y, z be the adjacent vertices. We can place u inside the triangle (x, y, z) to obtain a straight-line embedding of G_n. If $\text{DEG}(u) = 4$, the edge e is a diagonal of a quadrilateral in G_{n-1}. We can place u on the line representing e to obtain a straight-line embedding of G_n.

Suppose now that $\text{DEG}(u) = 5$. The edges $e = vx$ and $e' = vy$ are diagonals of a pentagon in G_{n-1}. This pentagon may have several possible polygonal shapes, which are illustrated in Figure 14.21. The triangle (v, x, y) is completely contained inside the pentagon. Inside (v, x, y), there is a "visible" region, shown shaded gray. The visible region can be calculated, by extending the lines of the adjacent triangles with sides vx and vy, and intersecting the half-planes with the triangle (v, x, y). Vertex u can then be placed inside the visible region to obtain a straight-line embedding of G_n. Thus in each case, a straight-line embedding of G_{n-1} can be extended to G_n.

This gives:

Theorem 14.26. (Fáry's theorem) *Every planar graph has a straight-line embedding.*

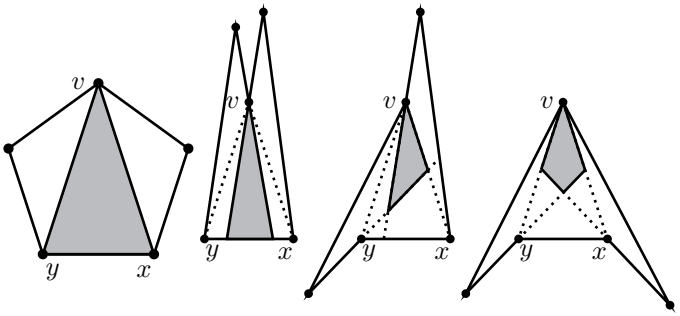


FIGURE 14.21
The "visible" region

Read's algorithm begins by triangulating G if $\varepsilon < 3n - 6$. It then deletes a sequence of vertices $u_1, u_2, \ldots, u_{n-4}$ to reduce G to K_4. It next assigns a planar coordinatization to the vertices of K_4, and then restores the deleted vertices in reverse order. For each vertex u_i deleted, it is necessary to store u_i and its degree, so that it can later be correctly restored to the graph. Finally, the triangulating edges are removed. The result is a straight-line embedding of G.

Algorithm 14.14.1: READSALGORITHM(G^p)

comment: { Given a simple planar graph G on $n \geq 4$ vertices with rotation system p, construct a straight-line drawing of G in the plane.

triangulate G without creating multiple edges or loops
mark all triangulating edges as "virtual" edges
$i \leftarrow 1$
while $n > 4$
do {
  $G \leftarrow$ REDUCEGRAPH(G)
  $u_i \leftarrow$ the vertex that was deleted
  $i \leftarrow i + 1$
  $n \leftarrow n - 1$
comment: G is now K_4

assign pre-chosen coordinates to the vertices of K_4
for $i = n - 4$ **downto** 1
do {
  calculate the visible region for u_i
  restore u_i to G
remove all virtual edges from G

It is easy to see that Read's algorithm is $O(n)$. It takes $O(n)$ steps to compute the degrees of G, and to triangulate G. It takes $O(n)$ steps to reduce G to K_4, and then $O(n)$ steps to rebuild G. Read's algorithm can be modified by initially choosing any facial cycle F of G, and assigning coordinates to the vertices of F so that they form a regular convex polygon. The reduction to K_4 is then modified so that vertices of the outer facial cycle F are never deleted. The result is a planar embedding with the given facial cycle as the outer face.

The embeddings produced by Read's algorithm are usually not convex embeddings. Tutte has shown how to produce a straight-line embedding of a graph such that all faces are convex regions, by solving linear equations. Consider any face of G, with facial cycle $(v_1, v_2, \ldots, v_k)$. We begin by assigning coordinates to $v_1, v_2, \ldots, v_k$ such that they form a convex polygon in the plane. This will be the outer face of a planar embedding of G.

Tutte then looks for a coordinatization of the remaining vertices with the special property: the coordinates of v_i, where $i > k$, are the *average* of the coordinates of all adjacent vertices. A coordinatization with this property is called a *barycentric coordinatization*. We can express it in terms of matrices as follows.

Let A be the adjacency matrix of G such that the first k rows and columns correspond to vertices $v_1, v_2, \ldots, v_k$. Let D be the $n \times n$ diagonal matrix such that entry D_{ii} equals 1, if $i \leq k$. If $i > k$, entry D_{ii} equals DEG(v_i). Let X be the vector of x-coordinates of the vertices, and let Y be the vector of y-coordinates. Construct a matrix B from A by replacing the first k rows with zeroes. Then the first k entries of BX are zero. But if $i > k$, the i^{th} entry is the sum of the x-coordinates of vertices adjacent to v_i. Let X_k denote the vector whose first n entries are the x-coordinates of

$v_1, \ldots, v_k$, and whose remaining entries are zero. Y_k is similarly defined. Then the barycentric condition can be written as

$$DX = X_k + BX, \qquad DY = Y_k + BY.$$

These equations can be written as $(D - B)X = X_k$ and $(D - B)Y = Y_k$. Consider the matrix $D - B$.

Lemma 14.27. *The matrix $D - B$ is invertible.*

Proof. Consider the determinant $\det(D - B)$. The first k rows of $D - B$ look like an identity matrix. Expanding the determinant along the first k rows gives $\det(D - B)$ = $\det(K)$ where K is the matrix formed by the last $n - k$ rows and columns. K looks very much like a Kirchhoff matrix, except that the degrees are not quite right. In fact, if we construct a graph G' from G by identifying $v_1, v_2, \ldots, v_k$ into a single vertex v_0, and deleting the loops created, then K is formed from the Kirchhoff matrix $K(G')$ by deleting the row and column corresponding to v_0. It follows that $\det(K) = \pm\tau(G')$, by the matrix-tree theorem. Because G' is a connected graph, $\det(K) \neq 0$, so that $D - B$ is invertible. □

It follows that the barycentric equations have a unique solution, for any assignment of the coordinates X_k and Y_k. Tutte has shown that if G is a 3-connected planar graph, this solution has remarkable properties: if we begin with a convex polygon for the outer face, the solution is a planar coordinatization with straight lines, such that no edges cross. All faces, except the outer face, are convex regions. No three vertices of any facial cycle are collinear.

It is fairly easy to see that a planar barycentric coordinatization of a 3-connected graph must have convex faces. For consider a non-convex face, as in Figure 14.22. Vertex v is a corner at which a polygonal face is non-convex. Clearly vertex v is not on the outer face. Let the two adjacent vertices of the polygon be u and w. Because G is 3-connected, v has at least another adjacent vertex. All other vertices adjacent to v must be in the angle between the lines vu and vw as indicated in the diagram, because G is 3-connected, and there are no crossing edges. But then all vertices adjacent to v are to the right of the dotted line, which is impossible in a barycentric coordinatization.

14.15 Coordinate averaging

Suppose that a straight-line drawing of a planar graph G has been obtained, with coordinates (x_i, y_i) for vertex i, where $i = 1, \ldots, n$. The faces may be convex regions, or not. There is a simple algorithm which frequently produces a significant improvement in the drawing, called *coordinate averaging*.

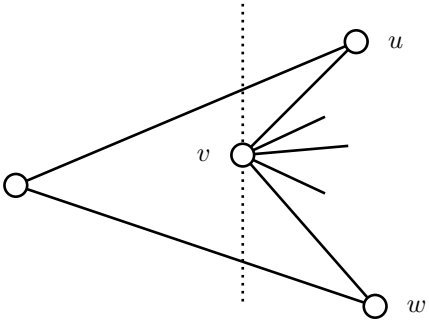


FIGURE 14.22
A non-convex face

Algorithm 14.15.1: COORDINATEAVERAGING(G^p)

comment: Given a straight-line drawing of a simple plane map G^p, on $n \geq 4$ vertices, with coordinates (x_i, y_i) for vertex i, perform coordinate averaging.

let $F_1, F_2, \ldots, F_f$ denote the facial cycles of G^p
let F_f be the facial cycle of the outer face
for $j \leftarrow 1$ **to** $f - 1$
do use algorithm FACIALCYCLE(F_j) to sum (x_i, y_i), for all $i \in F_j$
$(u_j, v_j) \leftarrow$ average of the coordinates of the vertices of F_j
comment: (u_j, v_j) are now coordinates of the dual G^{p*}, except for F_f

for $i \leftarrow 1$ **to** n **do if** $i \notin F_f$
then sum (u_j, v_j) for all faces F_j containing vertex i
$(x_i, y_i) \leftarrow$ average of the coordinates of the F_j containing i

The algorithm COORDINATEAVERAGING() uses FACIALCYCLE() to walk around each facial cycle F_j, summing the coordinates of the vertices on F_j, so as to compute their average. The result is coordinates (u_j, v_j) inside the polygon representing the face F_j. This is done for every face except the outer face. When this algorithm is used, it is usually convenient to first place the vertices of the outer face on a regular convex polygon. The remaining vertices will have coordinates (x_i, y_i) inside the outer polygon. Essentially, coordinates for the vertices of the dual are being constructed, except for the outer face F_f. Then a second loop re-calculates (x_i, y_i), by performing the same operation, but in the dual. This constitutes one application of COORDINATEAVERAGING(). Clearly this takes $O(n)$ steps. This algorithm can be iterated several times, to produce an improved drawing.

An example appears in Figure 14.23. The graph on the left in the diagram could be the result of using Read's algorithm to find a drawing of the line graph of the cube, placing the vertices of the outer face on a regular quadrilateral. The graph on the right

is after six applications of coordinate averaging. A constant number of applications of coordinate averaging still results in a $O(n)$ algorithm.

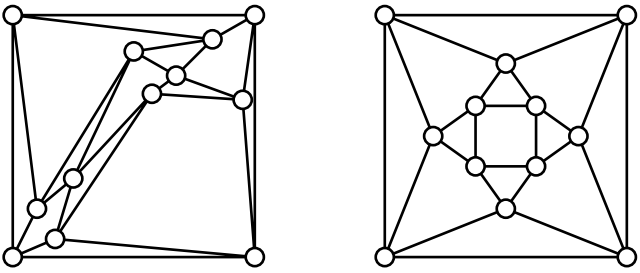


FIGURE 14.23
Coordinate averaging

Coordinate averaging can be expressed in terms of barycentric coordinates of a related graph. Given a plane map G^p, with dual G^{p*}, let $\{v_1, \ldots v_n\}$ denote the vertices of G, and let $\{F_1, \ldots, F_f\}$ denote the vertices of G^{p*}. Construct a bipartite graph whose vertices are $\{v_1, \ldots v_n\} \cup \{F_1, \ldots, F_f\}$ in which vertex v_i is incident with F_j if and only if v_i is on the boundary of F_j in G^p. This graph is called the *vertex-face-incidence graph* or equivalently, the *primal-dual graph* of G^p. Denote it by G^F. Apply Tutte's drawing algorithm to find a barycentric drawing of G^F. Coordinates for vertices on the outer face F_f are chosen, and then coordinates for the remaining vertices are found such that the coordinates of each vertex are the average of the coordinates of the adjacent vertices. This is a form of coordinate averaging.

Solving the linear equations of Lemma 14.27 to find the coordinates takes up to $O((n + f)^3)$ steps, using Gaussian elimination. But coordinate averaging can find an approximate solution in $O(n)$ steps. Coordinate averaging in G, which produces barycentric coordinates of G^F, tends to produce nicer drawings than barycentric coordinates for G. In fact, it often hilights symmetries of G.

14.16 Kuratowski's theorem

In this section we will prove Kuratowski's theorem. The proof presented is based on a proof by KLOTZ [101]. It uses induction on $\varepsilon(G)$.

If G is a disconnected graph, then G is planar if and only if each connected component of G is planar. Therefore we assume that G is connected. If G is a separable graph that is planar, let H be a block of G containing a cut-vertex v. H is also planar, because G is. We can delete $H - v$ from G, and find a planar embedding of the result. We then choose a planar embedding of H with v on the outer face, and embed H into a face of G having v on its boundary. This gives:

Lemma 14.28. *A separable graph is planar if and only if all its blocks are planar.*

So there is no loss in generality in starting with a 2-connected graph G.

Theorem 14.29. (Kuratowski's theorem) *A graph G is planar if and only if it contains no subgraph $TK_{3,3}$ or TK_5.*

Proof. It is clear that if G is planar, then it contains no subgraph $TK_{3,3}$ or TK_5. To prove the converse, we show that if G is non-planar, then it must contain $TK_{3,3}$ or TK_5. We assume that G is a simple, 2-connected graph with ε edges. To start the induction, notice that if $\varepsilon \leq 6$, the result is true, as all graphs with $\varepsilon \leq 6$ are planar. Suppose that the theorem is true for all graphs with at most $\varepsilon - 1$ edges. Let G be non-planar, and let $ab \in E(G)$ be any edge of G. Let $G' = G - ab$. If G' is non-planar, then by the induction hypothesis, it contains a $TK_{3,3}$ or TK_5, which is also a subgraph of G. Therefore we assume that G' is planar. Let $\kappa(a, b)$ denote the number of internally disjoint ab-paths in G'. Because G is 2-connected, we know that $\kappa(a, b) \geq 1$.

Case 1. $\kappa(a, b) = 1$.

G' has a cut-vertex u contained in every ab-path. Add the edges au and bu to G', if they are not already present, to get a graph H, with cut-vertex u. Let H_a and H_b be the blocks of H containing a and b, respectively. If one of H_a or H_b is non-planar, say H_a, then by the induction hypothesis, it contains a $TK_{3,3}$ or TK_5. This subgraph must use the edge au, as G' is planar. Replace the edge au by a path consisting of the edge ab plus a bu-path in H_b. The result is a $TK_{3,3}$ or TK_5 in G. If H_a and H_b are both planar, choose planar embeddings of them with edges au and bu on the outer face. Glue them together at vertex u, remove the edges au and bu that were added, and restore ab to obtain a planar embedding of G, a contradiction.

Case 2. $\kappa(a, b) = 2$.

Let P_1, and P_2 be two internally disjoint ab-paths in G'. Because $\kappa(a, b) = 2$, there is a vertex $u \in P_1$ and $v \in P_2$ such that all ab-paths contain at least one of $\{u, v\}$, and $G' - \{u, v\}$ is disconnected. If K_a denotes the connected component of $G' - \{u, v\}$ containing a, let G'_a be the subgraph of G' induced by $K_a \cup \{u, v\}$. Let K_b denote the remaining connected components of $G' - \{u, v\}$, and let G'_b be the subgraph of G' induced by $K_b \cup \{u, v\}$, except that uv, if it is an edge of G', is not included (because it is already in G'_a). Now add a vertex x to G'_a, adjacent to u, v, and a to obtain a graph H_a. Similarly, add y to G'_b adjacent to u, v, and b to obtain a graph H_b. Suppose first that H_a and H_b are both planar. As vertex x has degree three in H_a, there are three faces incident on x. Embed H_a in the plane so that the face with edges ux and xv on the boundary is the outer face. Embed H_b so that edges uy and yv are on the boundary of the outer face. Now glue H_a and H_b together at vertices u and v, delete vertices x and y, and add the edge ab within the face created, to obtain a planar embedding of G. Because G is non-planar,

we conclude that at least one of H_a and H_b must be non-planar. Suppose that H_a is non-planar. It must contain a subgraph TK_5 or $TK_{3,3}$. If the TK_5 or $TK_{3,3}$ does not contain x, then it is also contained in G, and we are done. Otherwise the TK_5 or $TK_{3,3}$ contains x. Now H_b is 2-connected (because G is), so that it contains internally disjoint paths P_{bu} and P_{bv} connecting b to u and v, respectively. These paths, plus the edge ab, can be used to replace the edges ux, vx, and ax in H_a to obtain a TK_5 or $TK_{3,3}$ in G.

Case 3. $\kappa(a, b) \geq 3$.

Let P_1, P_2, and P_3 be three internally disjoint ab-paths in G'. Consider a planar embedding of G'. Each pair of paths $P_1 \cup P_2$, $P_1 \cup P_3$, and $P_2 \cup P_3$ creates a cycle, which embeds as a Jordan curve in the plane. Without loss of generality, assume that the path P_2 is contained in the interior of the cycle $P_1 \cup P_3$, as in Figure 14.24. The edge ab could be placed either in the interior of $P_1 \cup P_2$ or $P_2 \cup P_3$, or else in the exterior of $P_1 \cup P_3$. As G is non-planar, each of these regions must contain a path from an interior vertex of P_i to an interior vertex of P_j. Let P_{12} be a path from u_1 on P_1 to u_2 on P_2. Let P_{13} be a path from v_1 on P_1 to u_3 on P_3. Let P_{23} be a path from v_2 on P_2 to v_3 on P_3. If $u_1 \neq v_1$, contract the edges of P_1 between them. Do the same for u_2, v_2 on P_2 and u_3, v_3 on P_3. Adding the edge ab to the resultant graph then results in a TK_5 minor. By Theorem 14.5, G contains either a TK_5 or $TK_{3,3}$.

□

We can also state Kuratowski's theorem in terms of minors.

Theorem 14.30. (Wagner's theorem) *A graph G is planar if and only if it does not have $K_{3,3}$ or K_5 as a minor.*

Proof. It is clear that if G is planar, then it does not have $K_{3,3}$ or K_5 as a minor. Conversely, if G does not have $K_{3,3}$ or K_5 as a minor, then it cannot have a subgraph $TK_{3,3}$ or TK_5. By Kuratowski's theorem, G is planar. □

The graphs K_5 and $K_{3,3}$ are called *Kuratowski graphs* for the plane. They are also said to be *obstructions* to planarity, because a graph containing a TK_5 or $TK_{3,3}$ is non-planar. In terms of minors, a graph having K_5 or $K_{3,3}$ as a minor is non-planar.

Exercises

14.16.1 Find a $TK_{3,3}$ or TK_5 in the Petersen graph.

14.16.2 Find a $TK_{3,3}$ or TK_5 in the graph of Figure 14.25.

14.16.3 Show that if G is a non-planar 3-connected graph, then either $G = K_5$, or else G contains a $TK_{3,3}$.

14.16.4 Let G be a graph with a separating set $\{u, v\}$. Let H' be a connected component of $G - \{u, v\}$, and let H denote the graph induced by $V(H') \cup$

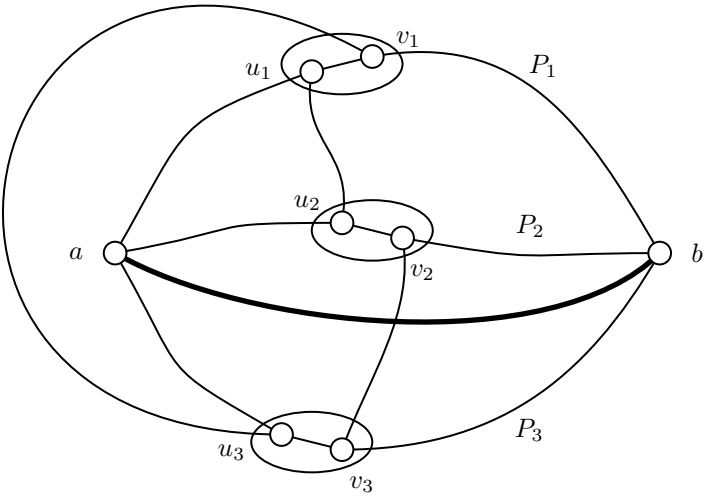


FIGURE 14.24
A $\boldsymbol{K_5}$ minor

$\{u, v\}$. Let K be the graph induced by $V(G) - V(H')$, so that $G = H \cup K$, and H and K intersect only in $\{u, v\}$. Let $H^+ = H + uv$ and $K^+ = K + uv$. Show that G is planar if and only if H^+ and K^+ are planar.

14.16.5 Let G be a 3-connected graph with at least five vertices. Show that G contains an edge xy such that $G \cdot xy$ is 3-connected. *Hint*: If $G \cdot xy$ is not 3-connected, choose xy so that the subgraph K of the preceding exercise is as large as possible, and find a contradiction. This was proved by THOMASSEN [169].

14.16.6 Suppose that a coordinatization in the plane of an arbitrary 3-connected graph G on n vertices is obtained by solving the barycentric equations $(D - B)X = X_k$ and $(D - B)Y = Y_k$. Describe an $O(n^3)$ algorithm which determines whether G is planar, and constructs a rotation system, using the coordinates X and Y.

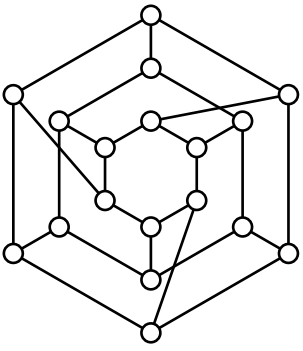


FIGURE 14.25
Find a $TK_{3,3}$

14.17 The Hopcroft-Tarjan algorithm

A number of different algorithms for planarity-testing have been developed. The first linear-time algorithm was the Hopcroft-Tarjan planarity algorithm. Given a 2-connected graph G on n vertices, it determines whether G is planar in $O(n)$ time. If G is found to be planar, it can be extended to construct a rotation system, too. If G is found to be non-planar, it can also be modified to construct a TK_5 or $TK_{3,3}$ subgraph, although this is somewhat more difficult. We present a simplified version of the Hopcroft-Tarjan algorithm here.

There is no loss in generality in starting with a 2-connected graph G. Suppose first that G is hamiltonian, and that we are given a hamilton cycle C, and number the vertices of C consecutively as $1, 2, \ldots, n$. The remaining edges of G are chords of C. For each vertex u, order the adjacent vertices $(v_1, v_2, \ldots, v_k)$ so that $v_1 < v_2 < \ldots < v_k$. We then start at vertex $u = n$, and follow the cycle C back to vertex 1. As we proceed, we will place each chord uv either inside C or outside C. When we have returned to vertex 1, we will have constructed a planar embedding of G. We draw the cycle C as in Figure 14.26, with the path from 1 to n starting near the top, and moving down the page.

Consider the example of Figure 14.26. The algorithm stores two linked lists of chords, one for the inside of C, and one for the outside of C. We denote these as L_i and L_o, respectively. Each linked list defines a sequence of chords $[u_1v_1, u_2v_2, u_3v_3, \ldots]$ as they are added to the embedding. The inside of C appears in the diagram to the left of the path from 1 to n. The outside of C appears to the right of the path. In the example, the algorithm begins at vertex $n = 7$ with adjacent vertices $(1, 3, 4, 6)$. It first places the "chord" $(7, 1)$, which really completes the cycle, on the inside linked list. It then places chords $(7, 3)$ and $(7, 4)$ also on L_i. The inside linked list is now $[(7, 1), (7, 3), (7, 4)]$. The chord $(7, 4)$ is called the *leading* chord

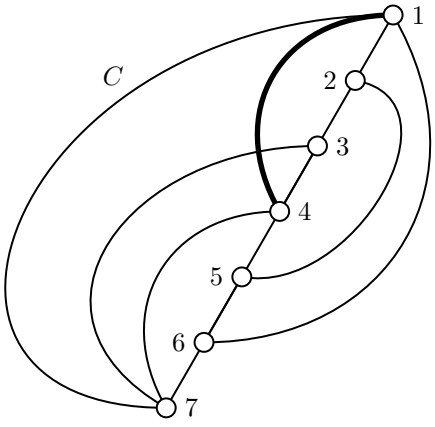


FIGURE 14.26
The Hopcroft-Tarjan algorithm

in the linked list. The next chord to be inserted is to be placed after it. To determine whether the next chord (u, v) fits on the inside, it need only compare its endpoint v with the upper endpoint of the current leading chord of L_i. After placing the chords incident on vertex 7, the algorithm moves to vertex 6, where it sees the chord $(6, 1)$. This will not fit on the inside (because $1 < 4$), but is easily placed on the outside linked list. It then moves to vertex 5, and places $(5, 2)$ also on L_o. The outside linked list is then $[(6, 1), (5, 2)]$, where $(5, 2)$ is the leading chord. It then moves to vertex 4, where it sees the chord $(4, 1)$. When the algorithm moves up the cycle to vertex 4, the leading chord of L_i is moved past $(7, 4)$ to $(7, 3)$, because the chord $(4, 1)$ is above $(7, 4)$. It then determines that $(4, 1)$ will not fit on the inside (because $1 < 3$, where $(7, 3)$ is the leading chord of L_i); and that $(4, 1)$ will not fit on the outside (because $1 < 2$, where $(5, 2)$ is the leading chord of L_o). Therefore G is non-planar. In fact the cycle C, together with the three chords $(7, 3), (5, 2)$, and $(4, 1)$ form a subgraph $TK_{3,3}$, which we know to be non-planar.

Algorithm 14.17.1: BASICHT(G, C)

```
comment: { Given a 2-connected graph G with a hamilton cycle
         { C = (1, 2, ..., n). Determine whether G is planar.
for u ← n downto 1
do { suppose that u is adjacent to (v_1, v_2, ..., v_k)
   { for j ← 1 to k
   { do { if v_j ≥ u − 1 go to L1
        { comment: uv_j is a chord with v_j above u
        { if uv_j fits inside C, place it in L_i
        {   else if uv_j fits outside C, place it in L_o
        {        else { m = SWITCHSIDES(u, v_j)
        {             { if m = 0 return (NonPlanar)
        {             { if m = 1
        {               then { comment: uv_j now fits inside C
        {                    { place uv_j inside C
        {               else if m = −1
        {               then { comment: uv_j now fits outside C
        {                    { place uv_j outside C
        { L1 :
```

We must still describe what SWITCHSIDES(u, v) does. Its purpose is to determine whether some chords of L_i and L_o can be interchanged to allow (u, v) to be embedded. Its description will follow.

This simplified version of the Hopcroft-Tarjan algorithm contains the main features of the complete algorithm, but is much easier to understand. The algorithm stores two linked lists, L_i and L_o. Each list has a *leading chord* – the chord after which the next chord is to be placed when inserted in the list as the sequence of chords is extended. Initially the leading chord will be the last chord in the linked list, but this will change as the algorithm progresses. It is convenient to initialize both linked lists with dummy chords, so that each list has at least one chord. Each chord stored in a linked list will be represented by a pair, denoted (*LowerPt*, *UpperPt*), where *UpperPt* is the endpoint with the smaller value – it is above the *LowerPt* in the diagram. As the chords are placed in the linked lists, a linear order is thereby defined on the chords in each list. Given a chord (u, v), where $u > v$, the next chord after (u, v) is the chord *nested immediately inside* (u, v), if there is such a chord. If there is no such chord, then the next chord after (u, v) is the *first chord below* (u, v), if there is one; that is, the first chord whose *UpperPt* $\geq u$. To determine whether a chord (u, v) fits either inside or outside C, we need only compare v with the leading chord's *UpperPt*. It fits if $v \geq$ *UpperPt*. As u moves up the cycle, we must adjust the pointer to the leading chord on both sides to ensure that the leading chord always satisfies *UpperPt* $< u$.

So we know how to store the chords, and how to determine whether a chord (u, v) fits in either side. If it fits, we insert it in the appropriate linked list and continue with the next chord. What if (u, v) will not fit in either side?

14.17.1 Bundles

Two chords (u_1, v_1) and (u_2, v_2) are said to be in *conflict* if either $u_1 > u_2 > v_1 > v_2$ or $u_2 > u_1 > v_2 > v_1$. Conflicting chords cannot both be placed on the same side of C. We define a *conflict graph* K whose vertices are the set of all chords currently in the linked lists. Two chords are adjacent in K if they are in conflict. A set of chords corresponding to a connected component of the conflict graph is called a *bundle* (originally called a "block" in HT's paper; however, the term "block" has another graph theoretical meaning). The conflict graph must always be bipartite, as two chords in the same linked list must never be in conflict. Therefore each bundle B is also bipartite – the bipartition of a bundle consists of the chords inside the cycle, denoted B_i, and the chords outside the cycle, denoted B_o. A typical bundle is shown as the shaded area in Figure 14.27. The two shaded zones represent B_i and B_o for one bundle. Any chord inside a shaded zone conflicts with all chords in the opposite shaded zone. Notice that if the conflict graph is not bipartite, that it is impossible to assign the chords to the inside and outside of C. Consequently, G must be non-planar in such a case.

Now the conflict graph K is changing dynamically as the algorithm progresses. Therefore the bundles are also changing. However, they change in a simple way. Each chord is initially a bundle by itself, until it is found to conflict with another chord. In a 1-chord bundle, one of B_i and B_o will be empty. The algorithm will store the bundles in a stack.

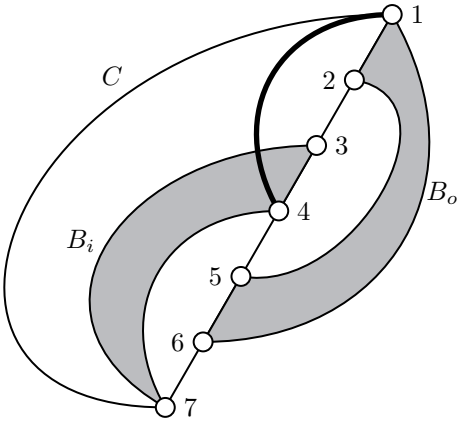


FIGURE 14.27
Bundles of chords

Consider the situation where no conflicts have yet been discovered, so that all

chords so far have been placed on L_i. The algorithm is visiting vertex u on C, attempting to place a chord (u, v). The leading inside chord satisfies *LowerPt* $\geq u >$ *UpperPt*. It belongs to a 1-chord bundle, the *current* bundle. The algorithm first attempts to nest (u, v) inside the leading inside chord. If it fits, then (u, v) becomes the new leading inside chord, and a new 1-chord bundle is created containing only uv, which becomes the current bundle – *the bundles are nested*. The innermost bundle is always the *current bundle* and is stored at the top of the bundle stack.

If (u, v) will not fit on the inside, it conflicts with the leading inside chord. (u, v) may conflict with several chords of the inside linked list. They will be *consecutive chords* of L_i, preceding the leading chord. These chords initially belong to different bundles, but will become merged into the current bundle B, thereby forming B_i, when (u, v) is placed in the outside list. B_o will consist of (u, v). If B denotes the current bundle, we will call B_i and B_o the current inside and outside bundles, although they are part of the same bundle.

At this point we can say that *the current inside and outside bundles consist of one or more consecutive chords in the two linked lists*. This will be true at each point of the algorithm.

So in order to represent a bundle B, we require two pointers into each of L_i and L_o, being the first and last chords of L_i that belong to B_i, and the first and last chords of L_o that belong to B_o. We could switch the chords of B_i to L_o and the chords of B_o to L_i by reassigning four pointers. Because the bundles are nested, we store them as a stack.

When a chord (u, v) is placed in one of the linked lists, and it does not conflict with the leading chord on either side, a new current bundle B containing (u, v) is created, nested inside the previous current bundle. The current bundle is the one at the top of the stack. As vertex u moves up the cycle, we eventually have $u \leq$ *UpperPt* for the uppermost chord in the current bundle. The current bundle B is then no longer relevant, as no more chords can conflict with it. Therefore B is removed from the stack and deleted. The next bundle B' on the stack becomes the current bundle. When this happens, the chords belonging to B_i or B_o often occur in L_i or L_o, respectively, as a consecutive subsequence of chords contained within B'_i or B'_o. When B is deleted, the effect is to merge the chords of B into B'. Then B' becomes the new current bundle. Its chords are again consecutive chords of L_i and L_o.

14.17.2 Switching bundles

We have a chord (u, v) that will not fit on the inside or outside of C. What do we do? There are several possible ways in which this can arise. Two of them are illustrated in Figure 14.28. The current bundle is shown shaded in gray.

In the left diagram of Figure 14.28, (u, v) conflicts with one or more chords in B_i and in B_o. If we form a subgraph consisting of the cycle C, the edge (u, v), and a conflicting chord from each of B_i and B_o, we have a subgraph $TK_{3,3}$, which we know is non-planar. In the right diagram of Figure 14.28, (u, v) conflicts with one or more chords in B_o. It does not conflict with any chords in B_i, but it does conflict with the leading chord (a, b) of L_i, which is in a previous inside bundle. If we interchange

the chords of B_i and B_o, there will be room to place (u, v) in L_o. This can be done in constant time, because the chords in each bundle are consecutive chords in the linked list. We only need to change a constant number of pointers to transfer a sequence of chords from L_i to L_o, and vice versa.

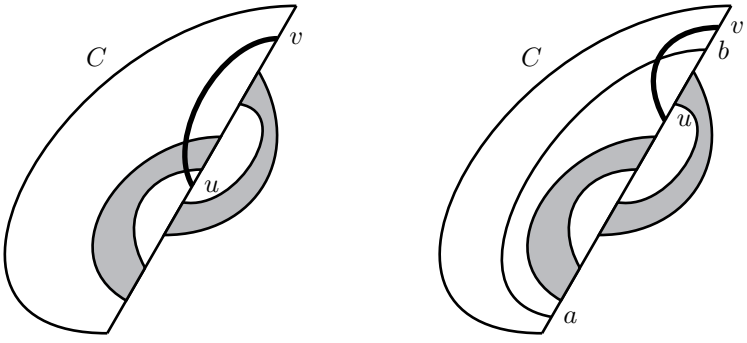


FIGURE 14.28
Switching bundles

A third situation also exists which is nearly the mirror image of the right diagram of Figure 14.28, in which (u, v) conflicts with one or more chords in B_i, but does not conflict with any chords in B_o, and does conflict with the leading chord (a, b) of L_o, which is in a previous outside bundle. It can be handled in a similar way.

Suppose that a situation similar to the right diagram of Figure 14.28 exists. B is the current bundle, and chord (u, v) conflicts with a chord of B_o, but not with B_i. The leading chord of L_o is in B_o. The leading chord of L_i is (a, b), which is not in B_i. Because every chord is contained in some bundle, (a, b) is in a previous bundle on the stack. The bundles are nested, so that B is nested within a bundle B', which may in turn be nested within a bundle B'', etc. Without loss of generality, suppose that there are at least three bundles on the stack, which begins B, B', B'', and that (a, b) is in B''.

Now B is nested within B', which is nested within B''. Because the leading chord of L_i is in B'', it follows that (u, v) does not conflict with any chord of B'_i, and that (u, v) does conflict with some chord of B'_o. So (u, v) conflicts with a chord of both B_o and B'_o. If (u, v) can be embedded, B_o and B'_o must be merged into one bundle, and they must both be on the same side of C. They will be in the same part of the bipartition of K. Therefore the algorithm merges B and B'. Call the result B. It then discovers that interchanging the chords of B_i and B_o allows (u, v) to be placed on L_o. Because (u, v) conflicts with (a, b) in L_i, the bundles B and B'' are then also merged.

The properties of bundles are summarized as follows:

1. The bundles correspond to connected components of the conflict graph.

2. The bundles are nested inside each other, and consequently stored on a stack.
3. The current bundle is the one on the top of the stack.
4. The chords of each B_i and B_o form a contiguous subsequence of L_i and L_o, respectively.

The description of SWITCHSIDES(u, v) Algorithm 14.17.2 can now be given. It is called when chord (u, v) conflicts with both the leading chord of L_i and the leading chord of L_o. The algorithm needs to know whether the leading chord of L_i is within B_i, and whether the leading chord of L_o is within B_o. This can be done by comparing the endpoints of the leading chords with the first and last chords of B_i and B_o. One of the leading chords is always in the current bundle. The other must be in a previous bundle, or the conflict graph will be non-bipartite, and G will be non-planar.

When B_i and B_o are interchanged, the leading chords in L_i and L_o also change. Suppose that the leading chord of L_o is within B_o, but that the leading chord of L_i is in a previous bundle, as in Figure 14.28. The new leading chord of L_o can easily be found by taking the first chord of L_o following B_o. The new leading chord of L_i is the former leading chord of L_o.

This procedure merges the bundles on the stack until either a non-bipartite conflict graph is found, in which case it returns zero, or until it becomes possible to place (u, v) in one of L_i or L_o. Notice that swapping the chords of B_i and B_o takes a constant number of steps, and that merging the current bundle with the previous bundle on the stack also takes a constant number of steps. The total number of bundles is at most the number of edges of G, so that the total number of steps required by SWITCHSIDES(u, v) is $O(n)$, summed over all iterations.

If SWITCHSIDES(u, v) returns either 1 or -1, then it is possible to place (u, v) inside or outside C. If SWITCHSIDES(u, v) returns 0, then the current bundle contains chords of L_i and L_o that conflict with (u, v), so that the conflict graph is not bipartite. In this case G is non-planar. However, it is not particularly easy to find a TK_5 or $TK_{3,3}$ in G in this situation. If the algorithm succeeds in placing all the chords, then a rotation system can be found from the order in which the chords occur in L_i and L_o.

So the three main components of this simplified version of the Hopcroft-Tarjan algorithm are:

1. Construct the linked lists L_i and L_o in the right order.
2. Construct the connected components of the conflict graph as a stack of bundles.
3. Keep the bundles up to date. Each time that a chord is added to one of L_i or L_o, the current bundle must be updated.

These steps can all be performed in linear time.

Algorithm 14.17.2: SWITCHSIDES(u, v)

comment: (u, v) conflicts with the leading chord of L_i and L_o

while (**true**)
do
- let B denote the current bundle
- let B' denote the previous bundle on the stack
- **if** the leading chord of L_o is within B_o
  - **then**
    - **if** the leading chord of L_i is within B_i
      - **then return** (0) "non-planar"
      - **comment:** otherwise the leading chord of L_i is in a bundle previous to B
    - **if** the leading chord of L_i is within B'_i
      - **then**
        - interchange the chords of B_i and B_o
        - merge B and B'
        - find the new leading chords of L_i and L_o
        - **if** (u, v) does not conflict with L_o
          - **then return** (-1) "(u, v) now fits in L_o"
      - **else** merge B and B'
  - **else**
    - **comment:** the leading chord of L_i is within B_i the leading chord of L_o is in a bundle previous to B
    - **if** the leading chord of L_o is within B'_o
      - **then**
        - interchange the chords of B_i and B_o
        - merge B and B'
        - find the new leading chords of L_i and L_o
        - **if** (u, v) does not conflict with L_i
          - **then return** (1) "(u, v) now fits in L_i"
      - **else** merge B and B'

14.17.3 The general Hopcroft-Tarjan algorithm

Up to now, we have assumed that we are given a hamilton cycle C in a 2-connected graph G which we are testing for planarity. If we are not given a starting hamilton cycle, the algorithm is the recursive extension of the hamilton cycle case. We give a brief sketch only. The first step is to perform a depth-first search in G starting from vertex 1, to assign a DF-numbering to the vertices, and to calculate the low-points of all vertices. The DFS will construct a DF-spanning tree T rooted at 1. Number the vertices of G according to their DF-numbers. It can happen that the tree T is a hamilton path, in which case we have the proceeding situation exactly – the vertices are numbered consecutively along T, and as the DFS returns from recursion, it visits the vertices in the order $n, n-1, \ldots, 1$. If we sort the chords incident on each vertex in order of increasing DF-number of the other endpoint, the algorithm is identical.

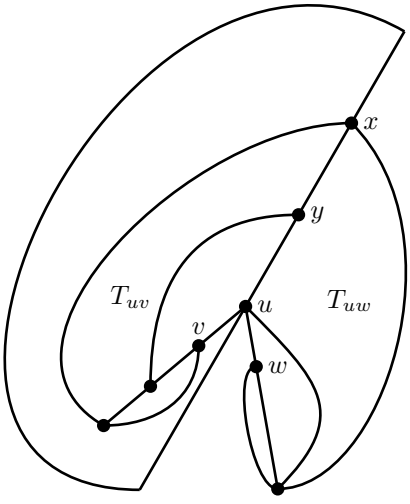


FIGURE 14.29
Subtrees T_{uv} and T_{uw}

If the tree T is not a hamilton path, consider the situation where the DFS is visiting vertex u, and a recursive call DFS(v) is made, due to an edge uv. The recursive call DFS(v) constructs a subtree T_{uv}. Refer to Figure 14.29. When T_{uv} is constructed, *LowPt*$[v]$ is calculated. Because G is 2-connected, this is a vertex somewhere above u in T. The entire subtree T_{uv} behaves very much like a single chord $(u,$ *LowPt*$[v])$. Therefore the vertices adjacent to u must be sorted according to *LowPt*$[\cdot]$ values, just as in the simplified algorithm (where v is used rather than *LowPt*$[v]$).

There are two kinds of possible subtree, and these are illustrated as T_{uv} and T_{uw} in Figure 14.29. Both T_{uv} and T_{uw} have the same *LowPt*, equal to x. Notice that T_{uv} has a frond with endpoint y between u and x, but that T_{uw} has no such frond. We will call a subtree like T_{uw} a type I subtree, and a subtree like T_{uv} a type II subtree. It is easy to distinguish type I and II subtrees. The DFS can compute the second low-point as well as the low-point. If the second low-point is between u and *LowPt*$[v]$, then the subtree T_{uv} is of type II; otherwise, it is of type I. Now a type I subtree T_{uw} behaves exactly like a chord $(u,$ *LowPt*$[w])$. There can be any number of them, and they can be nested inside each other in any order. However, they cannot be nested inside a type II subtree.

Therefore we must embed all type I subtrees at u with *LowPt* $= x$ before any type II subtrees at u with *LowPt* $= x$. This can be accomplished by ordering the adjacent vertices at u so that fronds ux and type I subtrees T_{uw} with *LowPt*$[w] = x$

precede type II subtrees T_{uv} with $\mathit{LowPt}[v] = x$. Hopcroft and Tarjan assign a weight to all edges incident on u. A frond ux has weight $2x$. A type I subtree T_{uv} has weight $2 \cdot \mathit{LowPt}[v]$. A type II subtree T_{uv} has weight $2 \cdot \mathit{LowPt}[v] + 1$. The adjacent vertices are then sorted by weight, which can be done by a bucket sort in linear time, because the weights are all in the range $1, \ldots, 2n + 1$.

The general algorithm takes place in two stages. The first stage is LOWPTDFS() which constructs a DF-tree, calculates the low-points and second low-points, and sorts the adjacency lists. The second stage is another DFS which we call EMBEDDINGDFS(). It is a DFS using the re-ordered adjacency lists and is given as Algorithm 14.17.3

Algorithm 14.17.3 constructs an embedding by placing the edges into two linked lists L_i and L_o, as in the simplified algorithm. The list L_i which originally corresponded to the chords placed inside C, now corresponds to the edges placed to the left of the DF-tree, because the drawings were made with the interior of C to the left of the path. Similarly, the list L_o now corresponds to edges placed to the right of the DF-tree, because the exterior of C was drawn to the right of the path. EMBEDDINGDFS() first descends the DF-tree. The first leaf it encounters will have a frond back to vertex 1. This is a consequence of the ordering of the adjacency lists. This creates a cycle C, which we draw to the left of the path. The remaining fronds and subtrees will be placed either in L_i or L_o, exactly as in the simplified algorithm.

A subtree T_{uv} is similar to a chord $(u, \mathit{LowPt}[v])$. T_{uv} fits in L_i if and only if a chord $(u, \mathit{LowPt}[v])$ does. In this case, we place a dummy chord $(u, \mathit{LowPt}[v])$ in L_i and a dummy chord (u, u) in L_o. A dummy chord is a chord that has an associated flag indicating that it is a placeholder for a subtree. If the dummy chord $(u, \mathit{LowPt}[v])$ is assigned to L_i, the subtree T_{uv} is to be embedded to the left of the path of the DF-tree containing u. The algorithm then calls EMBEDDINGDFS(v) recursively. The fronds placed in L_i by the recursion are placed to the left of the tree T_{uv}. The fronds placed in L_o are to the right of T_{uv}. The dummy chord (u, u) in L_o has the purpose of ensuring that any fronds placed in L_o by the recursive call must have $\mathit{UpperPt} \geq u$.

Algorithm 14.17.3: EMBEDDINGDFS(u)

comment: extend the embedding DFS from u

for all v adjacent to u

do
- **if** uv is a frond **and** v is above u
  - **then**
    - **if** (u, v) fits in L_i
      - **then** place (u, v) in L_i
      - **else if** (u, v) fits in L_o
      - **then** place (u, v) in L_o
      - **else**
        - $m =$ SWITCHSIDES(u, v)
        - **if** $m = 0$ **then return**(*NonPlanar*)
        - **if** $m = 1$
          - **then** place (u, v) in L_i
          - **else** place (u, v) in L_o
  - **else**
    - **comment:** uv is a tree edge
    - **if** $u =$ *Parent*$[v]$ **then**
      - $w =$ *LowPt*$[v]$
      - **if** (u, w) fits in L_i
        - **then** place (u, w) in L_i and (u, u) in L_o
        - **else if** (u, w) fits in L_o
        - **then** place (u, w) in L_o and (u, u) in L_i
        - **else**
          - $m =$ SWITCHSIDES(u, w)
          - **if** $m = 0$ **then return**(*NonPlanar*)
          - **if** $m = 1$
            - **then** place (u, w) in L_i and (u, u) in L_o
            - **else** place (u, w) in L_o and (u, u) in L_i
      - EMBEDDINGDFS(v)
      - **if** *NonPlanar* **then exit**

14.18 Notes

The important area of graph minors was developed in a series of over 20 papers by Robertson and Seymour. Some of their early work on graph minors is surveyed in their paper ROBERTSON and SEYMOUR [148]. They have proved a theorem of far-reaching significance, that in any infinite collection of graphs, there are always two graphs such that one is a minor of the other; or in other words, any set of graphs in which no graph is a minor of another, is finite. The books by DIESTEL [44] and ZIEGLER [196] contain excellent chapters on graph minors.

Rotation systems were developed by HEFFTER [81] and EDMONDS [46].

The Jordan curve theorem is non-trivial – see ROSS and ROSS [150] for some remarkable examples.

Read's algorithm to draw a planar graph by reducing a triangulation is from READ [144]. It was modified to use a regular polygon as the outer face by KOCAY and PANTEL [109]. Tutte's method of using barycentric coordinates to construct convex drawings of graphs appeared in TUTTE [173].

Whitney's theorem appeared in WHITNEY [191].

Good source books for polytopes are the books by GRÜNBAUM [76] and ZIEGLER [196]. A classic text on polyhedra is by COXETER [38].

The original proof of the four-color theorem appeared in APPEL and HAKEN [5] and [6]. An excellent survey article of the Appel-Haken proof is WOODALL and WILSON [194]. There are a number of excellent books on the 4-color problem, including SAATY and KAINEN [153], ORE [133], and FRITSCH and FRITSCH [56]. A very readable history of the 4-color problem can be found in WILSON [193]. A shorter proof was accomplished by ROBERTSON, SANDERS, SEYMOUR, and THOMAS in [149]. Much of the development of graph theory arose out of attempts to solve the 4-color problem. AIGNER [2] develops the theory of graphs from this perspective. A survey of work on problems related to nowhere-zero flows appears in JAEGER [94]. A detailed treatment of k-flows can be found in BONDY and MURTY [24] and DIESTEL [44].

Kuratowski's theorem is a famous theorem of graph theory. It originally appeared in KURATOWSKI [112]. The proof presented here is based on a proof of KLOTZ [101]. See also THOMASSEN [167].

The Hopcroft-Tarjan planarity algorithm is from HOPCROFT and TARJAN [88]. See also WILLIAMSON [192]. It is usually presented as a "path-addition" algorithm; that is, an algorithm that embeds one path at a time across a cycle. It is presented here as an equivalent algorithm that recursively embeds a branch of the DF-tree.

15

Graphs and Surfaces

15.1 Introduction

The plane and the sphere are the simplest topological surfaces. The structure of planar graphs, and algorithms for embedding graphs on the plane are well understood. Much less is known about graph embeddings on other topological surfaces, and the structure of these graphs. We begin with the torus, the doughnut-shaped surface shown in Figure 15.1. We imagine this surface made out of rubber, and using scissors, cut it along the two circumferences shown in the diagram. The surface of the torus then unfolds into a rectangle, which is indicated on the right. The opposite sides of the rectangle labeled a must be glued together with the arrows aligned, as must the sides labeled b, in order to reconstruct the torus. We could glue the edges in the order a, then b; or else b, then a. Both represent the same torus.

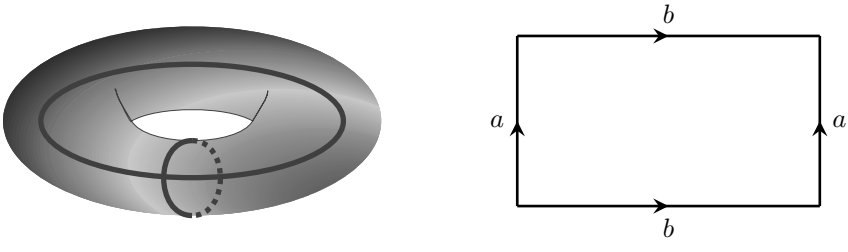


FIGURE 15.1
The torus

When a graph is drawn on the rectangle representing the torus, we must remember that the two sides labeled a (and the two sides b) are really the same, so that graph edges can "wrap around" the diagram. Notice that the four corners of the rectangle all represent the same point. Figure 15.2 shows two embeddings of $K_{3,3}$ on the torus.

These embeddings of $K_{3,3}$ are very different from each other. Unlike the plane in which a 3-connected graph has a unique embedding (up to orientation), some graphs have very many distinct embeddings in the torus, or other surfaces.

DEFINITION 15.1: An *embedding* of a graph G in a surface Σ is a function ψ that maps the vertices of G into points of Σ, and the edges of G into continuous

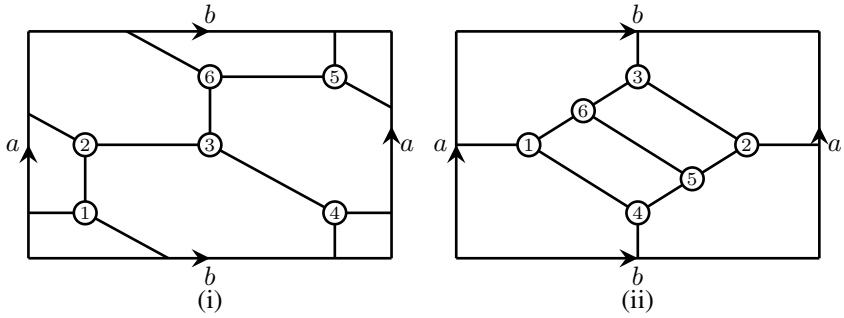


FIGURE 15.2
Two embeddings of $\boldsymbol{K_{3,3}}$ on the torus

curves in Σ, such that the curves representing two edges intersect only at a common endpoint. We write G^{ψ} for the image of G under the embedding ψ.

In the embeddings of Figure 15.2, we can assign a "coordinate system" to the rectangle representing the torus, and then construct ψ by assigning coordinates to the vertices, and then draw the edges as straight lines. This is how the diagram was constructed.

Definition 15.1 uses an intuitive notion of a *surface*, and an intuitive notion of *continuous*. Currently we have the plane, sphere, or torus in mind. We will later make the definition of a surface more precise. Because we have coordinate systems for the above surfaces, by "continuous" we mean continuous mappings of the coordinates. However, topological continuity does not require coordinates.

If we cut the torus along the edges of the embeddings of $K_{3,3}$, the torus surface falls apart into several connected regions. As in the case of the plane, we call these regions the *faces* of the embedding. Once again we are relying on an intuitive notion for the concepts of region and face. A *facial cycle* or *facial walk* is an oriented cycle or walk of the graph which bounds a face. The embedding of $K_{3,3}$ in Figure 15.2(i) has three faces, each bounded by a hexagon. The embedding in Figure 15.2(ii) also has three faces, two bounded by quadrilaterals, and one bounded by a facial walk which has length 10, and in which several vertices and edges appear twice. A convenient way of finding the faces and facial walks of an embedding is to think of walking along an edge of the graph, while holding a paint brush in one's right hand, and painting a stripe along the surface. When a vertex is reached, we take the first edge on our right, so that the paint brush does not cross an edge, and continue tracing out the facial boundary. Upon returning to the starting edge, tracing it in the same direction as before, a facial walk has been completed.

An *open disc* in the plane is the region interior to a circle. We will use an intuitive understanding of the notion of *homeomorphic* regions and surfaces. Two regions R_1 and R_2 are said to be homeomorphic if one can be transformed into the other by a one-to-one continuous deformation, whose inverse is also continuous. For example, an open disc is homeomorphic to a face bounded by a hexagon, or by any other

polygon. (However an open disc is not homeomorphic to a polygon with a hole in its interior.) A *homeomorphism* of R_1 and R_2 is any continuous one-to-one mapping from R_1 onto R_2, whose inverse is also continuous.

DEFINITION 15.2: A *2-cell* is a region homeomorphic to an open disc. An embedding G^ψ is a *2-cell embedding* if all faces are 2-cells.

Corresponding to the idea of a 2-cell are 0-cells (a single point), 1-cells (an open line segment), and 3-cells (the interior of a sphere in 3-dimensional space), etc.

Usually we will restrict embeddings of a graph G to 2-cell embeddings. For example, we could draw a planar embedding of a planar graph G, such as the cube, in the rectangle representing the torus. The result would not be a 2-cell embedding of G, for the outer face of the embedding would be homeomorphic to a torus with a hole in it, which is not a 2-cell. Embeddings which are not 2-cell embeddings really belong in a different surface. For the cube, there are five distinct 2-cell embeddings on the torus (see Figure 15.21).

DEFINITION 15.3: Two embeddings G^{ψ_1} and G^{ψ_2} in a surface Σ are *homeomorphic* if there is a homeomorphism of Σ which maps G^{ψ_1} to G^{ψ_2}. Otherwise G^{ψ_1} and G^{ψ_2} are *distinct embeddings*.

It is clear that a homeomorphism of G^{ψ_1} and G^{ψ_2} induces an automorphism of G, and that the faces of G^{ψ_1} map to the faces of G^{ψ_2}. The embeddings of Figure 15.2 are then easily seen to be distinct, because their facial walks have different lengths. In general, there is no easy method of determining all the embeddings of a graph on a given surface, or even to determine whether a graph is embeddable.

15.2 Surfaces

We can use the method of representing the torus as a rectangle, as in Figure 15.1, to represent a *cylinder*, and a *Möbius band*, shown in Figures 15.3 and 15.5.

The cylinder is glued along only one edge of the rectangle. We call it an *open surface* because it has a boundary (the edges which are not glued). If we project a cylinder onto a plane, one possible result is an annulus, that is, a disc with a hole in it. This immediately gives the following theorem:

Theorem 15.1. *A graph can be embedded on the cylinder if and only if it can be embedded on the plane.*

Proof. Given an embedding of a graph G on the plane, choose any face except the outer face, and cut a hole in it. The result is an embedding on the cylinder, and vice versa. □

Notice that an embedding of G on the cylinder corresponds to an embedding of G on the plane with two distinguished faces. They can be any two faces of a

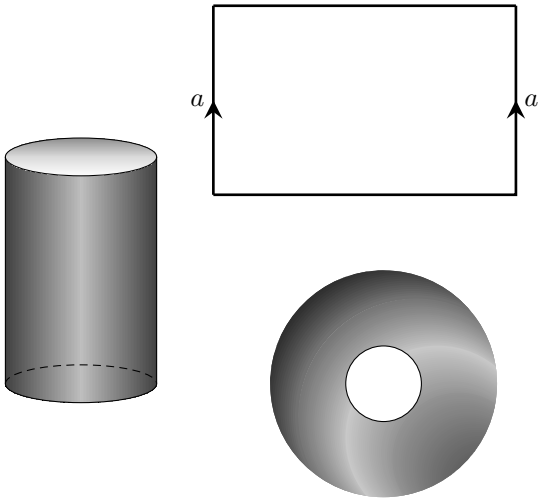


FIGURE 15.3
Three representations of the cylinder

planar embedding of G. By Whitney's theorem, we know that up to orientation, a 3-connected planar graph has just one embedding on the plane. This is not the case for embeddings on the cylinder. Two embeddings of the graph of the cube on the cylinder are shown in Figure 15.4.

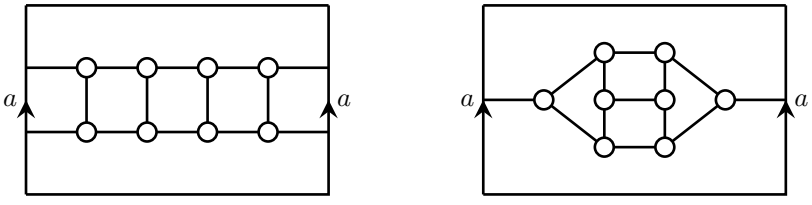


FIGURE 15.4
Two cylindrical embeddings of the cube

The *Möbius band* is constructed by giving one end of the rectangle a twist of 180 degrees before aligning and gluing the opposite edges. This is indicated by the opposite orientation of the arrows. Notice that if we follow the boundary of the Möbius band, it is a single closed curve, unlike the boundary of the cylinder.

The sphere, torus, and cylinder can all be considered as *two-sided* surfaces – they have an inside and an outside. One way to define this is to imagine a small clockwise-oriented circle drawn in the surface. If we reflect this circle in the surface, we obtain a circle of the opposite orientation. On the sphere, torus, and cylinder it is not possible

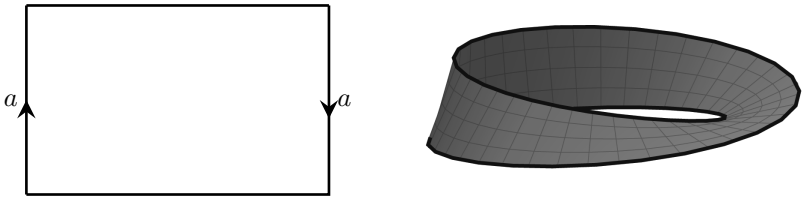


FIGURE 15.5
The Möbius band

to walk along the surface, taking the oriented circle with us, until it coincides with its opposite orientation (we are not allowed to walk over the boundary of the cylinder or Möbius band). On the Möbius band, it is possible to make these two circles coincide. We therefore say that the Möbius band is a *one-sided* surface. A two-sided surface is said to be *orientable*. We can assign an orientation to the surface, by partitioning the set of oriented circles defined at every point of the surface. One orientation is called the *inside*, and the other the *outside*. A one-sided surface is *non-orientable*, as the set of oriented circles does not have this partition into two subsets. For surfaces like the sphere and torus which can be constructed in euclidean 3-dimensional space, we could alternatively use a normal vector to the surface, and its reflexion in the surface in place of an oriented circle and its reflexion.

Aside from the cylinder and Möbius band, the surfaces we will be interested in are closed surfaces.

A *closed surface* is a generalized notion of a *polyhedron*. A *polyhedron* is a three-dimensional object consisting of a set of polygons, of three or more sides each. Each polygon is bounded by a sequence of p straight-line segments connecting p vertices in cyclic order, for some $p \geq 3$. The line segments are the edges of the polyhedron. Each edge is shared by exactly two polygons. Any two polygons may intersect only on a single common edge. There are at least three polygons meeting at each vertex, and the polygons meeting at any vertex form a single cycle.

This idea is generalized by allowing the polygons composing a polyhedron to be infinitely stretchable and interpenetrable. They are then called *curvilinear polygons*.

DEFINITION 15.4: A *closed surface* is a set of points homeomorphic to a polyhedron made of curvilinear polygons.

Thus a closed surface is an object that is capable of being represented as a collection of curvilinear polygons glued together along common edges. However, the surface is not any single one of these representations, because many different polygonal representations of the same closed surface are possible. For example, the surface of the sphere may be partitioned into many different polygonal forms. Similarly, the embeddings of $K_{3,3}$ of Figure 15.2 partition the torus into two different polyhedra.

When we identify opposite edges of a rectangle, we are identifying two edges of a single polygon. In order to conform to the definition of polyhedron, in which only

distinct polygons share a common edge, we can subdivide the rectangle into two or more polygons, as necessary. There are three more surfaces that can be made from a rectangle by identifying its edges in pairs. They are the sphere, projective plane, and Klein bottle, illustrated in Figures 15.6, 15.7, and 15.9. The sphere can also be represented as a *digon* (a "polygon" of two sides), as shown, where $c = ab$.

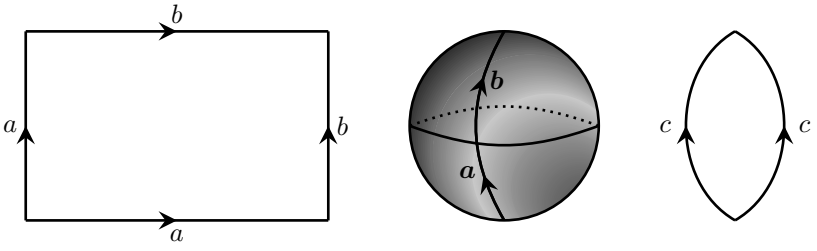


FIGURE 15.6
The sphere as a rectangle and as a digon

The *projective plane* is constructed from a rectangle by first using a twist to create a Möbius band, and then by gluing the boundary of the Möbius band to itself, to create a closed surface. This is illustrated in Figure 15.7, where the edges labeled b have been glued. This cannot be done in euclidean space, for the polygon must cross itself without intersecting itself. But mathematically, we can consider the surface to be constructed in this way. The projective plane can also be viewed as a digon, as illustrated in Figure 15.8, by combining the a and b sides of the rectangle into a single side, labeled c. Because of the orientation of the arrows, the two "corners" of the digon represent the same point. We can identify the corners, creating a "figure eight", and then identify the two lobes of the figure eight to complete the projective plane. If we then remove a disc from the projective plane by cutting along the circle containing the dotted line, the result is called a *crosscap*. It is a projective plane with a hole in it. In Exercise 15.2.1, it is proved that a crosscap is homeomorphic to a Möbius band.

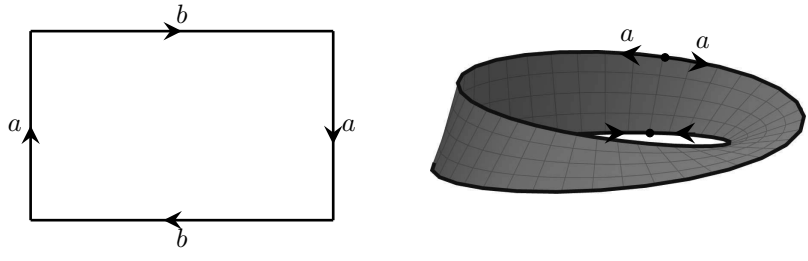


FIGURE 15.7
The projective plane as a rectangle, and as a Möbius band

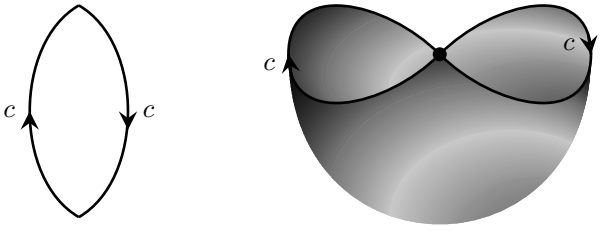


FIGURE 15.8
The projective plane as a digon, and as a crosscap

The Klein bottle can be constructed by gluing two edges of a rectangle to create a cylinder, and then by gluing the ends of the cylinder together, according to the orientation of the arrows. This also cannot be done in euclidean space, as the cylinder must cross through itself without intersecting itself.

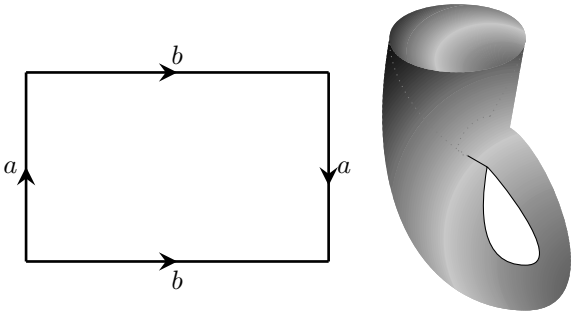


FIGURE 15.9
The Klein bottle as a rectangle, and as a "bottle"

These surfaces have been constructed from a rectangle by gluing edges together. By subdividing the rectangle into curvilinear polygons, closed surfaces represented as curvilinear polyhedra are obtained. Polygons with more sides than a rectangle could also be used. By the classification theorem of closed surfaces (Theorem 15.2), every closed surface can be constructed by gluing together edges of a single curvilinear polygon.

The rectangle representing the torus in Figure 15.1 can be written symbolically as $a^+b^+a^-b^-$. This means that we choose a clockwise orientation of the rectangle and write a^+ for the edge labeled a when the direction of the arrow is clockwise, and a^- when the direction of the arrow is counterclockwise. The boundary of the rectangle is then determined by the above symbol. Similarly the rectangle representing the

sphere (Figure 15.6) can be characterized as $a^+b^+b^-a^-$, that of the projective plane (Figure 15.7) as $a^+b^+a^+b^+$, and that of the Klein bottle (Figure 15.9) as $a^+b^+a^+b^-$.

In general, we have a polygon with an even number of sides, with labels $a_1, a_2, \ldots, a_p$, such that every label a_i appears on exactly two sides. We place an arrow on each edge of the polygon in some direction, and choose a clockwise orientation of the polygon. This defines a symbolic representation in terms of the a_i^+ and a_i^-. Every closed surface can be represented symbolically by a *normal form* of this type, as shown by the following theorem:

Theorem 15.2. (Dehn and Heegard – Normal forms for closed surfaces) *Every closed surface can be represented symbolically by one of the following normal forms. Two closed surfaces are homeomorphic if and only if they have the same normal form.*

1. a^+a^-
2. $a_1^+b_1^+a_1^-b_1^-a_2^+b_2^+a_2^-b_2^- \ldots a_p^+b_p^+a_p^-b_p^-$
3. $a_1^+a_1^+a_2^+a_2^+ \ldots a_q^+a_q^+$

The proof of this theorem can be found in FRÉCHET and FAN [54] or STILLWELL [161]. It is too lengthy to include here. It involves cutting and pasting a curvilinear polygon until the normal form is achieved. This is done in stages. The torus is represented in Figure 15.1 by $a^+b^+a^-b^-$, which is already in normal form. The sphere is represented by $a^+b^+b^-a^-$. It is clear from the diagram that the adjacent b^+b^- corresponds to edges that can be glued, so that they cancel from the formula, leaving a^+a^- as the normal form for the sphere. The projective plane is represented in Figure 15.7 as $a^+b^+a^+b^+$, which is not in normal form. We could transform it into normal form by letting $c = ab$, and obtain c^+c^+. Alternatively, we could illustrate the techinques used in the proof of Theorem 15.2 and convert it to normal form by the following sequence of operations, illustrated in Figure 15.10.

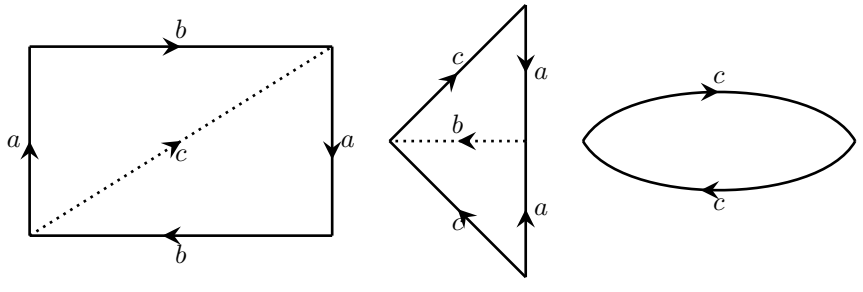


FIGURE 15.10
Transforming the projective plane to normal form

We make a diagonal cut across the rectangle, and label it c. We thereby obtain two triangles which we glue together along the edge b, which then disappears. The

symbolic form is now $c^+a^+a^-c^+$. The edges a^+a^- are then glued, and thereby cancel, to produce the digon in Figure 15.10, giving the normal form c^+c^+ for the projective plane.

15.2.1 Handles and crosscaps

Consider a surface with normal form $a^+b^+a^-b^-c^+d^+c^-d^-$, shown in Figure 15.11. We have an octagon with edges that are to be identified in pairs. Both endpoints of the upper edge marked b represent the same point, as they are the same endpoint of the arrows marked a. Therefore gluing the edges labeled a will make this b-edge the boundary of a hole in the surface. Consequently, the other edge labeled b also represents the boundary of a hole in the surface, and these two boundaries must be identified. One way to identify them is to attach the two ends of a cylindrical tube to each of these holes. The result is a *handle* attached to the surface.

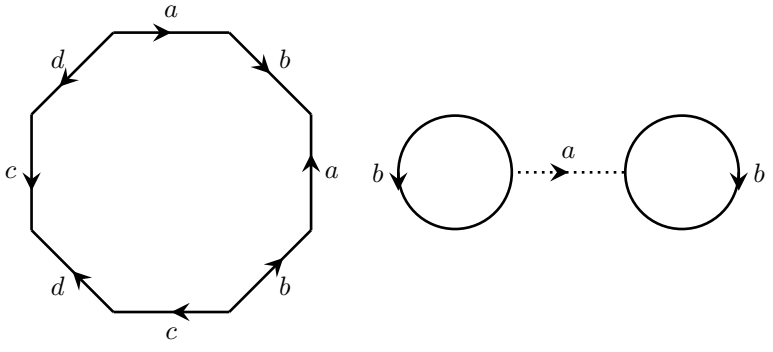


FIGURE 15.11
A sphere with two handles

Now the same can be done for the edges marked c in the diagram – we attach another handle. This same argument holds for any normal form of this type. This gives:

Theorem 15.3. *A surface with normal form*

$$a_1^+b_1^+a_1^-b_1^-a_2^+b_2^+a_2^-b_2^- \ldots a_p^+b_p^+a_p^-b_p^-$$

is homeomorphic to a sphere with p handles.

Because a sphere is an orientable surface, so is a sphere with p handles.

Consider now a surface with normal form $a^+a^+b^+b^+c^+c^+$, illustrated in Figure 15.11. We have a hexagon in which consecutive edges are to be identified in pairs. The vertex common to the two sides marked a is both the head and tail of the a-arrow. Therefore indentifying the endpoints of the edges marked a makes two holes in the surface, bounded by the a-edges. We must identify the boundaries of these two lobes.

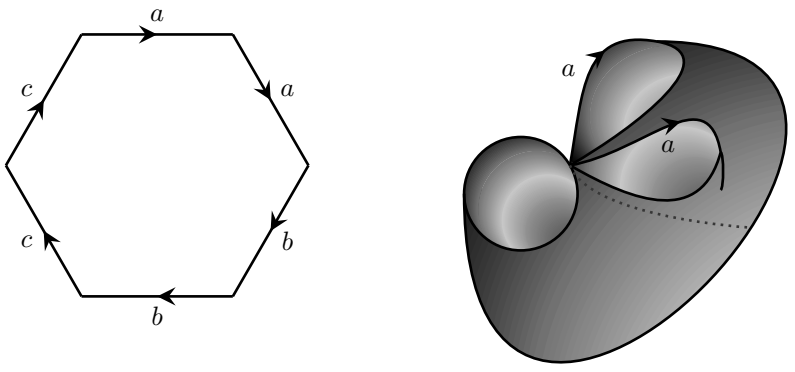


FIGURE 15.12
A sphere with three crosscaps

This is nearly identical to the construction of the projective plane from a digon, illustrated in Figure 15.8. If we draw a circle around the two a-lobes in Figure 15.12, and cut along the dotted line, we see that what we have is in fact a *crosscap* glued into a hole cut in the surface. We can then do the same for the remaining pairs b^+b^+ and c^+c^+ to get a sphere with three crosscaps. In general, this gives:

Theorem 15.4. *A surface with normal form*

$$a_1^+a_1^+a_2^+a_2^+\ldots a_q^+a_q^+$$

is homeomorphic to a sphere with q crosscaps.

Because a crosscap is an nonorientable surface, so is a sphere with q crosscaps. Gluing a crosscap to a hole in a sphere is equivalent to gluing the boundary of a Möbius band to a hole in the sphere.

15.2.2 The Euler characteristic and genus of a surface

We know from Chapter 14 that a connected planar graph with n vertices, ε edges, and f faces satisfies Euler's formula $n - \varepsilon + f = 2$. Furthermore, the skeleton of a polyhedron is a planar graph, so that any polyhedral division of the sphere also satisfies this formula. We say that the *Euler characteristic* of the sphere is 2.

DEFINITION 15.5: Let Σ be a closed surface represented by a curvilinear polyhedron with n vertices, f polygons, and ε edges. The *Euler characteristic* of the surface is the value $n - \varepsilon + f$. It is denoted $\chi(\Sigma)$.

It has been proved by Kerékjártó that the value $n-\varepsilon+f$ for a surface is invariant, no matter what polygonal subdivision is used to represent it. This is difficult to prove because of the vast numbers of polygonal subdivisions that are possible. However, we can get an understanding of it as follows. If we add a diagonal across a face, n does not change, but ε and f both increase by one. Thus $n-\varepsilon+f$ does not change. Similarly, if we subdivide an edge with a new vertex, n and ε increase by one, but f does not change. Modifying a polygonal subdivision by these operations does not change the value $n-\varepsilon+f$. Suppose that we now add a handle or a crosscap to a given polygonal division of a surface. Consider a polygonal division with n vertices, f polygons, and ε edges. Choose two polygons on the surface, cut a disc from the interior of each polygon, and attach a handle connecting them. Let the polygons be P_1 and P_2. Refer to Figure 15.13. We draw two curves along the handle connecting P_1 to P_2. The result is a polygonal division of the surface with an additional handle. Let it have n' vertices, f' polygons, and ε' edges. The effect of drawing the curves connecting P_1 to P_2 is to add two vertices and two edges to each of P_1 and P_2, plus two additional edges represented by the curves. The number of polygons does not change. We therefore have $n'=n+4$, $f'=f$, and $\varepsilon'=\varepsilon+6$, so that $n'-\varepsilon'+f'=(n-\varepsilon+f)-2$. Thus, when a handle is added to a surface, the Euler characteristic decreases by two. It does not matter to which faces the handle attaches. It follows that a sphere with p handles has Euler characteristic $2-2p$.

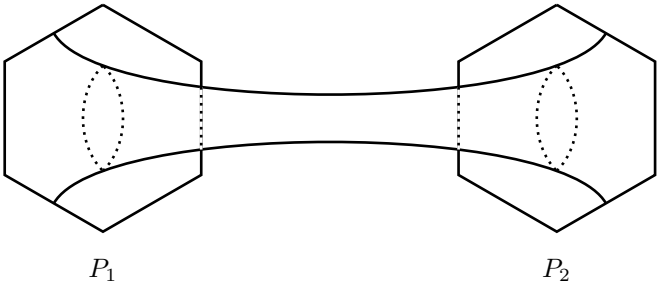


FIGURE 15.13
Attaching a handle to a surface

Suppose that we now add a crosscap to a polygonal division of a surface (e.g., the sphere). We choose a polygon P_1, cut a disc from its interior, and attach a crosscap. Let C be the boundary of the disc. Now a crosscap is a Möbius band, and the boundary of the disc becomes the boundary of the Möbius band. We draw a curve connecting P_1 to C, continue across the Möbius band to the opposite side of C, and continue to an opposite point of P_1. The result is a polygonal division of the surface with an additional crosscap. Let it have n' vertices, f' polygons, and ε' edges. The effect of drawing the curve across P_1 is to add two vertices and two edges to P_1, plus an additional edge represented by the curve. When the Möbius band is cut across, it remains connected. Therefore the number of polygons does not change. We therefore have $n'=n+2$, $f'=f$, and $\varepsilon'=\varepsilon+3$, so that $n'-\varepsilon'+f'=(n-\varepsilon+f)-1$. Thus,

when a crosscap is added to a surface, the Euler characteristic decreases by one. It does not matter to which face the crosscap attaches. It follows that a sphere with q crosscaps has Euler characteristic $2 - q$.

Consequently surfaces can be classified according to whether they are orientable or non-orientable, and their Euler characteristic. A related parameter of a surface is its *genus*.

DEFINITION 15.6: A Jordan curve in a surface Σ is *contractible* or *null-homotopic* if it can be continuously shrunk to a point within Σ.

Cutting a surface along a contractible Jordan curve always separates it into two pieces.

DEFINITION 15.7: The *genus* of a surface is the maximum number of non-intersecting Jordan curves that can be drawn on the surface such that cutting along the curves does not separate it into two or more pieces.

A consequence of the Jordan curve theorem is that the sphere has genus zero. We can then see that the torus has genus one.

If the Jordan curves of Definition 15.7 are allowed to intersect, then the maximum number of Jordan curves that can be drawn such that cutting along the curves does not separate it into two or more pieces is called the *connection number* (see [54]). It is easy to see that for the torus, two intersecting Jordan curves can be drawn, such that cutting along the curves leaves a rectangle. Thus the torus has connection number two.

In general, a sphere with p handles has genus p, as exactly one Jordan curve can be drawn around each handle without separating the surface. Because a sphere with handles is an orientable surface, we say it has *orientable genus* p.

The projective plane has genus one, as shown in Exercise 15.2.2. It then follows from Exercise 15.2.3 that a sphere with q crosscaps has genus q. We say it has *unorientable genus* q. Some texts use the term *crosscap number* in place of genus for a non-orientable surface. The relation with Euler characteristic can now be stated.

Theorem 15.5. *An orientable surface of genus p has Euler characteristic $2 - 2p$. A non-orientable surface of genus q has Euler characteristic $2 - q$.*

When a graph G is embedded in a surface Σ, cycles of G map to Jordan curves in Σ. We will be interested in cycles which are embedded as non-contractible curves.

DEFINITION 15.8: Let G^ψ be an embedding of a graph G in a surface Σ. A cycle C in G is an *essential* cycle, or *non-contractible* cycle of the embedding on Σ if C^ψ is not contractible.

For example, in the embedding of $K_{3,3}$ on the left in Figure 15.2, the cycles $(1, 2, 3, 4)$ and $(2, 3, 4, 5)$ are essential cycles, while $(1, 2, 3, 6, 5, 4)$ is not.

DEFINITION 15.9: The *genus* of a graph G is $g(G)$, the smallest genus of an orientable surface Σ such that G has a 2-cell embedding in Σ. The *unorientable genus* or *crosscap number* of G is $\overline{g}(G)$, the smallest genus of a non-orientable surface Σ such that G has a 2-cell embedding in Σ.

Suppose that G^ψ is a 2-cell embedding of G in an orientable surface Σ of genus p. Let G^ψ have n vertices, ε edges, and f faces. Then because the faces of G^ψ determine a polygonal division of the surface, we have $n-\varepsilon+f=\chi(\Sigma)=2-2p$. This is called the *Euler-Poincaré formula*. If G is 2-cell embedded on an unorientable surface of genus q, then $n-\varepsilon+f=\chi(\Sigma)=2-q$. This gives the following relations for graphs embedded on the:

plane: $n-\varepsilon+f=2$
torus: $n-\varepsilon+f=0$
projective plane: $n-\varepsilon+f=1$.

Lemma 15.6. *A triangulation of an orientable surface of genus p, with n vertices satisfies $\varepsilon=3n+6(p-1)$. A triangulation of a non-orientable surface of genus q satisfies $\varepsilon=3n+3(q-2)$.*

Proof. A triangulation satisfies $3f=2\varepsilon$. Combining this with $n-\varepsilon+f=\chi(\Sigma)$ gives the result. □

Exercises

15.2.1 Show that if a disc is removed from a projective plane, the result is a Möbius band.

15.2.2 Show that if the projective plane is cut along a non-separating Jordan curve, the result is a disc.

15.2.3 Show that exactly one Jordan curve can be drawn on the Möbius band without separating it. What is the result of cutting the Möbius band along a non-separating Jordan curve?

15.2.4 Use cutting and pasting to convert the representation $a^+b^+a^+b^-$ of the Klein bottle to normal form $c^+c^+d^+d^+$.

15.2.5 A sphere with one handle and one crosscap can be represented symbolically by $a^+b^+a^-b^-c^+c^+$. Find the normal form for this surface.

15.2.6 Find the facial cycles of the embeddings of $K_{3,3}$ on the torus shown in Figure 15.2.

15.2.7 Use Lemma 15.6 to obtain a lower bound on $g(G)$ and $\overline{g}(G)$ for an arbitrary graph G.

15.2.8 Show that $g(K_n)\geq(n-3)(n-4)/12$ and that $\overline{g}(K_n)\geq(n-3)(n-4)/6$.

15.2.9 Let G be a graph with no triangles embedded on a surface of genus g. Find an upper bound on the number of edges of G.

15.3 Isometries of surfaces

Graph embeddings on a surface are related to the geometrical properties of that surface. And the geometrical properties are closely related to the symmetries of the surface. This point of view was established by Felix Klein [100] in his lecture at the Erlanger University in 1872, which has become known as his *Erlanger program*. An *isometry* of a surface is a continuous transformation that preserves distance. For example, the isometries of the plane are the translations, reflexions, rotations, and combinations of these. The isometries of the sphere are composed of rotations, reflexions in a plane through the center of the sphere, and a reflection which interchanges each point with its antipodal point. It is easy to see that the isometries of a surface form a group.

It turns out that every translation or rotation of the plane can be expressed in terms of reflexions. For example, given a line ℓ, let R_ℓ denote the reflexion in that line. If ℓ_1 and ℓ_2 are two parallel lines in the plane, then the combination $R_{\ell_1} R_{\ell_2}$ (first R_{ℓ_1}, then R_{ℓ_2}) denotes a sequence of two reflexions. It is easy to see from Figure 15.14, that the result is a translation in the direction orthogonal to ℓ_1, towards ℓ_2, by an amount equal to twice the distance between ℓ_1 and ℓ_2. In the diagram, the numbers 1, 2, and 3 indicate three successive positions of a point as it is reflected, first in ℓ_1, then in ℓ_2.

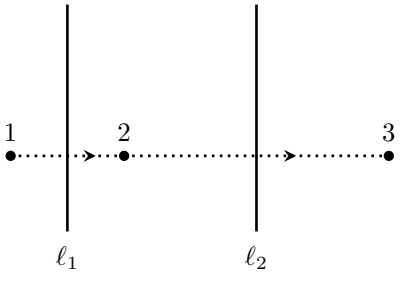


FIGURE 15.14
A translation from two reflexions

Similarly, if ℓ_1 and ℓ_2 are intersecting lines, with an acute angle θ between them, then the reflexions $R_{\ell_1} R_{\ell_2}$ produce a rotation about their point of intersection, by an angle 2θ, in the direction of the acute angle from ℓ_1 towards ℓ_2. Again the diagram in Figure 15.15 indicates three successive positions of a point by $1, 2, 3$.

It follows that the group of isometries of the plane can be generated by the reflexions. The subgroup generated by an even number of reflexions is orientation-preserving. The elements of the coset with an odd number of reflexions are all orientation-reversing.

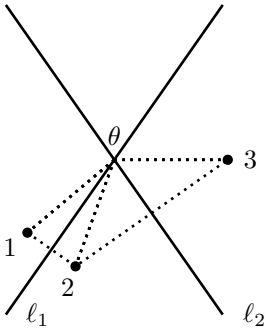


FIGURE 15.15
A rotation from two reflexions

It is convenient to represent the plane as the complex plane $\mathbb{C}$. Let Γ denote the group of isometries of the plane. Then a translation can be represented as $T_\alpha : z \mapsto z + \alpha$, where $z \in \mathbb{C}$, and α is a complex constant. If we choose α to be a real number, then T_α represents a horizontal translation. Similarly, if β is an imaginary number, then T_β is a vertical translation. T_α and T_β generate a subgroup $\Gamma(\alpha, \beta)$ of the full isometry group Γ. This is the group of all *integer* combinations of T_α and T_β. Every point $z \in \mathbb{C}$ is translated an integral number of multiples of α horizontally, and an integral number of multiples of β vertically. Now the values α and β determine a rectangle in the complex plane, of width $|\alpha|$ and height $|\beta|$, with one corner at the origin. The elements of $\Gamma(\alpha, \beta)$ map this rectangle to other rectangles such that the entire plane $\mathbb{C}$ is tiled by the rectangles. The original rectangle can be viewed as a rectangle representing the torus. It is called a *fundamental region* of the group $\Gamma(\alpha, \beta)$. Consequently, the torus can be viewed as a "factorization" of the plane $\mathbb{C}$ by the group $\Gamma(\alpha, \beta)$. This means the following: every point in the fundamental region is mapped to an infinite number of other points in the plane by the group $\Gamma(\alpha, \beta)$. All of these points are identified into a single point, so that they are considered to be a single point of the factorization $\mathbb{C}/\Gamma(\alpha, \beta)$. The torus is then *defined* to be the collection of these "meta-points". Another viewpoint would be to say that the plane has been decomposed into equivalent copies of the fundamental region. Note that α and β need not be chosen as real and imaginary. The fundamental rectangle then becomes a fundamental parallelogram, which still represents a torus.

Similarly, if we "factorize" the plane by the group generated by a single translation $\Gamma(\alpha)$, the result is $\mathbb{C}/\Gamma(\alpha)$, an infinite cylinder, i.e., a cylinder of infinite length, but finite circumference. The fundamental region would then be an infinitely long strip of finite width.

Observe that complex conjugation $z \mapsto \overline{z}$ represents a reflexion in the x-axis. If we factorize the plane by the group generated by T_α and complex conjugation, the result is the "twisted" cylinder, i.e., a Möbius band of infinite width.

This point of view is developed by Stillwell [162], where all possible factorizations of the plane, and their corresponding surfaces, are determined.

It is possible to extend the complex numbers to the surface of the sphere, using stereographic projection (see Section 14.9). Consider a sphere of radius one, centered at $(0,0,0)$. Given a point (x,y,z) on the surface of the sphere, let $w \in \mathbb{C}$ represent the corresponding point, projected onto the plane $\mathbb{C}$ by a line joining the north pole $(0,0,1)$ through (x,y,z), and vice versa. Only the north pole has no corresponding complex number. We assign it the symbol ∞, and let $\mathbb{C}^+$ denote $\mathbb{C} \cup \infty$. See Pedoe [135]. If $w \in \mathbb{C}$, then the arithmetic is extended by the rules

1. $w/\infty = 0$,
2. $w \pm \infty = \infty$,
3. $w/0 = \infty$, (where $w \neq 0$),
4. $w \cdot \infty = \infty$, (where $w \neq 0$).

The expressions $0/0$, $\infty - \infty$, ∞/∞, $0 \cdot \infty$ are not used, and remain undefined. Using the extended complex numbers $\mathbb{C}^+$, we can then do *linear fractional transformations*, also known as *Möbius transformations*, of the form

$$w \mapsto \frac{\alpha w + \beta}{\gamma w + \delta}$$

where $w \in \mathbb{C}^+$ and $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, with $\alpha\delta - \beta\gamma \neq 0$. For if $w = \infty$, we divide the numerator and denominator of the fraction by w and then use one of the above rules. The usefulness of this approach is the following theorem of Gauss [65] (see Stillwell [162]).

Theorem 15.7. *The rotations of the sphere are given by*

$$w \mapsto \frac{\alpha w + \beta}{-\overline{\beta} w + \overline{\alpha}}$$

where $w \in \mathbb{C}^+$, $\alpha, \beta \in \mathbb{C}$ *and* $|\alpha|^2 + |\beta|^2 = 1$.

Rotations of the sphere are orientation-preserving isometries. The orientation-reversing isometries can be represented as an orientation-preserving isometry, followed by a reflexion. These can also be expressed as linear fractional transformations of $\mathbb{C}^+$.

Theorem 15.8. *The orientation-reversing isometries of the sphere are given by*

$$w \mapsto \frac{\alpha \overline{w} + \beta}{-\overline{\beta} \overline{w} + \overline{\alpha}}$$

where $w \in \mathbb{C}^+$, $\alpha, \beta \in \mathbb{C}$ *and* $|\alpha|^2 + |\beta|^2 = 1$.

The transformation that maps each point (x,y,z) to its antipodal point $(-x,-y,-z)$ is also an isometry of the sphere. It generates a group of order two. If the sphere is factorized by this group, thereby identifying each point with its antipodal point, the result is the projective plane. Thus we can say that the projective plane *is* the sphere factorized by this isometry.

A surface is generally represented by a regular polygon. For example, the torus can be represented by a rectangle in the plane. And a rectangle has corner angles of $\pi/2$ radians, so that exactly four rectangles meet at each corner. The torus can also be represented by a regular hexagon, which can be seen as follows. Make a diagonal cut on the rectangle, as shown in Figure 15.16, and transfer the triangle obtained to the opposite side of the rectangle. Then distort the result into a hexagon. This gives the representation $a^+b^+c^+a^-b^-c^-$ for the torus, although this is not a normal form. Translations of the hexagon now give a tiling of the plane by hexagons. For the hexagon can be translated vertically by the width of the hexagon, or along a line of slope $1/2$ or $-1/2$, giving a group of translations. Note that the corner angle of a regular hexagon is $2\pi/3$, and that three hexagons meet at each corner in the tiling.

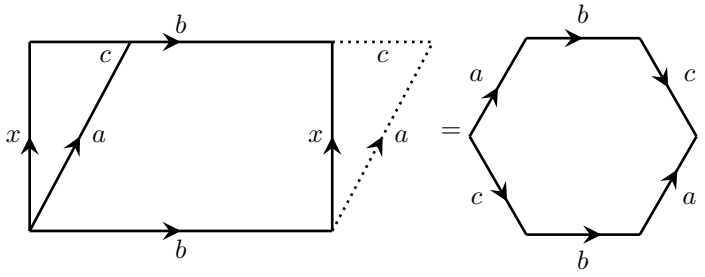


FIGURE 15.16
The torus represented as a hexagon

The normal forms of surfaces with more than one handle or more than two cross-caps require polygons with at least eight sides. A regular polygon in the plane with $2n$ sides has a corner angle of $(n-1)\pi/n$ radians. It follows that the plane cannot be tiled by regular polygons when $2n \geq 8$, for then the corner angles of the polygons meeting at each corner would sum to more than 2π radians. In Section 16.2 we will see that the *hyperbolic* plane can be tiled by octagons, and also by regular polygons with more sides. This will give a beautiful geometric representation of the *double torus*, i.e., the "torus with two holes", or equivalently, the sphere with two handles.

Exercises

15.3.1 Verify that the linear fractional transformations are well defined for all $w \in \mathbb{C}^+$ and all $\alpha, \beta, \gamma, \delta \in \mathbb{C}$.

15.3.2 Complex conjugation $w \mapsto \overline{w}$ in $\mathbb{C}$ is a linear fractional transformation. Determine the corresponding map of the sphere.

15.3.3 Find the linear fractional transformation which corresponds to a rotation of the sphere about the z-axis by $\pi/2$ radians.

15.3.4 Find the linear fractional transformation which corresponds to a reflexion of the sphere in a horizontal plane through its center.

15.3.5 The isometry $(x, y, z) \mapsto (-x, -y, -z)$ can be constructed by three successive reflexions, corresponding to reflexions in the xy, xz and yx-planes. Find the linear fractional transformation that represents this isometry.

15.4 Graph embeddings, obstructions

Three of the main algorithmic problems of graph embeddings are:

Problem 15.1:	Graph embeddability
Instance:	a graph G and a surface Σ.
Question:	is G embeddable in Σ?

Problem 15.2:	Graph embeddings
Instance:	a graph G and a surface Σ.
Question:	find all distinct embeddings of G in Σ.

Problem 15.3:	Graph genus
Instance:	a graph G and an integer k.
Question:	is $g(G) \leq k$? is $\bar{g}(G) \leq k$?

It was proved by THOMASSEN [168] that Graph Genus is NP-complete.

The first two problems are solved for the plane, but only partially solved for other surfaces. Several efficient algorithms are known for Graph embeddability on the projective plane.

For the plane, Kuratowski's theorem tells us that G is embeddable if and only if it has no subgraph isomorphic to TK_5 or $TK_{3,3}$. These graphs are called *obstructions* to planarity. There are two kinds of obstructions — those that are subgraphs, and those that are graph minors. A subgraph obstruction is called a *topological obstruction* or *forbidden subgraph*. An obstruction that is a graph minor is called a *forbidden minor*.

DEFINITION 15.10: Given a surface Σ, a *topological obstruction* for Σ is a graph K with $\delta(K) \geq 3$ such that any graph containing a subdivision TK cannot be embedded in Σ, and no proper subgraph of K has this property.

There are two topological obstructions for the plane, K_5 and $K_{3,3}$, as proved by Kuratowski's theorem 14.29. The definition requires that no proper subgraph of K is a topological obstruction, that is, that K be *minimal* with respect to this property (otherwise graphs such as K_6, K_7, etc., would all be considered as topological obstructions).

DEFINITION 15.11: Given a surface Σ, a *minor-order obstruction* (or *forbidden minor* or *excluded minor*) for Σ is a graph K, such that any graph having K as a minor cannot be embedded in Σ, but no proper minor of K has this property.

The graph relation "H is a minor of G" forms a partial order on the set of all graphs. If we restrict the set to all graphs which are not embeddable in the surface Σ, then the minimal graphs of this partial order are the minor-order obstructions. If K is a minor-order obstruction, then it is also a topological obstruction. K_5 and $K_{3,3}$ are both minor-order obstructions and topological obstructions for the plane, because any graph which has K_5 or $K_{3,3}$ as a minor necessarily contains either a TK_5 or $TK_{3,3}$; and TK_5 and $TK_{3,3}$ are not minors of each other. According to Wagner's theorem 14.30, K_5 and $K_{3,3}$ are the minor-order obstructions for the plane. For other surfaces, there is a distinction between the two concepts of topological obstruction and minor-order obstruction.

Robertson and Seymour have proved that there are a finite number of obstructions for any given surface, as a consequence of the *graph minor theorem*, which we state without proof.

Theorem 15.9. (Robertson-Seymour theorem) *In any infinite collection of graphs, there are always two graphs such that one is a minor of the other.*

Consider the set of all obstructions (topological or minor-order) for a surface Σ. The number of minimal graphs must be finite (or one minimal graph would be a minor of another by the Robertson-Seymour theorem). Thus there is a finite set of obstructions for any given surface.

It is known that there are 103 topological obstructions for the projective plane, of which 35 are minor-order obstructions, as found by Glover, Hunk and Wang [67, 7]. These are often called *Kuratowski subgraphs* for the projective plane. A list of them can be found in Mohar and Thomassen [126]. For the torus, the number of obstructions is in the hundreds of thousands, as shown by Myrvold [128]. From an algorithmic point of view, this is not an effective characterization, as there are too many obstructions.

15.5 Graphs on the torus

Given a 2-cell embedding ψ of a 2-connected graph G on the torus, there must be an essential cycle C in G. Cutting the torus along C^ψ results in a cylinder. Because the cylinder is not a 2-cell, but the embedding is a 2-cell embedding, there must be

another essential cycle C' in G, cutting the cylinder along an axis. Consequently C and C' must intersect, either in a path or a vertex.

DEFINITION 15.12: A *theta-graph* is a graph consisting of two vertices of degree three, connected by three paths of one or more edges each.

A theta-graph is illustrated in Figure 15.17.

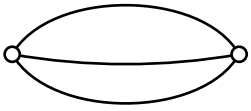


FIGURE 15.17
A theta-graph in schematic form

Thus, $C \cup C'$ must be either a theta-subraph of G, or two cycles with a vertex in common, and ψ is a 2-cell embedding of it. The simplest form of theta-subgraph is a *multigraph* consisting of two vertices connected by three parallel edges. A 2-cell embedding of it on the torus is shown in Figure 15.18. It is often necessary to consider embeddings of graphs with multiple edges and/or loops, as the duals of many graph embeddings (e.g., $K_4, K_5, K_{3,3}$) often have multiple edges, and sometimes loops. We shall always insist that in any embedding of a multigraph:

1. The cycle induced by any loop is an essential cycle (no face is a loop).
2. The cycle induced by any digon is an essential cycle (no face is a digon).

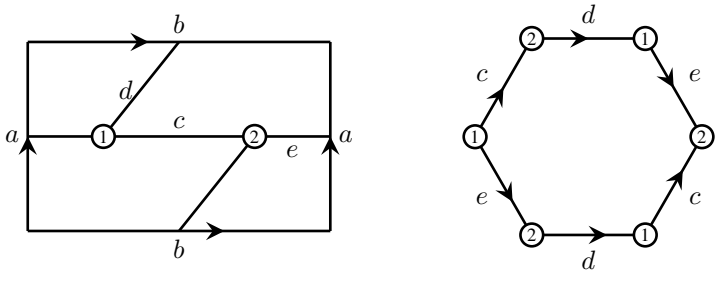


FIGURE 15.18
A 2-cell embedding of a theta-graph

If we cut the torus along the edges of this theta-graph, we find there is one face, a *hexagon*, shown in Figure 15.18 as $c^+d^+e^+c^-d^-e^-$. Thus we see that the embedding of a theta-graph on the torus gives the hexagonal representation (Figure 15.12). The rectangular form corresponds to an embedding of two cycles with a common vertex.

Given an embedding G^{ψ} on the torus, we choose an orientation of the torus, and walk around a vertex v^{ψ} in a small clockwise circle in the surface, and construct the cyclic adjacency list, just as for embeddings in the plane (Section 14.5). This determines a *rotation system* for G, exactly as in the planar case. We will denote a rotation system for a graph embedded on the torus by t. The faces of the embedding are completely determined by t, because Algorithm 12.5.1, FACIALCYCLE(), to find the facial cycles of a planar graph from its rotation system also applies to toroidal rotation systems, or to rotation systems for any orientable surface. Similarly, algorithm CONSTRUCTDUAL() applies equally to toroidal rotation systems as well as other orientable surfaces. Hence we denote a combinatorial toroidal embedding by G^t and its dual by G^{t*}. Now the rotation system determines the faces of the embedding. Hence, we can determine from t whether or not G^t has any faces which are digons or loops, but it cannot determine whether any digons or loops are embedded as essential cycles.

It is convenient to refer to a graph embedded on the torus as a torus *map*.

DEFINITION 15.13: A *torus map* is a combinatorial 2-cell embedding G^t, where t is a rotation system for an embedding of G on the torus, where G is a 2-connected graph.

We begin by embedding planar graphs on the torus. Let G be a 2-connected planar graph that is not a cycle, with a planar rotation system p. By Exercise 15.5.3, G has a theta-subgraph H. Let u and v be the two vertices with degree three in H, and let P_1, P_2, P_3 be the three uv-paths of H. Let w_1, w_2, and w_3 be the first vertices of P_1, P_2, P_3, respectively, adjacent to u. Refer to Figure 15.19.

Theorem 15.10. *Let p be a planar rotation system for G, and let H be a theta-subgraph of G, as described above. Let t be a rotation system constructed from p by interchanging w_2 and w_3 in the cyclic adjacency list of u, and leaving all other vertices the same. Then t is a toroidal rotation system for G.*

Proof. In the embedding G^p in the plane, the three paths P_1, P_2, P_3 divide the plane into three regions. Without loss of generality, let the paths occur in the order illustrated in Figure 15.19. Denote the subgraphs of G contained within the three regions as G_{12} (between paths P_1 and P_2), G_{23}, and G_{31}. Subgraph G_{ij} may have edges connecting it only to P_i and P_j, and to u and v. Construct the hexagonal representation of the torus with P_1, P_2, and P_3 on the boundary of the hexagon, and embed G_{12}, G_{23}, and G_{31} inside the hexagon as planar embeddings, as shown, resulting in a toroidal embedding of G. It is easy to verify from the diagram that any vertex in G_{12}, G_{23}, and G_{31} has the same cyclic adjacency list in the toroidal embedding as in the planar embedding. Similarly any vertex other than u or v of any P_i has the same cyclic adjacencies in both embeddings. The same is also true for v. The only vertex whose cyclic adjacencies differ is u. The adjacency list of u has been arranged so that $w_1 \in P_1$ is followed by the edges to G_{12}, followed by $w_3 \in P_3$, followed by the edges to G_{23}, followed by $w_2 \in P_2$, followed by the edges to G_{31}. The only difference to the planar adjacencies is that w_2 and w_3 have been interchanged for vertex u. □

It is evident from Figure 15.19 that there are several other ways to convert the planar rotation system to a toroidal rotation system.

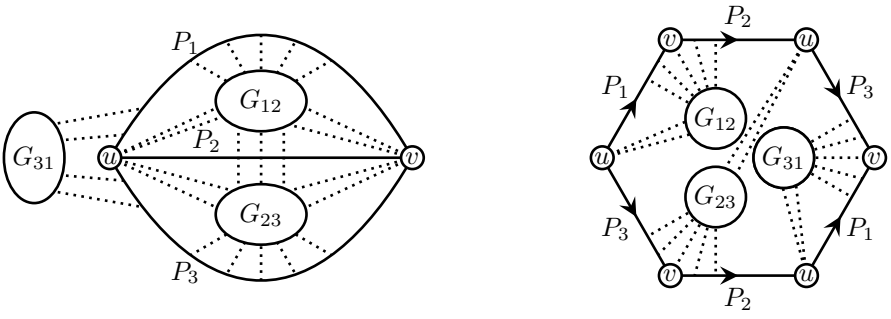


FIGURE 15.19
Constructing a toroidal rotation system

A planar graph also has non-2-cell embeddings on the torus. If a planar graph G is embedded in a disc on the surface of the torus, we will call this a *disc embedding* of G. If a planar graph is embedded such that one of the faces is homeomorphic to a cylinder, we will call this a *cylindrical embedding* of G. A cylindrical embedding on the torus of any graph G determines an embedding on the cylinder, so that G must be planar.

Lemma 15.11. *Every embedding of a non-planar graph on the torus is a 2-cell embedding.*

Proof. A non-2-cell embedding would necessarily be either a disc or cylindrical embedding. But a non-planar graph has no disc or cylindrical embedding. □

Currently, there is no satisfactory algorithm known to determine whether an arbitrary graph can be embedded on the torus, or to find all embeddings, or to characterize all possible embeddings. Whitney's theorem (14.20) on induced non-separating cycles does not apply to embeddings on the torus. There are several simple techniques that are useful in an exhaustive search to find the embeddings. Given two combinatorial embeddings with toroidal rotation systems t_1 and t_2, we need to distinguish whether G^{t_1} and G^{t_2} are equivalent embeddings. In general, two embeddings are considered equivalent if they have the same facial cycles, as the faces can be glued together along the facial boundaries in a unique way to construct the torus. Because the facial cycles are completely determined by the rotation system, we define equivalence in terms of rotation systems. Definitions 14.17 and 14.18 of equivalent embeddings and graph orientability apply equally well to toroidal graphs as to planar graphs, using a toroidal rotation system t in place of a planar rotation system p. We summarize the definitions as follows:

DEFINITION 15.14:

1. Embeddings G^{ψ_1} and G^{ψ_2} are *homeomorphic embeddings* if there is a homeomorphism of the torus mapping G^{ψ_1} to G^{ψ_2}. Otherwise they are distinct.
2. Embeddings G^{t_1} and G^{t_2} are *isomorphic* if there is an automorphism of G which induces a mapping of t_1 to t_2.
3. Embeddings G^{t_1} and G^{t_2} are *equivalent embeddings* if there is an automorphism of G which induces a mapping of t_1 to t_2 or $\overline{t}_2$, where $\overline{t}_2$ is obtained by reversing the cycles of t_2.
4. Embedding G^t is a *non-orientable embedding* if there is an automorphism of G inducing a mapping of t to $\overline{t}$. Otherwise it is an *orientable embedding*.

Now an embedding G^{ψ_1} determines a rotation system t_1. Homeomorphic embeddings G^{ψ_1} and G^{ψ_2} determine equivalent combinatorial embeddings G^{t_1} and G^{t_2}, because a homeomorphism can be either orientation preserving or orientation reversing. Conversely, if G^{t_1} and G^{t_2} are equivalent combinatorial embeddings of G, then they have the same facial cycles (up to orientation). The facial cycles can be glued together to construct a curvilinear polyhedron representing the torus. Therefore, topological embeddings G^{ψ_1} and G^{ψ_2} can be constructed from G^{t_1} and G^{t_2}, so that G^{ψ_1} and G^{ψ_2} are homeomorphic. This gives:

Theorem 15.12. *Topological embeddings G^{ψ_1} and G^{ψ_2} are homeomorphic if and only if the corresponding combinatorial embeddings G^{t_1} and G^{t_2} are equivalent.*

Now the homeomorphism between G^{ψ_1} and G^{ψ_2} was constructed by gluing curvilinear polygons (the faces of the embeddings) together along common edges; that is, it involves cutting and pasting the torus. For example, the two embeddings G^{t_1} and G^{t_2} shown in Figure 15.20 are equivalent, which can easily be verified from the rotation systems. However, they are homeomorphic only by cutting the torus along a non-contractible cycle to create a cylinder, then twisting one end of the cylinder by 360 degrees, and then re-gluing the cylinder to create a torus. It is not possible to transform one into the other without cutting and pasting.

It is also possible to *define* isomorphism of maps in terms of the facial cycles, as will be done for the projective plane, as the facial walks of G^t are completely determined by the rotation system t. Given the collection of facial walks, which are oriented, there is only one way to glue them together. The result is the torus, containing an embedding of G. If the vertices of G are permuted by some $\theta \in \text{AUT}(G)$, the vertices of the facial walks will also be permuted, thereby producing an equivalent embedding (possibly the very same embedding). For example, two embeddings of $K_{3,3}$ are shown in Figure 15.2. Now $|\text{AUT}(K_{3,3})| = 72$. If the vertices of $K_{3,3}$ in these two embeddings are permuted by any of these 72 automorphisms, equivalent embeddings will be produced, whose facial walks can be obtained by permuting the facial walks in Figure 15.2. We state a definition of isomorphism of torus maps, equivalent to 15.14.

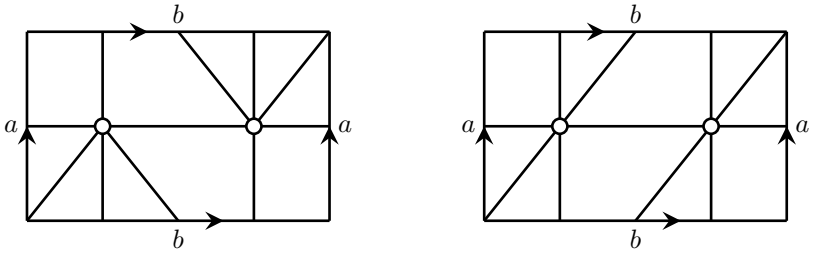


FIGURE 15.20
Two equivalent embeddings

DEFINITION 15.15: Torus maps G^{t_1} and G^{t_2} are isomorphic if there is a permutation of $V(G)$ that maps the collection of facial walks of G^{t_1} to that of G^{t_2}.

This definition uses the dual maps of G^{t_1} and G^{t_2} to determine isomorphism. For if we number their facial walks, then rotation systems of the dual maps are completely determined by the facial walks.

For graphs on a small number of vertices, it is possible to distinguish inequivalent embeddings by inspection. However, even for K_5 and K_6, it is reasonably difficult to determine the inequivalent embeddings by hand. One technique that helps is the dual graph – if G^{t_1} and G^{t_2} have non-isomorphic dual graphs, then the embeddings are distinct. More generally, we can use the *medial digraph* (Definition 14.20) to distinguish embeddings and orientations. It can also be used to determine the symmetries (automorphisms) of an embedding. The medial digraph was defined for planar rotation systems, but the definition is also valid for toroidal rotation systems. We use $M(G^t)$ to denote the medial digraph of G with a toroidal rotation system t. The medial digraph was defined for multigraphs. If we want to allow for loops as well, then the definition must be modified slightly (Exercise 15.5.7). Usually graph isomorphism software is necessary to make effective use of the medial digraph.

Theorem 15.13. *Torus embeddings G^{t_1} and G^{t_2} are isomorphic if and only if their medial digraphs $M(G^{t_1})$ and $M(G^{t_2})$ are isomorphic.*

Proof. If the torus embeddings G^{t_1} and G^{t_2} are isomorphic, their rotation systems t_1 and t_2 can be put into one-to-one correspondence. This determines an isomorphism of their medial digraphs. Conversely, if $M(G^{t_1})$ and $M(G^{t_2})$ are isomorphic, observe that every isomorphism from $M(G^{t_1})$ to $M(G^{t_2})$ must map $V(G)$ to $V(G)$. The cyclic ordering of the vertices adjacent to each vertex v is determined by a directed cycle in $M(G^{t_1})$ and $M(G^{t_2})$. This determines the rotation systems G^{t_1} and G^{t_2}. □

An embedding G^t is *orientable* if G^t and $G^{\overline{t}}$ are not isomorphic. It is *nonorientable* if G^t and $G^{\overline{t}}$ are isomorphic. Any automorphism of $M(G^t)$, where G^t is a torus embedding, induces a permutation of $V(G)$. These permutations constitute the automorphism group of G^t, similar to the situation for planar maps 14.22.

DEFINITION 15.16: Let G^t be a torus map. The *orientation preserving automorphism group* of G^t is $\text{AUT}(G^t)$, the group induced on $V(G)$ by the automorphisms of $M(G^t)$. The *full automorphism group* of G^t is $\text{AUT}^+(G^t)$ consisting of $\text{AUT}(G^t)$, plus those automorphisms that map G^t to $G^{\overline{t}}$.

With 3-connected planar maps G^p, Whitney's theorem tells us that there is a unique planar embedding, so that $\text{AUT}^+(G^p) = \text{AUT}(G)$. A similar result for torus maps is not known. For example, there are five torus embeddings of the graph of the cube, shown in Figure 15.21. The automorphism group of the cube has order 48. The embeddings shown have automorphism groups of orders varying from 2 to 24.

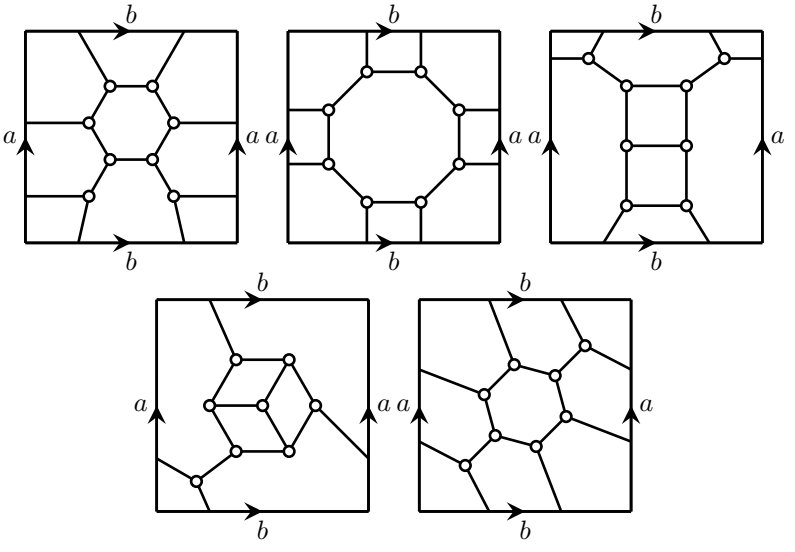


FIGURE 15.21
The five embeddings of the cube on the torus

Theorem 15.14. *There are two inequivalent embeddings of $K_{3,3}$ on the torus. They are both non-orientable.*

Proof. Take the rectangular representation $a^+b^+a^-b^-$ of the torus. An embedding of $K_{3,3}$ on the torus must be a 2-cell embedding, so that there must be an essential Jordan curve intersecting the a-curve and another intersecting the b-curve. Let G^t be an embedding of $K_{3,3}$ on the torus. Now $K_{3,3}$ is bipartite, with six vertices, so that the corresponding essential cycles in G^t must determine a theta subgraph. There are three possible theta subgraphs, illustrated in Figure 15.22.

The embeddings of the theta graphs have just one face, with a facial walk of length 12 or 14. These are illustrated in Figure 15.23. In the third diagram of Figure 15.23, the edges $(1, 6)$, $(3, 6)$, and $(5, 6)$ must be added. It is easy to see that there are two inequivalent ways to do this – one in which the edges to vertex 6 are equally

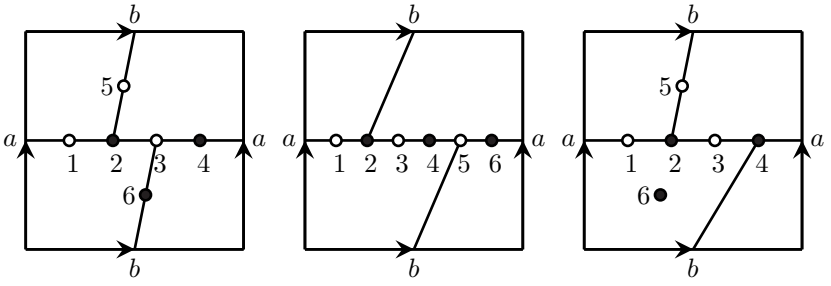


FIGURE 15.22
Three possible theta subgraphs of $\boldsymbol{K_{3,3}}$ on the torus

spaced by $2\pi/3$ radians, and one in which three edges are spaced by $\pi/3$ radians. These give the two embeddings of Figure 15.2, the first with three faces which are hexagons, and the second in which two faces are quadrilaterals.

In the first theta graph, the missing edges of the $K_{3,3}$ are $(1,6)$ and $(4,5)$. The possible places to embed them are shown as dotted lines in the diagram. In the second theta graph the missing edges are $(1,4)$ and $(3,6)$. In each case, non-intersecting edges must be chosen. The result is always the embedding with three hexagons, or the embedding with two quadrilaterals. It is easy to see that the embeddings of Figure 15.2 are non-orientable, as flipping the rectangle upside down, and possibly rotating it, produces an identical diagram. □

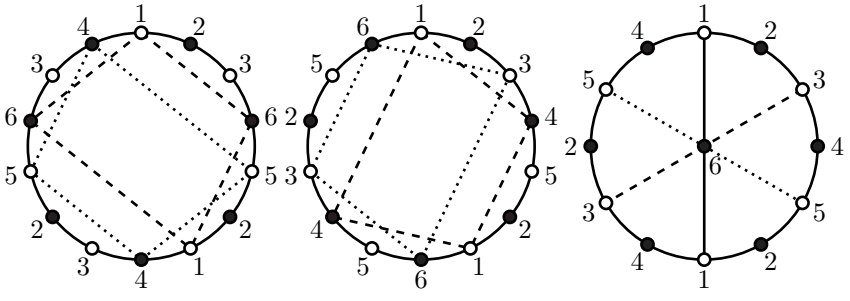


FIGURE 15.23
The face of a theta graph

More generally, to find all embeddings of a graph on the torus, we can proceed as follows. Suppose that G is a 2-connected non-planar graph. Choose a theta-subgraph H of G. We would like H to have as many edges as reasonably possible. We can do this by hand for small graphs. With larger graphs, a depth-first search can be used to find a theta-subgraph with a large number of edges. Every embedding of G on the torus induces an embedding of H. It will be either a 2-cell embedding, a cylindrical embedding, or a disc embedding. We start with a 2-cell embedding of H, and proceed as in Theorem 15.10 to find all ways of extending the embedding of H to G. This usually gives a number of embeddings. We then proceed to the cylindrical and disc embeddings of H. In each case, all possible ways of extending H to a 2-cell embedding must be exhaustively considered. For each embedding t, we construct $M(G^t)$, and compare the medial digraphs found for isomorphism, using graph isomorphism software.

If G is a 3-connected non-planar graph, we proceed recursively. We choose a vertex v and find all embeddings of $G - v$. Let the adjacent vertices to v be $u_1, u_2, \ldots, u_k$. If $G - v$ is non-planar, then every embedding of it is a 2-cell embedding. If $u_1, u_2, \ldots, u_k$ are all on the same facial cycle in some embedding of it, we can add vertex v to get an embedding of G, possibly in several ways. If $G - v$ is planar, instead we first find a $TK_{3,3}$ in G with as many edges as reasonably possible (assuming a $TK_{3,3}$ exists). For each embedding of $TK_{3,3}$ in the torus, we exhaustively consider all possible ways of extending it to an embedding of G, and then use medial digraphs to compare the results.

For example, consider the graph K_4. If uv is an edge of K_4, then $K_4 - uv$ is a theta-graph. We easily find that there are exactly two 2-cell embeddings of K_4 on the torus. We then consider K_5, one of the few non-planar graphs that does not contain $TK_{3,3}$. If v is a vertex of K_5, then $K_5 - v \cong K_4$. For each embedding of K_4 on the torus, including cylindrical and disc embeddings, we find all ways of adding v to the embedding. The result is six embeddings of K_5, of which three are orientable and three non-orientable. This is most easily determined using the medial digraph to distinguish embeddings. We proceed to K_6 by looking for a face of K_5 containing all five vertices. We find there are four inequivalent embeddings of K_6, of which two are orientable and two non-orientable. Exactly one of these has all six vertices on a common face. This gives one embedding of K_7, shown in Figure 15.24. It is an orientable embedding. Its dual is also shown. The dual is known as the *Heawood graph*.

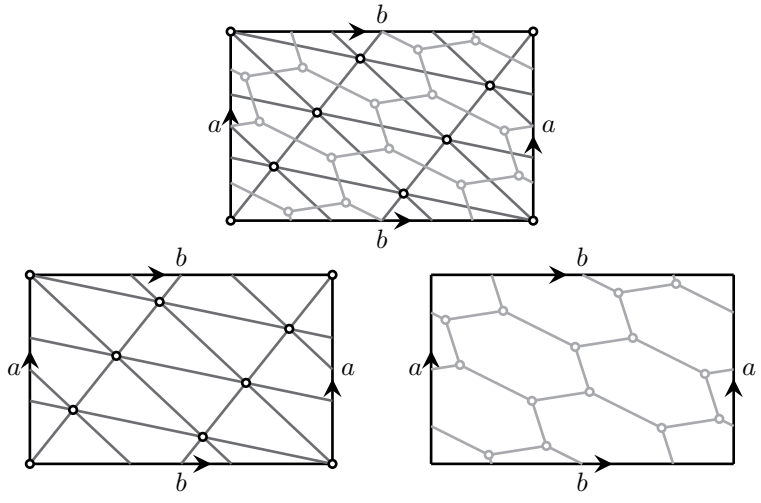
Exercises

15.5.1 Use cutting and pasting to convert the representation $c^+d^+e^+c^-d^-e^-$ of the torus to normal form.

15.5.2 Construct the duals of the embeddings of $K_{3,3}$ on the torus of Figure 15.2.

15.5.3 Show that every 2-connected graph that is not a cycle has a theta-subgraph.

15.5.4 Describe $O(\varepsilon)$ depth-first and breadth-first search algorithms to find a theta-subgraph of a 2-connected graph G.



K_7 Heawood graph

FIGURE 15.24
$\boldsymbol{K_7}$ and its dual, the Heawood graph, on the torus

15.5.5 Construct the two distinct embeddings of $K_{3,3}$ on the hexagonal form of the torus.

15.5.6 Verify that the two embeddings shown in Figure 15.20 are equivalent, and that the torus must be cut and pasted to construct a homeomorphism.

15.5.7 Show how to modify the definition of a medial digraph to allow for embeddings with loops (subdivide a loop with two vertices), and prove that it works.

15.5.8 Construct all distinct embeddings of K_4, K_5, and K_6 on the torus.

15.5.9 Construct all distinct embeddings of the 3-prism on the torus. Begin with a theta-graph containing all six vertices.

15.5.10 Construct all distinct embeddings of the Petersen graph on the torus. Begin with a theta graph containing all 10 vertices.

15.5.11 Determine which graphs are shown in the toroidal embeddings of Figure 15.25. Determine the dual graphs. (Note: None of the graph edges follow the boundary of the rectangles.)

15.5.12 Determine whether the embeddings in Figure 15.21 are orientable, and determine the orders of their automorphism groups.

15.5.13 Verify that the graphs in Figure 15.26 are distinct embeddings of the same graph. Do you recognize this graph? Are these embeddings orientable? Find the duals of both embeddings, and determine what graphs they are.

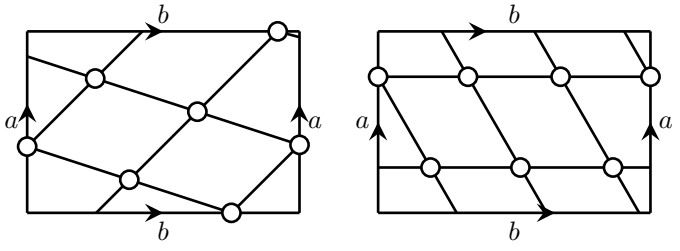


FIGURE 15.25
Two torus maps

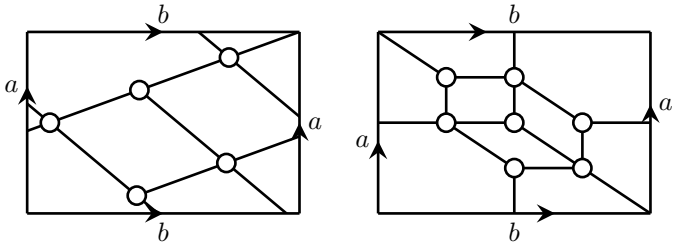


FIGURE 15.26
Two torus maps

15.5.1 Platonic maps on the torus

The embedding of K_7 in Figure 15.24 shows a triangulation of the torus in which each vertex has degree six. By translating it repeatedly horizontally and vertically, we obtain a symmetric tiling of the plane by triangles. Its dual gives a symmetric hexagonal cover of the plane in which three hexagons meet at each vertex. The graphs of Figure 15.25 give symmetric tilings by parallelograms. These embeddings all belong to families of graphs with these properties. We will call them *Platonic maps*.

Let G be a k-regular torus map on n vertices whose dual map is ℓ-regular. For the plane, such graphs are the graphs of the Platonic solids. Then $nk = 2\varepsilon = \ell f$. Using Euler's formula for the torus $n + f - \varepsilon = 0$, we obtain

$$\frac{1}{k} + \frac{1}{\ell} = \frac{1}{2}$$

The only integral solutions are $(k, \ell) = (4, 4), (3, 6)$, and $(6, 3)$. Clearly the last two are duals of each other. The graphs of Figure 15.25 are examples of the $(4, 4)$-pattern.

Consider a torus map G^t in which each vertex has even degree. Choose any edge uv. The incident edges at v are cyclically ordered by t. Let $\text{DEG}(v) = 2i$. The *diagonally opposite* edge to uv is vw, the i^{th} edge following uv in $t(v)$. Given a vertex v_0, with adjacent vertex v_1, we construct a *diagonal path* $v_0, v_1, v_2, \ldots$ by always

choosing $v_i v_{i+1}$ as the diagonally opposite edge to $v_{i-1}v_i$. Eventually a vertex must repeat, creating a cycle. Let the cycle be $C = (v_0, v_1, v_2, \ldots, v_m)$. C is a *diagonal cycle* if every edge is the diagonally opposite edge to its previous edge.

In the torus maps of K_7 (Figure 15.24) and Figure 15.25 there are many diagonal cycles. They are drawn as straight lines diagonally across the rectangle. As can be seen in K_7, a single diagonal cycle may wind around the torus several times.

Suppose now that G^t is a Platonic graph on the torus, with parameters $(6, 3)$ or $(4, 4)$. Consider a cycle $C = (v_0, v_1, v_2, \ldots, v_m)$ constructed by following a diagonal path.

Lemma 15.15. *If $C = (v_0, v_1, v_2, \ldots, v_m)$ is a diagonal cycle in G^t, then C is an essential cycle.*

Proof. Suppose first that G has parameters $(4, 4)$, and suppose that C is contractible, with interior $\text{INT}(C)$. For each $v_i \in C$, there is one adjacent vertex in $\text{INT}(C)$. This is illustrated in Figure 15.27. Because each face of G^t has degree four, the interior adjacent vertices to $v_0, v_1, v_2, \ldots, v_m$ form another diagonal cycle C' in $\text{INT}(C)$. Interior to C' is another diagonal cycle, etc., leading to an infinite sequence of diagonal cycles, a contradiction. Therefore C must be an essential cycle. If G has parameters $(6, 3)$, the argument is nearly identical, except that each $v_i \in C$ has two adjacent vertices in $\text{INT}(C)$. Because G^t is a triangulation, we again find C' in $\text{INT}(C)$, and so forth. □

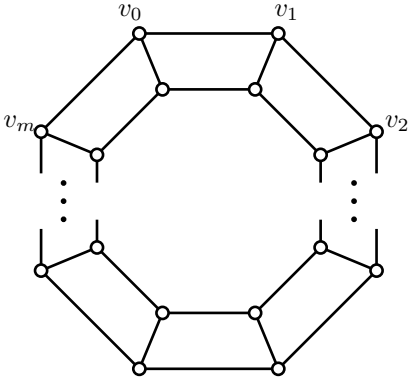


FIGURE 15.27
A diagonal cycle with $(k, \ell) = (4, 4)$

The proof of Lemma 15.15 also shows how to draw G^t. Given a diagonal cycle C_0, a sequence of "parallel" adjacent diagonal cycles is determined, $C_0, C_1, C_2, \ldots$. For any $v_0 \in C_0$, an edge not on C_0 can then be selected, and a diagonal cycle containing it can be constructed. We find that the edges of G can be partitioned into "orthogonal" diagonal cycles $C'_0, C'_1, \ldots$. Each C_i winds around the torus one or more

times, intersecting each C'_j in a regular pattern, as can be seen from Figures 15.24 and 15.25.

If $C = (v_0, v_1, v_2, \ldots, v_m)$ is any cycle constructed by following a diagonal path in a Platonic map, then the argument of Lemma 15.15 can be used to show that C must be a diagonal cycle. The only way in which C may fail to be a diagonal cycle is if one pair of edges, say $v_m v_0$ and $v_0 v_1$, are not diagonal edges. Suppose that G has parameters $(4, 4)$. We then find that v_0 has either 0 or 2 adjacent vertices to the right of C. Because every face has degree four, the parallel cycle C' is either shorter than C or longer than C, by two edges. If it is longer than C, then the parallel cycle C'' is again longer than C' by two edges, and so forth. As this leads to a contradiction, suppose that C' is two edges shorter than C. Then C'' is again two edges shorter than C', etc. Eventually we find a cycle of length four or five for which no parallel cycle can exist. If G has parameters $(6, 3)$, the argument is similar.

15.5.2 Drawing torus maps, triangulations

Read's algorithm for drawing a planar graph, given a rotation system, can be extended to torus maps. Let G^t be a 2-connected combinatorial embedding, with no vertices of degree two. Suppose that G has no loops, and that if there are multiple edges, no face is a digon. If G^t is not a triangulation, we can triangulate it by adding diagonal edges across non-triangular faces, so that no loops or digon faces are created. The smallest possible triangulation of the torus is shown in Figure 15.28. We denote it by T_3. It can be constructed from the theta-graph shown in Figure 15.18 by adding one vertex w, adjacent to each vertex u and v three times. Notice that T_3 is a 6-regular graph, whose faces are all triangles. It is the unique triangulation of the torus on three vertices (Exercise 15.5.4).

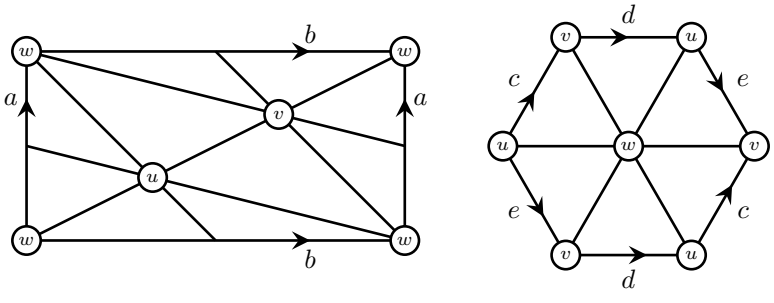


FIGURE 15.28
The triangulation $\boldsymbol{T_3}$

There is a triangulation on four points, denoted T_4, which can be constructed from the rectangular form of the torus. A 3-vertex graph consisting of two digon cycles (u, v) and (u, w) with a common vertex u is 2-cell embedded on the torus. There is a single face, of degree eight. A fourth vertex x is placed in this face, and joined to each vertex on the boundary of the face. It is illustrated in Figure 15.29. In this diagram, the sides a and b of the rectangle are also graph edges. Notice that T_4 has two vertices (u and x) of degree eight, and two (v and w) of degree four.

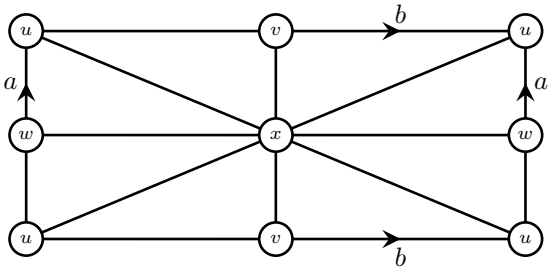


FIGURE 15.29
The triangulation $\boldsymbol{T_4}$

Suppose that G has n vertices, with n_3 of degree three, n_4 of degree four, etc. Then because G^t is a triangulation, we have $3f = 2\varepsilon$. Euler's formula then gives $\varepsilon = 3n$, and:

$$3n_3 + 2n_4 + n_5 = n_7 + 2n_8 + 3n_9 + \ldots$$

Lemma 15.16. *Either there is a vertex of degree three, four, or five, or else all vertices have degree six.*

Now any triangulation in which all vertices have degree six is a Platonic map of type $(6, 3)$, and we know how to draw it as a tiling of the plane. Otherwise, there is a vertex of degree three, four, or five. We can use a modification of the algorithm REDUCEGRAPH() of Section 15.5.2 to reduce the triangulation G^t on n vertices to a triangulation on $n - 1$ vertices, until either T_3 or T_4 results, or a 6-regular triangulation results. We must ensure that the reduction does not create any loops or digon faces.

Suppose that vertex u of G has degree three, four, or five, and suppose that $n \geq 4$. If $\text{DEG}(u) = 3$, then $G^t - u$ is a triangulation of the torus on $n - 1$ vertices. If $\text{DEG}(u) = 4$, suppose that u is adjacent to v, w, x, y, in cyclic order. If at least three of these vertices are distinct, then at least one of the diagonals of the 4-cycle (v, w, x, y) has distinct endpoints. Suppose it is vx. Then $G^t - u + vx$ will be a triangulation on $n-1$ vertices, without loops or digon faces. Otherwise there are only two distinct vertices in the 4-cycle, which is then (v, w, v, w); that is, there are four parallel edges connecting v and w. Three of these parallel edges form a theta-graph, whose only embedding has a single face, a hexagon, shown in Figure 15.30. The fourth parallel edge cuts the hexagon into two quadrilaterals, one of which contains u.

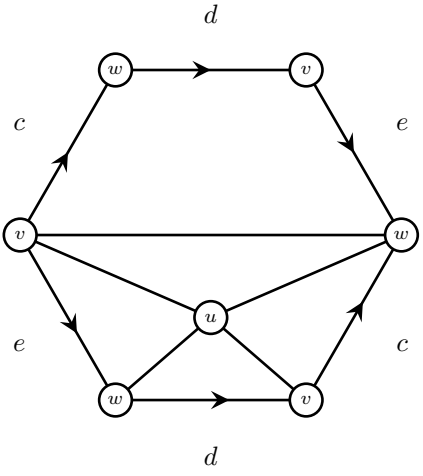


FIGURE 15.30
Reducing a triangulation, DEG$(u) = 4$

The remaining vertices of G are located in the other quadrilateral. If $n = 4$, then the map can only be the triangulation T_4, with v and w as the two vertices of degree eight. If $n \geq 5$, there are at least two other vertices in the other quadrilateral. This quadrilateral and the vertices it contains determine a planar graph, which must have several vertices of degree three, four, or five. We choose one of these, and delete it instead of u.

If DEG$(u) = 5$, let u be adjacent to v, w, x, y, z, in cyclic order. If v, x, and y are distinct, we proceed as in the planar case, deleting u and adding two diagonals across the pentagon. Otherwise, we can assume that $v = x$, because G has no loops.

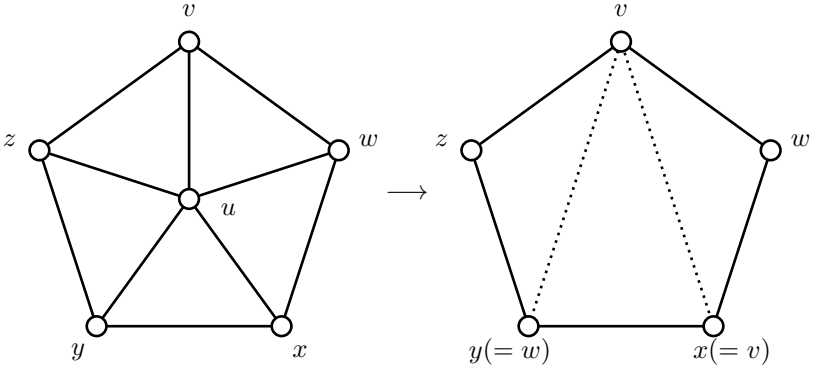


FIGURE 15.31
Reducing a triangulation, DEG$(u) = 5$

If w, y, and z are distinct, then we can add the diagonals wy and wz to get a triangulation. Otherwise we can assume that $w = y$. But then z, w, and x are distinct, so that we can add the diagonals zw and zx to obtain a triangulation with no loops or digon faces. There are always at least three distinct vertices on the boundary of the pentagon. This gives the following theorem:

Theorem 15.17. *Let G^t be a torus map on $n \geq 4$ vertices which is not 6-regular, with no loops or digon faces. Then G^t can be reduced to one of the following:*

1. *The triangulation T_3*
2. *The triangulation T_4*
3. *A 6-regular triangulation*

Algorithm 15.5.1 is input a triangulation of the torus, G, on $n \geq 4$ vertices with rotation system t, with no loops or digon faces. It constructs a triangulation G' on $n-1$ vertices, whenever possible. It can be used to successively reduce G^t to one of T_3, T_4, or a 6-regular triangulation. Drawings of T_3 and T_4 on the torus are shown in Figures 15.28 and 15.29. We can use these coordinatizations in a rectangle as topological embeddings. If a 6-regular triangulation is obtained, we can use diagonal cycles to obtain a coordinatization of it. These embeddings have no loops, and no digon faces. Every digon is embedded as an essential cycle. We then replace the deleted vertices in reverse order, exactly as in the planar case of READSALGORITHM(), using the visible region to assign coordinates to the deleted vertex. The result is a straight-line drawing of G^t on the torus; that is, a topological embedding G^{ψ}.

We summarize this as follows:

Theorem 15.18. *Every torus map has a straight-line embedding in the rectangle and hexagon models of the torus.*

Exercises

15.5.1 Find the duals of the embeddings of the triangulations T_3 and T_4 shown in Figures 15.28 and 15.29. What graphs are they?

15.5.2 Find the two tilings of the plane determined by a Platonic map of $K_{4,4}$.

15.5.3 Find the tiling of the plane determined by a Platonic map of $C_3 \times C_3$.

15.5.4 Show that T_3 is the unique triangulation of the torus on three vertices.

15.5.5 Show that there are two distinct triangulations of the torus on four vertices.

Algorithm 15.5.1: REDUCETORUSMAP(G, t)

if $G = T_3$ **or** $G = T_4$ **or** G is 6-regular **return** (**null**)
if there is a vertex u with DEG$(u) = 3$
then
- let $t(u) = (uv, uw, ux)$
- $G' \leftarrow G - u$
- **return** (G')

if there is a vertex u with DEG$(u) = 4$
then
- let $t(u) = (uv, uw, ux, uy)$
- **if** $v = x$ **and** $w = y$
  - **then** pick a new u from $V(G) - \{u, v, w\}$ of degree 4 or 5

if DEG$(u) = 4$
then
- let $t(u) = (uv, uw, ux, uy)$
- **if** $v \neq x$
  - **then**
    - $G' \leftarrow G' - u + vx$
    - vx replaces vu in $t(v)$ and xv replaces xu in $t(x)$
  - **else**
    - $G' \leftarrow G - u + wy$
    - wy replaces wu in $t(w)$ and yw replaces yu in $t(y)$
- **return** (G')

pick u of degree 5, let $t(u) = (uv, uw, ux, uy, uz)$
if v, x and y are distinct
then
- $G' \leftarrow G - u + vx + vy$
- vx, vy replace vu in $t(v)$
- xv replaces xu in $t(x)$ and yv replaces yu in $t(y)$

else if w, y and z are distinct
then
- $G' \leftarrow G - u + wy + wz$
- wy, wz replace wu in $t(w)$
- yz replaces yu in $t(y)$ and zw replaces zu in $t(z)$

else if x, z and v are distinct
then
- $G' \leftarrow G - u + xz + xv$
- xz, xv replace xu in $t(x)$
- zv replaces zu in $t(z)$ and vx replaces vu in $t(v)$

else if y, v and w are distinct
then
- $G' \leftarrow G - u + yv + yw$
- yv, yw replace yu in $t(y)$
- vw replaces vu in $t(v)$ and wy replaces wu in $t(w)$

else if z, w and x are distinct
then
- $G' \leftarrow G - u + zw + zx$
- zw, zx replace zu in $t(z)$
- wx replaces wu in $t(w)$ and xz replaces xu in $t(x)$

return (G')

15.6 Coordinate averaging

Coordinate averaging for torus maps is very similar to coordinate averaging for planar maps. Use the rectangular model of the torus. Let the coordinates of vertex i be (x_i, y_i), which are located inside the rectangle representing the torus. There is no outer face on the torus, and some edges of the graph "wrap around" the torus, that is, they may cross the boundary of the torus rectangle. Adapting the coordinate averaging algorithm from the plane to the torus, we obtain the following.

Algorithm 15.6.1: COORDINATEAVERAGING(G^t)

comment: Given a straight-line drawing of a torus map G^t, on $n \geq 4$ vertices, with coordinates (x_i, y_i) for vertex i, perform coordinate averaging.

let $F_1, F_2, \ldots, F_f$ denote the facial walks of G^t
for $j \leftarrow 1$ **to** f
do use algorithm FACIALCYCLE(F_j) to sum (x_i, y_i), for all $i \in F_j$
$(u_j, v_j) \leftarrow$ average of the coordinates of the vertices of F_j
comment: (u_j, v_j) are now coordinates of the dual G^{t*}

for $i \leftarrow 1$ **to** n
do sum (u_j, v_j) for all faces F_j containing vertex i
$(x_i, y_i) \leftarrow$ average of the coordinates of the F_j containing i

On the torus, even a 3-conneced graph can have facial walks in which vertices or edges are repeated, so that the facial walks F_j may not be cycles. As in the planar case, FACIALCYCLE() is used to walk around each F_j, summing the coordinates of the vertices on F_j, so as to compute their average. Let R denote the rectangle representing the torus. When an edge uv crosses a boundary of R, additional steps are necessary. Let ρ be the height of R, and σ its width. If an edge uv crosses the upper boundary of R as F_j is being traced, then ρ must be added to y_i, for each subsequent vertex i encountered on F_j. If uv crosses the lower boundary of R, then ρ must be subtracted from y_i, for each subsequent vertex encountered. A similar action is needed for x_i when the right or left boundary of R is crossed, adding or subtracting σ. Let the resulting coordinates representing the face F_j be (u_j, v_j). This point may not be inside R. In this case, ρ is added or subtracted from v_j, and σ is added or subtracted from u_j so as to place (u_j, v_j) inside R.

Coordinates (u_j, v_j) are calculated for every face F_j. Then a second loop recalculates (x_i, y_i), by performing the same operation, but in the dual. This constitutes one application of COORDINATEAVERAGING(). Clearly this takes $O(n)$ steps. This algorithm can be iterated several times, to produce an improved drawing.

An example appears in Figure 15.32. The graph on the left in the diagram is a torus embedding that could have been produced by the algorithm

REDUCETORUSMAP(G^t). The graph on the right is after six applications of coordinate averaging. A constant number of applications of coordinate averaging still results in a $O(n)$ algorithm.

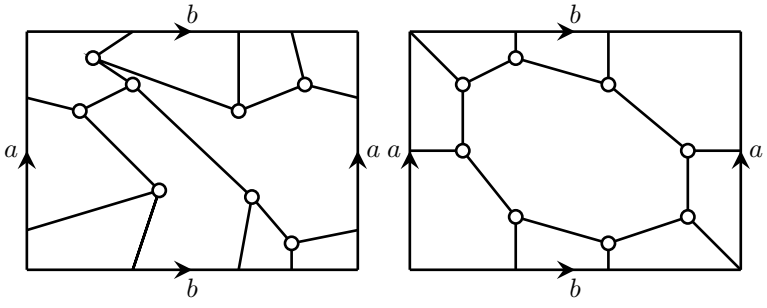


FIGURE 15.32
Coordinate averaging

Coordinate averaging can be expressed in terms of matrices, similar to barycentric coordinates for the *vertex-face-incidence graph* G^F. However, the edges which cross a boundary of R complicate the equations. As with the planar case, coordinate averaging in G^t often hilights symmetries of G.

15.7 Graphs on the projective plane

The projective plane is most conveniently represented as a disc with antipodal points identified. This is equivalent to the digon form c^+c^+ of the projective plane shown in Figures 15.8 and 15.10. An embedding of K_6 and its dual are shown in Figure 15.33. It is easy to verify that the dual of K_6 on the projective plane is the *Petersen graph*. As before, we shall only be concerned with *2-cell embeddings of 2-connected graphs*.

Now it can be a little tricky to visualize the faces of an embedding on the projective plane, because the projective plane is non-orientable. Each point on the circumference of the disc is identified with its antipodally opposite point. When an edge of the graph meets the disc boundary, it continues from the antipodal point. But the region immediately to the right of the edge as it meets the boundary is identified with the region immediately to the *left* of the antipodal point. A consequence is that rotation systems must be defined somewhat differently for non-orientable surfaces, and the algorithm FACIALCYCLE() which constructs the faces of an embedding must be modified.

Let G^ψ be an embedding of a 2-connected graph G in the projective plane, and let u be a vertex of G. If we walk around u^ψ in a small clockwise circle, we encounter the incident edges in a certain cyclic order, say $uv_1, uv_2, \ldots, uv_k$. If we walk along

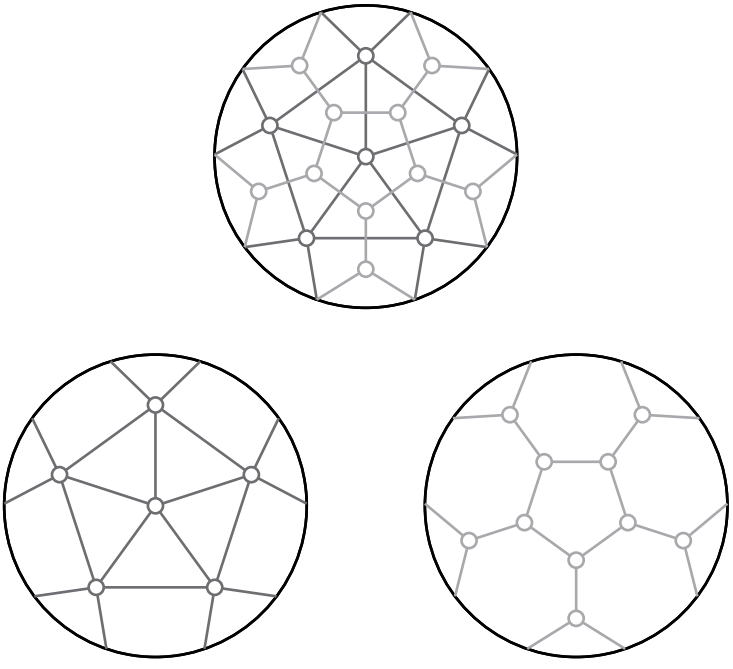


FIGURE 15.33
$\boldsymbol{K_6}$ and its dual, the Petersen graph, on the projective plane

the edges of G^ψ, always staying within the disc, then the embedding appears exactly like a planar embedding. If we traverse an edge that crosses the boundary of the disc, and continue on until we reach u^ψ again, we find that the cyclic order of the incident edges at u^ψ has been reversed.

Consequently a rotation system must be defined differently for a non-orientable surface. The projective plane is represented as a disc. We choose an orientation for this disc. Then given any vertex u, we walk around u^ψ in a small clockwise circle, and construct a cyclic list of incident edges. We assign a *signature* to each edge uv, denoted $\text{SGN}(uv)$. If an edge $(uv_i)^\psi$ crosses the boundary of the disc, then $\text{SGN}(uv_i) = -1$. Otherwise it is $+1$. The signature does not depend on the direction in which the edge is traversed. In this way, the embedding ψ determines a signed rotation system. In order for this to be well defined, it is necessary that vertex u *does not lie on the boundary of the disc*. Before constructing a rotation system, all vertices that lie on the boundary must be moved slightly so that they are no longer on the boundary. We will always assume that this has been done. However, once signatures have been assigned to the edges, we can always move u again, so that it is infinitesi-

TABLE 15.1
The two rotation systems for $K_{3,3}$ on the projective plane, corresponding to the two embeddings in Figure 15.34

(I)			(II)		
$\pi(1)$	$=$	$(12, 16, -14)$	$\pi(1)$	$=$	$(14, 16, -12$
$\pi(2)$	$=$	$(21, -25, 23)$	$\pi(2)$	$=$	$(23, -21, 25)$
$\pi(3)$	$=$	$(32, -36, 34)$	$\pi(3)$	$=$	$(34, 32, 36)$
$\pi(4)$	$=$	$(43, -41, 45)$	$\pi(4)$	$=$	$(41, -45, 43)$
$\pi(5)$	$=$	$(54, -52, 56)$	$\pi(5)$	$=$	$(52, -54, 56)$
$\pi(6)$	$=$	$(61, 65, -63)$	$\pi(6)$	$=$	$(61, 63, 65)$

mally close to the boundary. We also assume that each edge that crosses the boundary of the disc crosses it *exactly once*. If necessary, edges can be subdivided to ensure that this is the case.

We will use π to denote a rotation system for an embedding on the projective plane. For each vertex u, $\pi(u)$ denotes a cyclic list of *signed* incident edges. Two embeddings of $K_{3,3}$ are shown in Figure 15.34. The rotation systems corresponding to them are shown in Table 15.1. Although the rotation systems are different, the embeddings are equivalent.

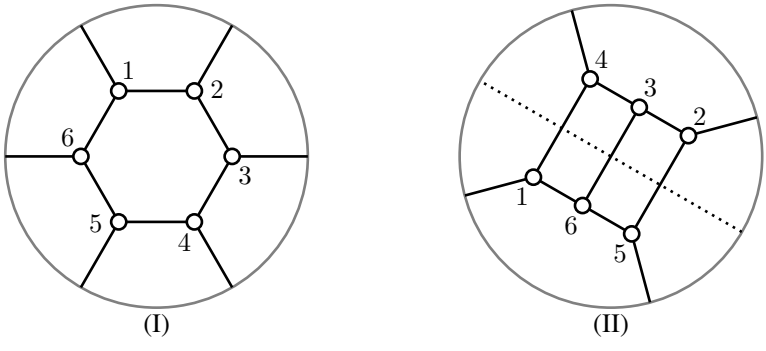


FIGURE 15.34
Two equivalent embeddings of $\boldsymbol{K_{3,3}}$ on the projective plane

Given a topological embedding G^ψ, this method of defining a rotation system for G is not unique, as it depends on the disc chosen to represent the projective plane. With an orientable surface, this situation does not arise. We must show that a signed rotation system uniquely determines the faces of an embedding, and that all rotation systems corresponding to ψ determine the same faces. In the embedding on the right of Table 15.1, we can cut the projective plane along the non-contractible Jordan curve indicated by the dotted line, call it C. We then flip one-half of the disc over, and glue the two pieces along the common boundary. We obtain another disc representing the projective plane, with antipodal points identified. This gives another rotation system π' for G. In Table 15.1, the edges crossed by C will now have a signature of -1; edges which previously had a signature of -1 will now have $+1$. It is easy to see that with respect to the new disc, the embedding will have a face which is a 6-cycle $(1, 2, 3, 4, 5, 6)$ with "spokes" to the boundary of the disc, exactly like the embedding on the left. Thus the embeddings are equivalent.

If π is a signed rotation system determined by an embedding G^ψ and a disc representation of the projective plane, we call G^π a combinatorial embedding. G is said to be *projective planar*, or simply *projective*. As we shall see, the faces of G^ψ are completely determined by G^π.

DEFINITION 15.17: A *projective map* is a combinatorial embedding G^π of a 2-connected graph G on the projective plane, where π is a signed rotation system corresponding to a disc representation of the projective plane.

In order to show that all signed rotation systems arising from G^ψ have the same facial cycles, we begin by rewriting the algorithm FACIALCYCLE() for a non-orientable surface. Notice that when traversing the facial cycles of a graph embedded on a disc representing the projective plane, the clockwise cyclic order of the incident edges viewed from above the disc appears counterclockwise when viewed from below, and vice versa. The algorithm uses a boolean variable *onTop* to indicate whether it is currently viewing the disc from above. Initially *onTop* has the value true. Each time an edge uv with $\text{SGN}(uv) = -1$ is encountered, it reverses the value of *onTop*. Any vertices visited while *onTop* is false will see a counterclockwise orientation for their incident edges. Those with *onTop* true will see a clockwise orientation.

When a graph is embedded on an orientable surface, the facial cycles are oriented cycles. We can assign a clockwise orientation to one cycle and the orientation of all adjacent cycles is determined, and so forth, so that an orientation can be assigned to the entire embedding. Reversing the cycles gives an equivalent, but reversed, embedding.

Graphs embedded on the projective plane do not have this property. If we try to assign an orientation to the facial cycles of an embedding G^ψ, we can then choose different signed rotation systems corresponding to G^ψ, and different orientations of the cycles will be obtained. However, if we are given the (unoriented) facial cycles, they can be glued together uniquely along their common edges to construct a polygonal representation of the projective plane. Therefore if we are given the facial cycles, they will determine a topological embedding G^ψ.

Algorithm 15.7.1 when given a signed rotation system π of an embedding G^ψ on

a non-orientable surface and a vertex u with an incident edge e, will find the facial cycle containing e.

Algorithm 15.7.1: FACIALCYCLESGN(G^π, u, e)

onTop $\leftarrow$ **true** "initially view the disc from above"
$e' \leftarrow e$
repeat
  comment: e' currently equals uv, for some v
  if SGN(e') $= -1$ **then** *onTop* $\leftarrow$ **not** *onTop*
  $v \leftarrow$ other end of e'
  $e'' \leftarrow$ edge of $\pi(v)$ corresponding to e'
  comment: e'' currently equals vu
  if *onTop*
    then $e' \leftarrow$ edge preceding e'' in $\pi(v)$
    else $e' \leftarrow$ edge following e'' in $\pi(v)$
  $u \leftarrow v$
until $e' = e$ **and** *onTop*

If this algorithm is applied to the combinatorial embeddings of $K_{3,3}$ given in Table 15.1, identical faces will be constructed. Because the projective plane is constructed by gluing together the faces, it follows that the topological embeddings they represent are equivalent.

In Definition 15.14 combinatorial embeddings G^{t_1} and G^{t_1} on the torus were defined to be equivalent if there exists an automorphism of G that induces a mapping of t_1 to t_2 or $\overline{t_2}$. This is inappropriate for signed rotation systems, as it cannot take the signatures into consideration. Therefore we define equivalence of signed rotation systems in terms of facial walks, as in Definition 15.15. Later we will see how to use a medial digraph and the double cover to determine equivalence of projective planar embeddings.

DEFINITION 15.18: Let G be a 2-connnected graph with projective rotation systems π_1 and π_2. Then G^{π_1} and G^{π_2} are *equivalent* or *isomorphic* if there is a permutation of $V(G)$ that maps the collection of facial walks of G^{π_1} to that of G^{π_2}, or to that of $G^{\overline{\pi}_2}$.

When a facial walk is traversed the vertices are encountered in a certain sequence. When glueing the faces of an embedding G^π to obtain a projective plane, there is only one way to do this, as each edge appears exactly twice in the collection of facial walks. The result is a projective plane with the graph G embedded on it. Equivalent embeddings must be isomorphic. Definition 15.18 determines the equivalence of two embeddings of a graph G on the projective plane in terms of their facial walks. Equivalence of embeddings G^{π_1} and H^{π_2}, where G and H are different graphs, requires in addition an isomorphism of G and H. Automorphisms of projective maps can also be defined in terms of the facial walks.

DEFINITION 15.19: Let G^π be a projective map. An *automorphism* of G^π is any permutation of $V(G)$ that maps the collection of facial walks of G^π to itself, or to those of $G^{\overline{\pi}}$. The *automorphism group* is $\text{AUT}(G^\pi)$.

In general, let G^ψ be a topological embedding, and consider two different representations of the projective plane as digons, a^+a^+ and b^+b^+. Let π_a be the signed rotation system corresponding to a^+a^+ and let π_b correspond to b^+b^+. The boundary of the disc given by b^+b^+ is a non-contractible Jordan curve C in the a^+a^+ representation. It will intersect the graph G^ψ in one or more points. Refer to Figure 15.35. When the disc is cut along the curve C, it may cut some edges of G^ψ more than once. If so, we subdivide those edges which are cut more than once. Hence we can assume that every edge of G^ψ is cut at most once by C. If the curve cuts through a vertex, we can move it slightly so that it misses the vertex. We can do this because the graph has a finite number of vertices.

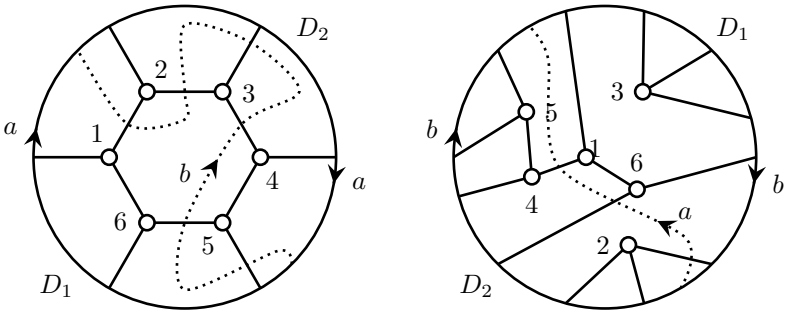


FIGURE 15.35
Representations $\boldsymbol{a^+a^+}$ and $\boldsymbol{b^+b^+}$ of the projective plane

Suppose first that C cuts the boundary of the a^+a^+ disc in exactly two points (which are antipodal points), as in Figure 15.35. To transform the a^+a^+ representation into the b^+b^+ representation, we cut the disc along the dotted curve, then flip one-half of the disc over, and glue the two equivalent parts of the a^+a^+ boundary together. We can flip either half over. Denote the two parts of the disc obtained as D_1 and D_2, where D_2 is the part that is flipped over. The result is a disc whose boundary is b^+b^+, shown as the disc on the right in Figure 15.35. We obtain π_b from π_a by *reversing* the cyclic adjacencies for all vertices in D_2, and by assigning -1 to those edges that are cut by C. We now compare the facial cycles of G^{π_a} and G^{π_b} constructed by the algorithm FACIALCYCLESGN(). Let $(v_1, v_2, \ldots, v_k)$ be a facial cycle constructed for G^{π_a}. Without loss of generality, suppose that FACIALCYCLESGN(G^{π_b}) begins to trace out this face from v_1 which is in D_1. If the entire face is within D_1, the result will be the same as the face obtained using π_a. If the facial cycle crosses from D_1 to D_2 via the a^+a^+ boundary, then because D_2 was flipped upside down, and the cyclic adjacencies of π_a were reversed to obtain π_b, the same facial boundary will be constructed using π_b or π_a. If the facial cycle crosses from D_1 to D_2 via C, then because π_b attaches a signature of -1 to

these edges, the cyclic adjacencies will be reversed by FACIALCYCLESGN(). But the cyclic adjacencies of π_a in D_2 were also reversed in π_b. The net effect is that the same facial boundary is constructed using π_b or π_a at each step of the algorithm. It follows that the two embeddings G^{π_a} and G^{π_b} have the same 2-cells as faces. Now we may have subdivided some edges of G before cutting and pasting the disc. Vertices of degree two do not affect the faces, and a cyclic order of two edges is invariant when reversed. Therefore, when traversing a facial cycle along a path created by subdividing an edge, the important factor is the number of -1's encountered. Hence we can contract any subdivided edges and assign a signature of -1 if the number of edges in the subdivided path was odd. The result will be the same facial cycle. We conclude that the faces of G^{π_a} and G^{π_b} are identical, so that the embeddings are equivalent.

Suppose now that C cuts the boundary of the a^+a^+ disc in more than one pair of antipodal points, where C is the boundary of the b^+b^+ disc. There are an infinite (in fact, uncountable) number of possible non-contractible Jordan curves C. But there are only a finite number of possible signed rotation systems for G^ψ, because G is finite. Therefore we will only consider non-contractible Jordan curves which meet the boundary a^+a^+ in a finite number of points.

If C cuts the boundary of the a^+a^+ disc in more than one pair of antipodal points, we proceed by induction. We are given a graph G embedded on the disc, with a rotation system π_a. As before we assume that C does not cut any vertices of G^ψ. Choose two consecutive points P and Q on C at which the disc boundary is cut such that one of the intervals $[P, Q]$ and $[Q, P]$ on the boundary is not cut by any other points of C. This is illustrated in Figure 15.36, where C is the dotted curve. Let the antipodal points of P and Q be P' and Q'. Let $C[P, Q]$ denote the portion of C from P to Q. Make a cut in the disc very close to $C[P, Q]$, cutting off a portion D_1 of the disc, so that $C[P, Q]$ is completely contained within D_1, but so that the only part of G that is affected by the cut are the edges of G that cross $C[P, Q]$. This is possible because the graph is finite. Let the remainder of the disc be D_2. We now flip D_1 over, and glue the matching boundaries of D_1 and D_2 near P' and Q'. The result is a disc representation of the projective plane such that C cuts its boundary in four fewer points. Let the boundary of the new disc be c^+c^+, and let the signed rotation system corresponding to it be π_c.

Consider the faces of G^{π_a} and G^{π_c}. The rotation systems π_a and π_c differ only in edges which are within D_1, or which cross from D_1 to D_2. Vertices within D_1 have adjacency lists of opposite orientation in π_a and π_c. FACIALCYCLESGN() will construct the same faces for both π_a and π_c. With respect to the π_c disc, C has fewer intersections with the boundary. We use induction to conclude that FACIALCYCLESGN() will construct the same faces for both π_c and π_b. It follows that G^{π_a} and G^{π_b} always have the same faces. Because the projective plane is constructed by gluing the faces together, this gives:

Theorem 15.19. *Let G^ψ be an embedding on the projective plane. Given any two signed rotation systems π_a and π_b for G^ψ, corresponding to different disc representations of the projective plane, G^{π_a} and G^{π_b} are equivalent embeddings.*

Theorem 15.20. *Let G^{ψ_1} and G^{ψ_2} be topological embeddings of G on the projective*

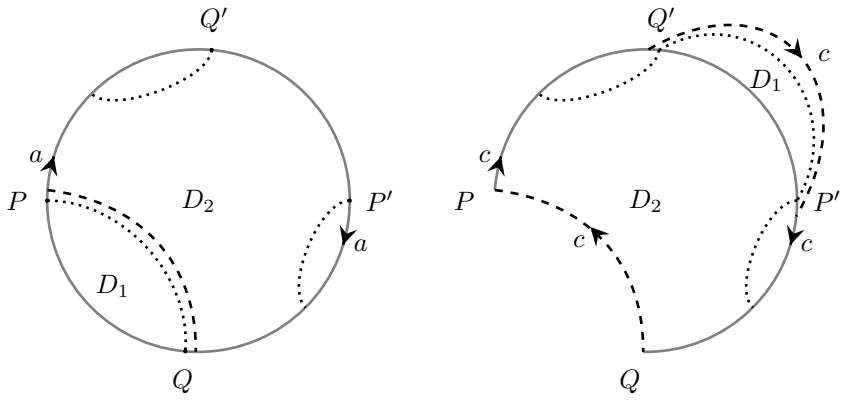


FIGURE 15.36
Transforming a disc representation of the projective plane

plane with corresponding combinatorial embeddings G^{π_a} and G^{π_b}, with respect to two disc representations of the projective plane. Then G^{ψ_1} and G^{ψ_2} are homeomorphic if and only if G^{π_a} and G^{π_b} are equivalent.

Proof. If G^{π_a} and G^{π_b} are equivalent, there is a permutation of $V(G)$ that maps the facial cycles of G^{π_a} to those of G^{π_b}. The faces are homeomorphic to 2-cells bounded by the facial cycles. It follows that the faces of G^{π_a} and G^{π_b} determine a homeomorphism of the embeddings G^{ψ_1} and G^{ψ_2}. Therefore G^{ψ_1} and G^{ψ_2} are homeomorphic if and only if G^{π_a} and G^{π_b} are equivalent. □

If G^p is a planar embedding of a 2-connected graph, it is very easy to convert p to a projective rotation system π. When G is drawn in the plane, one face is always the outer face. We draw G in a disc representing the projective plane. We then choose any edge e on the outer face, and reroute it so that it crosses the boundary of the disc. The result is a projective map. The two faces on either side of e in the planar map become one face in the projective map. The cyclic order of adjacent vertices is unchanged, for all vertices of G. Thus, p can be converted to a projective rotation system, by assigning a signature of -1 to any one edge of G. However, the embeddings constructed in this way are somewhat unsatisfactory, as there is a non-contractible Jordan curve in the surface which cuts the embedding in only one point.

15.7.1 The facewidth

The projective plane has unorientable genus one. The torus has orientable genus one. Although they both have genus one, these surfaces behave very differently. K_7 can be embedded on the torus. Yet it is easy to see that it cannot be embedded on the projective plane, as the unique embedding of K_6 shown in Figure 15.33 cannot be extended to K_7. Alternatively, Euler's formula can be used to show that K_7 has too

many edges to embed on the projective plane. However, there are infinite families of graphs that can be embedded on the projective plane, but not on the torus.

We begin with two families of graphs called the *Möbius ladder* and *Möbius lattice*, which can be embedded on both the projective plane and torus.

DEFINITION 15.20: The *Möbius ladder* L_{2n} is the graph with $2n$ vertices $\{v_1, v_2, \ldots, v_{2n}\}$ such that $v_i \longrightarrow v_{i+1}$, and $v_i \longrightarrow v_{i+n}$, where subscripts larger than $2n$ are reduced modulo $2n$.

The Möbius ladder L_6 is just $K_{3,3}$, shown in Figure 15.34. L_8 is shown in Figure 15.37. Notice that L_{2n} is always a 3-regular graph.

DEFINITION 15.21: The *Möbius lattice* L_{2n-1} is the graph with $2n-1$ vertices $\{v_1, v_2, \ldots, v_{2n-1}\}$ such that $v_i \longrightarrow v_{i+1}$, $v_i \longrightarrow v_{i+n-1}$, and $v_i \longrightarrow v_{i+n}$ where subscripts larger than $2n-1$ are reduced modulo $2n-1$.

The Möbius lattice L_5 is just K_5. L_7 is shown in Figure 15.37. Notice that L_{2n-1} is always a 4-regular graph.

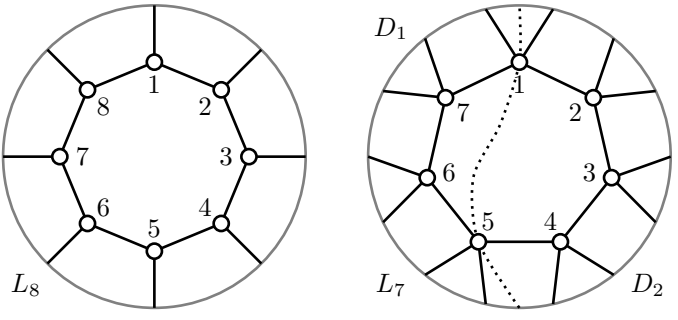


FIGURE 15.37
The Möbius ladder L_8 and Möbius lattice L_7

There is a clever trick that can be used to convert these projective embeddings of L_{2n} and L_{2n-1} to toroidal embeddings. Draw an essential cycle C across the disc as shown by the dotted line in Figure 15.37, dividing it into two parts, D_1 and D_2. Notice that C intersects the graph embedding in only two points, representing the vertices 1 and 5. Vertices 1 and 5 are both joined to several vertices located in D_1 and D_2. Now cut the disc along C, flip D_2 over, and glue D_1 and D_2 along the common boundary to get a new disc representation of the projective plane, as shown in Figure 15.38. Vertices 1 and 5 are on the boundary of the new disc. As we do not need to find a projective rotation system for this embedding, we can take these vertices on the boundary of the disc. These antipodal points are the only points of the embedding G^{ψ} on the disc boundary. Therefore we can convert the disc into the hexagonal form of the torus, obtaining an embedding on the torus.

DEFINITION 15.22: Let G^{ψ} be a graph embedding in a surface Σ. Let C be

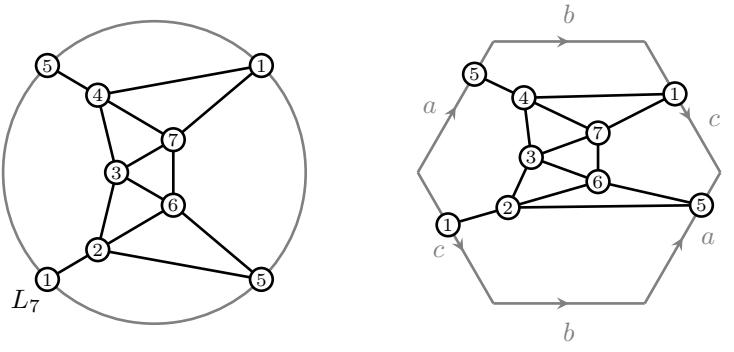


FIGURE 15.38
Converting a projective embedding to a toroidal embedding

a non-contractible Jordan curve in Σ. The *facewidth* of C is $\mathrm{fw}(C)$, the number of points of G^ψ common to C. The *facewidth* of G^ψ is $\mathrm{fw}(G^\psi)$, the minimum $\mathrm{fw}(C)$, where C is any non-contractible Jordan curve in Σ.

The facewidth is sometimes known as the *representativity* of an embedding. Let C be a non-contractible Jordan curve of minimum possible facewidth, for an embedding G^ψ. If C intersects a face F, then it also intersects the boundary of F. If an intersection point is not at a vertex, then it is at an interior point of an edge e. Then C also intersects the face on the other side of e. In such a case, we can alter C slightly so that it passes through e at an endpoint of the edge. The result is another non-contractible Jordan curve, also of minimum facewidth. This gives the following:

Lemma 15.21. *Given any embedding G^ψ, there is a non-contractible Jordan curve C of facewidth $\mathrm{fw}(G^\psi)$ such that C intersects G^ψ only at images of vertices.*

Now the faces of an embedding G^ψ are determined completely by its rotation system. If C intersects G^ψ only at images of vertices, then C determines a cyclic sequence of vertices $(v_1, v_2, \ldots, v_k)$, where $k = \mathrm{fw}(C)$, such that consecutive vertices are on the same facial boundary. It follows that $\mathrm{fw}(G^\psi)$ depends only on the rotation system, so that we can also write $\mathrm{fw}(G^\pi)$. We show that the method used above to convert a projective embedding of L_7 to a toroidal embedding works in general, whenever $\mathrm{fw}(G^\pi)$ is at most three.

Theorem 15.22. *Let G^π be a projective embedding with facewidth at most three. Then G can be embedded on the torus.*

Proof. The proof proceeds as in the previous example. Let C be a non-contractible Jordan curve with $\mathrm{fw}(C) \leq 3$. Use the hexagonal form of the torus, $a^+b^+c^+a^-b^-c^-$, as in Figure 15.39. We cut the disc of the projective plane along C obtaining D_1 and D_2, which we glue together to obtain a new disc. Without loss of generality, assume that $\mathrm{fw}(C) = 3$, so that there are three pairs of antipodal points of G^π on the boundary of the new disc, call them p, q, and r, and suppose that they occur in this

order along the boundary of the disc. We convert the disc into the hexagonal form of the torus, as in Figure 15.39. We place p on the side of the hexagon labeled a, q on the side labeled b, and r on the side labeled c. In the hexagon, the identified pairs of sides are not antipodally reversed as in the disc model of the projective plane. However, there is only one point p, q, or r on each side, so that the sequence of points on the boundary is the same. The result is a toroidal embedding of G. □

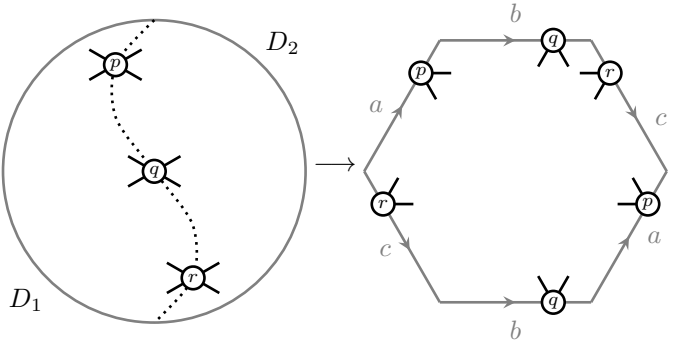


FIGURE 15.39
Converting a projective embedding to a toroidal embedding

This transformation will only work when $\mathrm{fw}(C) \leq 3$. It has been shown by FIEDLER, HUNEKE, RICHTER, and ROBERTSON [51] that the converse of this theorem is also true; so that if a toroidal graph can be embedded on the projective plane, then the facewidth on the projective plane is at most three. We can use this theorem to construct projective graphs which are not toroidal. The construction uses Möbius ladders or lattices. Suppose that we start with a Möbius ladder L_{2n} with vertices $\{v_1, v_2, \ldots, v_{2n}\}$. Add $2n$ more vertices $u_1, u_2, \ldots, u_{2n}$ forming a cycle in which $u_i \longrightarrow u_{i+1}$, and add the edges $u_i \longrightarrow v_i$, for all i. The result is a graph as shown in Figure 15.40. The facewidth of this graph is four, if $n \geq 4$, so that it cannot be embedded in the torus. The facewidth can be made arbitrarily high by increasing n and adding more cycles in this fashion.

15.7.2 Double covers

The projective plane can be viewed as exactly one-half of a sphere, by considering the disc representing the projective plane as the upper hemisphere of a sphere. We make another copy of the projective plane as follows. Let O denote the center of the disc representing the projective plane. Reflect each point Q of the disc in O, i.e., draw a diameter through O and Q. Map Q to its diametrically opposite point Q'. Use this reflected copy of the disc as the bottom hemisphere of a sphere. The two discs meet on the equator. Now the points on the boundary of the disc representing the projective plane are identified with each other. Therefore the equatorial points of the

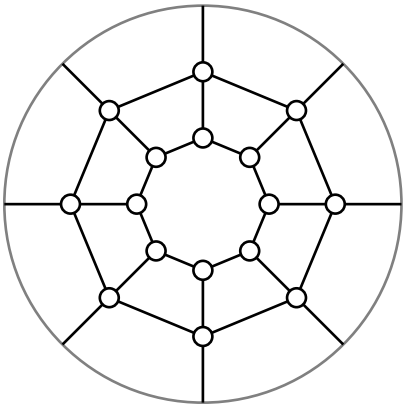


FIGURE 15.40
An embedding with facewidth four

sphere all have a corresponding antipodal point on the equator, which is identified in the projective plane models. Antipodal points of the sphere always correspond to the same point of the projective plane. The result is a two-to-one mapping of the sphere onto the projective plane. Thus, we say that the sphere is a *double cover* of the projective plane.

If G is a graph embedded on the disc of the projective plane, then the reflected disc will contain a reflected copy G' of G. Thus we have a sphere with $V(G)$ on the upper hemisphere, and $V(G')$ on the bottom hemisphere. We assume that no vertices of G are situated exactly on the boundary of the disc. Any edge uv of G that crosses the boundary of the disc will now meet the equator at the point where $v'u'$ meets the boundary of the reflected disc. So the edge uv is represented by two edges, uv' and $u'v$ on the sphere. We obtain a *double cover* of the embedding of G on the projective plane. For example, Figure 15.41 shows that a double cover of K_4 is the graph of the cube. We also find that the dodecahedron is a double cover of the Petersen graph, as can be seen from Figure 15.33. It is interesting to note that the double covers of projective graphs obtained by this method must be planar graphs. If G has several different embeddings on the projective plane, it will likely also have several different double covers.

DEFINITION 15.23: Let G and H be simple graphs such that there is a two-to-one mapping $\gamma : V(H) \to V(G)$ with the property that γ induces a two-to-one mapping of edges, $\gamma : E(H) \to E(G)$. Then H is said to be a *double cover* of G. γ is called the *double cover map*.

We denote the double cover of a projective embedding G^π by $DC(G^\pi)^p$, where p is the corresponding planar rotation system. If $(G')^{\pi'}$ is the reflected map of G^π, then π and π' are inverses of each other. Let $u \in V(G)$ have corresponding vertex $u' \in V(G')$. Then $\{u, u'\}$ is called an *antipodal pair* of vertices. If u

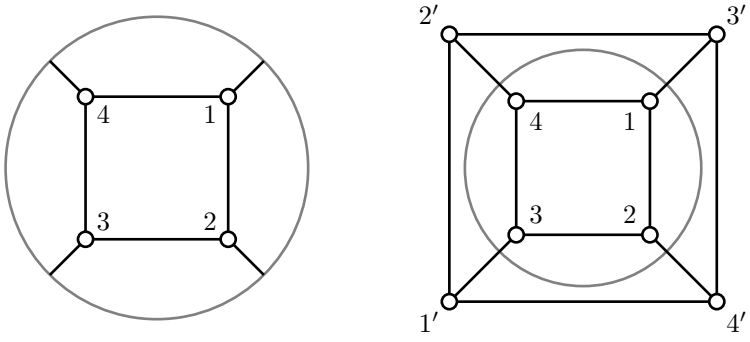


FIGURE 15.41
The cube is a double cover of $\boldsymbol{K_4}$

has rotation $\pi(u) = (u_1, u_2, \ldots, u_m)$, then the antipodal $u' \in V(G')$ has rotation $\pi'(u') = (u'_m, u'_{m-1}, \ldots, u'_1)$. If the edges uu_i all have $\text{SGN}(uu_i) = +1$, then $p(u) = (u_1, u_2, \ldots, u_m)$ and $p(u') = (u'_m, u'_{m-1}, \ldots, u'_1)$. But if some edge uu_i has $\text{SGN}(uu_i) = -1$, then in $\pi(u)$, u'_i is substituted for u_i to obtain $p(u)$, and in $\pi'(u')$, u_i is substituted for u'_i to obtain $p(u')$. This is illustrated in Figure 15.41.

Consider a facial walk $W = (v_1, v_2, \ldots, v_k)$ in G^π. Let $W' = (v'_k, v'_{k-1}, \ldots, v'_1)$ be the corresponding facial walk in $(G')^{\pi'}$. If all edges v_iv_{i+1} of W have signature $+1$, then $W_1 = W$ and $W_2 = W'$ are both facial walks of $DC(G^\pi)^p$. Notice that W_1 and W_2 are vertex-disjoint, and that W_2 consists of the antipodal vertices of W_1. Otherwise let v_iv_{i+1} be the first edge of W with $\text{SGN}(v_iv_{i+1}) = -1$. In $DC(G^\pi)^p$, $(v_1, v_2, \ldots, v_i, \ldots)$ is the beginning of a facial walk. In G^π, W continues on the opposite side of the disc, so that beginning with v_{i+1}, the rotations $\pi(v_j)$ are reversed in determining W, until the next edge with signature -1 is encountered. But in $DC(G^\pi)^p$, the rotation $p(v'_{i+1})$ is $\pi(v_{i+1})$ reversed, so that the facial walk in $DC(G^\pi)^p$ continues with $(\ldots, v'_{i+1}, v'_{i+2}, \ldots)$, until the next edge with signature -1 is encountered. Thus, a facial walk W_1 of $DC(G^\pi)^p$ is determined by W and its edges with signature -1. Similarly W_2 is determined from W'. We find that W_2 consists of the antipodal vertices of W_1, they are said to be *antipodal walks*. For example, in Figure 15.41, the facial walk $W = (1, 3, 4, 2)$ of K_4 has corresponding facial walks $W_1 = (1, 3', 4', 2)$ and $W_2 = (4, 3, 1', 2')$ in the double cover, where $W' = (4', 3', 1', 2')$. This gives:

Lemma 15.23. *Let $DC(G^\pi)^p$ be the double cover of G^π, with double cover map γ. Then γ induces a two-to-one map of the facial walks of $DC(G^\pi)^p$ to those of G^π. Pairs of facial walks W_1 and W_2 where $(W_1)^\gamma = (W_2)^\gamma$ are antipodal walks.*

Proof. The previous paragraph proves that the double cover contains two antipodal facial walks W_1 and W_2 corresponding to each W of G^π. By Euler's formula for planar maps ($n + f - \varepsilon = 2$), this comprises all facial walks of the double cover. □

Equivalence of projective maps is defined in terms of the facial walks (Defini-

tion 15.18). So it follows from Lemma 15.23 that isomorphism of projective maps is closely related to the isomorphism of their double covers.

Theorem 15.24. *Let $DC(G^\pi)^p$ be the double cover of G^π, with double cover map γ. Let $\theta \in \text{AUT}(DC(G^\pi)^p)$ have the property that θ maps every antipodal pair $\{v, v'\}$ to an antipodal pair. Then θ induces an automorphism $\phi \in \text{AUT}(G^\pi)$.*

Proof. Let $(G')^{\pi'}$ be the reflected copy of G^π contained in the double cover, with $V' = V(G')$. Construct ϕ from θ as follows. Given any $v \in V(G)$, let $w = v^\theta$. Define $v^\phi = w^\gamma$. Now because θ maps antipodal pairs to antipodal pairs, it must be the case that $(v')^\theta$ is the antipodal vertex of w. Thus ϕ is completely determined by θ. We show that $\phi \in \text{AUT}(G^\pi)$. Consider the collection of facial walks of $DC(G^\pi)^p$. Every facial walk is mapped by θ to a facial walk in the same collection. The facial walks of the double cover occur in pairs W_1, W_2 which are mapped by γ to a common facial walk W of G^π. W_2 consists of the antipodal vertices of W_1, but reversed. W_1 and W_2 are mapped by θ to a pair of facial walks W_1^θ, W_2^θ, where W_2^θ consists of the antipodal vertices of W_1^θ, but reversed, by the properties of θ. Therefore W_1^θ, W_2^θ are both mapped by γ to W^ϕ or to the reversal of W^ϕ. It follows that ϕ maps the collection of facial walks of G^π to itself, or to that of $G^{\overline{\pi}}$, so that it is an automorphism of G^π. □

Given $\theta \in \text{AUT}(DC(G^\pi)^p)$ we write $\phi = \theta^\gamma$ for the induced automorphism, for ϕ is obtained by replacing each vertex v in θ with v^γ.

Theorem 15.25. *Let $DC(G^\pi)^p$ be the double cover of G^π, with double cover map γ. Let $\phi \in \text{AUT}(G^\pi)$. Then there is an automorphism $\theta \in \text{AUT}(DC(G^\pi)^p)$ such that $\phi = \theta^\gamma$.*

Proof. Let $(G')^{\pi'}$ be the reflected copy of G^π contained in the double cover. The construction of the double cover describes how to link the embeddings G^π and $(G')^{\pi'}$ on the surface of the sphere uniquely to create $DC(G^\pi)^p$. If we now permute $V(G)$ on the disc representing the projective plane according to the automorphism ϕ, an identical-looking embedding $(G^\pi)^\phi$ is produced, with each vertex v replaced by v^ϕ. The collection of facial walks is either unchanged or reversed. The reflected embedding $(G')^{\pi'}$ has rotations that are reversed from G^π, so that when v' is replaced by $(v^\phi)'$, the reflected embedding also looks identical to $(G')^{\pi'}$. So when the permuted embedding is uniquely extended to the sphere to obtain the double cover, an identical-looking double cover is obtained, with v^ϕ replacing vertex v, and $(v^\phi)'$ replacing vertex v'. Thus we have an automorphism θ of $DC(G^\pi)^p$. Consider a facial walk W of G^π. There is a corresponding walk W' of $(G')^{\pi'}$. They correspond to antipodal walks W_1 and W_2 in the double cover. When ϕ maps W to W^ϕ, W' is mapped to $(W^\phi)'$. Therefore θ maps W_1 and W_2 to the pair of antipodal walks that correspond to W^ϕ, so that $\phi = \theta^\gamma$. □

Theorem 15.25 shows that corresponding to $\phi \in \text{AUT}(G^\pi)$, there is an automorphism $\theta \in \text{AUT}(DC(G^\pi)^p)$, such that $\phi = \theta^\gamma$. Consider a vertex $u \in V(G)$. It is mapped to u^ϕ. Then u^θ is either u^ϕ or $(u^\phi)'$. Either choice can be made, because the permutation that exchanges all antipodal pairs $\{v, v'\}$ is an automorphism

of $DC(G^\pi)^p$. This also determines $(u')^\theta$. Having chosen u^θ, the edges uv incident on u are taken. Consider an edge uv of G with $\text{SGN}(uv) = +1$. It may map to an edge of signature -1. Then $(uv)^\theta = u^\theta v^\theta$ will either be $(u^\phi)(v^\phi)'$ or $(u^\phi)'(v^\phi)$. As u^θ has already ben chosen, this determines v^θ uniquely, and consequently $(v')^\theta$ is also determined. Similarly there are two choices for $(uv)^\theta$ when uv maps to an edge of signature $+1$, namely $(u^\phi)(v^\phi)$ or $(u^\phi)'(v^\phi)'$. Again v^θ and $(v')^\theta$ are determined uniquely. We continue like this until all vertices of the double cover have been mapped by θ.

As the sphere is an orientable surface, the double cover provides a convenient means of determining isomorphisms and automorphisms of projective embeddings. The *medial digraph* of an embedding as expressed in Definition 14.20 is inappropriate for non-orientable surfaces. Although it encapsulates the cyclic adjacencies of each vertex, it does not take into account the signature of the edges. The signatures are required to ensure that the facial walks are correctly determined. In order to distinguish inequivalent projective embeddings of a graph, and to determine the symmetries (automorphisms) of an embedding, we can use the double cover.

By Theorems 15.24 and 15.25, every automorphism of G^π arises from an automorphism of $DC(G^\pi)^p$ that maps antipodal pairs to antipodal pairs. Let M be the medial digraph of $DC(G^\pi)^p$. By Lemma 14.21, $\text{AUT}(M)$ determines all automorphisms of $DC(G^\pi)^p$. In order to ensure that only those automorphisms that map antipodal pairs to antipodal pairs are used, M can be modified slightly. If G has no vertices of degree two, an undirected path of length two connecting v and v' is added to M, for each antipodal pair $\{v, v'\}$. Call the result M^+, the *antipodal medial digraph*. Then any automorphism of M^+ must permute the vertices of degree two, thereby ensuring that antipodal pairs are mapped to antipodal pairs. But if G has vertices of degree two, the same technique will work, using a slightly longer path connecting v to v'. Then $\text{AUT}(M^+)$ contains all automorphisms of $DC(G^\pi)^p$ that map antipodal pairs to antipodal pairs. Let γ be the double cover map. For each generator of $\text{AUT}(M^+)$, we first restrict it to $V(DC(G^\pi)^p)$ to obtain θ, and we then find θ^γ. The result is $\text{AUT}(G^\pi)$. We state this as a theorem.

Theorem 15.26. *Let G^π be a projective map with double cover $DC(G^\pi)^p$, and double cover map γ. Let M^+ be the antipodal medial digraph of the double cover. Then $\text{AUT}(G^\pi)$ is obtained from $\text{AUT}(M^+)$ by restricting each element of $\text{AUT}(M^+)$ to $V(DC(G^\pi)^p)$, and then transforming it by γ.*

Theorem 15.26 provides a very convenient method of determining the automorphism groups of projective embeddings. It usually requires graph isomorphism software, but is purely combinatorial. It can also be used to determine whether embeddings G^{π_1} and G^{π_2} are isomorphic. We just construct the antipodal medial digraphs M_1^+ and M_2^+ of their double covers, and test them for isomorphism. In this way it is possible to find all non-equivalent projective embeddings of a given graph G.

Theorem 15.27. *There is only one embedding of $K_{3,3}$ on the projective plane, up to isomorphism.*

Proof. Let G^π be an embedding of $K_{3,3}$ on the projective plane. Consider a longest

essential cycle in G^π. If the length of C is 6, let the vertices of C be $(1, 2, 3, 4, 5, 6)$, in that order. When C is embedded in the projective plane it has one face, with boundary $(1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6)$. The edges still to be embedded are $14, 25$ and 36. Without loss of generality, 14 must be embedded in the face as shown in Figure 15.42. There is then only one way to complete the embedding by adding edges 25 and 36.

If the longest essential cycle in G^π has length four, it can be taken as $C = (1, 2, 3, 4)$. When C is embedded in the projective plane it has one face, with boundary $(1, 2, 3, 4, 1, 2, 3, 4)$. Vertices 5 and 6 must be embedded within this face, as shown in Figure 15.43. The edge 56 must be drawn within the face, and without loss of generality, the edges 16 and 36 can be drawn as shown. There is then only one way to embed the edge 25, but two equivalent ways to embed the edge 45. Now one of the edges of C must be assigned a signature of -1. For each possibility, there is an essential cycle of length 6, a contradiction. □

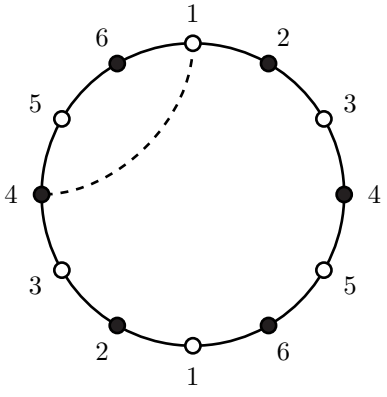


FIGURE 15.42
Embedding $\boldsymbol{K_{3,3}}$ in the projective plane, $\boldsymbol{C}$ has length 6

Exercises

15.7.1 Find all embeddings of K_4 and K_5 on the projective plane.

15.7.2 Find a projective embedding of $K_{3,4}$ and find its projective dual. What graph is it?

15.7.3 Let G^ψ be a projective embedding of G, and let π be an associated rotation system. Let C be any cycle of G. Show that C is an essential cycle of G^ψ if and only if the number of edges of C with signature -1 is congruent to 2 (mod 4).

15.7.4 Find the dual maps of the embeddings shown in Figure 15.37.

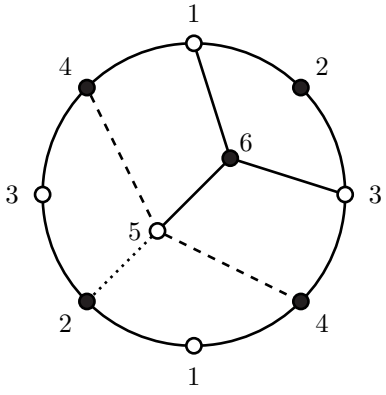


FIGURE 15.43
Embedding $\boldsymbol{K_{3,3}}$ in the projective plane, $\boldsymbol{C}$ has length 4

15.7.5 Show that the Möbius ladder L_{2n} contains a topological subgraph TL_{2n-2}, when $n \geq 3$.

15.7.6 Show that the Möbius lattice L_{2n-1} is a minor of L_{2n+1}, if $n \geq 3$.

15.7.7 Show that the Möbius ladder L_{2n} has an embedding on the torus in which all faces are hexagons. Construct the embedding for $n = 4, 5$, and find the dual map. (*Hint*: In the rectangular representation of the torus, draw a cycle of length $2n$ which wraps around the torus twice. Show how to complete this to a hexagonal embedding of L_{2n}.)

15.7.8 Show that the Möbius lattice L_{2n-1} has an embedding on the torus in which all faces are quadrilaterals. Construct the embedding for $n = 4, 5$, and find the dual map. (*Hint*: In the rectangular representation of the torus, draw a cycle of length $2n - 1$ which wraps around the torus twice. Show how to complete this to a quadrilateral embedding of L_{2n-1}.)

15.7.9 Show that there is a unique triangulation of the projective plane with three vertices and six edges.

15.7.10 Show that Read's algorithm for drawing a planar graph can be adapted to the projective plane. Show that there is a unique triangulation of the projective plane on three vertices, and that any triangulation can be reduced to it by deleting vertices of degrees three, four, or five, and adding diagonals to the faces obtained. Conclude that every projective graph has a straight-line drawing in the disc model of the projective plane.

15.7.11 Show that the graph of the $2n$-prism is a double cover of the Möbius ladder L_{2n}.

15.7.12 The cube is a double cover of K_4. Find another double cover of K_4.

15.7.13 Find a double cover of K_6, as illustrated in Figure 15.33.

15.7.14 Find a projective embedding of the graph of the cube, and find its double cover.

15.7.15 The *Desargues* graph is shown in Figure 15.44. The Desargues graph is non-planar, non-projective, and non-toroidal. Show that it is a double cover of the Petersen graph.

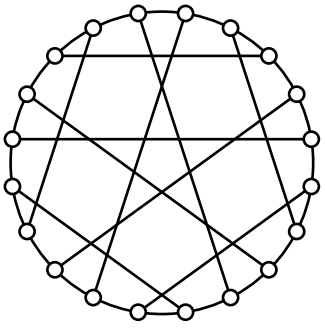


FIGURE 15.44
The Desargues graph

15.8 Embedding algorithms

In this section, we outline an algorithm to determine whether a 2-connected graph G can be embedded on the projective plane, and to find an embedding G^π. It is modeled on algorithms of Gagarin, Mohar, and Myrvold and Roth. If G is planar, we know how to convert a planar embedding to a projective embedding. Hence we can assume that G is non-planar, so that it contains a Kuratowski subgraph TK_5 or $TK_{3,3}$.

There is exactly one embedding of $K_{3,3}$ on the projective plane, shown in Figure 15.34, and two embeddings of K_5, shown in Figure 15.45. These embeddings all have the property that there are no repeated vertices on any facial cycle. In Figure 15.34, the hamilton cycle $(1, 2, 5, 6, 3, 4)$ is an essential cycle. Because $K_{3,3}$ has a unique embedding on the projective plane, this gives:

Lemma 15.28. *In any embedding of $K_{3,3}$ in the projective plane, some hamilton cycle is an essential cycle.*

If we now cut the projective plane along this cycle, the result is a disc in which each vertex $1, 2, \ldots, 6$ appears twice on the boundary. The resulting diagram, shown in Figure 15.46, is a very convenient representation of the projective plane.

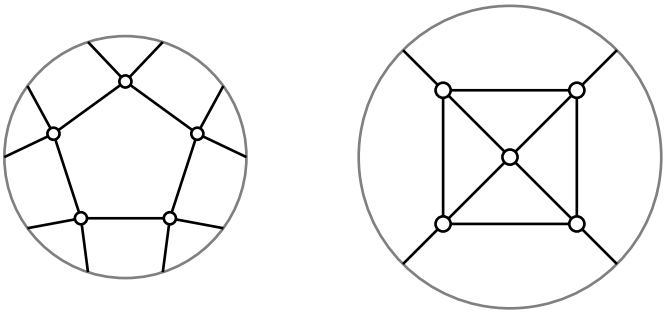


FIGURE 15.45
The embeddings of $\boldsymbol{K_5}$ on the projective plane

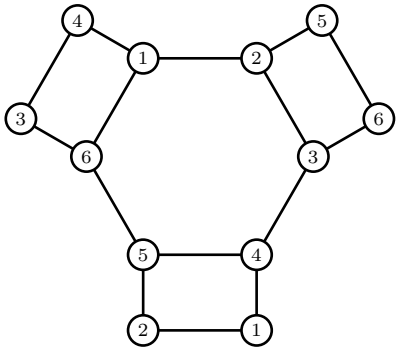


FIGURE 15.46
A representation of the projective plane

Consider a subgraph $TK_{3,3}$ in G. We want to determine whether G can be embedded in the projective plane. We will begin by embedding the subgraph $TK_{3,3}$. There are six hamilton cycles of $K_{3,3}$. Each corresponds to a cycle of $TK_{3,3}$. One of them must be essential. Exercise 15.8.2 describes an easy way to enumerate the hamilton cycles of $K_{3,3}$. We take each of the six cycles in turn, and construct an embedding of $TK_{3,3}$, as in Figure 15.46, and try to extend it to an embedding of G. If any one succeeds, then G is projective. Otherwise we conclude that G is non-projective.

The embedding of $K_{3,3}$ divides the projective plane into four faces – a hexagon, and four quadragons. The remaining vertices and edges of G must be placed in one of these faces. If we delete $V(TK_{3,3})$ from G, the result is a subgraph consisting of a number of connected components. If H is such a connected component, then because G is 2-connected, there must be at least two edges with one endpoint in H and the other in $TK_{3,3}$.

DEFINITION 15.24: A *bridge* of G with respect to $TK_{3,3}$ is either:

1. An edge uv, where $u, v \in V(TK_{3,3})$ but $uv \notin E(TK_{3,3})$, or
2. A connected component H of $G - V(TK_{3,3})$ together with all edges connecting H to $TK_{3,3}$

If B is a bridge, then a *vertex of attachment* of B is any vertex u of B such that $u \in V(TK_{3,3})$.

We can use a breadth-first (BFS) or depth-first search (DFS) to find the bridges of G with respect to an embedding of $TK_{3,3}$. Each bridge has at least two vertices of attachment. Because each face is a 2-cell, and each bridge must be embedded in a face of $TK_{3,3}$, each bridge must be planar. Into which faces of $TK_{3,3}$ can the bridges be placed?

The embedding of $K_{3,3}$ in Figure 15.46 determines a classification of the vertices and edges of $K_{3,3}$. Edges on the boundary of the hexagon are called *hexagon edges*. Edges which are on the boundary of a quadragon, but not on the hexagon are called *quadragon edges*. Hexagon edges like $\{1, 2\}$ and $\{4, 5\}$ are called *opposite* edges. Vertices like 1 and 4 are called *diagonally opposite* vertices, because they are diagonally opposite on the hexagon. By a *path* of $TK_{3,3}$ we mean a path connecting two corner vertices. The paths of $TK_{3,3}$ corresponding to hexagon edges of $K_{3,3}$ are called *hexagon paths*, those corresponding to opposite edges of the hexagon are called *opposite paths*, and so forth. In general, a path of $TK_{3,3}$ will be either a hexagon or a quadragon path of $TK_{3,3}$. The following lemmas on bridges can be proved by considering all possibilities of placing a bridge in Figure 15.46.

Lemma 15.29. *A bridge B can be placed in three faces of $TK_{3,3}$ if and only if B has exactly two vertices of attachment, which are diagonally opposite vertices.*

Lemma 15.30. *A bridge B can be placed in exactly two faces of $TK_{3,3}$ if and only if all vertices of attachment of B are on the same path, or on opposite paths of $TK_{3,3}$.*

It follows from these lemmas that a bridge B can be placed in at most three faces, and that bridges for which these lemmas do not apply, either cannot be placed in any face, or can be placed in at most one face. A bridge is *compatible* with a face if it can be placed in the face. A bridge B is a *k-face* bridge if it can be placed in exactly k faces of $TK_{3,3}$. Thus, we have 3-face, 2-face, 1-face, and 0-face bridges with respect to an embedding of $TK_{3,3}$. We can determine which faces a bridge may embed in by using its vertices of attachment and the previous lemmas.

Two bridges B_1 and B_2 *conflict* in face F if they can both be placed in face F, but cannot be simultaneously placed in face F. Suppose that B_1 can be embedded in face F and that it has k vertices of attachment $v_1, v_2, \ldots, v_k$, where $k \geq 2$, and where the vertices occur in that order on the facial cycle of F. The vertices divide the facial cycle into k intervals $[v_1, v_2], [v_2, v_3], \ldots, [v_{k-1}, v_k], [v_k, v_1]$, where each interval is a path from v_i to v_{i+1}. If B_2 is another bridge that can also be embedded in face F, then B_1 and B_2 do not conflict if and only if all vertices of attachment of B_2 lie in one interval of B_1, and vice versa.

Suppose that B is a 3-face bridge, with vertices of attachment u and v. All 3-face bridges with these vertices of attachment can be combined into one bridge, as they can always all be embedded in the same face if any embedding is possible. Thus, we can assume that there are at most three 3-face bridges, one for each pair of diagonally opposite vertices. Furthermore, any two distinct 3-face bridges conflict in the hexagon, so that at most one 3-face bridge can be embedded in the hexagon. The algorithm looks for embeddings with no 3-face bridges in the hexagon, or with exactly one 3-face bridge in the hexagon. Thus there are four subproblems to consider.

If we choose a DFS to find the bridges, it can be organized as follows. The procedure uses a variable *nBridges* to count the number of bridges found so far. It is initially zero. We take each vertex $u \in V(TK_{3,3})$ in turn, and consider all incident edges $uv \notin E(TK_{3,3})$. Edge uv belongs to exactly one bridge. We store a value $B(uv)$ for each edge, indicating the bridge to which uv belongs. If $B(uv) = 0$, then $B(uv)$ has not yet been assigned. If $v \in V(TK_{3,3})$, then edge uv is a bridge, and we assign $B(uv)$. Otherwise we call a procedure BRIDGEDFS(v) to build the bridge B. Because of the nature of a DFS, it will visit all vertices of B before returning, and will explore only edges of B. For each edge $xy \in B$, $B(xy)$ is assigned to be the current value of *nBridges*. Each time it encounters a vertex of $TK_{3,3}$, it has found a vertex of attachment. For each bridge, a list of vertices of attachment is saved. If two bridges B and B' are both found to have exactly two vertices of attachment, and they are the same two vertices, then B and B' are combined into a single bridge. The vertices of attachment are later used to determine the compatible faces, using the previous lemmas, and to sort the adjacency list of each u.

Algorithm 15.8.1: CONSTRUCTBRIDGES$(G, TK_{3,3})$

```
comment: Construct all bridges of G with respect to TK_{3,3}
nBridges ← 0
for each u ∈ V(TK_{3,3})
do { for each edge uv such that uv ∉ E(TK_{3,3})
     do { if B(uv) = 0 then
          { comment: the bridge of uv has not been visited
            nBridges ← nBridges + 1
            if v ∈ V(TK_{3,3})
              then B(uv) ← nBridges
              else BRIDGEDFS(v) }
     comment: all bridges incident on u have now been constructed }
```

We can now present an algorithm for determining the conflicts of bridges in a face F. Let $(u_1, u_2, \ldots, u_k)$ denote the facial cycle of F. We assign a numbering to the facial cycle, such that u_i is numbered i. This defines an ordering $u_1 < u_2 < \ldots < u_k$. For each bridge B that is compatible with F, we sort the vertices of attachment according to this ordering. The purpose of this is to determine whether the vertices of

attachment of each bridge lie completely within an interval of all other bridges. Let $b_{\min}$ denote the smallest vertex of attachment of bridge B, and let $b_{\max}$ denote the largest. The algorithm then walks along the facial cycle from u_k to u_1 and sorts the adjacency list of each u_i. The edges incident on u_i can be divided into the following three classes:

1. Edges u_iv belonging to a bridge B such that $u_i = b_{\min}$
2. Edges u_iv belonging to a bridge B such that $b_{\min} < u_i < b_{\max}$
3. Edges u_iv belonging to a bridge B such that $u_i = b_{\max}$

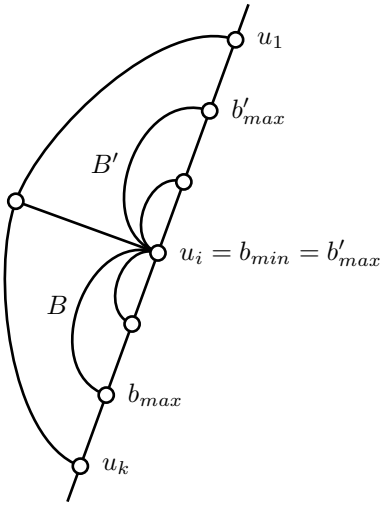


FIGURE 15.47
Determining whether bridges conflict

Refer to Figure 15.47. The adjacency list is ordered so that edges in the first class precede those in the second class, which precede those in the third class, and so that edges of each bridge are contiguous in each of the three classes. The edges in the first class are further sorted so that if u_iv and u_iv' belong to bridges B and B', respectively, where $b_{\max} < b'_{\max}$, then u_iv precedes u_iv'. If $b_{\max} = b'_{\max}$, then u_iv precedes u_iv' if B has more vertices of attachment. The edges in the third class are further sorted so that if u_iv and u_iv' belong to bridges B and B', respectively, where $b_{\min} < b'_{\min}$, then u_iv precedes u_iv'. If $b_{\min} = b'_{\min}$, then u_iv precedes u_iv' if B has fewer vertices of attachment.

In Figure 15.47 the u_1u_k-path of the facial cycle is drawn approximately vertically, and the bridges are placed to the left of the path. With this representation,

the ordering of the adjacency lists appears as a clockwise circle drawn at each u_i. If $u_i = b_{\min}$ for some bridge B, there can be several edges $u_i v$ belonging to bridge B. The last such edge is saved as $B_{\min}$. Similarly, if $u_i = b_{\max}$ for some bridge B, the first edge $u_i v$ of B is saved as $B_{\max}$.

The algorithm then walks along the facial cycle from u_k to u_1. Every edge $u_i v$ such that $B(u_i v)$ is compatible with F is placed on a stack. When $B_{\min}$, the last edge of bridge B is encountered, all edges of B are removed from the stack, and conflicts with B are determined. The algorithm stores a linked list of conflicting bridges for each bridge B.

Algorithm 15.8.2: BRIDGECONFLICTS(F)

```
comment: { Determine the conflicts among bridges incident on face F
         { The facial cycle of F is (u_1, u_2, ..., u_k)
         { The adjacency list of each u_i has been sorted
for u_i ← u_k downto u_1
do { for each edge u_i v such that u_i v ∉ E(TK_{3,3})
   do { B ← B(u_i v)
        if B is not compatible with face F go to L1
        place u_i v on Stack
        if u_i v = B_min
        then { let u_j be the vertex of attachment of B_max
             { u_ℓ w ← top of Stack
             { while u_ℓ w ≠ B_max do
             { { if B(u_ℓ w) = B
               {   then remove u_ℓ w from Stack
               {   else { if u_ℓ ≠ u_i and u_ℓ ≠ u_j
               {        {   then B and B(u_ℓ w) conflict
               { u_ℓ w ← next edge on Stack
             { remove u_ℓ w from Stack
        L1 :
```

We prove that the algorithm works. Suppose that B is a bridge with $b_{\max} = u_i$ and $b_{\min} = u_j$, with extreme edges $B_{\max} = u_i v$ and $B_{\min} = u_j w$. All vertices of attachment of B lie between u_i and u_j. If no other bridge has an attachment u_ℓ here, such that $u_i \neq u_\ell \neq u_j$, then B does not conflict with other bridges. Consider the point in the algorithm when $B_{\min}$ is reached. The algorithm will have stacked each edge of B incident on the facial cycle, including $B_{\min}$. It then removes all these edges. If there is no u_ℓ between u_i and u_j, no conflicts are discovered. But if B' is a bridge with a vertex of attachment u_ℓ in this range, then an edge $u_\ell x$ of B' will have been stacked after $B_{\max}$ and before $B_{\min}$. Because the edges of B' are still on the stack while edges of B are being removed, $b'_{\min} \leq b_{\min}$. If $b'_{\min} < b_{\min}$, then B and B' are in conflict, and this is discovered. If $b'_{\min} = b_{\min}$, then because $B_{\min}$

precedes $B'_{\min}$ in the adjacency lists, we know that $b'_{\max} \geq b_{\max}$. If $b'_{\max} > b_{\max}$, then B and B' are in conflict, and this is discovered. Otherwise $b'_{\max} = b_{\max}$ and $b'_{\min} = b_{\min}$, and u_ℓ is strictly between these limits. The ordering of the adjacency list tells us that B has at least as many vertices of attachment as B'. Therefore B and B' both have a vertex of attachment strictly between u_i and u_j. We conclude that the bridges conflict.

Once all bridges have been constructed and all conflicts have been determined, we construct an instance of the 2-Sat problem to represent this embedding problem. The 2-Sat problem will have boolean variables corresponding to the placement of bridges, and boolean expressions to characterize the conflicts of bridges. Let the bridges be $B_1, B_2, \ldots, B_m$. If there are any 0-face bridges, the embedding of $TK_{3,3}$ cannot be extended. If B_i is a 1-face bridge, embeddable in face F, create a boolean variable x_i for it. We require $x_i =$ **true**, and consider this to mean that B_i is assigned to face F. If B_i is a 2-face bridge embeddable in faces F and F', create boolean variables x_i and y_i for it. We consider $x_i =$ **true** to mean that B_i is assigned to F and $y_i =$ **true** to mean that B_i is assigned to F'. Because we do not want x_i and y_i both to be true, or both to be false, we construct the clauses

$$(x_i + y_i)(\overline{x}_i + \overline{y}_i)$$

This ensures that exactly one of x_i and y_i will be true.

If B_i is a 3-face bridge, create boolean variables x_i, y_i, and z_i for it as above, where $z_i =$ **true** means that B_i is embedded in the hexagon. The 3-face bridges require special treatment. We take $z_i =$ **false** and $z_i =$ **true** as separate cases. If $z_i =$ **false**, B_i becomes a 2-face bridge, and we construct the clauses

$$(x_i + y_i)(\overline{x}_i + \overline{y}_i)$$

to represent this bridge. If $z_i =$ **true**, B_i becomes a 1-face bridge, and we require $x_i = y_i =$ **false**.

If B_i and B_j are bridges that conflict in a face F, suppose without loss of generality that x_i represents the assignment of B_i to face F, and that w_j represents the assignment of B_j to face F, where w_j is one of x_j, y_j, or z_j. We then construct the clause

$$(\overline{x}_i + \overline{w}_j)$$

to ensure that at most one of B_i and B_j can be placed in face F.

When a variable is required to have a certain value (e.g., $x_i =$ **true**), we construct clauses

$$(x_i + x_0)(x_i + \overline{x}_0)$$

where x_0 is an additional boolean variable. Notice that this can only be satisfied if $x_i =$ **true**. If $x_i =$ **false** is required, we construct the clauses

$$(\overline{x}_i + x_0)(\overline{x}_i + \overline{x}_0)$$

which can only be satisfied if $x_i =$ **false**.

Thus, we can represent all conflicts and all placements of bridges by instances

of 2-Sat. Suppose that an algorithm for 2-Sat finds a solution satisfying the constraints. If $B_1, B_2, \ldots, B_k$ are the bridges assigned to face F, they are all mutually non-conflicting bridges. Consequently, all vertices of attachment of any B_i lie in an interval determined by two vertices of attachment of every B_j. If each B_i has a planar embedding in F, then they all have a common planar embedding in F, and conversely. If there is no common planar embedding of the bridges in F, then some bridge B_i has no planar embedding in F. It then follows that B_i cannot be embedded in any other face F'. Thus, we can complete the projective embedding of G by determining whether there is a planar embedding of the bridges in F. We construct a graph $G(F)$ which is the union of the facial cycle of F, and all bridges $B_1, B_2, \ldots, B_k$ assigned by 2-Sat to F. We add one additional vertex u_0, joined to every vertex of the facial cycle, in order to distinguish an "inside" and "outside" for F. We have:

Lemma 15.31. *$G(F)$ is planar if and only if bridges $B_1, B_2, \ldots, B_k$ have a common embedding in F.*

We can now present the algorithm for projective planarity.

The bridges $B_1, B_2, \ldots, B_m$ are constructed using a BFS or DFS. This takes $O(n)$ steps, because $\varepsilon \leq 3n-3$. There are six embeddings of $TK_{3,3}$ that are considered. For each embedding, the bridges are classified according to the faces they can be assigned to. The conflicts between bridges are then calculated. As the number of bridges m is bounded by n, the number of conflicts is at most $O(n^2)$. An instance of 2-Sat is then constructed with at most $3m+1$ variables and at most $4m+m(m-1)$ clauses. This can be solved in $O(n^2)$ time. If a solution is found, a planar embedding algorithm must be used for each face to find the actual embedding. If a linear or quadratic planarity algorithm is used, the result is at most $O(n^2)$ steps to complete the projective embedding. The result is a $O(n^2)$ algorithm for finding a projective embedding of G, if one exists, when we are given a $TK_{3,3}$ in G.

Now it would be possible to consider the embeddings of TK_5 in a similar way, and construct the bridges with respect to each embedding of TK_5, etc. There are 27 (labeled) embeddings of TK_5 in the projective plane. However, there is an easier way. In Section 14.3 we found that most graphs containing a subgraph TK_5 also contain $TK_{3,3}$, and that a simple breadth-first search algorithm can find a $TK_{3,3}$, given a TK_5. Thus, if we are given a TK_5 in G, we first try to find a $TK_{3,3}$ in its place, and then use Algorithm PROJECTIVEPLANARITY() to extend it to G.

Algorithm 15.8.3: PROJECTIVEPLANARITY$(G, TK_{3,3})$

comment: Given a graph G with a subgraph $TK_{3,3}$, determine whether G is projective.

let n and ε denote the number of vertices and edges of G
if $\varepsilon > 3n - 3$
  then return (*NonProjective*)
construct the bridges $B_1, B_2, \ldots, B_m$ with respect to $TK_{3,3}$
for each embedding of $TK_{3,3}$
do
- classify the bridges as 0-face, 1-face, 2-face, or 3-face
- **if** there is a 0-face bridge **go to** $L1$
- determine all conflicts of bridges
- construct the clauses representing all conflicts of bridges
- wlog, assume that B_1, B_2 and B_3 are 3-face bridges
- $z_1, z_2, z_3 \leftarrow$ **false**
- **for** $i \leftarrow 0$ **to** 3
  - **do**
    - construct the clauses representing B_1, B_2 and B_3
    - solve the resulting 2-Sat problem
    - **if** there is no solution **go to** $L2$
    - **for each** face F of $TK_{3,3}$ **do**
      - **do**
        - take all bridges assigned to F, construct graph $G(F)$
        - **if** $G(F)$ is non-planar **go to** $L2$
    - **comment:** we now have a projective embedding
    - **return** (*Projective*)
    - $L2:$
    - $z_i \leftarrow$ **false**
    - $z_{i+1} \leftarrow$ **true**
- $L1:$

return (*NonProjective*)

—

If TK_5 cannot be extended to $TK_{3,3}$, then the structure of G is limited. Let $\{v_1, v_2, v_3, v_4, v_5\}$ be the corners of TK_5. Then $G - \{v_1, v_2, v_3, v_4, v_5\}$ is a disconnected graph. Each component is adjacent to exactly two of the corner vertices of TK_5. Let G_{ij} denote the subgraph induced by all components adjacent to v_i and v_j, together with all edges connecting them to v_i or v_j. G_{ij} is called a K_5-*component* of G. Notice that v_i and v_j are vertices of G_{ij}, and that $E(G) = \cup_{i,j} E(G_{ij})$. An *augmented* K_5-*component* is the graph G_{ij}^+ with the additional edge $v_i v_j$; namely, $G_{ij}^+ = G_{ij} + v_i v_j$. We have the following theorem:

Theorem 15.32. *Suppose that G has a subgraph TK_5 which cannot be extended to $TK_{3,3}$. Then G is projective if and only if all augmented K_5-components G_{ij}^+ are planar.*

The proof of this theorem is left as an exercise. A consequence of it is that algorithms for projective planarity can focus on $TK_{3,3}$ subgraphs, which have fewer embeddings. A similar, but more complicated result holds for toroidal graphs containing a TK_5 which cannot be extended to $TK_{3,3}$.

Exercises

15.8.1 Show that a graph can be embedded on the projective plane if and only if it can be embedded on the Möbius band.

15.8.2 Show that $K_{3,3}$ has six hamilton cycles. If $C = (1, 2, 5, 6, 3, 4)$ is a hamilton cycle, show that all hamilton cycles can be obtained by successively applying the permutation $(1)(3, 5)(2, 4, 6)$ to C.

15.8.3 Show how to find a projective rotation system for a graph G containing $TK_{3,3}$, when the algorithm PROJECTIVEPLANARITY() determines that G is projective. *Hint*: Use the projective embedding of $K_{3,3}$ in Figure 15.48.

15.8.4 Prove Theorem 15.32.

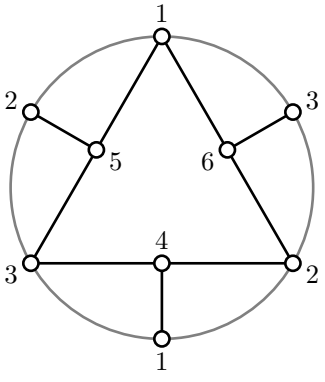


FIGURE 15.48
A projective embedding of $\boldsymbol{K_{3,3}}$

15.9 Heawood's map coloring theorem

We conclude this chapter with a discussion of Heawood's map coloring theorem. The 4-color theorem states that $\chi(G) \leq 4$, for graphs of genus zero. Heawood's map coloring theorem gives an analogous result for graphs of genus one or more.

Lemma 15.33. *Let $n \geq 3$. Then $g(K_n) \geq \lceil (n-3)(n-4)/12 \rceil$.*

Proof. By Lemma 15.6, $\varepsilon(K_n) = n(n-1)/2 \leq 3n + 6(g-1)$. Solving for g gives the result. □

Theorem 15.34. (Heawood's theorem) *Let G be a graph on n vertices with genus $g \geq 1$. Then $\chi(G) \leq \lfloor \frac{1}{2}(7 + \sqrt{1+48g}) \rfloor$.*

Proof. Let $\chi(G) = k$. If G is not a critical graph, then it contains a critical subgraph. Because a k-critical graph has minimum degree at least $k-1$, we conclude that the sum of degrees of G is at least $(k-1)n$, so that $\varepsilon \geq (k-1)n/2$. Lemma 15.6 gives $\varepsilon \leq 3n + 6(g-1)$. These two inequalities together give

$$k \leq 7 + \frac{12(g-1)}{n},$$

with equality only if both inequalities above are equalities. Now $g \geq 1$ so that for fixed g, this is a non-increasing function of n, so that $\chi(G)$ will be bounded. This arises because the number of edges in a k-critical graph increases as $kn/2$, whereas the maximum number of edges in a graph embedded in a surface of genus g increases as $3n$. We also know that $k \leq n$, an increasing function. Therefore the largest possible value of k is when $k = n = 7 + 12(g-1)/n$. The equation then becomes $n^2 - 7n - 12(g-1) = 0$, which gives the solution $n = \frac{1}{2}(7 + \sqrt{1+48g})$. Because $k \leq n$ and k must be an integer, the result follows. □

If $\frac{1}{2}(7 + \sqrt{1+48g})$ is an integer, the inequalities used in the proof of Heawood's theorem must be equalities. This requires that $\varepsilon = (n-1)n/2$, so that $G = K_n$. The quantity $\frac{1}{2}(7 + \sqrt{1+48g})$ represents the largest number of vertices that a graph can have, and still satisfy $n(n-1)/2 \leq 3n + 6(g-1)$. If it is not an integer, this means that $n(n-1)/2 < 3n+6(g-1)$, so that a complete graph on $n = \lfloor \frac{1}{2}(7+\sqrt{1+48g}) \rfloor$ vertices will not be a triangulation. In general, we have:

Theorem 15.35. *Let G be a graph on n vertices with genus $g \geq 1$ and chromatic number $\chi(G) = \lfloor \frac{1}{2}(7 + \sqrt{1+48g}) \rfloor$. Then G contains a spanning complete graph.*

Proof. Let $h = \lfloor \frac{1}{2}(7 + \sqrt{1+48g}) \rfloor$. Let G have n vertices. Because $\chi(G) = h$, we know that $n \geq h$, and $\varepsilon(G) \geq n(h-1)/2 \geq h(h-1)/2$. But h is the largest integer such that $h(h-1)/2 \leq 3h + 6(g-1)$. Therefore $n = h$ and G contains a spanning complete graph. Note that a complete graph may not triangulate the surface, so that the number of edges in a triangulation, $3h + 6(g-1)$, may be larger than $h(h-1)/2$. □

We conclude that the extreme value of χ will be achieved only if a complete graph with this many vertices can be embedded in the surface.

This theorem gives $\chi(G) \leq 7$ for the torus. An embedding of K_7 in the torus is shown in Figure 15.24, so that seven colors are necessary for the torus. We say that the *chromatic number* of the torus is seven, because all toroidal graphs can be colored in at most seven colors, and seven colors are necessary for some graphs. The dual of

the embedding of K_7 on the torus is the Heawood graph. Heawood was coloring the faces of an embedding rather than the vertices of a graph, and discovered this graph. The 4-color theorem tells us that the chromatic number of the plane is four. The formula of Heawood's theorem gives the bound $\chi(G) \leq 4$ for the plane. However, the proof is invalid when $g = 0$, and there are many planar graphs other than K_4 which require four colors. Lemma 15.33 gives $g(K_n) \geq \lceil (n-3)(n-4)/12 \rceil$. A proof that $g(K_n)$ equals this bound would mean that K_n can always be embedded in a surface of this genus. Because $\chi(K_n) = n$, the inequality of Heawood's theorem could then be replaced by an equality. Calculating the genus of K_n was accomplished by a number of people (Heffter, Gustin, Ringel, Youngs, and Mayer) over many years. We state the result, without proof, as the following theorem.

Theorem 15.36. (Ringel-Youngs) *Let $n \geq 3$. Then $g(K_n) = \lceil (n-3)(n-4)/12 \rceil$.*

A complete proof of this result can be found in the survey paper of WHITE [190]. A consequence is the following:

Theorem 15.37. (Heawood map coloring theorem) *The chromatic number of an orientable surface of genus $g \geq 1$ is $\lfloor \frac{1}{2}(7 + \sqrt{1+48g}) \rfloor$.*

The corresponding results for non-orientable surfaces of genus $\overline{g} \geq 1$ are as follows. Corresponding to Lemma 15.33 is the bound $\overline{g}(K_n) \geq \lceil (n-3)(n-4)/6 \rceil$. Corresponding to Heawood's theorem is the bound $\chi(G) \leq \lfloor \frac{1}{2}(7 + \sqrt{1+24\overline{g}}) \rfloor$, which is proved in an analogous way. Again, the graphs which meet the bound are the complete graphs. The non-orientable version of the Ringel-Youngs theorem is the following:

Theorem 15.38. *Let $n \geq 5$. Then $\overline{g}(K_n) = \lceil (n-3)(n-4)/6 \rceil$, except that $\overline{g}(K_7) = 3$.*

The formula $(n-3)(n-4)/6$ gives $\overline{g}(K_7) \geq 2$. However, $\overline{g}(K_7) = 3$, as K_7 does not embed on the Klein bottle. The map coloring theorem for non-orientable surfaces is then:

Theorem 15.39. *The chromatic number of a non-orientable surface of genus $\overline{g} \geq 1$ is $\lfloor \frac{1}{2}(7 + \sqrt{1+24\overline{g}}) \rfloor$, except that the chromatic number of the Klein bottle is 6.*

Exercises

15.9.1 Let t be a positive integer, and let tK_3 denote the graph with three vertices, and t parallel edges connecting each pair of vertices, so that $\varepsilon(tK_3) = 3t$. Consider embeddings of tK_3 in which there are no digon faces. Show that $g(tK_3) \geq (t-1)/2$ and that $\overline{g}(tK_3) \geq t-1$.

15.9.2 Show that $g(tK_3) = (t-1)/2$ and $\overline{g}(tK_3) = t-1$ by constructing embeddings of tK_3 on the appropriate surfaces.

15.9.3 Let G be a graph with n vertices and genus g, and let n_k denote the number of vertices of degree k. Suppose that $n_1 = n_2 = 0$. Construct an inequality satisfied by $n_3, n_4, \ldots$ in terms of g, using the number of edges in a triangulation. Do the same for $\overline{g}(G)$.

15.9.4 The *maximum genus* of a graph G is the largest value g such that G has a 2-cell embedding on a surface of genus g. If g' is the maximum genus of a graph G on n vertices, use the Euler-Poincaré formula to show that $g' \leq (\varepsilon - n + 1)/2$. Find the maximum orientable genus of K_4.

15.9.5 Show that K_7 does not embed on the Klein bottle.

15.10 Notes

An excellent source book related to topology and geometry is HILBERT and COHN-VOSSEN [83]. It is perhaps one of the best and most readable mathematics books ever written. Proofs of the Dehn-Heegard theorem can be found in FRÉCHET and FAN [54] and in STILLWELL [161]. Both contain very readable accounts of combinatorial topology. Fréchet and Fan call the Euler-Poincaré formula *Descartes' formula*. An excellent source book for the relation between isometries of surfaces and topological surfaces is STILLWELL [162]

Some beautiful computer models of the projective plane can be found in APÉRY [4].

There are excellent chapters in DIESTEL [44] and ZIEGLER [196] on the graph minor theorem. The minor order obstructions for the projective plane were found by GLOVER, HUNEKE and WANG [67]. ARCHDEACON [7] proved that the list is complete. MYRVOLD [128] has found over 200,000 topological obstructions for the torus, and the list may not be complete.

An excellent source for graphs and surfaces is the book by MOHAR and THOMASSEN [126], or the book on topological graph theory by GROSS and TUCKER [74]. See THURSTON [171] for topological and geometric insights into the torus and other surfaces.

THOMASSEN [168] has proved that Graph Genus is NP-complete.

The algorithm for drawing graphs on the torus, given a toroidal rotation system is from KOCAY, NEILSON, and SZYPOWSKI [104]. It is adapted from Read's algorithm READ [144] for planar graphs.

Theorem 15.22 relating projective embeddings to toroidal embeddings is from FIEDLER, HUNEKE, RICHTER, and ROBERTSON [51].

The algorithm for embedding a graph on the projective plane is based on algorithms by GAGARIN [59], MOHAR [125], and MYRVOLD and ROTH [130]. Theorem 15.32 is from GAGARIN and KOCAY [60].

The survey article by WHITE [190] contains a complete proof of the Ringel-Youngs theorem and the Heawood map coloring theorem.

16

The Klein Bottle and the Double Torus

16.1 The Klein bottle

The *Klein bottle* is the non-orientable surface of genus two. The *double torus* is the orientable surface of genus two. We begin with the Klein bottle. It has several possible polygonal representations. The most common one seems to be a rectangle with one pair of opposite sides glued with a twist, as in Figure 16.1, (also Figure 15.9), giving the form $a^+b^+a^+b^-$, which is not the standard form of Theorem 15.2. Numerous interesting 3D models of the Klein bottle can be found on the internet. We see that in this rectangular representation, if the two horizontal edges of the rectangle are glued, the result is a cylinder, in which opposite ends have opposite orientation. But if the vertical edges are glued, we obtain a Möbius band, whose boundary must be glued to itself.

We now make a diagonal cut across the rectangle, indicated by the dotted line c, and re-assemble the rectangle so as to obtain another rectangular form, in standard form $c^+c^+e^+e^+$, where $e^+ = b^-$ and $e^- = b^+$.

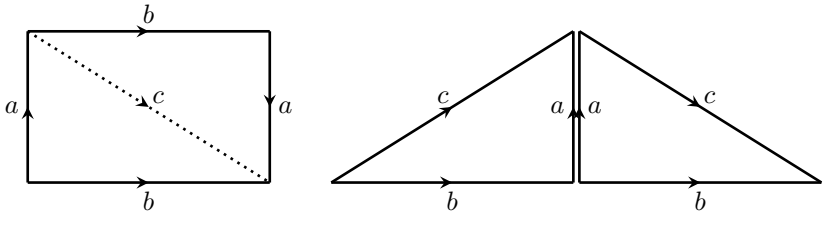


FIGURE 16.1
The Klein bottle as rectangles, $\boldsymbol{a^+b^+a^+b^-}$ and $\boldsymbol{c^+c^+b^-b^-}$

Alternatively, we can make a horizontal cut through the middle of the rectangle, and re-assemble it to obtain a hexagonal form for the Klein bottle (see Figures 16.2, 16.3, and 16.4). The edge marked a in Figure 16.1 is cut into a_1 and a_2, and the two a_2's are glued in Figure 16.4, to obtain a hexagon, denoted here by $a^+c^+c^+a^-b^-b^-$, where we have renamed a_1 to a. In this form it is evident that the Klein bottle contains two projective planes c^+c^+ and b^-b^-, as well as a cylinder (by glueing the two a's), and can be thought of as a sphere with two crosscaps. The circular arcs numbered $1, 2, \ldots, 8$ in Figure 16.3 will be considered later.

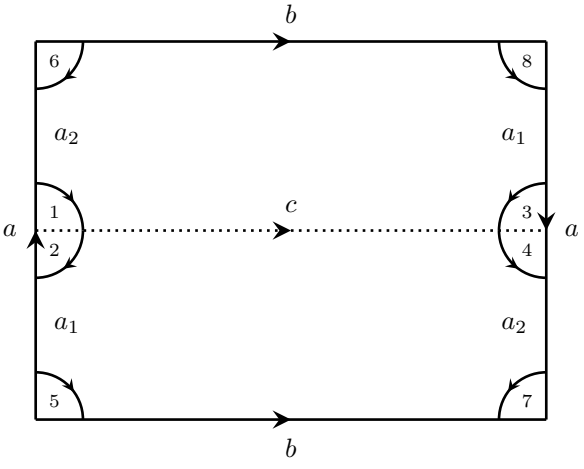


FIGURE 16.2
Constructing a hexagonal form of the Klein bottle, a horizontal cut c

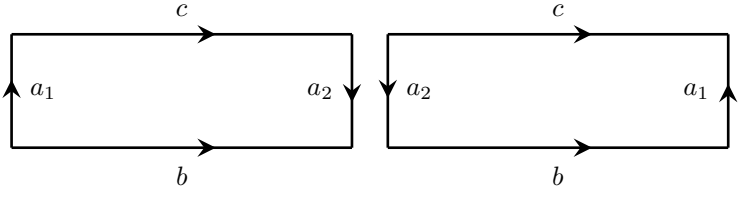


FIGURE 16.3
Constructing a hexagonal form of the Klein bottle, continued

Because the Klein bottle contains two projective planes, it is clear that any graph which can be embedded on the projective plane has a non-2-cell embedding on the Klein bottle. In fact, *two disjoint copies* of $K_{3,3}$ or K_5 could easily be embedded as a non-2-cell embedding on the Klein bottle. Existing algorithms for embedding graphs on the projective plane or torus rely on the fact that the Kuratowski graphs for the plane K_5 and $K_{3,3}$ have only 2-cell embeddings on the projective plane or torus. Thus, they are good starting points for an embedding algorithm. They are not as effective for the Klein bottle, because they are not obstructions for the projective plane, and even allow non-2-cell embeddings of two disjoint copies.

16.1.1 Rotation systems

Consider the point of Figure 16.2 at the midpoint of the left edge of the rectangle. If we walk around this point in a small clockwise circle, it begins with the arcs marked 1 and 2, then continues counterclockwise with the arcs marked 3 and 4, to produce a

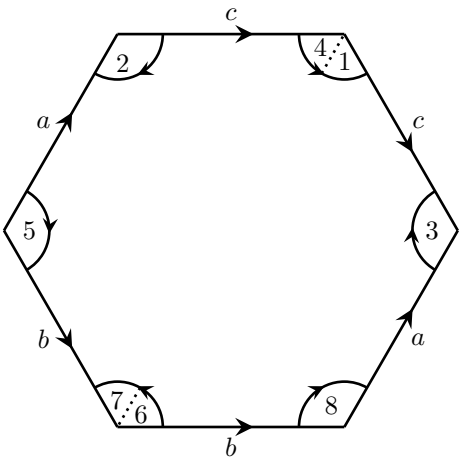


FIGURE 16.4
A hexagonal form of the Klein bottle

cycle $(1, 2, 3, 4)$. Similarly, if we start at the bottom of the left edge, and walk around a small clockwise circle, we find the cycle $(5, 6, 7, 8)$. If we then look for these cycles in Figure 16.4 we find them in the locations indicated, with the given orientations. This will be helpful in understanding the *two traversals* of the Jordan curves b and c in the hexagonal model, when a graph is embedded on the surface.

As the Klein bottle is a non-orientable surface, an embedding of a graph G on it is represented by a signed rotation system. We will denote a Klein bottle embedding by G^{κ}, where κ is a signed rotation system. Now the Klein bottle can be represented in several different ways as a polygonal disc, with pairs of sides identified. A graph edge uv which crosses a side of the polygon from one surface of the polygonal disc to the opposite surface will have a sign of -1. An edge which crosses from one surface of the disc to the same surface, or which does not cross a boundary, will have a sign of $+1$. If a vertex is placed along an edge of the polygon, the determination of the sign of incident edges can be ambiguous. Therefore we will always move all such vertices slightly (even an infinitesimal amount), so that they do not lie on the boundary, and then determine the signatures of all incident edges. Similarly, we must be careful with edges that cross through the corners of the polygon, as this can also be ambiguous. We will always reroute them slightly so that they cross the polygonal edges meeting at the corner separately, not both simultaneously. In the representation $a^+b^+a^+b^-$, edges which cross the essential cycle a will have a sign of -1. In the representation $a^+c^+c^+a^-b^-b^-$, edges which cross the cycles c or b will have a sign of -1, but all others will have a sign of $+1$, etc. The signed rotation system representing the embedding in Figure 16.5(i) is the following:

1	2,-4, 6
2	1, 5,-3
3	2, 4, 6
4	3,-1, 5
5	2, 4, 6
6	1, 5, 3

Given a signed rotation system, the facial boundaries can be traced out using Algorithm 15.7.1, FACIALCYCLESGN(). *The faces are independent of the polygonal representation used for the Klein bottle.* When one representation is converted to another, using the cuts shown in Figures 16.2, 16.3, and 16.4, the signatures of some of the edges change. Therefore, a signed rotation system based on a 2-cell embedding on $a^+b^+a^+b^-$ may not be a 2-cell embedding, or even an embedding at all, on $a^+c^+c^+a^-b^-b^-$ or $a^+a^+b^+b^+$, and vice versa. Thus the *edge signatures of a rotation system depend on the polygonal representation being used*. They are not part of the embedding, but are necessary so that the facial walks can be determined by the rotation system. Algorithm 15.7.1 will work for every polygonal representation, although a particular rotation system will have been constructed using a given polygonal representation.

DEFINITION 16.1: A *Klein map* is a combinatorial embedding G^κ of a 2-connected graph G on the Klein bottle, where κ is a signed rotation system corresponding to a given polygonal representation of the Klein bottle.

Equivalence of Klein maps is defined in terms of their faces. When a facial walk is traversed, its edges are followed in the direction of traversal. Each edge of a facial walk occurs in exactly one other facial walk (which may be the same facial walk). Therefore there is only one way to glue the facial walks together. Once they have been glued, the result is the Klein bottle, with a graph embedded on it. The facial walks can also all be reversed, and the result is the same, because the surface is non-orientable.

DEFINITION 16.2: Klein maps G^{κ_1} and G^{κ_2} are *equivalent* or *isomorphic* if there is a permutation of $V(G)$ that maps the collection of facial walks of G^{κ_1} to those of G^{κ_2}, or to those of $G^{\overline{\kappa}_2}$.

Two distinct 2-cell embeddings of $K_{3,3}$ on the Klein bottle are shown in Figure 16.5. They appear to be almost identical to the embeddings of $K_{3,3}$ on the torus (see Figure 15.2), but the facial walks are different. This observation gives the following theorem.

Theorem 16.1. *Let G^t be a 2-cell embedding on the rectangular form of the torus of a graph G. Let C be an essential Jordan curve in the torus, intersecting G^t in at most two points, which are not vertices. Then G^t can be transformed into a 2-cell embedding G^κ on the Klein bottle.*

Proof. Cutting the torus along C creates a cylinder, so that we can draw the torus as a rectangle with C as the right and left sides of the torus rectangle. Choose a

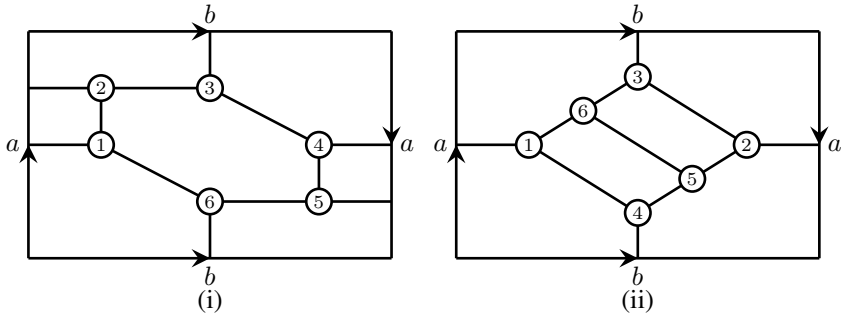
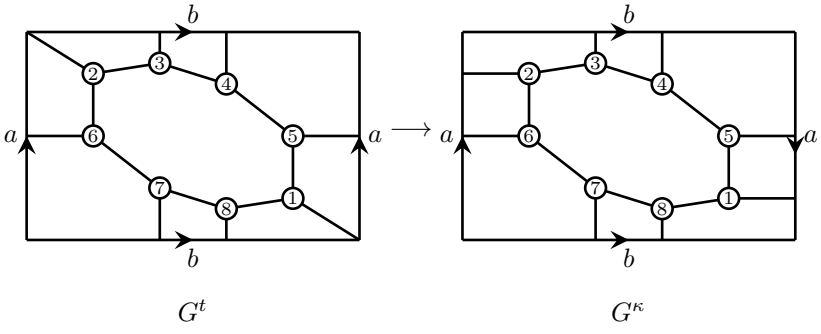


FIGURE 16.5
Two embeddings of $\boldsymbol{K_{3,3}}$ on the Klein bottle

rectangular representation of the Klein bottle of the form $a^+b^+a^+b^-$, with a on the right and left sides, as in Figure 16.1. Suppose first that only one edge uv crosses C, draw G^t so that uv crosses C in the center of the right boundary of the torus rectangle. This is also an embedding on the rectangular representation of the Klein bottle. (An example is diagram (ii) of Figure 16.5.) Thus, a rotation system for G^κ representing G embedded on the Klein bottle is identical to G^t, except that the edge uv now has a sign of -1. Note that the faces (regions) of G^κ will be the same as those of G^t, although the facial walk containing uv will be different in G^t and G^κ.

If C intersects two edges of G^t, let the two edges be uv and xy, crossing C from left to right in the rectangular representation. Without loss of generality, we assume that uv crosses C above the point where xy crosses C. Refer to Figure 16.6, where $u = 5$, $v = 6$, $x = 1$, and $y = 2$. See also Figure 16.5 (i). Furthermore, we can take $v \neq y$ (otherwise $u \neq x$ and we can perform a horizontal flip on the torus rectangle, and relabel the points). Now G^t is a 2-cell embedding, so if we start at u and follow the facial boundary W beginning with uv, the vertices u, v, y, x occur on W, in that order. Let B be the Jordan curve representing the top and bottom side of the torus rectangle. Now uv or xy may contain the intersection point of B and C, as in Figure 16.6. If this is not the case, as in Figure 16.5(i), we adjust the embedding slightly to make it so, as follows. If uv or xy cross both B and C, say xy, but not at their intersection, we subdivide the edge which crosses B with a new vertex z, so that xz crosses C but not B, and then take xz in place of xy. Then uv and xy do not intersect B. The walk W bounds a disc face, so that the torus can be distorted slightly, by moving vertex y downwards, below the bottom side of the rectangle, until edge xy intersects the bottom-right corner of the torus rectangle, where B and C meet. This has been done in the example of the left diagram of Figure 16.6, where the edge $xy = 12$ intersects the bottom-right corner of the torus rectangle. But the rectangle $a^+b^+a^-b^-$ representing the torus can now be viewed as a rectangle $a^+b^+a^+b^-$ representing the Klein bottle. The edge xy then touches B, but does not cross it, because the orientation of the right a-side of the rectangle has been reversed.

The curve representing edge xy can then be redrawn so that the point where it previously intersected C and B is moved upwards slightly along C, so that edge xy no longer intersects B. This is illustrated in the right diagram of Figure 16.6. Any subdividing vertices that were added are then removed. The facial boundaries of G^t that do not contain uv or xy are the same in G^κ. The union of the faces whose boundaries contain uv or xy in G^t, plus the edges uv and xy, comprise the faces that touch the right boundary of the torus rectangle. Because G^t is a 2-cell embedding, their union forms a cylinder. In G^κ their union also forms a cylinder, though different because the orientation of the right a-side of the rectangle has been reversed. When the cylinder is cut along uv and xy, 2-cell faces result. Thus the embedding G^κ is a 2-cell embedding. □



G^t G^κ

FIGURE 16.6
Converting $\boldsymbol{G^t}$ to $\boldsymbol{G^\kappa}$

Note that the condition that the Jordan curve C intersects G^t in at most two points is necessary for the proof to work, because the orientation of one of the a-sides of the torus rectangle is reversed when transforming it to a Klein bottle.

A similar construction can be used to transform a Klein map to a torus map. There are several kinds of essential Jordan curves on the Klein bottle (see the exercises at the end of this section). Notice that a non-contractible Jordan curve that cuts b in the rectangular form $a^+b^+a^+b^-$ creates a cylinder when cut. Call this a *cylindrical* Jordan curve.

Theorem 16.2. *Let G^κ be a 2-cell embedding of a graph G in the Klein bottle. Let C be an essential cylindrical Jordan curve in the Klein bottle, intersecting G^κ in at most two points, which are not vertices. Then G^κ can be transformed into a 2-cell embedding G^t on the torus.*

Proof. The proof is the inverse transformation of Theorem 16.1. □

A similar transformation works for graphs embedded on the projective plane.

Theorem 16.3. *Let G^π be a 2-cell embedding of a graph G in the projective plane. Let C be an essential Jordan curve in the projective plane, intersecting G^π in a non-vertex point of edge xy, such that $G^\pi - xy$ is also a 2-cell embedding. And suppose that C intersects at least one other edge of G. Then G^π can be transformed into a 2-cell embedding G^κ on the Klein bottle.*

Proof. Draw the projective plane so that C is the bounding circle, so that the edge xy and the other edges intersecting C have signature -1. Represent the Klein bottle as a rectangle $a^+b^+a^+b^-$, and let P be the point of intersection of C with edge xy. We assume that G^π is finite, so that a portion of C adjacent to P can be chosen as b, and the remainder of C is chosen as a. The disc of the projective plane can then be deformed into a rectangle such that xy crosses the side b (and therefore has signature $+1$ in the Klein bottle), and the remaining edges of signature -1 cross the curve a, and therefore have signature -1. The result is a Klein map G^κ. The facial walks of G^κ that do not contain xy are the same as in G^π, and therefore these faces are 2-cells in G^κ. The face of G^κ whose boundary contains xy is homeomorphic to the union of the faces of G^π whose boundary contains xy. Now $G^\pi - xy$ is also a 2-cell embedding. It follows that in G^κ, the face whose boundary contains xy is the union of two 2-cells and the single common edge xy on their boundaries. Therefore G^κ is a 2-cell embedding. □

An example of transforming a projective embedding into a Klein map is shown in Figure 16.7, where an embedding of the Petersen graph on the projective plane is converted to an embedding on the Klein bottle. Clearly the transformation of Theorem 16.3 can also be done in reverse to transform certain Klein maps into projective maps.

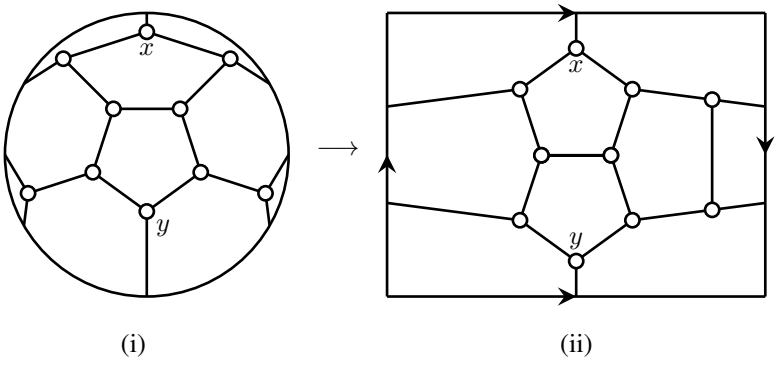


FIGURE 16.7
Converting G^π to G^κ, the Petersen graph

Just as a 2-cell planar map G^p can be transformed into a torus map using a theta subgraph of G, we can use a barbell subgraph to transform G^p into a Klein map G^κ.

DEFINITION 16.3: A *barbell graph* consists of a uv-path P, a cycle C_u containing u, and a cycle C_v containing v such that C_u and C_v are vertex-disjoint; P and C_u have only u in common; and P and C_v have only v in common. Refer to Figure 16.8.

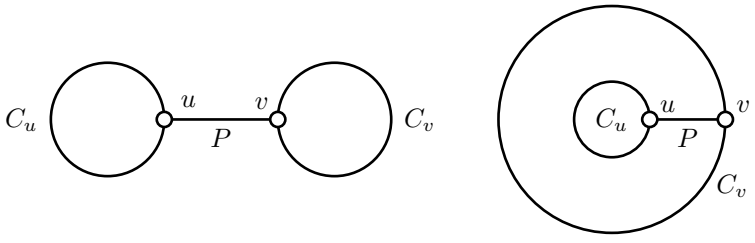


FIGURE 16.8
Schematic of a barbell graph, two planar embeddings

Let G^p be a 2-connected planar map containing a barbell subgraph H. The following theorem shows how to convert G^p to a Klein bottle embedding of G.

Theorem 16.4. *Let G^p be a 2-connected planar map containing a barbell subgraph H. Then G^p can be converted to a Klein bottle embedding G^κ, such that the subgraph H forms the boundary of the hexagon representation of the Klein bottle.*

Proof. Suppose first that in the embedding G^p in the plane, the barbell is embedded as in the left diagram of Figure 16.9. Let A be the part of G embedded outside the barbell, and let B and C be the parts embedded inside the Jordan curves C_u and C_v, respectively. The orientations of A, B, and C in G^p are indicated by arrows in Figure 16.9. The barbell itself can be embedded in the hexagonal form of the Klein bottle, with the vertices and edges of the barbell forming the boundary of the hexagon. Note that the cycles C_u and C_v and path P are each traversed twice in the Klein bottle. The two traversals of C_u and C_v have the same orientation on the boundary of the hexagon; those of P have opposite orientations. The two traversals always indicate the region of the Klein bottle to the right and left sides of the path as it is followed. If we walk along P from u to v in G^p, there may be edges to A on both sides of P. The edges on the left side of P attach to A in the counterclockwise direction in G^p. Those on the right side of P attach to A in the clockwise direction. These same attachments to A occur in the Klein bottle in Figure 16.9, in the same order. The right and left sides of P agree in G^p and G^κ.

Following v, we walk along C_v in G^p clockwise from v, in the direction opposite to the arrow. Edges on the outside of C_v to A in G^p continue in the counterclockwise direction, and also appear counterclockwise in the Klein bottle diagram. Edges on the inside of C_v in G^p are indicated on the second traversal of C_v in the Klein bottle. The orientation of C_v requires that C be reversed in G^κ for these edges to attach in the correct order. This is indicated in the diagram.

Then, we walk backwards along P in G^p, from v to u, looking at edges to the

right of P. There can be edges to C_v, A, or C_u, in that order. Possible edges from P to C_u determine that the direction of traversal of C_u in G^p must be clockwise, as indicated by the arrow. Then C_u is traversed. Edges to B on the other side of C_u in G^p attach in a counterclockwise direction. This requires that the orientation of B be reversed in the Klein bottle, as indicated in the diagram.

We can now read the rotation system for G^κ from the diagram. All vertices of B and C have their rotations reversed. All other vertices have the same rotations in G^p and G^κ. In order to determine the signatures of the edges of G^κ, it is necessary to move the vertices of the barbell slightly, so that they are not on the boundary of the hexagon, and it can then be determined which edges cross the Jordan curves C_u and C_v.

In the event that in G^p the barbell is embedded as in the right diagram of Figure 16.8, the proof is almost identical (to be completed as an exercise).

□

It is evident from Figure 16.9 that there are several other ways to convert the planar rotation system to a Klein bottle rotation system.

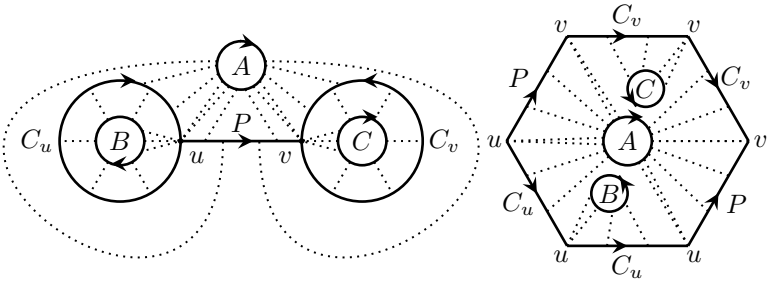


FIGURE 16.9
Converting a planar rotation system to a Klein bottle rotation system

Theorem 16.5. *There are two 2-cell embeddings of $K_{3,3}$ on the Klein bottle.*

Proof. By Theorem 15.27 there is a unique embedding of $K_{3,3}$ on the projective plane (see Figure 15.34). The embedding has two kinds of edges – those that separate two quadrilateral faces, and those that separate a hexagon and a quadrilateral. When Theorem 16.3 is used to convert the projective embedding of $K_{3,3}$ to a Klein map, the edge xy of the theorem can be either type of edge. The result is the two Klein embeddings of $K_{3,3}$ in Figure 16.5.

Let G^κ be an arbitrary embedding of $K_{3,3}$ on the Klein bottle represented as $a^+b^+a^+b^-$. If the Jordan curve b is intersected by only one edge xy, then the transformation of Theorem 15.27 can be used in reverse to obtain a projective embedding of $K_{3,3}$, so that G^κ must be one of the embeddings obtained from the projective plane. Otherwise there are at least two edges that intersect b.

Similarly there are two distinct embeddings of $K_{3,3}$ on the torus, by Theorem 15.14, shown in Figure 15.2. To use Theorem 16.1 an essential Jordan curve C must be chosen that intersects either one or two edges of the torus embedding. Consideration of Figure 15.2 shows that C can cross at most one facial walk, meeting it either in one edge, or two. As there are only three faces, all possibilities for C are quickly determined, resulting once again in the two Klein maps of Figure 16.5. If the Klein bottle contained a cylindrical Jordan curve intersecting G^κ in at most two edges, then Theorem 16.2 can be used to convert the embedding to one of the torus embeddings of $K_{3,3}$. Otherwise, at least three edges of $K_{3,3}$ must intersect every cylindrical Jordan curve.

Given the rectangular representation $a^+b^+a^+b^-$ of the Klein bottle, a 2-cell embedding must contain a Jordan curve intersecting the b-side of the rectangle, and another intersecting the a-side. Hence the Jordan cycles must induce a theta graph in $K_{3,3}$. Now $K_{3,3}$ is bipartite, so that the shortest possible essential cycle induced by the Jordan curves must have length either four or six. There are five possibilities for this theta graph, three of which are shown in Figure 16.10. The other two are obtained by rotating two of the rectangles through 90 degrees.

Is it possible that every cylindrical Jordan curve intersects G^κ in at least three edges, and that the curve b is intersected in at least two edges of G^κ?

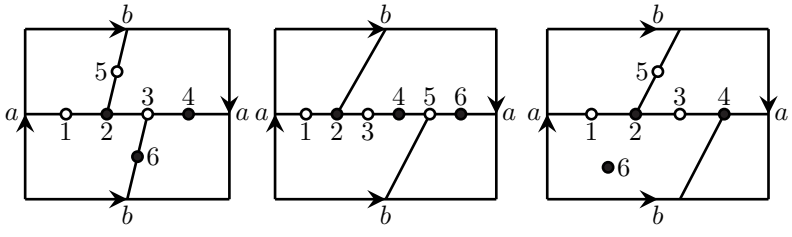


FIGURE 16.10
Three theta graphs on the Klein bottle.

In each theta graph embedding there is a single face, whose facial walk has length 12 or 14. The facial walks are shown in Figure 16.11. The edges missing from the first theta graph are $(1, 6)$ and $(4, 5)$. Those missing from the second are $(1, 4)$ and $(3, 6)$. Their possible locations are shown as dotted lines in the diagram. In the third theta graph the edges $(1, 6)$ and $(3, 6)$ can be drawn as shown without loss of generality, leaving two choices for $(5, 6)$, which are dotted. In each case, non-intersecting lines must be chosen for the missing edges. Each possibility leads to an embedding which can be converted either to a projective embedding or a torus embedding. The missing two theta graphs can be completed as an exercise. □

The other Kuratowski graph for the plane, K_5, has 14 2-cell embeddings on the Klein bottle. This is more difficult to prove.

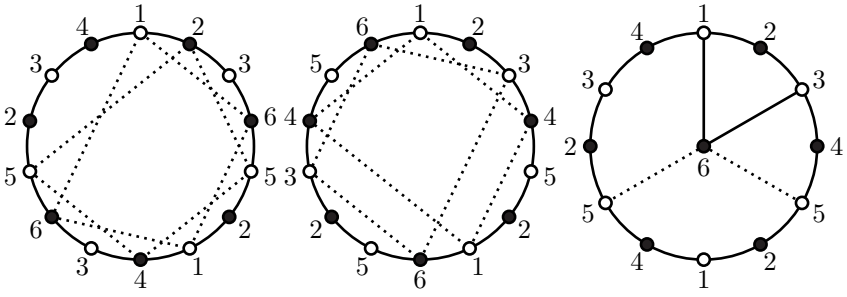


FIGURE 16.11
The face of a theta graph.

16.1.2 The double cover

Given a rectangular representation $a^+b^+a^+b^-$ of the Klein bottle, we can make an identical copy of it, $a'^+b'^+a'^+b'^-$, flip the copy over, align the two copies so that corresponding a and a' sides match, and then glue the sides a and a' together, as shown in Figure 16.12. The result is a torus $a^+c^+a^-c^-$, where $a = a'$ and $c = bb'$. Thus, the torus is a double cover of the Klein bottle. If there is a graph G^κ embedded on the Klein bottle, then there will be a graph H^t embedded on the torus, where H is a double cover of G. Just as for the projective plane, this provides a means to determine equivalence of Klein bottle embeddings, and to determine their automorphism groups.

DEFINITION 16.4: Let G^κ be a graph embedding on the Klein bottle. The *automorphism group* of G^κ is $\text{AUT}(G^\kappa)$, the set of all permutations of $V(G)$ that map the collection of facial walks of G^κ to itself, or to the facial walks of $G^{\overline{\kappa}}$.

The double cover of a Klein map G^κ is very similar to the double cover of a projective map. Let the representation of the Klein bottle be $a^+b^+a^+b^-$. We assume that G^κ is a 2-cell embedding of a 2-connected graph. A copy $(G')^{\kappa'}$ of the embedding is made, and they are connected through the edges that cross the a boundary. Each vertex $v \in V(G)$ has a corresponding *antipodal* vertex v'. An edge uv with $\text{SGN}(uv) = -1$ is doubled to obtain edges uv' and $u'v$. Let $DC(G^\kappa)^t$ denote the toroidal double cover constructed in this way. There is a two-to-one *double cover map* γ from $V(DC(G^\kappa)^t)$ to $V(G)$ that induces a two-to-one mapping of edges and of facial walks. Given a facial walk W of G^κ, there are two facial walks W_1 and W_2 that correspond to W. W_2 consists of the antipodal vertices of W_1, but reversed. Every automorphism $\phi \in \text{AUT}(G^\kappa)$ induces an automorphism θ of $DC(G^\kappa)^t$ that maps antipodal pairs to antipodal pairs. And every such automorphism θ induces an automorphism ϕ of G^κ. See Lemma 15.23 and Theorems 15.24 and 15.25. The proofs are nearly identical.

Thus, the *antipodal medial digraph* of $DC(G^\kappa)^t$ can be used to determine $\text{AUT}(G^\kappa)$. It is formed from the medial digraph of $DC(G^\kappa)^t$ by adding a suitable

path of length two or more connecting every pair $\{v, v'\}$ of antipodal vertices. Similar to Theorem 15.26, we have:

Theorem 16.6. *Let G^κ be a Klein map with double cover $DC(G^\kappa)^t$, and double cover map γ. Let M^+ be the antipodal medial digraph of the double cover. Then* $\text{AUT}(G^\kappa)$ *is obtained from* $\text{AUT}(M^+)$ *by restricting each element of* $\text{AUT}(M^+)$ *to $V(DC(G^\kappa)^t)$, and then transforming it by γ.*

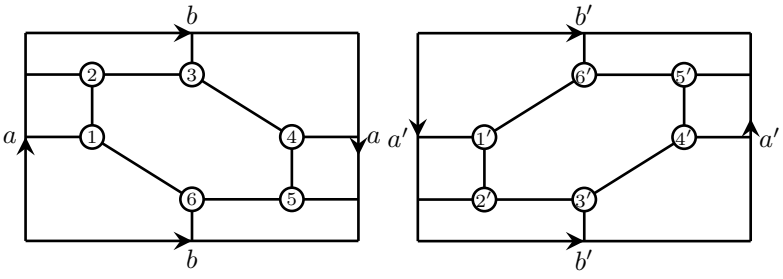


FIGURE 16.12
The torus is a double cover of the Klein bottle

This provides a simple combinatorial means to determine equivalence of Klein maps. The automorphism groups of the two inequivalent Klein maps of $K_{3,3}$ shown in Figure 16.5 have orders 4 and 2. The group of the $K^\kappa_{3,3}$ in the left diagram contains non-identity permutations $(1,4)(2,3)(5,6)$ and $(1,2)(3,4)(5,6)$ and $(1,3)(2,4)$. The group of the $K^\kappa_{3,3}$ on the right is generated by $(1,4)(2,3)(5,6)$.

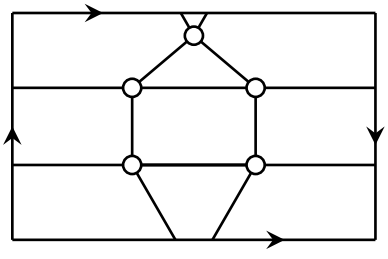


FIGURE 16.13
An embedding of $\boldsymbol{K_5}$ on the Klein bottle

Exercises

16.1.1 Write down the signed rotation systems for the Klein maps of Figure 16.5. Use Algorithm 15.7.1 to find the facial cycles.

16.1.2 Determine whether the rotation systems of the previous exercise are em-

beddings on the representations $a^+c^+c^+a^-b^-b^-$ and $a^+a^+b^+b^+$ of the Klein bottle.

16.1.3 Convert the Klein embedding of Figure 16.6 to an embedding on the hexagonal form of the Klein bottle, using the cuts of Figure 16.3. Then write down the rotation system for the hexagonal embedding.

16.1.4 Determine the possible results if the Klein bottle is cut along a single non-contractible Jordan curve, for each of the forms $a^+b^+a^+b^-$, $a^+a^+b^+b^+$, and $a^+b^+b^+a^-c^+c^+$. Is it possible to cut the Klein bottle into two pieces with just one non-contractible Jordan curve? If so, what are pieces?

16.1.5 Convert the Klein maps of Figure 16.5 to embeddings on the representations $a^+c^+c^+a^-b^-b^-$ and $a^+a^+b^+b^+$ of the Klein bottle, and write down their rotation systems. Then use Algorithm 15.7.1 to find the facial cycles.

16.1.6 Find the facial walks of the embedding of the Petersen graph on the Klein bottle in Figure 16.7.

16.1.7 Find an embedding of K_6 on the projective plane, and then use Theorem 16.3 to convert it to an embedding on the Klein bottle.

16.1.8 A Klein map of K_5 is shown in Figure 16.13. Find its automorphism group.

16.1.9 Complete the proof of the second case of Theorem 16.4.

16.1.10 Find the double cover of the Petersen graph embedded on representation $a^+b^+a^+b^-$ of the Klein bottle, from Figure 16.7. Then use it to find the automorphism group of this Klein map.

16.1.11 Given the standard-form representation $a^+a^+b^+b^+$ of the Klein bottle, show that there is a double cover that is also a Klein bottle. Given the Klein bottle map G^κ of K_5 in Figure 16.13, find its double cover as another Klein map.

16.2 The double torus

The *double torus*, also known as the *2-holed torus*, is illustrated in Figure 16.14 as a "doughnut with two holes".

It is the unique orientable surface of genus two, and can be represented in standard form as an octagon, with sides labelled $a^+b^+a^-b^-c^+d^+c^-d^-$. The conversion between the octagon and the doughnut is shown in Figure 16.15. (This diagram is based on a diagram in Hilbert and Cohn-Vossen [83].) Inspection of the octagon shows that the origin of the a-side is also the terminus of the b-side, which is also the terminus of the a-side. Further inspection then shows that all eight vertices of the octagon are the same point. In Figure 16.15, the octagon is first shaped so as to glue

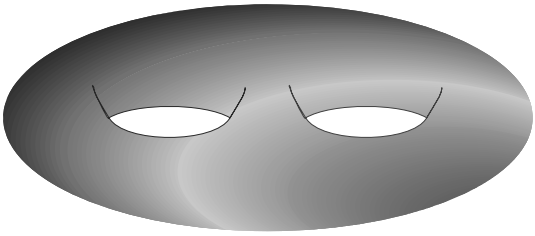


FIGURE 16.14
The double torus

the b and d edges. Then the a and c sides are formed into closed curves, and glued to the corresponding a and c sides, thereby producing the double torus.

If we wanted to tile the plane with regular octagons, as the plane was tiled with rectangles by the torus, we would need to have two edges labelled with each of a, b, c, d at each vertex, i.e., there would be eight octagons meeting at each corner. The interior angle at each vertex would then need to be $2\pi/8$. However, in the Euclidean plane, the interior angle of a regular octagon is $3\pi/4$, so that this cannot be done in the Euclidean plane. Fortunately, there is another plane that can be tiled by regular octagons, the *hyperbolic* plane.

The hyperbolic plane is a non-Euclidean plane, in which there are many lines parallel to a given line ℓ through a point P not on ℓ. It was discovered independently by Lobachevsky [1829] and Bolyai [1832] (see [117] and [163]) as a consequence of attempting to prove the parallel postulate of Euclidean geometry. In the hyperbolic plane there are several kinds of parallel lines. There are lines which do not meet, but which become arbitrarily close as they move towards infinity, and those that always maintain a minimum distance. The first type of parallel lines can be called *asymptotic lines*. The others can be called *ultra-parallel*. Much information on the hyperbolic plane can be found in the books [162, 160, 93, 140, 135]. We outline some of the key ideas of the geometry of the hyperbolic plane. Just as in section 15.3, we saw that isometries of the Euclidean plane can be used to produce the torus as a factorizaton of the Euclidean plane, it turns out that the isometries of the hyperbolic plane produce the double torus as a factorization of the hyperbolic plane.

There are various models of the hyperbolic plane — the *half-plane model*, the *Poincaré disc model*, and the *Beltrami-Klein disc model*. We will use the half-plane model and the Poincaré disc model of the hyperbolic plane. Let $\mathbb{H}$ denote the half-plane model. It consists of all complex numbers $z = u + iv$, such that $v > 0$. Hyperbolic lines are modeled by vertical Euclidean lines (with equation $z = u =$ constant), and by Euclidean semi-circles whose centers are on the x-axis (with equation $|z - c| = r$, where $c, r \in \mathbb{R}$ are constants, and $r > 0$). Any two of these intersect in at most one point, and they are determined by any two points they contain. Thus, they have properties that lines in the Euclidean plane satisfy.

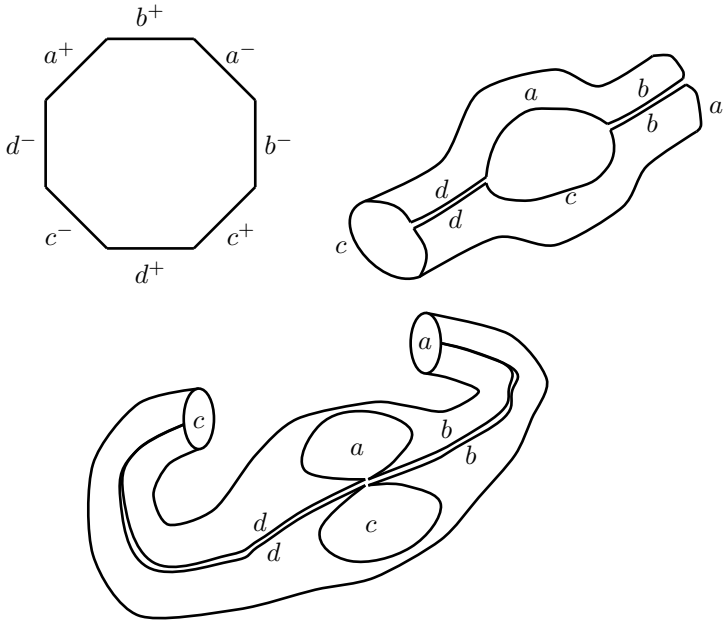


FIGURE 16.15
Gluing an octagon into a double torus

The mapping

$$z \mapsto \frac{iz+1}{z+i}$$

is then used to map $\mathbb{H}$ to the interior of the unit disc in the complex plane. The interior of this disc will be denoted $\mathbb{D}$. It is the Poincaré disc model of the hyperbolic plane. The x-axis gets mapped to the boundary of the disc, which becomes a *limit* circle — as a line in the disc approaches the circle, distances become greater and greater, so that the bounding circle is never reached. It represents an infinitely distant circle bounding the hyperbolic plane. Vertical lines of $\mathbb{H}$ map to diameters of $\mathbb{D}$, so that diameters of the unit circle are lines in the $\mathbb{D}$ model of the hyperbolic plane. The semi-circles representing lines in $\mathbb{H}$ map to portions of circles that intersect the limit circle at right angles (however the centers of these circles are outside the limit circle). So the lines of $\mathbb{D}$ are modelled in two ways:

a) as diameters of the limit circle;
b) as semi-circles that intersect the limit circle at right angles.

Parallel hyperbolic lines can meet on the limit circle (which is not part of the plane), these represent asymptotic lines. Or they do not meet at all, these represent ultra-parallel lines. There can be very many hyperbolic lines parallel to a given hyperbolic

line. Figure 16.16 shows several lines in the Poincaré disc model. Some of them are asymptotic, and some are ultra-parallel lines.

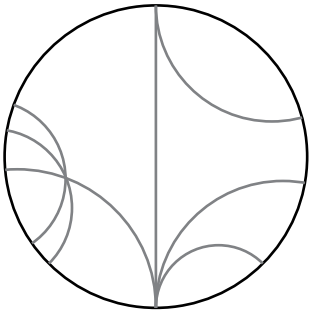


FIGURE 16.16
The Poincaré disc model of the hyperbolic plane

16.2.1 Isometries of the hyperbolic plane

In section 15.3, we saw that the isometries of the Euclidean plane are reflexions, translations, rotations, and their combinations, and that all Euclidean isometries are generated by reflexions. In the hyperbolic plane there are also reflexions, rotations, translations, as well as other isometries, and their combinations. And again, they are all generated by the reflexions. However, these are not Euclidean reflexions.

DEFINITION 16.5: Let a circle of radius r in the complex plane have center $c \in \mathbb{C}$. *Inversion* in the circle is the map that sends a point z at distance ρ from c to the unique point at distance r^2/ρ from c along a ray through z from c. It has the equation

$$z \mapsto c + \frac{r^2}{\overline{z} - \overline{c}}$$

Inversion in a circle maps the circle to itself, and exchanges the interior and exterior of the circle. The center is mapped to infinity, and vice-versa. In $\mathbb{H}$, inversion in a semi-circle representing a hyperbolic line is considered a reflexion in the hyperbolic line. In the limiting case when the semi-circle becomes a vertical line, inversion becomes a Euclidean reflexion in the line. When $\mathbb{H}$ is mapped to $\mathbb{D}$, we find that reflexions in the lines of $\mathbb{D}$ are also inversions in the circles. Thus, hyperbolic reflexions in $\mathbb{D}$ are modeled as inversions in the Euclidean circles that represent lines, and as reflexions in Euclidean lines that are diameters of the unit circle.

Just as is the case of the Euclidean plane (section 15.3), it turns out that all hyperbolic isometries are products of hyperbolic reflexions. Hyperbolic isometries are slightly different from Euclidean isometries, but there are great similarities. For example, the product of two reflexions in ultra parallel hyperbolic lines produces a hyperbolic translation. The following theorem is from Stillwell [162].

Theorem 16.7. *Every hyperbolic isometry can be written as a product of at most three hyperbolic reflexions.*

Reflexions in $\mathbb{D}$ will give us transformations of graph embeddings on the double torus. Similarly to Gauss's Theorem 15.7 on spherical isometries, the isometries of the hyperbolic plane can be written as Möbius transformations, as shown by Poincaré [136] (see Stillwell [162]).

Theorem 16.8. *The orientation-perserving isometries of* $\mathbb{D}$ *are given by*

$$w \mapsto \frac{\alpha w + \beta}{\overline{\beta} w + \overline{\alpha}}$$

where $w \in \mathbb{C}^+$, $\alpha, \beta \in \mathbb{C}$, *and* $|\alpha|^2 - |\beta|^2 = 1$.

Theorem 16.9. *The orientation-reversing isometries of* $\mathbb{D}$ *are given by*

$$w \mapsto \frac{\alpha \overline{w} + \beta}{\overline{\beta} \overline{w} + \overline{\alpha}}$$

where $w \in \mathbb{C}^+$, $\alpha, \beta \in \mathbb{C}$, *and* $|\alpha|^2 - |\beta|^2 = 1$.

Similar formulas apply to the $\mathbb{H}$ model of the hyperbolic plane. (see Stillwell [162]).

Exercises

16.2.1 Show that a circle of radius r in the Euclidean plane intersects the unit circle at right angles if its center is at distance $\sqrt{1+r^2}$ from the origin.

16.2.2 Show that a regular octagon in the hyperbolic plane can be constructed with adjacent sides meeting at an angle of $\pi/4$ if the lines are represented by Euclidean circles with radius $r = \sqrt{(\sqrt{2}-1)/2}$.

16.2.3 Show that $\mathbb{D}$ can also be tiled by regular octagons with interior angle $\pi/2$ (so that four octagons meet at each vertex).

16.2.2 The double torus as an octagon

In section 15.3 we saw that the Euclidean plane can be factorized by a group of translation isometries so as to produce a torus. This corresponds to a tiling of the Euclidean plane by isometric rectangles. Thus we can say that the torus *is* a factorization of the Euclidean plane by a group of isometries of the plane.

The double torus has standard form $a^+b^+a^-b^-c^+d^+c^-d^-$, represented by an octagon. The Euclidean plane cannot be factorized so as to produce a tiling by regular octagons, but the hyperbolic plane can be factorized by a group of translation isometries so as to produce a double torus, giving a tiling of the hyperbolic plane by regular octagons. So we can say that the double torus *is* a factorization of the hyperbolic plane by a group of isometries — the geometry of the double torus is

hyperbolic. A portion of a tiling of the Poincaré disc model by octagons is shown in Figure 16.17, where the central octagon is *traversed in a clockwise orientation* to obtain $a^+b^+a^-b^-c^+d^+c^-d^-$. In this tiling there are eight octagons meeting at each corner. Adjacent sides of each octagon meet at an angle of $\pi/4$. The octagons are all isometric, even though some appear to be (much) smaller than others, because in the model $\mathbb{D}$, distances increase as one moves closer to the limit circle.

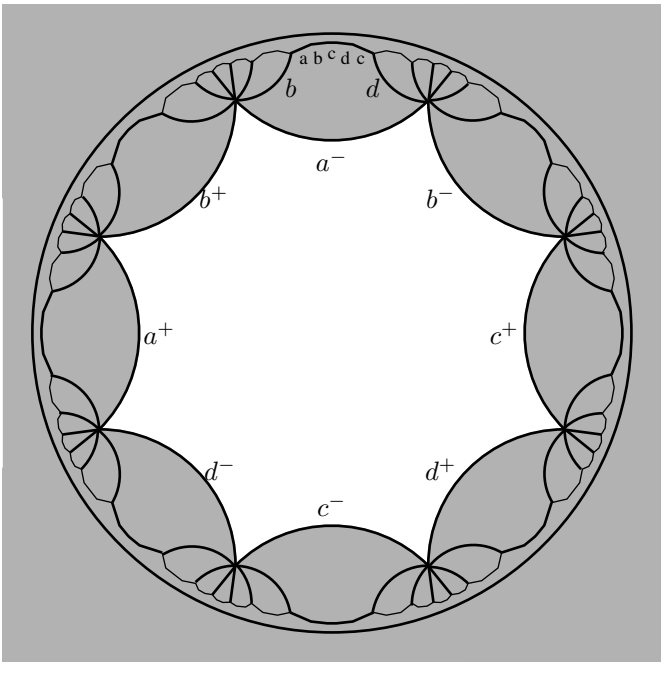


FIGURE 16.17
A partial tiling of the hyperbolic plane by octagons

Consider the corners of the octagon in Figure 16.17. In the tiling they are all equivalent. Two of the corners have an edge labeled a directed outwards. If we walk around those corners in a small clockwise circle, we see that in one of them, the out-edge labelled a is followed on the central octagon by an edge labelled d, directed outwards as the octagon is traversed. There is a second corner with an out-edge labeled d. When we walk around it in a small circle, it is followed by an in-edge labeled c. There is a second corner with an in-edge labeled c. When we walk around it, it is followed by an in-edge labeled d, and so forth. Thus we can determine the cyclic order of edges, it is the same at each corner of the octagon. The result is shown in Figure 16.18, where the superscript indicates the direction of the edge as it is traversed outwards from the corner.

We can now use hyperbolic reflexions to produce translations that map octagons to octagons in this factorization. Let R_{a^+} denote a reflexion in the hyperbolic line

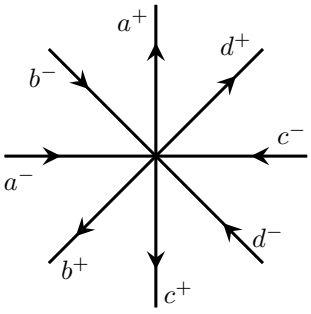


FIGURE 16.18
The edges of the octagon meeting at each corner of the octagon

labelled a^+, and so forth. Let R_{ac^-} denote a reflexion in the diameter of the unit circle that meets the midpoints of a^- and c^-. Consider the product $R_{a^+}R_{ac^-}$ (first R_{a^+}, then R_{ac^-}). Note that the center of the circle representing the hyperbolic line a^+ is outside the unit disc. The reflexion R_{a^+} maps the central octagon to the inside of the a^+ circle. Then the reflexion R_{ac^-} maps it to the inside of the c^+ circle. We see that the central octagon is translated into the c^+ circle. Similarly, the interior of the a^+ circle is translated into the central octagon. The "smaller" octagons contained within the a^+ circle are translated into the various secondary octagons surrounding the central octagon. And these in turn are translated into the "smaller" octagons contained within the c^+ circle. The net effect is a smooth translation moving the points within $\mathbb{D}$ from the top-left towards the bottom-right. Similarly, other translations can be formed using reflexions in other pairs of hyperbolic lines.

The translations of the hyperbolic plane that are isometries of the double torus must respect the labeling of the sides of the octagons: a^+ must map to a^+, b^+ to b^+, and so forth. Consider the product of reflexions $R_{a^-}R_{bd^+}$. The side of the octagon labeled a^- is first mapped to itself by R_{a^-}, then reflected by R_{bd^+} onto the side labeled a^+, such that the orientation of the edges match. The side labeled b^- is first reflected inside the a^- circle onto the edge labeled d^+ (refer to Figure 16.18) by R_{a^-}, then reflected by R_{bd^+} inside the a^+ circle onto the edge labeled b^-. Similarly, the edge labeled $c+$ is first reflected into the a^- circle onto an edge labelled c^- (refer to Figure 16.18). Then it is reflected inside the a^+ circle onto an edge labeled c^+. Similarly the other edges labelled c and d are mapped onto edges of the correct label. Thus the translation $R_{a^-}R_{bd^+}$ is an isometry of the double torus. In the same way, we find that the products $R_{a^+}R_{bd^+}$, $R_{b^+}R_{ac^-}$, $R_{c^+}R_{bd^+}$, etc., are also isometries of the double torus.

We now use these ideas to construct an embedding of $K_{3,3}$ in this hyperbolic model of the double torus, shown in Figure 16.19. The diagram was constructed as follows. The Hamilton path $(2, 1, 6, 5, 4, 3)$ in $K_{3,3}$ was drawn through the central octagon. Then translations that are isometries of the double torus were used to map the six vertices into each of the secondary eight octagons surrounding the central oc-

tagon, and the appropriate edges of $K_{3,3}$ were added to the diagram. Then the vertices in the secondary octagons were translated further into the "smaller" octagons, and so forth. Only the vertices in the central octagon and secondary octagons are shown in the diagram. However, the paths continue through the "smaller" octagons, as indicated. The paths can be determined by considering translations from the central octagon into these "smaller" octagons. The embedding has one face. By tracing the edges in the diagram, we can see that the facial walk of this embedding of $K_{3,3}$ on the double torus is $(2, 3, 4, 5, 2, 1, 4, 3, 6, 5, 4, 1, 6, 3, 2, 5, 6, 1)$, containing all nine edges twice each, and all six vertices three times each on the walk. As each vertex has degree three, the walk enters each vertex three times. The facial walk is shown separately in Figure 16.20.

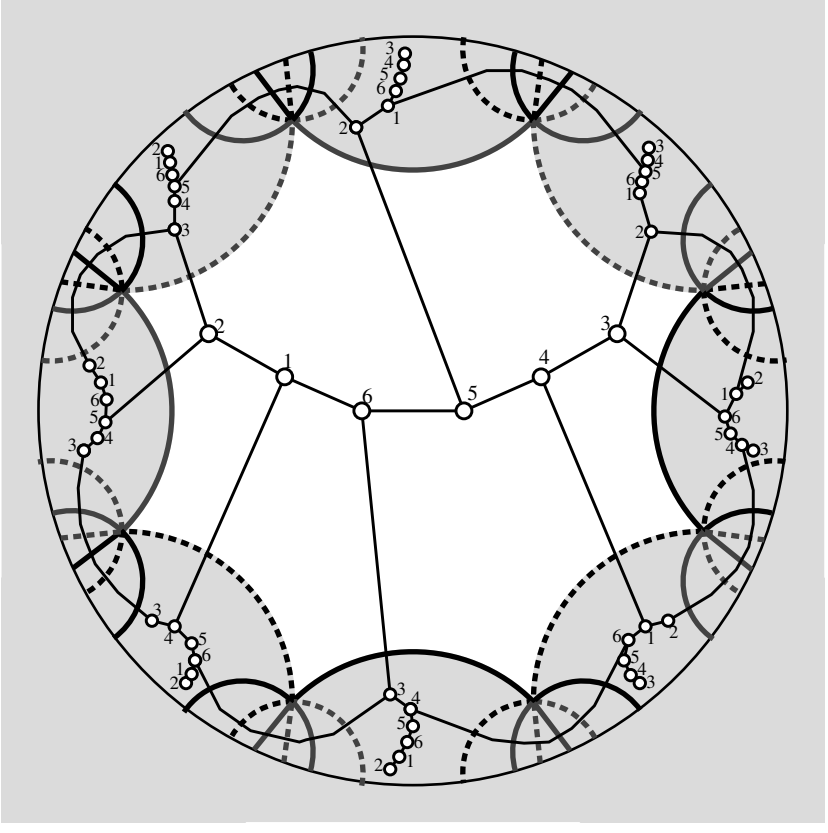


FIGURE 16.19
An embedding of $\boldsymbol{K_{3,3}}$ on the double torus

As with the torus, an embedding on the double torus is represented by a rotation system. We will write G^{τ} to indicate a combinatorial embedding of a graph G on

the double torus, given by the rotation system τ. The rotation system of $K_{3,3}$ in Figure 16.19 can be read off the diagram.

We can now find the automorphism group of this embedding using the facial walk. Notice that vertices 1, 3, and 5 are each distributed evenly around the circle, with five vertices intervening between successive occurrences. But vertices 2, 4, and 6 are each distributed in a "T-fashion", with one occurrence centered between the other two occurrences. We can see that the facial walk can be rotated clockwise by six vertices, giving $(2,4,6)$ as an orientation-preserving automorphism. Thus, the orientation-preserving automorphism group has order three. There are also three orientation reversing automorphisms which can be seen by flipping the facial walk on a $\{2,5\}$-diameter, a $\{4,5\}$-diameter, or a $\{6,5\}$-diameter, giving $(1,3)(4,6)$, $(1,3)(2,6)$, and $(1,3)(2,4)$ as automorphisms.

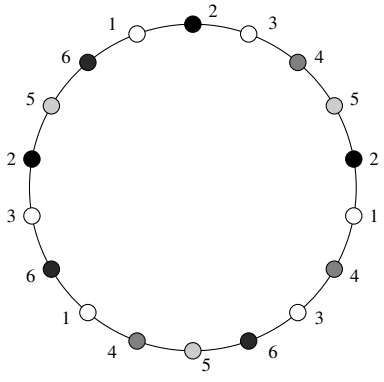


FIGURE 16.20
The facial walk of an embedding of $\boldsymbol{K_{3,3}}$ on the double torus

Equivalence of double torus mappings is defined as for the torus (see 15.15).

DEFINITION 16.6: Double torus maps G^{τ_1} and G^{τ_2} are isomorphic if there is a permutation of $V(G)$ that maps the collection of of facial walks of G^{τ_1} to those of G^{τ_2}.

DEFINITION 16.7: Let G^{τ} be a double torus map. $\text{AUT}(G^{\tau})$ consists of all permutations of $V(G)$ that map the collection of facial walks of G^{τ} to itself.

Similar to Theorem 15.13, we have:

Theorem 16.10. *Double torus embeddings G^{τ_1} and G^{τ_2} are isomorphic if and only if their medial digraphs $M(G^{\tau_1})$ and $M(G^{\tau_2})$ are isomorphic.*

The double torus is an oriented surface, so that there are two automorphism groups — the *orientation preserving automorphism group* of G^{τ} is $\text{AUT}(G^{\tau})$, it is the group induced on $V(G)$ by the automorphisms of $M(G^{\tau})$. The *full automorphism group* of G^{τ} is $\text{AUT}^{+}(G^{\tau})$ consisting of $\text{AUT}(G^{\tau})$, plus those automorphisms that map G^{τ} to $G^{\overline{\tau}}$. In general, the orientation-preserving automorphism

group $\text{AUT}(G^\tau)$ and the full automorphism group $\text{AUT}^+(G^\tau)$ can be found from the medial digraph, as is the case for the other orientable surfaces, such as the plane and the torus.

In Theorem 15.10, a theta subgraph of a plane map was used to construct a 2-cell embedding of an arbitrary planar graph on the torus. A similar method can be used to construct a 2-cell embedding of a graph G on the double torus, given a 2-cell embedding G^t on the torus.

Theorem 16.11. *Let G^t be a 2-cell embedding of a graph G on the torus, and let H be a $TK_{3,3}$ subgraph of G. Then G^t can be converted to a 2-cell embedding G^τ on the double torus.*

Proof. There are two distinct embeddings of $K_{3,3}$ on the torus, shown in Figure 15.2. The subgraph H must must be embedded as one of these. We take each in turn. The embedding on the left of Figure 15.2 has three hexagonal faces, shown in Figure 16.21. Each edge of a hexagon corresponds to a path of the $TK_{3,3}$.

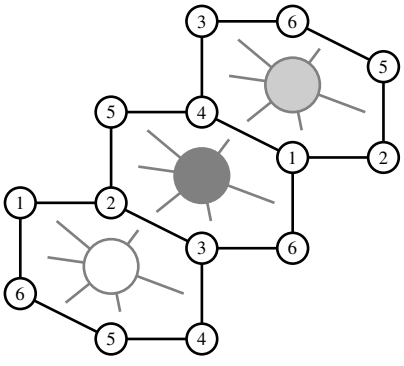


FIGURE 16.21
Three hexagonal faces of $\boldsymbol{K_{3,3}}$ on the torus

The portion of G embedded inside each face is planar, indicated schematically in Figure 16.21. The facial cycles are $(1,2,3,4,5,6)$, $(1,6,3,2,5,4)$, and $(1,4,3,6,5,2)$. Consider now the embedding of $K_{3,3}$ on the double torus, shown in Figure 16.19. It has one face, whose facial walk $(1,2,3,4,5,2,1,4,3,6,5,4,1,6,3,2,5,6)$ is an 18-gon, shown in Figure 16.20. The planar subgraphs of G^t within the three hexagons can be embedded inside the single face of $TK_{3,3}$ on the double torus, as shown in Figure 16.22. The facial cycles of the hexagons form successive portions of the 18-gon. The result is a 2-cell embedding of G on the double torus.

Consider now the second embedding of $K_{3,3}$ of Figure 15.2. Its facial walks are a 10-gon, and two quadrilaterals, shown in Figure 16.23. As before, each face contains a planar portion of G^t. These planar portions of G^t can also be embedded in the face of $K_{3,3}$ on the double torus, as shown in Figure 16.24. Hence, every torus

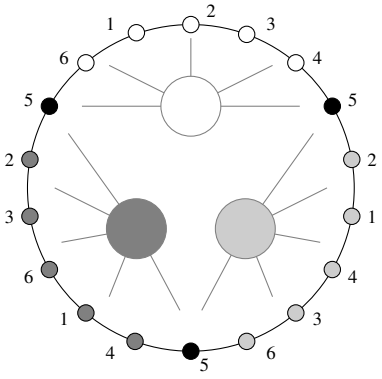


FIGURE 16.22
Converting $\boldsymbol{G^t}$ to $\boldsymbol{G^\tau}$ using hexagonal faces.

embedding G^t containing a $TK_{3,3}$ subgraph can be converted to a 2-cell embedding on the double torus.

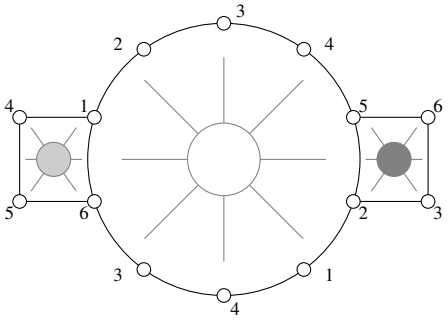


FIGURE 16.23
A 10-gon and two quadrilateral faces of $\boldsymbol{K_{3,3}}$ on the torus.

□

Note that the proof of Theorem 16.11 depends on the numbering of the vertices of the facial walks of $K_{3,3}$ on the torus and double torus. If the vertices of the embeddings of $TK_{3,3}$ in G^t and $K_{3,3}$ on the double torus were numbered differently, one of them would first have to be renumbered before constructing G^τ.

Exercises

16.2.1 Verify that the cyclic ordering shown in Figure 16.18 is correct.

16.2.2 Verify that $R_{a^-}R_{bd^+}$ is an isometry of the double torus, i.e., show that it maps the a, b, c, and d sides of the octagon correctly.

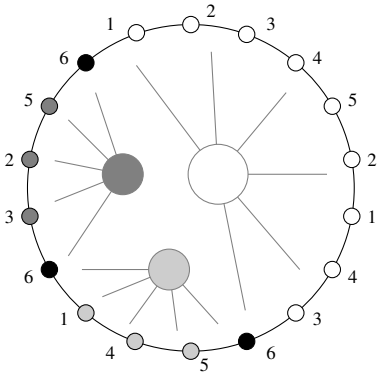


FIGURE 16.24
Converting G^t to G^τ using a 10-gon face.

16.2.3 Given the translation $R_{a^-}R_{bd^+}$, determine the mapping it induces on the central octagon, the left octagon, the top-left octagon, and the top octagon in Figure 16.17.

16.2.4 Find a translation isometry that maps the top octagon of Figure 16.17 onto the bottom octagon.

16.2.5 Show that the group of translation isometries of the double torus is transitive on the octagons of the tiling of Figure 16.17.

16.2.6 Is there a 2-cell embedding of K_4 on the double torus?

16.2.7 Is there a 2-cell embedding of $K_{2,3}$ on the double torus?

16.2.8 Find all 2-cell embeddings of $K_{3,3}$ on the double torus.

16.2.9 Use Theorem 16.11 to find a 2-cell embedding of the Petersen graph on the double torus.

16.2.10 Find a 2-cell embedding of the graph of the cube on the double torus.

16.2.11 Determine whether the hyperbolic plane can be tiled by isometric rectangles, using a group of translations.

16.3 Notes

Some computer drawings of the Klein bottle can be found in APÉRY [4]. Theorems 16.4, 16.1, and 16.3 provide a means of transforming planar maps, torus maps, and projective maps to the Klein bottle. However, an efficient general algorithm to determine 2-cell embeddings of graphs on the Klein bottle is not yet known.

One could say that the double torus "lives" in the hyperbolic plane, as the octagon representing the double torus tiles the hyperbolic plane, by means of hyperbolic translations. The world of hyperbolic geometry is very inclusive. Hyperbolic three-dimensional space contains isometric copies of both the hyperbolic plane, and the Euclidean plane. Hyperbolic geometry is developed axiomatically in IVERSEN [93]. Interesting presentations can be found in STAHL [160], STILLWELL [162], THURSTON [171], and HILBERT and COHN-VOSSEN [83]. Much of the artwork of ESCHER [49] is based on symmetries of the hyperbolic plane. Drawings of tesselations of the hyperbolic plane can be found in MAGNUS [121].

17

Linear Programming

17.1 Introduction

Problems which seek to find a "best" configuration to achieve a certain goal are called *optimization problems*. *Programming problems* deal with determining optimal allocations of limited resources to meet given objectives. They deal with situations in which a number of resources such as manpower, materials, and land are available to be combined to yield one or more products. There are, however, restrictions imposed, such as the total number of resources available and the quantity of each product to be made. *Linear programming* deals with the class of programming problems in which these restrictions and any other relations among the variables are *linear*. In particular, the constraints imposed when searching graphs and networks for say shortest paths, matchings, and maximum flow can be expressed as linear equations.

17.1.1 A simple example

Let us consider a furniture shop that produces tables, chairs, cabinets, and stools. There are three types of machines used: table saw, band saw, and drill press. We assume that the production is continuous and each product first uses the table saw, then the band saw, and finally the drill press. We also assume that setup time for each machine is negligible. Table 17.1 shows

1. The hours required on each machine for each product, and
2. The profit realized on the sale of each product

We wish to determine the weekly output for each product in order to maximize profit. Let x_1, x_2, x_3, and x_4 denote the number of tables, chairs, cabinets, and stools produced per week, respectively. We want to find the values of x_1, x_2, x_3, x_4 which maximizes the profit. The available machine time is limited so we cannot arbitrarily increase the output of any one product. Thus we must allocate machine hours among the products without exceeding the maximum number of machine hours available.

Consider the restriction imposed by the table saw. According to Table 17.1 it will be used a total of

$$1.5x_1 + x_2 + 2.4x_3 + x_4$$

hours per week, but can only be used for at most 2000 hours. This yields the linear

TABLE 17.1
Data for the simple example

Machine type	Table	Chair	Cabinet	Stool	Time available
table saw	1.5	1	2.4	1	2000
band saw	1	5	1	3.5	8000
drill press	1.5	3	3.5	1	5000
profit	5.24	7.30	8.34	4.18	

inequality

$$1.5x_1 + x_2 + 2.4x_3 + x_4 \quad \leq \quad 2000.$$

Similarly, the band saw and drill press yield the following restrictions:

$$\begin{aligned} x_1 + 5x_2 + x_3 + 3.5x_4 &\leq 8000 \\ 1.5x_1 + 3x_2 + 3.5x_3 + x_4 &\leq 5000 \end{aligned}$$

Furthermore, we cannot produce a negative amount of a product, so we also have

$$x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0, \text{ and } x_4 \geq 0.$$

Now we have all the restrictions. The profit is

$$z = 5.24x_1 + 7.30x_2 + 8.34x_3 + 3.4.18x_4.$$

Thus our goal is to solve the following linear program.

$$\begin{aligned} &\text{Maximize:} && z = 5.24x_1 + 7.30x_2 + 8.34x_3 + 3.4.18x_4 \\ &\text{Subject to:} && \begin{aligned} 1.5x_1 + x_2 + 2.4x_3 + x_4 &\leq 2000 \\ x_1 + 5x_2 + x_3 + 3.5x_4 &\leq 8000 \\ 1.5x_1 + 3x_2 + 3.5x_3 + x_4 &\leq 5000 \end{aligned} \\ & && x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0 \end{aligned}$$

17.1.2 Simple graphical example

We can graphically solve linear programs that involve only two variables. For example, consider the linear program

$$\begin{aligned} &\text{Maximize:} && z = 5x_1 + 3x_2 \\ &\text{Subject to:} && \begin{aligned} 3x_1 + 5x_2 &\leq 18 \\ 5x_1 + 2x_2 &\leq 11 \end{aligned} \\ & && x_1 \geq 0, x_2 \geq 0 \end{aligned}$$

Each of the constraints determines a half plane consisting of the set of points (x_1, x_2) that satisfy the inequality. For example, the inequality $x_1 \geq 0$ determines the half plane of points lying to the right of the x_2-axis. The intersection of these four half planes is the set of *feasible solutions* to the problem. In this example the feasible solutions form a bounded polygonal region. It is not difficult to show that such a region must be convex. (It is possible to have an unbounded region.) The four vertices of this region are easily determined by pairwise solving the four bounding equations.

1. Solving $x_1 = 0$ and $3x_1 + 5x_2 = 18$ gives the vertex $(0, 3.6)$.
2. Solving $3x_1 + 5x_2 = 18$ and $5x_1 + 2x_2 = 11$ gives the vertex $(1, 3)$.
3. Solving $5x_1 + 2x_2 = 11$ and $x_2 = 0$ gives the vertex $(2.2, 0)$.
4. Solving $x_1 = 0$ and $x_2 = 0$ gives the vertex $(0, 0)$.

The set of feasible solutions is depicted in Figure 17.1. We have also drawn the objective function $z = 5x_1 + 3x_2$, when $z = 3.5$, 11, and 38.5.

Consider any line segment joining points $P = (p_1, p_2)$ and $Q = (q_1, q_2)$. Let

$$z_0 = 5p_1 + 3p_2$$

be the value of the objective function at P and let

$$z_1 = 5q_1 + 3q_2$$

be the value of the objective function at Q. Assume without loss that $z_0 \leq z_1$. The coordinates of any point on the line segment between P and Q is given by

$$((1-t)p_1 + tq_1, (1-t)p_2 + tq_2)$$

for $0 \leq t \leq 1$. The value of the objective function at this point is

$$\begin{aligned} z_t &= 5((1-t)p_1 + tq_1) + 3((1-t)p_2 + tq_2) \\ &= (1-t)(5p_1 + 3p_2) + t(5q_1 + 3q_2) \\ &= (1-t)z_0 + tz_1. \end{aligned}$$

Observe that

$$z_0 = (1-t)z_0 + tz_0 \leq (1-t)z_1 + tz_1 = z_t$$

and

$$z_t = (1-t)z_0 + tz_1 \leq (1-t)z_1 + tz_1 = z_1.$$

Thus $z_0 \leq z_t \leq z_1$ for any $0 \leq t \leq 1$. Therefore the maximum value of the the objective function among the points on any line segment occurs at one of the endpoints. *It follows that the maximum value of any (linear) objective function among the points of a compact region occurs at some point on the boundary*. Also, *if the boundary is a polygon, then the maximum value of the objective will occur at one of the vertices.* Hence for our example the maximum value of the objective function

$$z = 5x_1 + 3x_2$$

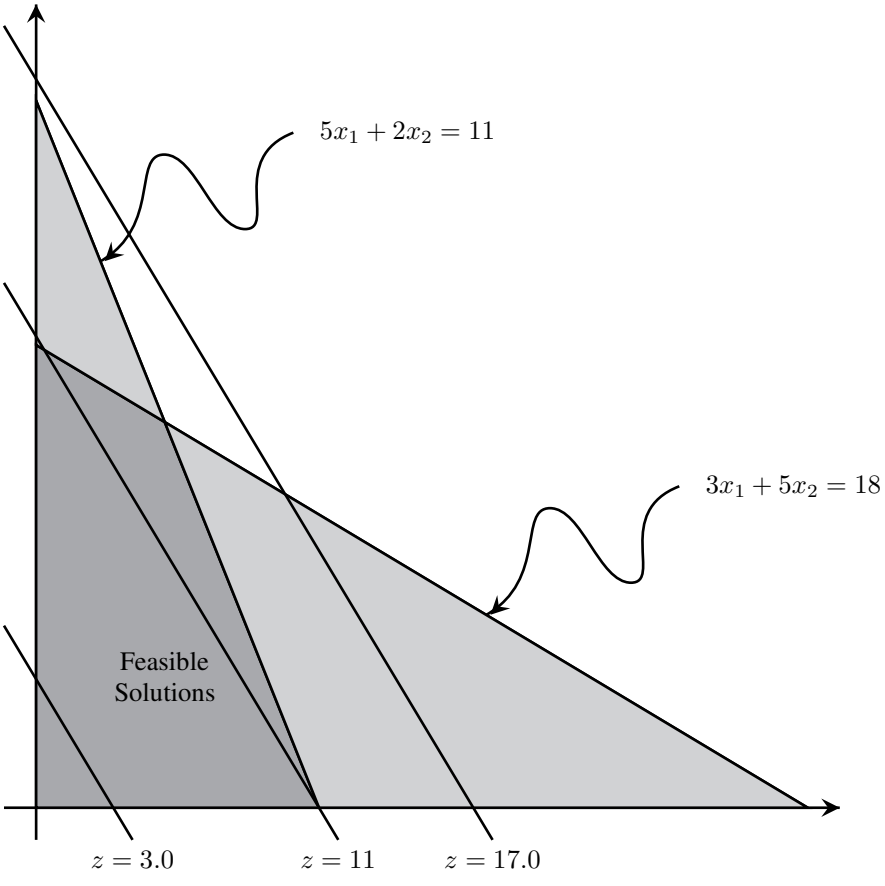


FIGURE 17.1
Graphical example

occurs at least one of $(0, 3.6)$, $(1, 3)$, $(2.2, 0)$, or $(0, 0)$. The value at these points is 10.8, 14, 11, and 0. Thus the maximum occurs at $x_1 = 1$, $x_2 = 3$.

If the region of feasible solutions is unbounded, then there may be no point in the region for which the objective function achieves its absolute maximum. Of course there is also no point for which the objective function achieves its maximum, if there are no feasible solutions.

If the objective function achieves its maximum at two adjacent vertices P and Q, and the region of feasible solutions is connected and polygonally bounded, then it will achieve its maximum at infinitely many points: namely, those lying on the line segment joining P and Q.

In a general linear program the region of feasible solutions is the intersection of the half hyper-planes determined by the linear constrains. Thus the region of feasible solutions to the general linear program is a compact convex region, bounded by facets (hyper-plane segments). That is, it is a polyhedron.[1] Consequently it follows that:

Theorem 17.1. *In a general linear program the objective function achieves its maximum either at exactly one point, at infinitely many points, or at no point. Furthermore if it does achieve a maximum, it does so on one of the vertices of the polyhedral boundary of the feasible solutions.*

17.1.3 Slack and surplus variables

It is easier to work with equalities, than with inequalities because we can take advantage of linear algebra. We can convert linear inequalities into equalities by introducing surplus and slack variables.

For example, consider an inequality of the form

$$a_1x_1 + a_2x_2 + \cdots + a_tx_t \quad \leq \quad b. \tag{17.1}$$

Given fixed assignments to the variables $x_1, x_2, \ldots, x_t$ that satisfy this inequality, there will be *slack* or "room for improvement" amounting to

$$x_j = b - a_1x_1 + a_2x_2 + \cdots + a_tx_t \geq 0.$$

Thus introducing a new variable x_j and requiring $x_j \geq 0$ we obtain the equality

$$a_1x_1 + a_2x_2 + \cdots + a_tx_t + x_j \quad = \quad b$$

which is equivalent to Inequality 17.1.

The variable x_j is called a *slack variable*. Similarly an inequality of the form

$$a_1x_1 + a_2x_2 + \cdots + a_tx_t \quad \geq \quad b \tag{17.2}$$

represents a *surplus* of

$$x'_j = a_1x_1 + a_2x_2 + \cdots + a_tx_t - b$$

[1] O.K. it is only a polyhedron if it is bounded. It could have some open sides, but the closed sides are bounded by hyper-planes.

for a fixed assignment to the variables $x_1, x_2, \ldots, x_t$. Thus introducing $x_{j'}$ as a new variable and requiring $x_{j'} \geq 0$ we obtain the equality

$$a_1x_1 + a_2x_2 + \cdots + a_tx_t - x_{j'} = b$$

which is equivalent to Inequality 17.2. The variable $x_{j'}$ is called a *surplus variable*. Adding slack and surplus variables in this way will reduce the the system of inequalities to a system of equalities and variables x_i that are either unbounded or satisfy $x_i \geq 0$. If x_i is unbounded, we find an inequality that x_i satisfies, solve for x_i, and substitute to eliminate x_i from the set of equations. A linear program in which all the variables are required to be non-negative and the remaining constraints are equality constraints is said to be in *standard form* or a *standard linear program*.

For example, to convert the linear program

$$\begin{aligned} &\text{Maximize:} && z = 5x_1 + 3x_2 + 3x_3 + x_4 \\ &\text{Subject to:} && \begin{aligned} 2x_2 + x_4 &= 2 \\ x_1 + x_2 + x_4 &\leq 3 \\ -x_1 - 2x_2 + x_3 &\geq 1 \\ x_1 \leq 0, x_2, x_3 &\geq 0 \end{aligned} \end{aligned}$$

into standard form we introduce slack and surplus variables x_5, x_6, and x_7 obtaining

$$\begin{aligned} &\text{Maximize:} && z = 5x_1 + 3x_2 + 3x_3 + x_4 \\ &\text{Subject to:} && \begin{aligned} 2x_2 + x_4 &= 2 \\ x_1 + x_2 + x_4 + x_5 &= 3 \\ -x_1 - 2x_2 + x_3 - x_6 &= 1 \\ x_1 + x_7 &= 0 \\ x_2, x_3, x_5, x_6, x_7 &\geq 0 \end{aligned} \end{aligned}$$

Now variables x_1 and x_4 are unbounded, so we solve for them

$$\begin{aligned} x_1 &= -x_7 \\ x_4 &= 2 - 2x_2. \end{aligned}$$

Substituting, we obtain

$$\begin{aligned} &\text{Maximize:} && z = x_2 + 3x_3 - 5x_7 + 2 \\ &\text{Subject to:} && \begin{aligned} -x_2 + x_5 - x_7 &= 1 \\ -2x_2 + x_3 - x_6 + x_7 &= 1 \\ x_2, x_3, x_5, x_6, x_7 &\geq 0 \end{aligned} \end{aligned}$$

Finally, to convert to a problem of minimization, we set

$$Z = 2 - z$$

and we have

$$\begin{array}{lrcl} \text{Minimize:} & Z = -x_2 - 3x_3 + 5x_7 & & \\ \text{Subject to:} & -x_2 + x_5 - x_7 & = & 1 \\ & -2x_2 + x_3 - x_6 + x_7 & = & 1 \\ & x_2, x_3, x_5, x_6, x_7 \geq 0 & & \end{array}$$

a linear program in standard form. In matrix form we set

$$\begin{aligned} X &= [x_2, x_3, x_5, x_6, x_7]^T \\ c &= [-1, -3, 0, 0, 5]^T \\ b &= [1, 1]^T \\ A &= \begin{bmatrix} -1 & 0 & 1 & 0 & -1 \\ -2 & 1 & 0 & -1 & 1 \end{bmatrix} \end{aligned}$$

and we see that the original linear program is equivalent to

$$\begin{array}{rcl} \text{Minimize: } Z & = & c^T X \\ \text{Subject to: } AX & = & b \\ X & \geq & 0. \end{array}$$

Exercises

17.1.1 Solve the following graphically and shade the region representing the feasible solutions:

$$\begin{array}{lrcl} \text{Maximize:} & z = x_1 + 1.5x_2 & & \\ \text{Subject to:} & 2x_1 + 3x_2 & \leq & 6 \\ & x_1 + 4x_2 & \leq & 4 \\ & x_1, x_2 \geq 0 & & \end{array}$$

17.1.2 Solve the following graphically and shade the region representing the feasible solutions:

$$\begin{array}{lrcl} \text{Minimize:} & Z = 6x_1 + 4x_2 & & \\ \text{Subject to:} & 2x_1 + x_2 & \geq & 1 \\ & 3x_1 + 4x_2 & \geq & 1.5 \\ & x_1, x_2 \geq 0 & & \end{array}$$

17.1.3 Carefully examine Exercises 17.1.1 and 17.1.2. How are the solutions related? They form what is called a pair of dual problems. Note that they involve the same constants, but in a rearranged order.

17.1.4 Put the following linear program into standard form:

$$\begin{array}{lrcl} \text{Maximize:} & z = 2x_1 + 3x_2 + 5x_3 & & \\ \text{Subject to:} & 3x_1 + 10x_2 + 5x_3 & \leq & 15 \\ & 33x_1 - 10x_2 + 9x_3 & \leq & 33 \\ & x_1 + 2x_2 + x_3 & \geq & 4 \\ & x_1, x_2 \geq 0 & & \end{array}$$

17.2 The simplex algorithm

17.2.1 Overview

The simplex algorithm can be used to solve linear programs of the form

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X & & \\ \text{Subject to:} & AX & = & b \\ & X & \geq & 0 \end{array} \tag{17.3}$$

There are three phases to the algorithm.

Phase 0: Find a basis solution or show that the linear program is infeasible.
Phase 1: Find a basis feasible solution or show that the linear program is infeasible.
Phase 2: Improve the basis feasible solution until

1. it's optimal, or
2. the linear program can be shown to be unbounded.

17.2.2 Some notation

In order to study the linear program in Equation 17.3 we first introduce some notation. Denote the columns of the m by n matrix A by

$$A = [A_1, A_2, A_3, \ldots, A_n].$$

Without loss, assume that the rank of A is m. Suppose

$$\mathcal{B} = \{A_{j_1}, A_{j_2}, A_{j_3}, \ldots A_{j_m}\}$$

is a set of m linearly independent columns of A. Then the m by m matrix

$$B = [A_{j_1}, A_{j_2}, A_{j_3}, \ldots A_{j_m}]$$

is nonsingular. Thus B^{-1} exists. Consequently, $B^{-1}A$ contains the identity matrix on columns $j_1, j_2, j_3, \ldots, j_m$ of A. Let $X_B = B^{-1}b$. Then X given by

$$X[j] = \begin{cases} X_B[i] & \text{if } j = j_i \\ 0 & \text{if not} \end{cases}$$

satisfies

$$AX = b$$

and is called the *basis solution* given by $\mathcal{B}$. If $X_B \geq 0$, then X also satisfies

$$X \geq 0$$

with the remaining constraints of the linear program given in Equation 17.3. Thus in this case we call X a *basis feasible solution*. Also corresponding to $\mathcal{B}$ we define c_B by

$$c_B[i] = c[j_i], \text{ for } i = 1, 2, \ldots, m.$$

The value Z of the objective function at X is

$$Z = c^T X = c_B^T X_B$$

It is also convenient to set

$$z_j = c_B^T Y_j;$$

where $Y_j = B^{-1} A_j$. Using this notation we can safely identify X_B with X and refer to X_B as the basis solution.

17.2.3 Phase 0: finding a basis solution

A basis solution if one exists can be found by *pivoting*. Pivoting is also known as Gaussian elimination or row reduction. To pivot on non-zero entry a_{ij} of the m by n matrix A we replace row k by

$$a_{ij}[a_{k1}, a_{k2}, \ldots, a_{kn}] - a_{kj}[a_{i1}, a_{i2}, \ldots, a_{in}]$$

for $k \neq i$, and we replace row i by

$$\frac{1}{a_{ij}}[a_{i1}, a_{i2}, \ldots, a_{in}].$$

The result is that column j becomes

$$[0, 0, \ldots, 0, \underbrace{1}_{i\text{th}}, 0, 0, \ldots, 0]^T.$$

We record this procedure as Algorithm 17.2.1.

Algorithm 17.2.1: PIVOT(i, j)

for $k \leftarrow 1$ **to** m
do **if** $k \neq i$
  then **for** $h \leftarrow 1$ **to** n
    do $A[k, h] \leftarrow A[i, j] \cdot A[k, h] - A[k, j] \cdot A[i, h]$
  else **for** $h \leftarrow 1$ **to** n
    do $A[i, h] \leftarrow A[i, h]/A[i, j]$

Thus to find a basis solution we iteratively select a non-zero entry a_{ij_i} in row i and pivot on a_{ij_i} for $i = 1, 2, \ldots, m$. That is, we have selected the linearly independent columns

$$\{A_{j_1}, A_{j_2}, A_{j_3}, \ldots, A_{j_m}\}$$

of A and have determined the basis solution X_B; where

$$B = [A_{j_1}, A_{j_2}, A_{j_3}, \ldots A_{j_m}].$$

17.2.4 Obtaining a basis feasible solution

Suppose that $A = [A_1, A_2, \ldots, A_n]$ has rank m and that X is a feasible solution to the linear program in Equation 17.3. Let $p \leq n$ be the number of positive variables in X. Without loss we may assume that the first p variables are positive. Then

$$X = [x_1, x_2, \ldots, x_p, \underbrace{0, 0, \ldots, 0}_{n-p}]^T$$

and so

$$\sum_{j=1}^{p} x_j A_j = b. \tag{17.4}$$

If the p columns $A_1, A_2, \ldots, A_p$ are linearly independent, then $p \leq m$, and there exists an additional $m - p$ columns of A whose inclusion with the first p are a linear independent set. Thus we can form a basis solution with $m - p$ zeros.

If the p columns are linearly dependent, then there exists α_j not all zero such that

$$\sum_{j=1}^{p} \alpha_j A_j = 0.$$

Let A_r be any column for which $\alpha_r \neq 0$, $j = 1, 2, \ldots, p$. Then

$$A_r = - \sum_{\substack{j=1 \\ j \neq r}}^{p} \frac{\alpha_j}{\alpha_r} A_j. \tag{17.5}$$

Substituting Equation 17.5 into Equation 17.5, we obtain

$$\sum_{\substack{j=1 \\ j \neq r}}^{p} (x_j - x_r \frac{\alpha_j}{\alpha_r}) A_j = b. \tag{17.6}$$

Thus we have a solution with at most $p - 1$ non-zero entries. However, we are not certain that they are non-negative. We need

$$x_j - x_r \frac{\alpha_j}{\alpha_r} \geq 0$$

for $j = 1, \ldots, p$, $j \neq r$. If $\alpha_j = 0$, then this is automatically satisfied. If $\alpha_j \neq 0$, then dividing by α_j gives

$$\frac{x_j}{\alpha_j} - \frac{x_r}{\alpha_r} \geq 0 \qquad \text{if } \alpha_j > 0$$

and

$$\frac{x_j}{\alpha_j} - \frac{x_r}{\alpha_r} \leq 0 \qquad \text{if } \alpha_j < 0$$

Thus we may select A_r such that

$$\frac{x_r}{\alpha_r} = \text{MIN}_j\{\frac{x_j}{\alpha_j} : \alpha_j > 0\}$$

or such that

$$\frac{x_r}{\alpha_r} = \text{MAX}_j\{\frac{x_j}{\alpha_j} : \alpha_j < 0\}.$$

Then each entry in Equation 17.6 will be non-negative and a feasible solution with no more than $p - 1$ non-zero variables has been found. We can continue this process of selecting columns A_j until $p \leq m$ in which case a basis feasible solution has been found.

17.2.5 The tableau

The *initial tableau* for the linear program in Equation 17.3 is the array

$$\left[\begin{array}{c|c} A & b \\ \hline c^T & 0 \end{array}\right]. \tag{17.7}$$

Note that other authors use a more elaborate tableau, but this is sufficient. Suppose B is the submatrix of A corresponding to the basis $\mathcal{B}$ as in Section 17.2.2. For the purpose of explanation assume B is the first m columns of A; then the tableau has the form

$$\left[\begin{array}{c|ccc|c} B & \cdots & A_j & \cdots & b \\ \hline c_B^T & \cdots & c_j & \cdots & 0 \end{array}\right];$$

where A_j is the j^{th} column of A. Multiplying the tableau with

$$\left[\begin{array}{c|c} B^{-1} & 0 \\ \hline -c_B^T B^{-1} & 1 \end{array}\right]$$

we obtain the new tableau

$$\left[\begin{array}{c|ccc|c} I & \cdots & Y_j & \cdots & X_B \\ \hline 0 & \cdots & c_j - z_j & \cdots & -z \end{array}\right]. \tag{17.8}$$

This is because $Y_j = B^{-1}A_j$, $X_B = B^{-1}b$, and $z_j = c_B^T Y_j$ as defined in Section 17.2.2. Thus selecting a new basis solution is equivalent to pivoting on entries of the first m rows of the tableau 17.7 to obtain a tableau similar to the tableau 17.8.

(The identity matrix need not be among the first columns.) Thus in the process of selection of a basis the values of $c_j - z_j$, z, and X_B are also easily determined.

An algorithm for the Phase 0 portion of the simplex algorithm that takes as input a tableau for a linear program whose adjacency matrix has m rows and n columns is provided in Algorithm 17.2.2.

Algorithm 17.2.2: PHASE0(*Tableau*, m, n)

```
Infeasible ← false
for r ← 1 to m
do { c ← 1
     while Tableau[r, c] = 0 do c ← c + 1
     if c > n
       then { if Tableau[r, n + 1] ≠ 0
                then { comment: the linear program is infeasible
                       Infeasible ← true
                       exit
              else { comment: { the linear program has
                              { rank < M – shift up the rows
                     for i ← r + 1 to m
                       do for j ← 1 to n + 1
                       do Tableau[i − 1, j] ← Tableau[i, j]
                     m ← m − 1
                     r ← r − 1
       else { PIVOT(r, c)
              pivots[r] ← c
```

17.2.6 Phase 2: improving a basis feasible solution

Let $X_B = B^{-1}b$ be a basis feasible solution of the linear program given in Equation 17.3. The value of the objective function at X_B is

$$Z = c_B^T X_B.$$

Some questions naturally arise:

- *Can we find another basis feasible solution with better Z?*
- *Furthermore, can we do this by changing only one column of B?*
- *Can we remove one column B_r of B and replace it with a column A_j of A and get a smaller value Z of the objective function $Z = c^T X$?*

Column A_j is a linear combination of the columns in B because B is a nonsingular m by m submatrix of A. Thus

$$A_j = \sum_{i=1}^{m} y_{ij} B_i. \tag{17.9}$$

Then A_j can replace any B_r for which

$$y_{rj} \neq 0$$

because the new set of vectors

$$\{B_1, B_2, \ldots, B_{r-1}, A_j, B_{r+1}, B_{r+2}, \ldots, B_m\}$$

will be linearly independent. Let

$$B^\star = [B_1, B_2, \ldots, B_{r-1}, A_j, B_{r+1}, B_{r+2}, \ldots, B_m].$$

Then $X_{B^\star}$ is a basis solution, but it may not be feasible. Observe from Equation 17.9 that

$$B_r = \frac{1}{y_{rj}} A_j - \sum_{\substack{i=1 \\ i \neq r}}^{m} \frac{y_{ij}}{y_{rj}} B_i.$$

Also

$$b = BX_B = \sum_{i=1}^{m} X_B[i] B_i = \sum_{\substack{i=1 \\ i \neq r}}^{m} X_B[i] B_i + X_B[r] B_r.$$

So, substituting we have:

$$\begin{aligned} b &= \sum_{\substack{i=1 \\ i \neq r}}^{m} X_B[i] B_i + X_B[r] \left(\frac{1}{y_{rj}} A_j - \sum_{\substack{i=1 \\ i \neq r}}^{m} \frac{y_{ij}}{y_{rj}} B_i \right) \\ &= \sum_{\substack{i=1 \\ i \neq r}}^{m} \left(X_B[i] - X_B[r] \frac{y_{ij}}{y_{rj}} \right) B_i + \frac{X_B[r]}{y_{rj}} A_j. \end{aligned}$$

Thus

$$X_{B^\star}[i] = \begin{cases} X_B[i] - X_B[r] \frac{y_{ij}}{y_{rj}} & \text{if } i \neq r \\ \frac{X_B[r]}{y_{rj}} & \text{if } i = r \end{cases} \tag{17.10}$$

is feasible if and only if

$$X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} \quad \geq \quad 0 \tag{17.11}$$

and

$$\frac{X_B[r]}{y_{rj}} \quad \geq \quad 0. \tag{17.12}$$

Thus if $X_B[r] \neq 0$, we see from Equation 17.12 that we must have

$$y_{rj} > 0.$$

If $y_{ij} \leq 0$, then Equation 17.11 automatically holds. So, we need to only be concerned with coordinates i for which $y_{ij} > 0$. When $y_{ij} > 0$, the condition given by Equation 17.11 can be rewritten as

$$\frac{X_B[r]}{y_{rj}} \leq \frac{X_B[i]}{y_{ij}}.$$

Thus we need to choose that column r of B such that

$$\frac{X_B[r]}{y_{rj}} = \text{MIN}_i \left\{ \frac{X_B[i]}{y_{ij}} : y_{ij} > 0 \right\} = \theta. \tag{17.13}$$

To summarize:

We began with a nonsingular submatrix B of A

$$B = [A_{i_1}, A_{i_2}, A_{i_3}, \ldots A_{i_m}].$$

If $X_B = B^{-1}b$ is a basis feasible solution to the linear program given in Equation 17.3, then we selected an arbitrary column A_j of A, not in B and wrote

$$A_j = BY_j,$$

a linear combination of the columns of B, where

$$Y_j = [y_{1j}, y_{2j}, \ldots, y_{mj}].$$

If some $y_{ij} > 0$, there is a column B_r of B, which we can replace with A_j to get a basis feasible solution $X_{B^\star}$. Equation 17.13 shows how to select B_r so that this is possible.

Now what about the value of the objective function – is it better? Let

$$B^\star = [B_1^\star, B_2^\star, \ldots, B_m^\star]$$

be the new matrix obtained by replacing B_r with A_j. That is $B_i^\star = B_i$, for $i \neq r$

and $B_r^* = A_j$. The new basis feasible solution is $X_{B^\star}$ given in Equation 17.10. The objective function corresponding to $B^\star$ is

$$Z^\star = c_{B^\star}^T X_{B^\star}$$

where

$$c_{B^\star}[i] = \begin{cases} c_B[i] & \text{if } i \neq r \\ c_j & \text{if } i = r. \end{cases}$$

Therefore

$$\begin{aligned} Z^\star &= \sum_{\substack{i=1 \\ i \neq r}}^{m} \left(X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} \right) c_B[i] + \frac{X_B[r]}{y_{rj}} c_j \\ &= \sum_{i=1}^{m} \left(X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} \right) c_B[i] + \frac{X_B[r]}{y_{rj}} c_j, \\ &\qquad \text{because } X_B[i] - X_B[r]\frac{y_{ij}}{y_{rj}} = 0 \text{ when } i = r \\ &= Z - \frac{X_B[r]}{y_{rj}} \sum_{i=1}^{m} y_{ij} c_B[i] + \frac{X_B[r]}{y_{rj}} c_j \\ &= Z + \frac{X_B[r]}{y_{rj}} (c_j - c_B^T Y_j) \end{aligned}$$

Setting $z_j = c_B^T Y_j = c_B^T B^{-1} A_j$ we have

$$Z^\star = Z + \frac{X_B[r]}{y_{rj}} (c_j - z_j) = Z + \theta(c_j - z_j).$$

Therefore if we can find A_j such that

$$c_j - z_j < 0$$

and at least one $y_{ij} > 0$, then it is possible to replace one of the columns of the columns of B by A_j and obtain a new value $Z^\star$ of the objective function satisfying

$$Z^\star \leq Z.$$

If the given basis solution is not degenerate, then

$$Z^\star < Z.$$

In terms of the tableau given in Equation 17.8, this means we can find a column j with a negative entry in last row and a positive entry y_{ij} in row i. For each positive entry y_{ij} compute the ratio

$$\theta_i = \frac{X_B[i]}{y_{ij}}$$

and chose i so that θ_i is smallest. Recall that

$$[X_B[1], X_B[2], \ldots, X_B[m], -Z]^T$$

is the last column of the tableau. Then pivoting on y_{ij} produces a new tableau with smaller Z. Note that $-Z$ is the entry in the last row and column of the tableau.

17.2.7 Unbounded solutions

In Section 17.2.6, given a basis feasible solution $X_B = B^{-1}b$ we found a column A_j that had at least one $y_{ij} > 0$, $i = 1, 2, \ldots, m$ where

$$Y_j = B^{-1}A_j.$$

For this column, we found a column B_r in B which when replaced by A_j, resulted in a basis feasible solution $X_{B^\star}$. The value of the objective function for $X_{B^\star}$ was

$$Z^\star = Z + \theta(c_j - z_j).$$

Let us consider a column A_j for which $y_{ij} \leq 0$ for each $i = 1, 2, \ldots, m$. We have

$$b = BX_B = \sum_{i=1}^{m} X_B[i]B_i$$

with value of the objective function equal to $Z = c_B^T X_B$.

Adding and subtracting θA_j for any θ yields

$$\begin{aligned} b &= BX_B - \theta A_j + \theta A_j \\ &= \sum_{i=1}^{m} X_B[i]B_i - \theta A_j + \theta A_j \end{aligned}$$

but

$$A_j = BY_j = \sum_{i=1}^{m} y_{ij}B_i.$$

So substituting we obtain:

$$b = \sum_{i=1}^{m} (X_B[i] - \theta y_{ij})B_i + \theta A_j. \tag{17.14}$$

When $\theta > 0$, then $(X_B[i]B_i - \theta y_{ij}) \geq 0$ because we have assumed that $y_{ij} \leq 0$, for $i = 1, 2, \ldots, m$. Thus Equation 17.14 is feasible. The value of the objective function is again

$$\begin{aligned} Z^\star &= \sum_{i=1}^{m} c_B[i](X_B[i] - \theta y_{ij}) + c_j\theta \\ &= Z + \theta(c_j - z_j). \end{aligned}$$

Thus choosing θ arbitrarily large we can make $Z^\star$ arbitrarily small if

$$c_j - z_j < 0.$$

To summarize:

Given any basis feasible solution to a linear program, if there is a column A_j not in the basis for which

$$c_j - z_j < 0$$

and

$$Y_j = B^{-1}A_j \leq 0,$$

then the linear program in Equation 17.3 has an unbounded solution.

In terms of the tableau given in Equation 17.8, this means that if we can find a column j with *every* entry less than or equal to zero with a negative entry in last row, then the linear program in Equation 17.3 has an unbounded solution.

17.2.8 Conditions for optimality

Assume that $X_B = B^{-1}b$ is a basis feasible solution of the linear program given in Equation 17.3 and that the value of the objective function at X_B is

$$Z_0 = c_B^T X_B.$$

In addition, suppose that

$$c_j - z_j \geq 0$$

for every column A_j of A not in B. Thus the value of the objective function cannot be improved by replacing a column of B with a column of A. We will show that Z_0 is the minimal value of the linear program and hence that X_B is an optimal solution. Set $\overrightarrow{z} = [z_1, z_2, \ldots, z_n]^T$. So,

$$\overrightarrow{z} \leq c.$$

Let X be any feasible solution of linear program given in Equation 17.3. Then

$$x_1A_1 + x_2A_2 + \cdots + x_nA_n \quad = \quad b \tag{17.15}$$

and $X \geq 0$. Let $Z^\star = C^T X$ be the value of the objective function at X.

Every column A_j of A can be written as a linear combination of the columns of B:

$$A_j \quad = \quad BY_j.$$

Setting $Y = [Y_1, Y_2, \ldots, Y_n]$ we have $A = BY$, and $\overrightarrow{z} = c_B^T Y$. Then

$$BX_B = b = AX = BYX.$$

Therefore,

$$X_B = YX$$

because B is nonsingular. Hence

$$Z_0 = c_B^T X_B = c_B^T YX = \overrightarrow{z}\ X \leq c^T X = Z^\star,$$

which proves that Z_0 is the optimal value of the objective function and hence X_B is an optimal solution to the linear program in Equation 17.3.

In terms of the tableau given in Equation 17.8, this means if there is no negative entry in the last row, then we have found an optimal solution. The optimal value is Z the negative of the entry in the last row and column. To find the optimal solution, first discover the columns $\{j_1, j_2, \ldots, j_m\}$ that contain the identity matrix on the first m rows of the tableau. That is, column j_i is

$$[0, 0, \ldots, 0, \underbrace{1}_{i\textit{th}}, 0, 0, \ldots, 0]^T.$$

Then the optimal solution is X where

$$X[j] = \begin{cases} X_B[i] & \text{if } j = j_i \\ 0 & \text{if not.} \end{cases}$$

The Phase 2 portion of the simplex algorithm is given in Algorithm 17.2.3.

Algorithm 17.2.3: PHASE2(*Tableau*, m, n)

```
c ← 1
while c < n
do { if Tableau[m+1, c] < 0
     then { i ← 0
            while Tableau[i, c] ≤ 0 and i ≤ m do i ← i + 1
            if i ≥ m then { Unbounded ← true
                            return
            M ← Tableau[i,n+1] / Tableau[i,c]
            r ← i
            i ← i + 1
            while i ≤ m do
            { if (Tableau[i, c] > 0) and Tableau[i,n+1] / Tableau[i,c] < M
              then { M ← Tableau[i,n+1] / Tableau[i,c]
                     r ← i
              i ← i + 1
            PIVOT(r, c)
            pivots[r] ← c
            c ← 1
     c ← c + 1
```

17.2.9 Phase 1: initial basis feasible solution

Suppose that after **Phase 0** we have obtained a basis solution X_B to the linear program

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X & & \\ \text{Subject to:} & AX & = & b \\ & X & \geq & 0 \end{array} \tag{17.16}$$

where A is a m by n matrix of rank m. Then

$$B = [B_1, B_2, \ldots, B_m]$$

is a nonsingular submatrix of A and $X_B = B^{-1}b$. The result of pivoting on the columns in B is the tableau

$$\left[\begin{array}{ccc|c} \cdots & Y_j & \cdots & X_B \\ \hline \cdots & c_j - z_j & \cdots & -z \end{array}\right].$$

If $X_B[i] < 0$ for some row i, then the basis solution is infeasible, and we cannot proceed to **Phase 2**. Let E be the m-dimensional vector defined by

$$E[i] = \begin{cases} -1, & \text{if } X_B[i] < 0 \\ 0, & \text{if } X_B[i] \geq 0. \end{cases}$$

Let x_0 be a new artificial variable and define the **Phase 1** *linear program* as follows:

$$\begin{array}{rcl} \text{Minimize: } w & = & [1, 0, 0, \ldots, 0] \begin{bmatrix} x_0 \\ X \end{bmatrix} \\ \text{Subject to: } [E, Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} & = & X_B \\ x_0, X & \geq & 0. \end{array} \tag{17.17}$$

The **Phase 1** *tableau* is:

$$\left[\begin{array}{c|ccc|c} E & \cdots & Y_j & \cdots & X_B \\ \hline 0 & \cdots & c_j - z_j & \cdots & -Z \\ \hline 1 & 0, 0, \cdots & 0 & \cdots 0, 0 & 0 \end{array}\right].$$

It has columns $0, 1, \ldots, n$. The second to last row corresponds to the cost equation to the linear program 17.16 and can almost be ignored for this discussion. Let $w(x_0, X)$ denote the value of the objective function for the **Phase 1** linear program at a given x_0 and X.

Lemma 17.2. *The* **Phase 1** *linear program is always feasible and has a non-negative optimal objective value.*

Proof. Let i be the row for which $X_B[i]$ is smallest (i.e., most negative), and pivot on row i column 0. The result is a tableau with a feasible solution to the **Phase 1** linear program. We can apply the techniques of Section 17.2.6 to obtain an optimal

feasible solution or show that it is unbounded below. It cannot be unbounded below, however, because

$$w = w(x_0, X) = [1, \overrightarrow{0}] \begin{bmatrix} x_0 \\ X \end{bmatrix} = x_0 \geq 0.$$

Hence w is non-negative. □

Theorem 17.3. *A linear program is feasible if and only if the artificial objective function has minimum value $w_{min} = 0$. Moreover, if $w_{min} = 0$ for $x_0 \geq 0$ and $X \geq 0$, then X is a feasible solution of the original linear program.*

Proof. By Lemma 17.2, the **Phase 1** linear program has an optimal solution for which the minimum value $w_{min} \geq 0$. First suppose $w_{min} = 0$. Then there exist X and x_0 such that $w(x_0, X) \geq 0$, $x_0 = 0$, $X \geq 0$, and

$$[E, Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} = x_0 E + YX = X_B.$$

Then $B^{-1}AX = 0 + YX = x_0 E + YX = X_B = B^{-1}b$ and hence $AX = b$. Thus the linear program 17.16 is feasible.

Conversely, suppose there exists a feasible solution $\hat{X}$ of $AX = b$. Then $Y\hat{X} = B^{-1}A\hat{X} = B^{-1}b = \hat{X}_B$ and thus

$$[E, Y] \begin{bmatrix} x_0 \\ X \end{bmatrix} = X_B$$

is solvable in non-negative variables, by choosing $x_0 = 0$, and $X = \hat{X}$. For these values of x_0 and X, $w(0, \hat{X}) = 0$. Because $w \geq 0$ must hold for every feasible solution of the **Phase 1** linear program, we have $w_{min} = 0$. □

Algorithm 17.2.4: PHASE1$(Tableau, m, n)$

Unbounded $\leftarrow$ **false**
Infeasible $\leftarrow$ **false**
comment: Use column 0 and row $m+2$ to create the Phase 1 tableau
for $i \leftarrow 1$ **to** m **do** { **if** *Tableau*$[i, n+1] < 0$ **then** *Tableau*$[i, 0] \leftarrow -1$ **else** *Tableau*$[i, 0] \leftarrow 0$
Tableau$[m+2, 0] \leftarrow 1$
for $j \leftarrow 1$ **to** n **do** *Tableau*$[m+2, j] \leftarrow 0$
Tableau$[m+2, n+1] \leftarrow 0$
comment: find the Phase 1 first pivot
$r = 0$
$M \leftarrow 0$
for $i \leftarrow 1$ **to** m **do** { **if** *Tableau*$[i, n+1] < M$ **then** { $M \leftarrow$ *Tableau*$[i, n+1]$; $r \leftarrow i$
if $r = 0$
  then { **comment:** no phase 1 is necessary ; **return**
PIVOT$(r, 0)$
$c \leftarrow 1$
while $c \leq n$
  do { **if** *Tableau*$[m+2, c] < 0$
    then { $i \leftarrow 1$
      while *Tableau*$[i, c] \leq 0$ **do** $i \leftarrow i + 1$
      $M \leftarrow \frac{\textit{Tableau}[i,n+1]}{\textit{Tableau}[i,c]}$
      $r \leftarrow i$
      $i \leftarrow i + 1$
      while $i \leq m$ **do** { **if** (*Tableau*$[i, c] > 0$) **and** $\frac{\textit{Tableau}[i,n+1]}{\textit{Tableau}[i,c]} < M$ **then** { $M \leftarrow \frac{\textit{Tableau}[i,n+1]}{\textit{Tableau}[i,c]}$; $r \leftarrow i + 1$ } ; $i \leftarrow i + 1$
      PIVOT(r, c)
      pivots$[r] \leftarrow c$
      $c \leftarrow 1$
    $c \leftarrow c + 1$
if *Tableau*$[m+2, n+1] \neq 0$
  then *Infeasible* $\leftarrow$ **true**

We can now give the complete simplex algorithm.

Algorithm 17.2.5: SIMPLEX(*Tableau*, m, n)

Unbounded $\leftarrow$ **false**
Infeasible $\leftarrow$ **false**
PHASE0(*Tableau*, m, n)
if *Infeasible*
  then { **output** ("The Linear Program is infeasible.")
  return
PHASE1(*Tableau*, m, n)
if *Infeasible*
  then { **output** ("The Linear Program is infeasible.")
  return
PHASE2(*Tableau*, m, n)
if *Unbounded*
  then { **output** ("The Linear Program is unbounded.")
  return
$Z \leftarrow -$*Tableau*$[m+1, n+1]$
for $j \leftarrow 1$ **to** n **do** $X[j] \leftarrow 0$
for $i \leftarrow 1$ **to** m **do** $X[$*pivots*$[r]] =$ *Tableau*$[i, n+1]$
return (X, Z)

17.2.10 An example

$$\begin{array}{lrrrrrrr}
\text{Minimize:} & Z = 7x_1 + 2x_2 & & & & & & \\
\text{Subject to:} & -x_1 & +2x_2 & & +x_3 & & = & 4 \\
& 4x_1 & +3x_2 & +x_3 & +x_4 & & = & 24 \\
& -2x_1 & -2x_2 & & & +x_5 & = & -7 \\
& & & & & & & \\
& & & & & x_1, x_2, x_3, x_4 & \geq & 0.
\end{array}$$

The initial tableau is

$$\left[\begin{array}{rrrrr|r}
-1 & 2 & 1 & 0 & 0 & 4 \\
4 & 3 & 1 & 1 & 0 & 24 \\
-2 & -2 & 0 & 0 & 1 & -7 \\
\hline
7 & 2 & 0 & 0 & 0 & 0
\end{array}\right]$$

and we start **Phase 0**. Pivoting on the $[1, 3]$ entry obtains

$$\left[\begin{array}{rrrrr|r}
-1 & 2 & 1 & 0 & 0 & 4 \\
5 & 1 & 0 & 1 & 0 & 20 \\
-2 & -2 & 0 & 0 & 1 & -7 \\
\hline
7 & 2 & 0 & 0 & 0 & 0
\end{array}\right].$$

Thus we have the basis solution

$$X = [0, 0, 4, 20, -7]^T$$

corresponding to basis consisting of columns 3, 4, and 5. This ends **Phase 0**. The basis solution found was not feasible so we start **Phase 1**. In this phase we have columns 0, 1, 2, 3, 4, and 5.

$$\left[\begin{array}{c|cccccc|c} 0 & -1 & 2 & 1 & 0 & 0 & 4 \\ 0 & 5 & 1 & 0 & 1 & 0 & 20 \\ -1 & -2 & -2 & 0 & 0 & 1 & -7 \\ \hline 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

First we price out column 0.

$$\left[\begin{array}{c|cccccc|c} 0 & -1 & 2 & 1 & 0 & 0 & 4 \\ 0 & 5 & 1 & 0 & 1 & 0 & 20 \\ 1 & 2 & 2 & 0 & 0 & -1 & 7 \\ \hline 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ \hline 0 & -2 & -2 & 0 & 0 & 1 & -7 \end{array}\right]$$

We now have a feasible solution, but the **Phase 1** objective value is 7. We proceed to reduce this value to 0. In the last row we find a negative entry in column 1. The smallest ratio of the last column entries with positive entries in column 1 is $\theta = 7/2$, so we pivot on the $[3, 1]$ entry to obtain the tableau below.

$$\left[\begin{array}{c|cccccc|c} \frac{1}{2} & 0 & 3 & 1 & 0 & -\frac{1}{2} & 7\frac{1}{2} \\ -2\frac{1}{2} & 0 & -4 & 0 & 1 & 2\frac{1}{2} & 2\frac{1}{2} \\ \frac{1}{2} & 1 & 1 & 0 & 0 & -\frac{1}{2} & 3\frac{1}{2} \\ \hline -3\frac{1}{2} & 0 & -5 & 0 & 0 & 3\frac{1}{2} & -24\frac{1}{2} \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

The **Phase 1** objective value is now zero and we have a feasible solution. We proceed to **Phase 2** by dropping column 0 and the last row.

$$\left[\begin{array}{cccccc|c} 0 & 3 & 1 & 0 & -\frac{1}{2} & 7\frac{1}{2} \\ 0 & -4 & 0 & 1 & 2\frac{1}{2} & 2\frac{1}{2} \\ 1 & 1 & 0 & 0 & -\frac{1}{2} & 3\frac{1}{2} \\ \hline 0 & -5 & 0 & 0 & 3\frac{1}{2} & -24\frac{1}{2} \end{array}\right]$$

There is a negative entry in the last row in column 2, so the objective value can be reduced. The smallest ratio is $\theta = (7\frac{1}{2})/3$, so we pivot on the $[1, 2]$–entry.

$$\left[\begin{array}{cccccc|c} 0 & 1 & \frac{1}{3} & 0 & -\frac{1}{6} & 2\frac{1}{2} \\ 0 & 0 & 1\frac{1}{3} & 1 & 1\frac{5}{6} & 12\frac{1}{2} \\ 1 & 0 & -\frac{1}{3} & 0 & -\frac{1}{3} & 1 \\ \hline 0 & 0 & 1\frac{2}{3} & 0 & 2\frac{2}{3} & -12 \end{array}\right]$$

There are no negative entries left in the last row so an optimal solution has been found. This solution is

$$X = [1, 2\frac{1}{2}, 0, 12\frac{1}{2}, 0]^T$$

and has objective value $Z = 12$.

17.3 Cycling

It is prudent to ask whether it is possible for the simplex algorithm to go through an endless sequence of iterations without terminating. Consider the following linear program:

$$\begin{array}{lllllllll}
\text{Minimize:} & Z = -10x_1 + 57x_2 + 9x_3 + 24x_4 & & & & & & & \\
\text{Subject to:} & 0.5x_1 & -5.5x_2 & -2.5x_3 & +9x_4 & +x_5 & & = & 0 \\
& 0.5x_1 & -1.5x_2 & -0.5x_3 & +x_4 & & +x_6 & = & 0 \\
& x_1 & & & & & +x_7 & = & 1 \\
& & & x_1, x_2, x_3, x_4, x_5, x_6, x_7 & & \geq & 0. & &
\end{array}$$

The initial tableau is

$$\left[\begin{array}{rrrrrrr|r}
0.5 & -5.5 & -2.5 & 9 & 1 & 0 & 0 & 0 \\
0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
-10 & 57 & 9 & 24 & 0 & 0 & 0 & 0
\end{array}\right].$$

If we adopt the following rule:

> *Always pivot on the column with the smallest (most negative) entry in the bottom row, choosing the first row that achieves the smallest ratio. If there are two such columns always choose the first one, (i.e., the one with smallest index).*

Then the sequence of iterations is:

1. Pivot on (1, 1).

$$\left[\begin{array}{rrrrrrr|r}
1 & -11 & -5 & 18 & 2 & 0 & 0 & 0 \\
0 & 4 & 2 & -8 & -1 & 1 & 0 & 0 \\
0 & 11 & 5 & -18 & -2 & 0 & 1 & 1 \\
\hline
0 & -53 & -41 & 204 & 20 & 0 & 0 & 0
\end{array}\right]$$

2. Pivot on (2, 2).

$$\left[\begin{array}{rrrrrrr|r}
1 & 0 & 0.5 & -4 & -0.75 & 2.75 & 0 & 0 \\
0 & 1 & 0.5 & -2 & -0.25 & 0.25 & 0 & 0 \\
0 & 0 & -0.5 & 4 & 0.75 & -2.75 & 1 & 1 \\
\hline
0 & 0 & -14.5 & 98 & 6.75 & 13.25 & 0 & 0
\end{array}\right]$$

3. Pivot on (1, 3).

$$\left[\begin{array}{rrrrrrr|r} 2 & 0 & 1 & -8 & -1.5 & 5.5 & 0 & 0 \\ -1 & 1 & 0 & 2 & 0.5 & -2.5 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 29 & 0 & 0 & -18 & -15 & 93 & 0 & 0 \end{array}\right]$$

4. Pivot on (2, 4).

$$\left[\begin{array}{rrrrrrr|r} -2 & 4 & 1 & 0 & 0.5 & -4.5 & 0 & 0 \\ -0.5 & 0.5 & 0 & 1 & 0.25 & -1.25 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 20 & 9 & 0 & 0 & -10.5 & 70.5 & 0 & 0 \end{array}\right]$$

5. Pivot on (1, 5).

$$\left[\begin{array}{rrrrrrr|r} -4 & 8 & 2 & 0 & 1 & -9 & 0 & 0 \\ 0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline -22 & 93 & 21 & 0 & 0 & -24 & 0 & 0 \end{array}\right]$$

6. Pivot on (2, 6).

$$\left[\begin{array}{rrrrrrr|r} 0.5 & -5.5 & -2.5 & 9 & 1 & 0 & 0 & 0 \\ 0.5 & -1.5 & -0.5 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline -10 & 57 & 9 & 24 & 0 & 0 & 0 & 0 \end{array}\right]$$

The tableau obtained after iteration 6 is identical to the initial tableau and so this cycle of iterations would repeat and the simplex algorithm would never terminate. It is easy to see that this is the only way that the simplex algorithm can fail to terminate. That is

Theorem 17.4. *If the simplex algorithm fails to terminate, then it must cycle.*

Several methods have been proposed to avoid this cycling phenomenon. The easiest is to adopt the *smallest index rule*.

> *Always pivot on the column with the first negative entry in the bottom row (i.e., the one with the smallest index), and choose the first row in that column that achieves the smallest ratio.*

We leave it as an exercise to prove the following result:

Theorem 17.5. (R.G. Bland, 1977) *The simplex method always terminates if the smallest index rule is adopted.*

Proof. Exercise 17.3.2. □

Exercises

17.3.1 Solve the following linear programs using the simplex algorithm.

(a)

$$\begin{array}{lrcrcrcrcl} \text{Maximize:} & z = 3x_1 & + & 2x_2 & + & x_3 & + & 4x_4 & & \\ \text{Subject to:} & 4x_1 & + & 5x_2 & + & & - & 3x_4 & = & 5, \\ & 2x_1 & - & 3x_2 & - & 4x_3 & + & 5x_4 & = & 7, \\ & x_1 & + & 4x_2 & + & 2.5x_3 & - & 4x_4 & = & 6, \end{array}$$

$$x_1, x_2, x_3, x_4 \geq 0.$$

(b)

$$\begin{array}{lrcrcrcrcl} \text{Maximize:} & z = 3x_1 & + & 4x_2 & + & x_3 & + & 7x_4 & & \\ \text{Subject to:} & 8x_1 & + & 3x_2 & + & 4x_3 & + & x_4 & \leq & 5, \\ & 2x_1 & + & 6x_2 & + & x_3 & + & 5x_4 & \leq & 7, \\ & x_1 & + & 4x_2 & + & 5x_3 & + & 2x_4 & \leq & 6, \end{array}$$

$$x_1, x_2, x_3, x_4 \geq 0.$$

(c)

$$\begin{array}{lrcrcrcrcl} \text{Maximize:} & z = 2x_1 & - & 3x_2 & + & 4x_3 & + & x_4 & & \\ \text{Subject to:} & x_1 & + & 5x_2 & + & 9x_3 & - & 6x_4 & \geq & -2, \\ & 3x_1 & - & 1x_2 & + & x_3 & + & 3x_4 & \leq & 10, \\ & -2x_1 & - & 3x_2 & + & 7x_3 & - & 8x_4 & \geq & 0, \end{array}$$

$$x_1, x_2, x_3, x_4 \geq 0.$$

17.3.2 Prove Theorem 17.5.

17.4 Notes

The topic of linear programming appears in a variety of different subjects, for example operations research, mathematical programming, and combinatorial optimization. There are thus numerous books in which it is discussed and among them are CHVÁTAL [35], HADLEY [78], NEMHAUSER and WOLSEY [132], PAPADIMITRIOU and STEIGLITZ [134], TAHA [165], and WALKER [185].

In this chapter we have only discussed the simplex algorithm which was invented in the late 1940s by DANZIG [40] (see also DANTZIG [41]). A thorough discussion of the history of linear programming can be found in DANTZIG's celebrated work [42].

The example of cycling in the simplex method found in Section 17.3, is from the book by CHVÁTAL [35]. Theorem 17.5 appears in BLAND [18].

It can be shown that in the worst case the running time of simplex algorithm is not polynomial bounded (see KLEE and MINTY [99]) and hence the simplex algorithm

is theoretically not satisfactory. In practice it is eminently useful and except for very contrived problems exceedingly fast. In 1979, KHACHIAN [97] provided a method called the *ellipsoid method* that solves linear programs in polynomial time. This is a marvelous, elegant, and simple jewel of pure mathematics. However, we believe that it is unlikely that the ellipsoid method will ever be a serious challenger to the simplex method.

18

The Primal-Dual Algorithm

18.1 Introduction

In Chapter 17 we found it convenient to convert every linear program consisting of constraints that are a mix of inequalities and equalities:

$$\sum a_{ij}x_i\{\leq=\geq\}b_i, \quad i = 1, 2, \ldots, m$$

to a system of equations $Ax = b$, $b \geq 0$. A slightly different formulation of constraints will prove useful here. We convert every equality to the equivalent pair of inequalities, so that

$$\sum a_{ij}x_i \quad = \quad b_i$$

becomes the two inequalities

$$\begin{aligned}\sum a_{ij}x_i \quad &\geq \quad b_i \\ \sum a_{ij}x_i \quad &\leq \quad b_i.\end{aligned}$$

We then multiply all inequalities with the relation $\geq$ through by a -1 so that each has the form

$$\sum a_{ij}x_i \leq b_i.$$

Now we have a linear program of the form

$$\begin{array}{lrcl} \text{Maximize:} & z = c^T X & & \\ \text{Subject to:} & DX & \leq & d \\ & X & \geq & 0 \end{array} \qquad \text{(Primal)}$$

which we call the *primal linear program*. Corresponding to the primal linear program is another linear program which we call the *dual linear program*.

$$\begin{array}{lrcl} \text{Minimize:} & Z = d^T W & & \\ \text{Subject to:} & D^T W & \geq & c \\ & W & \geq & 0 \end{array} \qquad \text{(Dual)}$$

Lemma 18.1. *The dual of the dual is the primal:*

Proof. The dual linear program:

$$\begin{array}{lrcl} \text{Minimize:} & Z = d^T W & & \\ \text{Subject to:} & D^T W & \geq & c \\ & W & \geq & 0 \end{array}$$

is equivalent to :

$$\begin{array}{lrcl} \text{Maximize:} & z^\star = (-d)^T W & & \\ \text{Subject to:} & (-D)^T W & \leq & -c \ . \\ & W & \geq & 0 \end{array}$$

This is in the form of a primal linear program. The dual linear program corresponding to it is:

$$\begin{array}{lrcl} \text{Minimize:} & Z^\star = (-c)^T X & & \\ \text{Subject to:} & -DX & \geq & -d \ . \\ & X & \geq & 0 \end{array}$$

This linear program is equivalent to

$$\begin{array}{lrcl} \text{Maximize:} & z = c^T X & & \\ \text{Subject to:} & DX & \leq & d \ . \\ & X & \geq & 0 \end{array}$$

□

Lemma 18.2. *If X is a feasible solution to the primal and W is a feasible solution to the dual, then $c^T X \leq d^T W$ (implying $z \leq Z$).*

Proof. Suppose X is a feasible solution to the primal. Then $DX \leq d$. If W is a feasible solution to the dual, then $W \geq 0$, so we see

$$W^T DX \leq W^T d \quad = \quad d^T W.$$

Similarly, because $X \geq 0$ and W is a feasible solution,

$$\begin{array}{rcl} D^T W & \geq & c \\ X^T D^T W & \geq & X^T c \\ W^T DX & \geq & c^T X. \end{array}$$

Therefore

$$c^T X \leq W^T DX \leq d^T W. \tag{18.1}$$

□

Lemma 18.3. *If $\widehat{X}$ is a feasible solution to the primal and $\widehat{W}$ is a feasible solution to the dual such that $c^T \widehat{X} = d^T \widehat{W}$, then $\widehat{X}$ and $\widehat{W}$ are optimal solutions to the primal and dual, respectively.*

Proof. By assumption, $c^T \widehat{X} = d^T \widehat{W}$, but for any feasible solution X to the primal,

$$c^T X \leq d^T \widehat{W} = c^T \widehat{X}$$

(with the inequality from Lemma 18.2). Therefore, $\widehat{X}$ is an optimal solution to the primal.

By the same logic, for any feasible solution W of the dual,

$$d^T W \geq c^T \widehat{X} = d^T \widehat{W}.$$

Thus, $\widehat{W}$ is an optimal solution to the dual. □

Lemma 18.4. *If the dual or the primal has an optimal solution, then the other also must have an optimal solution and their optimal values are the same.*

Proof. Because of Lemma 18.1, we need only show that if the primal linear program has an optimal solution, then so does the dual linear program. Recall that the primal linear program is:

$$\begin{array}{lrcl} \text{Maximize:} & z = c^T X & & \\ \text{Subject to:} & DX & \leq & d \\ & X & \geq & 0. \end{array}$$

Add slack variables X_s and convert it to a minimization problem to get the standard form linear program:

$$\begin{array}{lrcl} \text{Minimize:} & (-z) = (-c)^T X & & \\ \text{Subject to:} & DX + IX^s & = & d \\ & X, X_s & \geq & 0 \end{array} , \tag{18.2}$$

where $I = [E_1, E_2, \ldots, E_m]$ is the m by m identity matrix. Let $D = [D_1, \ldots, D_n]$. The tableau for the simplex algorithm is

$$\left[\begin{array}{cccccccccc|c} D_1 & \cdots & D_j & \cdots & D_n & E_1 & \cdots & E_{j'} & \cdots & E_m & d \\ \hline -c_1 & \cdots & -c_j & \cdots & -c_n & 0 & \cdots & 0 & \cdots & 0 & 0 \end{array}\right].$$

If the linear program 18.2 has an optimal solution X_B, with optimal value $(-z) = (-c_B)^T X_B$, then there is a rank m submatrix B of the columns $D_1, D_2, \ldots, D_n, E_1, E_2, \ldots, E_m$ such that $X_B = B^{-1}d$ and after multiplying the tableau by

$$\left[\begin{array}{c|c} B^{-1} & 0 \\ \hline c_B^T B^{-1} & 1 \end{array}\right]$$

we obtain the tableau

$$\left[\begin{array}{ccccccccc|c} \cdots & \cdots & Y_j & \cdots & \cdots & \cdots & Y_{j'} & \cdots & \cdots & X_B \\ \hline \cdots & \cdots & z_j - c_j & \cdots & \cdots & \cdots & z_{j'} & \cdots & \cdots & z \end{array}\right]$$

where,

$$\begin{array}{rcl} z & = & c_B^T B^{-1} d, \\ Y_j & = & B^{-1} D_j, \\ Y_{j'} & = & B^{-1} E_{j'}, \end{array}$$

and because of optimality,

$$\begin{aligned} 0 &\leq z_j - c_j = c_B^T B^{-1} D_j - c_j \\ 0 &\leq z_{j'} = c_B^T E_{j'}. \end{aligned}$$

So these equations show that

$$\begin{aligned} c_B^T B^{-1} D - c^T &\geq 0 \\ c_B^T B^{-1} I &\geq 0, \text{ and} \\ z &= c_B^T B^{-1} d. \end{aligned}$$

If we let $W^T = c_B^T B^{-1}$, then these equations show that W satisfies

$$\begin{aligned} D^T W &\geq c \\ W &\geq 0, \text{ and} \\ z &= d^T W. \end{aligned}$$

Thus W is an optimal feasible solution to the dual linear program. Optimality follows from the last equation and Lemma 18.3. □

Observe the similarity between Lemma 18.4 and the *max-flow-min-cut* Theorem (Theorem 10.4). Indeed the Ford-Fulkerson algorithm for solving the maximum network flow problem that was presented in Section 10.2 is a primal-dual algorithm. We re-examine this algorithm in Section 18.6.3.

In the proof of Lemma 18.4 we discovered how to construct an optimal solution of the dual linear program given an optimal solution to the primal. We record this useful fact as the following corollary:

Corollary 18.5. *If X is an optimal solution to*

$$\begin{aligned} \text{Maximize:} \quad & z = c^T X \\ \text{Subject to:} \quad & DX \leq d \\ & X \geq 0 \end{aligned}$$

with basis B, then $W = B^{-T} c_B$ is an optimal solution to the dual linear program

$$\begin{aligned} \text{Minimize:} \quad & Z = d^T W \\ \text{Subject to:} \quad & D^T W \geq c \\ & W \geq 0. \end{aligned}$$

Theorem 18.6. *Given a primal-dual pair, exactly one of the following can occur:*

a. Both the primal and the dual have a finite optimum.

b. The primal is unbounded and the dual is infeasible.

c. The primal is infeasible but the dual is unbounded.

d. Both the primal and the dual are infeasible.

Proof. We saw in Chapter 17 that every linear program either (i) has a finite optimum, (ii) is unbounded, or (iii) is unfeasible. Thus for a primal-dual pair there are nine possibilities. Namely:

1. Both the primal and the dual have a finite optimum.
2. The primal has a finite optimum but the dual is unbounded.
3. The primal has a finite optimum but the dual is infeasible.
4. The primal is unbounded but the dual has a finite optimum.
5. Both the primal and the dual are unbounded.
6. The primal is unbounded and the dual is infeasible.
7. The primal is infeasible but the dual has a finite optimum.
8. The primal is infeasible but the dual is unbounded.
9. Both the primal and the dual are infeasible.

Lemma 18.4 shows that possibilities 2, 3, 4, and 7 cannot occur. Equation 18.1 tells us that if either the primal or the dual is unbounded, then the other cannot have a feasible solution and thus possibility 5 is eliminated. It is easy to construct examples of the remaining four possibilities, 1, 6, 8, and 9.

1. A primal-dual pair in which both the primal and the dual have a finite optimum:

$$\begin{array}{ll} \text{Maximize:} & z = x_1 \\ \text{Subject to:} & x_1 \leq 1 \\ & x_1 \geq 0 \end{array}$$

(Primal)

$$\begin{array}{ll} \text{Minimize:} & Z = w_1 \\ \text{Subject to:} & w_1 \geq 1 \\ & w_1 \geq 0 \end{array}$$

(Dual)

6. A primal-dual pair in which the primal is unbounded and the dual is infeasible:

$$\begin{array}{ll} \text{Maximize:} & Z = -x_1 + 2x_2 \\ \text{Subject to:} & -x_1 + x_2 \leq 1 \\ & x_1, x_2 \geq 0 \end{array}$$

(Primal)

$$\begin{array}{ll} \text{Minimize:} & z = w_1 \\ \text{Subject to:} & -w_1 \geq -1 \\ & w_1 \geq 2 \\ & w_1 \geq 0 \end{array}$$

(Dual)

8. A primal-dual pair in which the primal is infeasible and the dual is unbounded:

$$\begin{array}{ll} \text{Maximize:} & z = x_1 \\ \text{Subject to:} & x_1 \leq 1 \\ & -x_1 \leq -2 \\ & x_1 \geq 0 \end{array}$$

(Primal)

$$\begin{array}{ll} \text{Minimize:} & Z = w_1 - 2w_2 \\ \text{Subject to:} & w_1 - w_2 \geq 1 \\ & w_1, w_2 \geq 0 \end{array}$$

(Dual)

9. A primal-dual pair in which both the primal and the dual are infeasible:

$$\begin{array}{lrcl} \text{Maximize:} & z = -x_1 + 2x_2 & & \\ \text{Subject to:} & x_1 - x_2 & \leq & 1 \\ & -x_1 + x_2 & \leq & -2 \\ & x_1, x_2 \geq 0 & & \\ & \text{(Primal)} & & \end{array} \qquad \begin{array}{lrcl} \text{Minimize:} & Z = w_1 - 2w_2 & & \\ \text{Subject to:} & w_1 - w_2 & \geq & -1 \\ & -w_1 + w_2 & \geq & 2 \\ & w_1, w_2 \geq 0 & & \\ & \text{(Dual)} & & \end{array}$$

□

18.2 Alternate form of the primal and its dual

It is often more convent to write the primal linear program as:

$$\begin{array}{lrcll} \text{Maximize:} & z = c^T X & & & \\ \text{Subject to:} & AX & = & b & \text{(Primal: equality form)} \\ & X \geq 0. & & & \end{array}$$

This is equivalent to

$$\begin{array}{lrcl} \text{Maximize:} & z = c^T X & & \\ \text{Subject to:} & AX & \leq & b \\ & -AX & \leq & -b \\ & X \geq 0, & & \end{array}$$

and this has the dual linear program

$$\begin{array}{lrcl} \text{Minimize:} & Z = [b^T, -b^T] \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} & & \\ \text{Subject to:} & [A^T, -A^T] \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} & \geq & c \\ & W_1, W_2 \geq 0. & & \end{array}$$

This dual is equivalent to:

$$\begin{array}{lrcl} \text{Minimize:} & Z = b^T(W_1 - W_2) & & \\ \text{Subject to:} & A^T(W_1 - W_2) & \geq & c \\ & W_1, W_2 \geq 0. & & \end{array}$$

If we let $W = W_1 - W_2$, then the entries of W are unrestricted in sign and the dual linear program is equivalent to

$$\begin{array}{lrcl} \text{Minimize:} & Z = b^T W & & \\ \text{Subject to:} & A^T W & \geq & c \\ & W \text{ unrestricted.} & & \end{array}$$

Similarly, if we take the dual linear program to be

$$\begin{array}{lll} \text{Minimize:} & Z = b^T W & \\ \text{Subject to:} & A^T W = c & \text{(Dual: equality form)} \\ & W \geq 0. & \end{array}$$

Then its corresponding primal linear program is

$$\begin{array}{ll} \text{Maximize:} & z = c^T X \\ \text{Subject to:} & AX \leq b \\ & X \text{ unrestricted.} \end{array}$$

A similar proof of Corollary 18.5 found in Lemma 18.4 establishes Corollary 18.7.

Corollary 18.7. *If W is an optimal solution to*

$$\begin{array}{lll} \textit{Minimize:} & Z = b^T W & \\ \textit{Subject to:} & A^T W = c & \textit{(Dual: equality form)} \\ & W \geq 0 & \end{array}$$

with basis B, then $X = B^{-T} b_B$ is an optimal solution to the dual linear program

$$\begin{array}{ll} \text{Maximize:} & z = c^T X \\ \text{Subject to:} & AX \leq b \\ & X \text{ unrestricted.} \end{array}$$

18.3 Geometric interpretation

A geometric interpretation can be given to the dual linear program.

Let $A = [A_1, A_2, \ldots, A_n]$ and write the primal linear program as:

$$\begin{array}{lll} \text{Maximize:} & z = c^T X & \\ \text{Subject to:} & x_1 A_1 + x_2 A_2 + \cdots + x_n A_n = b & (18.3) \\ & X \geq 0. & \end{array}$$

Then the dual is

$$\begin{array}{ll} \text{Minimize:} & Z = b^T W \\ \text{Subject to:} & A_1^T W \geq c_1 \\ & A_2^T W \geq c_2 \\ & \vdots \\ & A_n^T W \geq c_n \\ & W \text{ unrestricted.} \end{array} \qquad (18.4)$$

The vectors A_j in the primal linear program 18.3 are the normals to the half-spaces that represent the constraints in the dual linear program 18.4. Furthermore the requirement vector of the primal is normal to the hyperplane $Z = b^T W$ in the dual. This is easy to illustrate in two dimensions by means of an example.

18.3.1 Example

Given the linear program:

$$\begin{array}{lrcl} \text{Maximize:} & z = -3x_1 - 23x_2 - 4x_3 & & \\ \text{Subject to:} & x_1A_1 + x_2A_2 + \cdots + x_5A_5 & = & b \\ & x_1, x_2, \ldots, x_5 \geq 0, & & \end{array}$$

where

$$A_1 = \begin{bmatrix} 2 \\ -1 \end{bmatrix},\ A_2 = \begin{bmatrix} -3 \\ -4 \end{bmatrix},\ A_3 = \begin{bmatrix} -2 \\ 3 \end{bmatrix},\ A_4 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ A_5 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ \text{and } b = \begin{bmatrix} -2 \\ 1 \end{bmatrix},$$

the dual linear program is:

$$\begin{array}{lrcl} \text{Minimize:} & Z = -2w_1 + w_2 & & \\ \text{Subject to:} & 2w_1 - 1w_2 & \geq & -3 \\ & -3w_1 - 4w_2 & \geq & -23 \\ & -2w_1 + 3w_2 & \geq & -4 \\ & w_1 & \geq & 0 \\ & w_2 & \geq & 0. \end{array}$$

(Note that the appearance of slack variables in the primal linear program have caused the variables in the dual to be non-negative.) In Figure 18.1, we have drawn the *requirement-space configuration* of the primal and in Figure 18.2, the *convex set of feasible solutions* is shown as a shaded region.

Whenever two of the constraints hold as strict equalities, the vectors normal to these constraints are a basis for the primal (if the normals are linearly independent). In w_1w_2-space the point w where two dual constraints hold as strict equalities is the intersection of the two lines representing these two constraints. A basis solution to the primal can then be associated with the intersection of each pair of bounding lines for the half-spaces representing the dual constraints.

There are $\binom{5}{2} = 10$ basis solutions to the primal. They are represented by the points P_{ij}, $1 \leq i < j \leq 5$, in Figure 18.2. The point P_{ij} corresponds to having A_i, A_j in the primal basis. In Table 18.1 we display the simplex tableaux corresponding to the various choice of basis $\{A_i, A_j\}$. Two of them yield feasible solutions to the primal, but only one corresponds to a point that is also feasible in the dual. This is basis $\{A_2, A_3\}$, corresponding to the point $P_{23} = (5, 2)$. Furthermore this basis yields the value $z = -8$, obtained by setting $x_1 = x_4 = x_5 = 0$, $x_2 = 0.24$ and $x_3 = 0.65$. This yields the value $Z = [-2, 1]^T[5, 2] = -8$. Thus by Lemma 18.3, this an optimal point. Therefore using the dual simplex method we move from one extreme point of the convex polyhedron to an adjacent one until an optimal extreme point is reached. At this point, the corresponding solution to the primal becomes feasible.

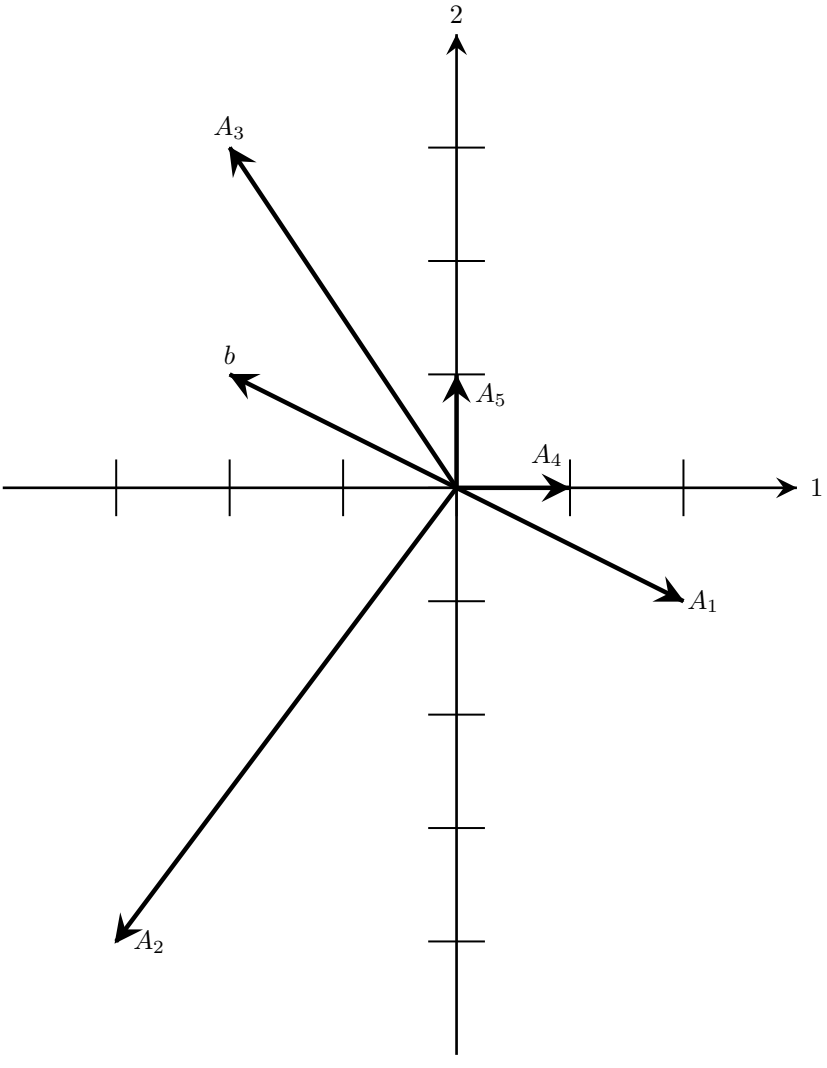


FIGURE 18.1
Requirement-space configuration for the primal

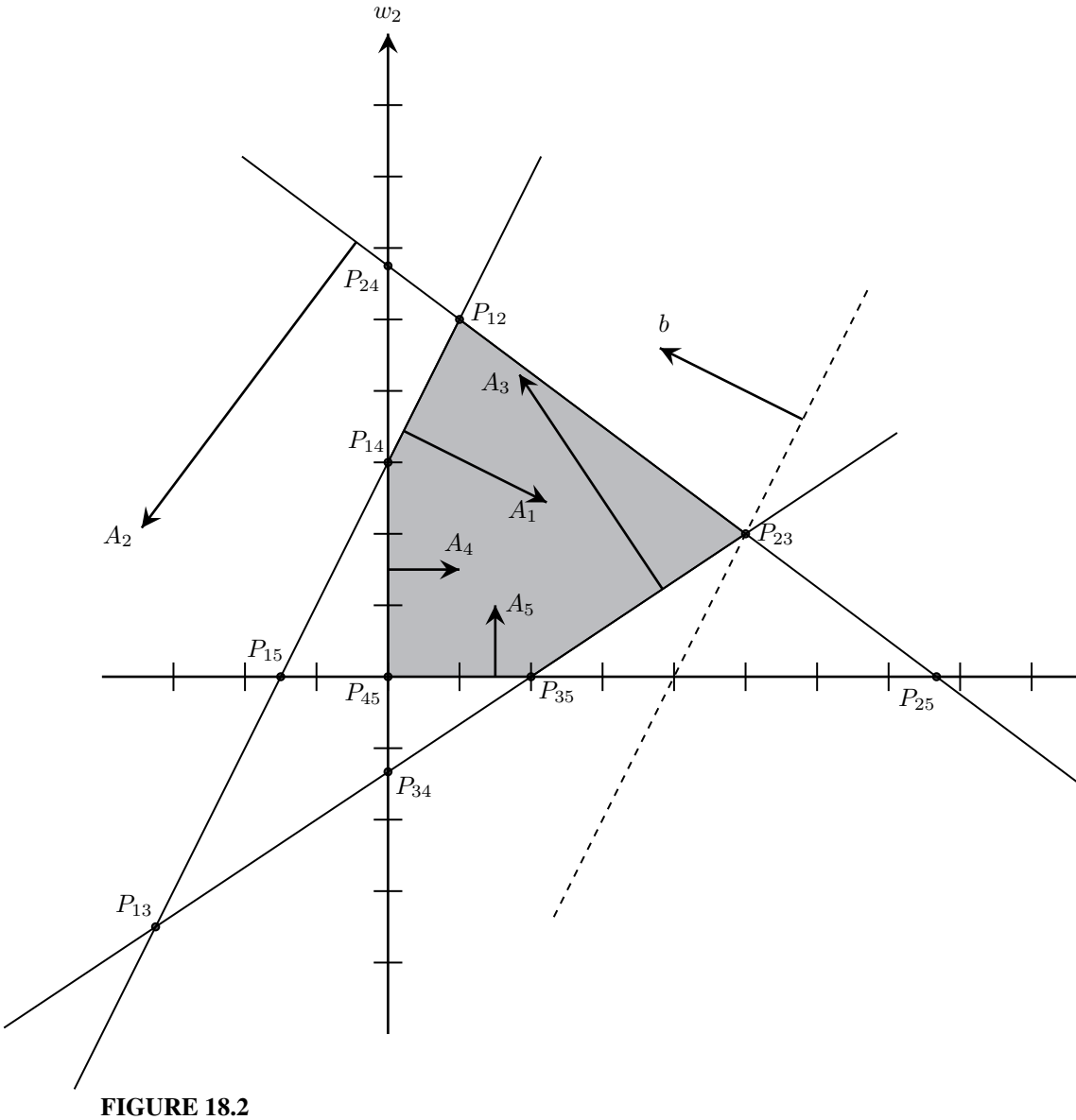


FIGURE 18.2
The convex set of feasible solutions to the dual

TABLE 18.1
Table of simplex tableaux

$\left[\begin{array}{ccccc\|c} 1 & 0 & -1.55 & 0.36 & -0.27 & -1 \\ 0 & 1 & -0.36 & -0.09 & -0.18 & 0 \\ \hline 0 & 0 & -17 & -1 & -5 & -3 \end{array}\right]$	Basis $\{A_1, A_2\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & -4.25 & 0 & 0.75 & 0.50 & -1 \\ 0 & -2.75 & 1 & 0.25 & 0.50 & 0 \\ \hline 0 & -46.75 & 0 & 3.25 & 3.50 & -3 \end{array}\right]$	Basis $\{A_1, A_3\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & 4 & -3 & 0 & -1 & -1 \\ 0 & -11 & 4 & 1 & 2 & 0 \\ \hline 0 & -11 & -13 & 0 & -3 & -3 \end{array}\right]$	Basis $\{A_1, A_4\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 1 & -1.50 & -1 & 0.50 & 0 & -1 \\ 0 & -5.50 & 2 & 0.50 & 1 & 0 \\ \hline 0 & -27.50 & -7 & 1.50 & 0 & -3 \end{array}\right]$	Basis $\{A_1, A_5\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} -0.24 & 1 & 0 & -0.18 & -0.12 & 0.24 \\ -0.65 & 0 & 1 & -0.24 & 0.18 & 0.65 \\ \hline -11 & 0 & 0 & -5 & -2 & 8 \end{array}\right]$	Basis $\{A_2, A_3\}$ yields $z = 8$.
$\left[\begin{array}{ccccc\|c} 0.25 & 1 & -0.75 & 0 & -0.25 & -0.25 \\ 2.75 & 0 & -4.25 & 1 & -0.75 & -2.75 \\ \hline 2.75 & 0 & -21.25 & 0 & -5.75 & -5.75 \end{array}\right]$	Basis $\{A_2, A_4\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} -0.67 & 1 & 0.67 & -0.33 & 0 & 0.67 \\ -3.67 & 0 & 5.67 & -1.33 & 1 & 3.67 \\ \hline -18.33 & 0 & 11.33 & -7.67 & 0 & 15.33 \end{array}\right]$	Basis $\{A_2, A_5\}$ yields $z = 15.33$.
$\left[\begin{array}{ccccc\|c} -0.33 & -1.33 & 1 & 0 & 0.33 & 0.33 \\ 1.33 & -5.67 & 0 & 1 & 0.67 & -1.33 \\ \hline -4.33 & -28.33 & 0 & 0 & 1.33 & 1.33 \end{array}\right]$	Basis $\{A_3, A_4\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} -1 & 1.50 & 1 & -0.50 & 0 & 1 \\ 2 & -8.50 & 0 & 1.50 & 1 & -2 \\ \hline -7 & -17 & 0 & -2 & 0 & 4 \end{array}\right]$	Basis $\{A_3, A_5\}$ is infeasible.
$\left[\begin{array}{ccccc\|c} 2 & -3 & -2 & 1 & 0 & -2 \\ -1 & -4 & 3 & 0 & 1 & 1 \\ \hline -3 & -23 & -4 & 0 & 0 & 0 \end{array}\right]$	Basis $\{A_4, A_5\}$ is infeasible.

18.4 Complementary slackness

There is a battle of balance between the primal and dual linear programs.

As the constraints tighten in one, they loosen in the other.

In this section we denote by $[Row_i(D)]$ and $[Col_j(D)]$ the i^{th} row and j^{th} column of the matrix D, respectively. We also use $X \cdot Y = X^T Y$, the so called dot product.

Theorem 18.8. (Complementary slackness conditions) *A primal-dual feasible solution pair X, W is optimal if and only if*

$$x_j([Col_j(D)] \cdot W - c_j) = 0 \quad \text{for all } j \tag{18.5}$$
$$w_i(d_i - [Row_i(D)] \cdot X) = 0 \quad \text{for all } i. \tag{18.6}$$

Proof. Let

$$u_j = x_j([Col_j(D)] \cdot W - c_j), \text{ and}$$
$$v_i = w_i(d_i - [Row_i(D)] \cdot X).$$

Then, because of the feasibility and the duality relations we have $u_j \geq 0$ for all j and $v_i \geq 0$ for all i. Let

$$u = \sum_j u_j$$
$$v = \sum_i v_i.$$

Then $u, v \geq 0$, $u = 0$ if and only if Equation 18.5 holds for all j and $v = 0$ if and only if Equation 18.6 holds for all j. Observe that

$$\begin{aligned}
u + v &= \sum_j u_j + \sum_i v_i \\
&= \sum_j x_j([Col_j(D)] \cdot W - c_j) + \sum_i w_i(d_i - [Row_i(D)] \cdot X) \\
&= -\sum_j x_j c_j + \sum_i w_i d_i + \sum_j x_j([Col_j(D)] \cdot W - \sum_i w_i [Row_i(D)] \cdot X \\
&= -c \cdot X + d \cdot W + \sum_j x_j \sum_i D[i,j] w_i - \sum_i w_i \sum_j D[i,j] x_j \\
&= -c \cdot X + d \cdot W + \sum_i \sum_j w_i D[i,j] x_j - \sum_i \sum_j w_i D[i,j] x_j \\
&= -c \cdot X + d \cdot W.
\end{aligned}$$

Therefore Equations 18.5 and 18.6 hold for all j and i, respectively, if and only if $u + v = 0$ if and only if $c \cdot X = d \cdot W$ if and only if X and W are both optimal. □

Note that Theorem 18.8 says at optimality if a constraint is not met with equality, i.e., has slack, in the primal linear program, then the corresponding variable in the dual is zero and vice versa.

18.5 The dual of the shortest-path problem

In this section we study the *shortest-path problem* for directed graphs.

Problem 18.1:	Shortest Path (directed graph)
Instance:	a directed graph $G = (V, E)$, nodes $s, t \in V$ and non-negative weights c_j, for each edge $e_j \in E$.
Find:	a directed path P from s to t with minimum total weight $C(P) = \sum_{e_j \in E(P)} c_j$

Let $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$ define the m by n *node-edge incidence matrix* A by

$$A[i,j] = \begin{cases} +1, & \text{if } e_j = (v_i, u) \text{ for some vertex } u, \\ -1, & \text{if } e_j = (u, v_i) \text{ for some vertex } u, \\ 0, & \text{otherwise.} \end{cases}$$

In Figure 18.3 an example is given.

We model the shortest-path problem as a network flow problem by assigning a capacity of 1 on each edge. Let w_j denote the flow on edge j.

The conservation of flow constraints are

$$[Row_i(A)] \cdot W = \begin{cases} 0, & i \notin \{s, t\} \\ +1, & i = s \\ -1, & i = t. \end{cases}$$

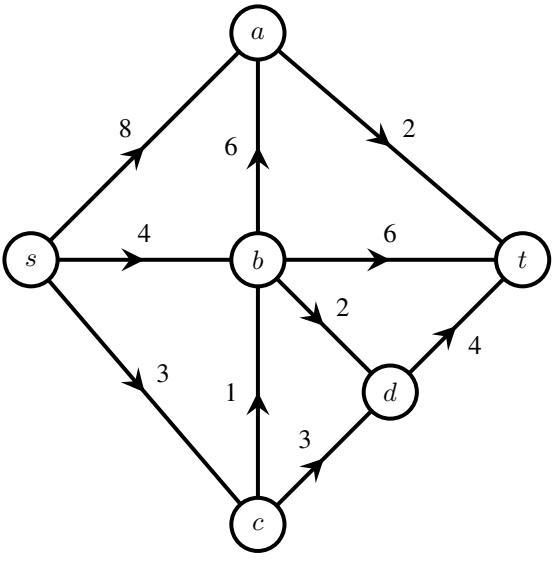
Then W satisfies

$$AW = \begin{bmatrix} +1 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \end{matrix}$$

and the capacity constrains are

$$0 \leq w_j \leq 1.$$

A solution to the shortest-path problem is given by a flow W that minimizes



$$A = \begin{array}{c|cccccccccc} & (s,a) & (s,b) & (s,c) & (a,t) & (b,a) & (b,d) & (b,t) & (c,b) & (c,d) & (d,t) \\ \hline s & +1 & +1 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a & -1 & 0 & 0 & +1 & -1 & 0 & 0 & 0 & 0 & 0 \\ b & 0 & -1 & 0 & 0 & +1 & +1 & +1 & -1 & 0 & 0 \\ c & 0 & 0 & -1 & 0 & 0 & 0 & 0 & +1 & +1 & 0 \\ d & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & +1 \\ t & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 \end{array}$$

$$c^T = [\ 8,\ 4,\ 3,\ 2,\ 3,\ 2,\ 6,\ 3,\ 3,\ 4]$$

FIGURE 18.3
An instance of the shortest-path problem

$Z = c^T W$. As far as we know it is possible that the w_j in general take on non-integer values, but in Section 19.4 we will show that there is an optimal solution W to linear program 18.7 with only integer entries.

$$\begin{aligned} &\text{Minimize:} \quad Z = c^T W \\ &\text{Subject to:} \quad AW = \begin{bmatrix} +1 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \end{matrix} \\ &\qquad\qquad W \geq 0. \end{aligned} \tag{18.7}$$

Indeed it is not to hard to see that there is an optimal solution to linear program 18.7 in which each w_i is either zero or one. A one represents a unit flow along a shortest path from s to t.

The dual linear program for the shortest-path problem is

$$\begin{aligned} &\text{Maximize:} \quad z = x_s - x_t \\ &\text{Subject to:} \quad x_i - x_j \leq c_{ij} \quad \text{for each } (i,j) \in E \\ &\qquad\qquad X \text{ unrestricted.} \end{aligned} \tag{18.8}$$

The complementary slackness conditions (Theorem 18.8) are easy to interpret in the shortest-path problem. A path W and an assignment of variables X are jointly optimal if and only if

1. Each edge in the shortest path (i.e., a positive w_i in the primal linear program 18.7) corresponds to equality in the corresponding constraint in the dual linear program 18.8, and
2. Each strict inequality in the dual linear program 18.8 corresponds to an edge not in the shortest path.

For the graph in Figure 18.3 an optimal solution to the primal is

$$w_{(s,a)} = 0, w_{(s,b)} = 1, w_{(s,c)} = 0, w_{(a,t)} = 1, w_{(b,a)} = 1,$$

$$w_{(b,d)} = 0, w_{(b,t)} = 0, w_{(c,b)} = 0, w_{(c,d)} = 0, w_{(d,t)} = 0,$$

which has cost $= 9$. In the dual we see by complementary slackness that this means

$$\begin{aligned} x_s - x_b &= 4 \\ x_b - x_a &= 3 \\ x_a - x_t &= 2. \end{aligned}$$

Summing these equations we obtain $z = x_s - x_t = 9$.

Exercises

18.5.1 An engineer takes measurements of a variable $y(x)$; the results are in the form of pairs (x_i, y_i). The engineer wishes to find the straight line that fits this data best in the sense that the maximum vertical distance between any point (x_i, y_i) and the line is as small as possible. Formulate this as a linear program. Why might you decide to solve the dual?

18.5.2 Consider the node-edge incidence matrix A of the directed graph $G = (V, E)$ as described in Section 18.5. Show that a set of $|V| - 1$ columns is linearly independent if and only if the corresponding edges, when considered as undirected edges, form a tree. (Thus a basis corresponds to a tree.) If a network problem is formulated with this graph, what does this result say about the pivot step?

18.5.3 It was shown in Section 18.2 that the dual of

$$\begin{array}{ll} \text{Maximize:} & z = c^T X \\ \text{Subject to:} & AX = b \\ & X \geq 0 \end{array}$$

has unrestricted variables. However, if some slack and/or surplus variables appear in A, show that the dual variable for a constraint having a slack variable is non-negative and the dual variable for a constraint having a surplus variable is non-positive. Hence, show that the only dual variables which are really unrestricted are those that correspond to constraints that were originally equations and not inequalities. Thus show that the dual of any linear program is essentially unique and is independent of the particular manner in which we write the primal.

18.6 The primal-dual algorithm

Consider a linear program in standard form

$$\begin{array}{ll} \text{Minimize:} & Z = c^T X \\ \text{Subject to:} & AX = b \\ & X \geq 0 \end{array} \qquad \text{(P)}$$

and its dual

$$\begin{array}{ll} \text{Maximize:} & z = b^T W \\ \text{Subject to:} & A_1^T W \leq c_1 \\ & A_2^T W \leq c_2 \\ & \vdots \\ & A_n^T W \leq c_n \\ & W \text{ unrestricted.} \end{array} \qquad \text{(D)}$$

We may assume that $b > 0$, because the equalities in (P) can be multiplied by -1 where necessary. The complementary slackness conditions (Theorem 18.8) are: if X is a feasible solution to (P) and W is a feasible solution to (D), then X and W are both optimal if and only if

$$w_i([\mathit{Row}_i(A)]^T X - b_i) \quad = \quad 0 \text{ for all } i \tag{18.9}$$

and

$$(c_j - A_j^T W)x_j \quad = \quad 0 \text{ for all } j. \tag{18.10}$$

Condition 18.9 is automatically satisfied because of the equality in (P), so we will focus on condition 18.10. The main idea of the primal dual algorithm is:

> *Given a feasible solution W to (D), find a feasible solution X to (P) such that*
>
> $$X_j = 0, \textit{ whenever } A_j^T W < c_j.$$

In order to construct such a pair W, X we will iteratively improve W, while maintaining its feasibility in (D). Suppose W is a feasible solution to (D). Then with respect to W some of the inequalities $A_j^T W \leq c_j$ in (D) still have slack and some do not. Let

$$J = \{j : A_j^T W = c_j\} = \{j_1, j_2, \ldots, j_{n'}\},$$

be the set of *admissible columns*. So X, a feasible solution to (P), is optimal if $x_j = 0$ for all $j \notin J$. Let

$$A_J \quad = \quad [A_{j_1}, A_{j_2}, \ldots, A_{j_{n'}}]$$

and

$$X_J \quad = \quad [x_{j_1}, x_{j_2}, \ldots, x_{j_{n'}}].$$

If we can find X_J such that

$$\begin{array}{rcl} A_J X_J & = & b \\ X_J & \geq & 0, \end{array} \tag{18.11}$$

then by complementary slackness, the X defined by

$$X[j] = \begin{cases} 0 & \text{if } j \notin J \\ X_{j_\ell} & \text{if } j = j_\ell \in J \end{cases}$$

is optimal in (P). To find X_J we construct a new linear program called the *restricted primal* (RP).

$$\begin{array}{lll} \text{Minimize:} & \zeta = \gamma \begin{bmatrix} X_J \\ Y \end{bmatrix} = y_1 + y_2 + \cdots + y_m & \\ \text{Subject to:} & A_J X_J + Y = [A_J, I] \begin{bmatrix} X_J \\ Y \end{bmatrix} \quad = \quad b & \text{(RP)} \\ & X_J \geq 0 & \\ & Y \geq 0 & \end{array}$$

where $Y = [y_1, y_2, \ldots, y_m]^T$ are new variables one for each equation in (P) and

$$\gamma = [\underbrace{0, 0, \ldots, 0}_{|J|}, \underbrace{1, 1, \ldots, 1}_{m}].$$

Let ζ_{OPT} be the optimal value of (RP) and suppose it occurs at $Y = Y_{\text{OPT}}$, with basis B. If $\zeta_{\text{OPT}} = 0$, then $Y_{\text{OPT}} = 0$ and the corresponding X_J solves the constraints 18.11. Thus we have an optimal solution to (P). What happens when $\zeta_{\text{OPT}} > 0$?

The dual of (RP) is

$$\begin{array}{lrcl} \text{Maximize:} & z = b^T W & & \\ \text{Subject to:} & A_J^T W & \leq & 0 \\ & W & \leq & \overrightarrow{1} \\ & W \text{ unrestricted} & & \end{array} \qquad \text{(DRP)}$$

where $\overrightarrow{1} = [1, 1, 1, \ldots, 1]^T$. Let W_{OPT} be the solution to (DRP) corresponding to Y_{OPT}, that is $W_{\text{OPT}} = B^{-T}\gamma_B$ (see Theorem 18.7). We call (DRP) the *dual-restricted primal*. The situation is that we tried to find a feasible X in (P) using only the columns in J, but failed. However, we do have the optimal feasible solution pair Y_{OPT}, W_{OPT} to (RP) and (DRP), respectively. We also know $\zeta_{\text{OPT}} > 0$, the value of (RP) at Y_{OPT} is positive. Let's try correcting W by a linear combination of the old W and W_{OPT}. Let

$$W_\star = W + \theta W_{\text{OPT}}.$$

The value of the objective function at $W_\star$ is

$$b^T W_\star = b^T W + \theta b^T W_{\text{OPT}}.$$

Now we know $b^T W_{\text{OPT}} = \zeta_{\text{OPT}} > 0$, because $Y_{\text{OPT}}, W_{\text{OPT}}$ is a primal-dual feasible solution pair. Thus to maximize the value of the objective function we can take $\theta > 0$ and large. We also need to maintain feasibility so we need

$$A^T W_\star = A^T W + \theta A^T W_{\text{OPT}} \leq c.$$

Consider the j-th equation

$$A_j^T W_\star = A_j^T W + \theta A_j^T W_{\text{OPT}} \leq c_j.$$

If $A_j^T W_{\text{OPT}} \leq 0$, then this equation is satisfied because W is a feasible solution to (D). Thus, in particular, $W_\star$ is feasible, if $A_j^T W_{\text{OPT}} \leq 0$ for all j, but in this case we may take $\theta > 0$ arbitrarily large and the value of the objective function at $W_\star$ will be unbounded. Therefore the primal (P) is infeasible, by Theorem 18.6. Hence:

Theorem 18.9. *If $\zeta_{\text{OPT}} > 0$ in (RP) and W_{OPT} the optimal solution to (DRP) satisfies*

$$A_j^T W_{\text{OPT}} \leq 0, \text{ for all } j \notin J$$

then (P) is infeasible.

Therefore we need only worry when

$$A_j^T W_{\text{OPT}} > 0 \text{ for some } j \notin J.$$

Consequently, the feasibility conditions are

$$A_j^T W_\star = A_j^T W + \theta A_j^T W_{\text{OPT}} \le c_j, \text{ whenever } j \notin J \text{ and } A_j^T W_{\text{OPT}} > 0.$$

Thus for all $j \notin J$ and $A_j^T W_{\text{OPT}} > 0$ we need to chose θ such that

$$\theta \le \frac{c_j - A_j^T W}{A_j^T W_{\text{OPT}}}.$$

This yields the following theorem.

Theorem 18.10. *When $\zeta_{\text{OPT}} > 0$ in (RP) and there is a $j \notin J$ with $A_j^T W_{\text{OPT}} > 0$, the largest θ that maintains feasibility of $W_\star = W + \theta W_{\text{OPT}}$ is*

$$\theta_\star \quad = \quad \textit{min} \left\{ \frac{c_j - A_j^T W}{A_j^T W_{\text{OPT}}} : j \notin J \textit{ and } A_j^T W_{\text{OPT}} > 0 \right\}. \tag{18.12}$$

Given a feasible solution W to (D) Algorithm 18.6.1 constructs in W an optimal solution to (D) or determines if (P) is infeasible.

Algorithm 18.6.1: PRIMAL-DUAL(W)

```
FEASIBLE ← true
OPTIMAL ← false
while FEASIBLE and not OPTIMAL
do { J ← {j : A_j^T W = c_j}
     if |J| = n
       then OPTIMAL ← true
       else { Solve (RP) by Simplex Algorithm
              obtain solution with basis B and objective value ζ_OPT
              if ζ_OPT = 0
                then OPTIMAL ← true
                else { W_OPT ← B^{-T} γ_B
                       if A_J^T W_OPT ≤ 0 for all j ∈ J
                         then FEASIBLE ← false
                         else { compute θ_⋆ using equation 18.12
                                W ← W + θ_⋆ W_OPT }
```

18.6.1 Initial feasible solution

In order to start Algorithm 18.6.1 we must first have a feasible solution W to (D). If $c_i \geq 0$ for all i, we can take $W = \overrightarrow{0}$ as an initial feasible solution. When $c_i < 0$ for some i, we can use the following method to obtain a feasible solution W to (D). Introduce a new variable x_0 to the primal problem (P), set $c_0 = 0$ and add the constraint :

$$x_0 + x_1 + \cdots + x_n = b_0,$$

where b_0 is taken to be larger than $\sum_i^n x_i$, for every basis solution $X = [x_1, \ldots, x_n]$ to (P). For example, take $b_0 = nM$ where M is given in Lemma 18.11. This new primal is

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X = [\,0, c^T] \begin{bmatrix} x_0 \\ X \end{bmatrix} & & \\ \text{Subject to:} & x_0 + x_1 + \cdots + x_n & = & b_0 \\ & AX & = & b \\ & x_0 \geq 0, X \geq 0 & & \end{array} \qquad \text{(P')}$$

and its dual is

$$\begin{array}{lrcl} \text{Maximize:} & z = w_0 b_0 + b^T W & & \\ \text{Subject to:} & w_0 & \leq & 0 \\ & w_0 + A_1^T W & \leq & c_1 \\ & w_0 + A_2^T W & \leq & c_2 \\ & \vdots & & \\ & w_0 + A_n^T W & \leq & c_n \\ & W \text{ unrestricted} & & \end{array} \qquad \text{(D')}$$

where w_0 is the new variable corresponding to the new equation in the new primal. A feasible solution to this new dual is

$$W_i = \begin{cases} \text{MIN}\{c_1, c_2, \ldots, c_n\}, & \text{if } i = 0 \\ 0, & \text{if } i = 1, 2, \ldots, m. \end{cases}$$

Also $[x_0, X]$ is an optimal solution to the new primal (P') if and only if X is an optimal solution to (P). A suitable value for b_0 is provided by the following lemma.

Lemma 18.11. *Let $X = [x_1, x_2, \ldots, x_n]$ be a basis solution to (P). Then*

$$|x_j| \leq M = m! \alpha^{m-1} \beta$$

where

$$\alpha = \text{MAX}_{i,j}\{|a_{i,j}|\}$$

and

$$\beta = \text{MAX}\{b_1, b_2, \ldots, b_m\}.$$

Proof. Without loss of generality we assume that the entries of A, b, and c are integers. If X is a basis solution, then there is a set J of m columns such that $X[j] = 0$ whenever $j \notin J$, and $B = [A_j : j \in J]$ is nonsingular. Thus $BX_J = b$, where $X_J = [X[j] : j \in J]$ and so by Cramer's rule

$$x_j = \frac{\det(B')}{\det(B)},$$

where B' is the matrix B with column j replaced with b. By integrality we have $|\det(B)| \geq 1$, and so

$$|x_j| \quad \leq \quad |\det(B')|$$

column in the optimal basis of (RP) remains admissible at the start of the next iteration. □

Theorem 18.12.

Proof. Suppose that the optimal basis B of (RP) includes the column A_j Then by Corollary 18.7 the optimal solution to (DRP) corresponding to X is $W_{\text{OPT}} = B^{-T}\gamma_B$, where γ_B are the entries of

$$\gamma = [\underbrace{0, 0, \ldots, 0}_{|J|}, \underbrace{1, 1, \ldots, 1}_{m}]$$

corresponding to the columns of $[A_J, I]$ that are in B. Consequently, for some ℓ, $1 \leq \ell \leq |J|$,

$$\begin{aligned} A_j^T W_{\text{OPT}} &= A_j^T (B^{-T}\gamma_B) \\ &= (A_j^T B^{-T})\gamma_B \\ &= (B^{-1}A_J)^T \gamma_B \\ &= E_\ell^T \gamma_B \\ &= \gamma_B[\ell] = 0, \end{aligned}$$

where $E_\ell = [0, 0, \ldots, 0, 1, 0, 0, \ldots, 0]^T$ with 1 in position ℓ. This implies that

$$\begin{aligned} A_j^T W_\star &= A_j^T (W + \theta W_{\text{OPT}}) \\ &= A_j^T W + \theta W_{\text{OPT}}^T A_j \\ &= \gamma_j + 0 \\ &= \gamma_j. \end{aligned}$$

Thus if $j \in J$ at the start of an iteration of the while loop in Algorithm 18.6.1, then j remains in J in the next iteration. □

How do we know that the primal-dual algorithm will terminate? If the minimum calculation of $\theta_\star$ by Equation 18.12 occurs at $j = j_\star$, then

$$\theta_\star = \frac{c_{j_\star} - A_{j_\star}^T W}{A_{j_\star}^T W_{\text{OPT}}}$$

and so

$$c_{j_\star} = A_{j_\star}^T W + \theta_\star A_{j_\star}^T W_{\text{OPT}} = A_{j_\star}^T W_\star.$$

Consequently, $j_\star$ becomes a new column of J, and we see that $|J|$ monotonically increases. If $J = \{1, 2, \ldots, n\}$, then W satisfies the complementary slackness condition in Equation 18.10 and is therefore optimal. Consequently we have the following theorem.

Theorem 18.13. *Algorithm 18.6.1 correctly solves (P) in at most n iterations.*

18.6.2 The shortest-path problem

We illustrate the primal-dual method with the shortest-path problem introduced in Section 18.5. Observe that the row sum of the coefficient matrix in linear program 18.8 is zero. Hence any row may be omitted because it is redundant. If we choose to omit row t, the resulting linear program 18.13 is our primal.

$$\begin{aligned} &\text{Minimize:} && Z = c^T X \\ &\text{Subject to:} && AX = \begin{bmatrix} +1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } s \\ & && X \geq 0 \end{aligned} \qquad (18.13)$$

The vector of flow is X and each entry of X is 0 or 1. The dual linear program for the shortest-path problem is

$$\begin{aligned} &\text{Maximize} && z = w_s \\ &\text{Subject to:} && w_i - w_j \leq c_{ij} \text{ for each edge} (i, j) \in E \\ & && W \text{ unrestricted, except } w_t = 0. \end{aligned} \qquad (18.14)$$

We fix $w_t = 0$, because its row was omitted from the primal. The set of admissible columns (i.e., edges) is

$$J = \{(i, j) : x_i - x_j = c_{ij}\}$$

and the restricted primal is

$$\begin{array}{ll} \text{Minimize:} & \zeta = y_1 + y_2 + \cdots + y_{m-1} \\ \text{Subject to:} & AX + Y = \begin{bmatrix} +1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } s \\ & x_j \geq 0 \text{ for all } j \in J \\ & x_j = 0 \text{ for all } j \notin J \\ & Y \geq 0. \end{array} \tag{18.15}$$

Consequently, the dual-restricted primal is

$$\begin{array}{ll} \text{Maximize:} & z = w_s \\ \text{Subject to:} & w_i - w_j \leq c_{ij} \text{ for each edge} (i,j) \in J \\ & W \leq \overrightarrow{1} \\ & W \text{ unrestricted.} \end{array} \tag{18.16}$$

It is easy to solve the dual-restricted primal. Note that $w_s \leq 1$, and that we are maximizing w_s, so we can try $w_s = 1$. If $w_i = 1$ and $(i,j) \in J$, then we can satisfy the constraint

$$w_i - w_j \leq 0,$$

by also setting $w_j = 1$. Hence if $P = i_1 i_2 \cdots i_h$ is a path with $(i_a, i_{a+1}) \in J$ for all $a = 1, 2, \ldots, h-1$ and $w_{i_1} = 1$. Then we can set $w_{i_a} = 1$ for each a without violating the constraints. Hence if there is no st-path using edges only in J, then an optimal solution to the dual-restricted primal is given by

$$W_{\text{OPT}}[i] = \begin{cases} 1, & \text{if there is an } si\text{-path using only edges in } J \\ 0, & \text{if there is an } it\text{-path using only edges in } J \\ 1, & \text{for all other vertices } i. \end{cases}$$

We then calculate

$$\theta_\star = \text{MIN}\{c_{ij} - (w_i - w_j) : (i,j) \notin J \text{ and } W_{\text{OPT}}[i] - W_{\text{OPT}}[j] > 0\}$$

and update W and J, and obtain and solve the new dual-restricted primal. If we get to a point where there is an st-path using edges in J, then $w_s = 0$ and we are at an optimal solution, because minimum ζ is equal to the maximum z which is $w_s = 0$. Notice that any st-path that uses only edges in J is optimal.

The primal-dual algorithm reduces the shortest-path problem to the easy calculation which vertices are reachable from the source s.

We illustrate the primal-dual algorithm when applied to the shortest-path problem with the directed graph given in Figure 18.4.

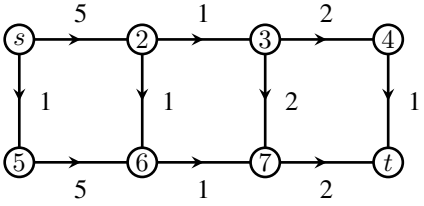
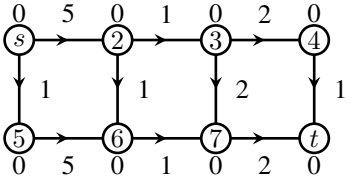
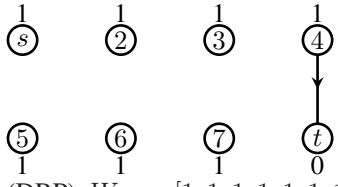


FIGURE 18.4
Shortest-path example

Iteration 1

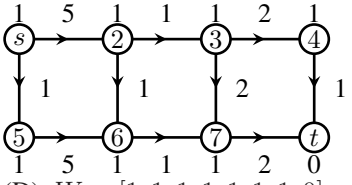


(D): $W = [0, 0, 0, 0, 0, 0, 0, 0, 0]$
$J = \{\}$

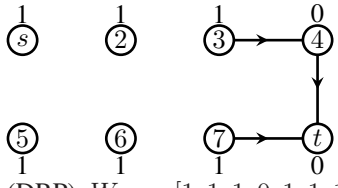


(DRP): $W_{\text{OPT}} = [1, 1, 1, 1, 1, 1, 1, 0]$
$\theta^\star = 1$ for edge $(4, 7)$

Iteration 2

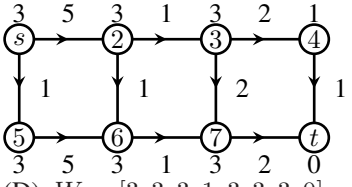


(D): $W = [1, 1, 1, 1, 1, 1, 1, 1, 0]$
$J = \{(4, 7)\}$

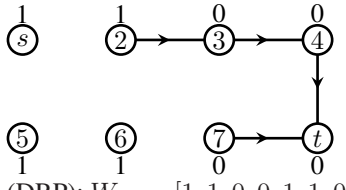


(DRP): $W_{\text{OPT}} = [1, 1, 1, 0, 1, 1, 1, 0]$
$\theta^\star = 2$ for edges $(3, 4)$ and $(7, t)$

Iteration 3

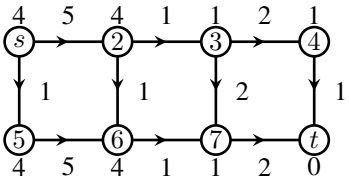


(D): $W = [3, 3, 3, 1, 3, 3, 3, 0]$
$J = \{(4, t), (3, 4), (7, t)\}$



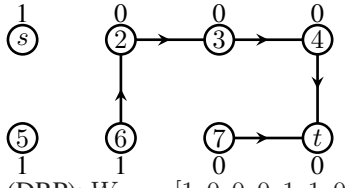
(DRP): $W_{\text{OPT}} = [1, 1, 0, 0, 1, 1, 0, 0]$
$\theta^\star = 1$ for edge $(2, 3)$

Iteration 4



(D): $W = [4, 4, 1, 1, 4, 4, 1, 0]$

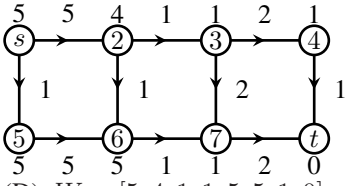
$J = \{(4, t), (3, 4), (7, t), (2, 3)\}$



(DRP): $W_{\text{OPT}} = [1, 0, 0, 0, 1, 1, 0, 0]$

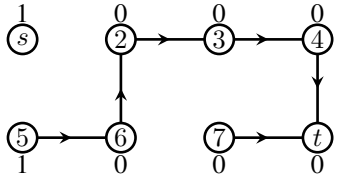
$\theta^\star = 1$ for edge $(6, 2)$

Iteration 5



(D): $W = [5, 4, 1, 1, 5, 5, 1, 0]$

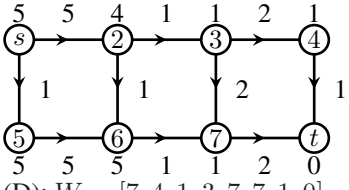
$J = \left\{\begin{matrix}(4, t), (3, 4), (7, t), (2, 3), \\ (6, 2)\end{matrix}\right\}$



(DRP): $W_{\text{OPT}} = [1, 0, 0, 1, 0, 0, 0, 0]$

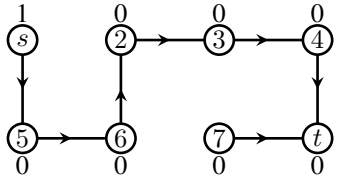
$\theta^\star = 2$ for edge $(5, 6)$

Iteration 6



(D): $W = [7, 4, 1, 3, 7, 7, 1, 0]$

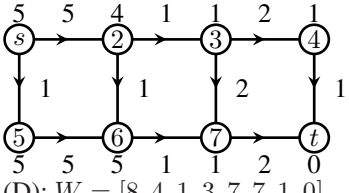
$J = \left\{\begin{matrix}(4, t), (3, 4), (7, t), (2, 3), \\ (6, 2), (5, 6)\end{matrix}\right\}$



(DRP): $W_{\text{OPT}} = [1, 0, 0, 0, 0, 0, 0, 0]$

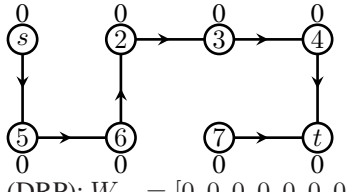
$\theta^\star = 1$ for edge $(s, 5)$

Iteration 7



(D): $W = [8, 4, 1, 3, 7, 7, 1, 0]$

$J = \left\{\begin{matrix}(4, t), (3, 4), (7, t), (2, 3), \\ (6, 2), (5, 6), (s, 5)\end{matrix}\right\}$



(DRP): $W_{\text{OPT}} = [0, 0, 0, 0, 0, 0, 0, 0]$

18.6.3 Maximum flow

A network N with source s and target t can be described as a linear program as follows:

$$\begin{aligned}
&\text{Maximize:} \quad v \\
&\text{Subject to:} \quad Af = \begin{bmatrix} +v \\ -v \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \leftarrow \text{row } s \\ \leftarrow \text{row } t \\ \\ \\ \end{matrix} \\
&\qquad\qquad f \leq c \\
&\qquad\qquad f \geq 0,
\end{aligned} \tag{18.17}$$

where $v = \text{VAL}(f)$ is the value of the flow f and $c = [\text{CAP}(e) : e \in E(N)]^T$ are the capacities of the edges e in N. This is of course equivalent to

$$\begin{aligned}
&\text{Maximize} \quad v \\
&\text{Subject to:} \quad Af + vT \leq 0 \\
&\qquad\qquad\qquad\qquad f \leq c \\
&\qquad\qquad\qquad\qquad -f \leq 0,
\end{aligned} \tag{18.18}$$

where

$$T[x] \quad = \quad \begin{cases} -1, \text{ if } x = s \\ +1, \text{ if } x = t \\ 0, \text{ otherwise.} \end{cases}$$

(Note that $Af + vT \leq 0$ implies $Af + vT = 0$ because we maximize v.) The set of admissible columns (i.e., edges) is

$$J = \{\overrightarrow{uv} \colon f(\overrightarrow{uv}) = c \text{ or } -f(\overrightarrow{uv}) = 0\}.$$

Thus J is the set of saturated edges together with the zero-flow edges and the dual-restricted primal is:

$$\begin{aligned}
&\text{Maximize:} \quad v \\
&\text{Subject to:} \quad Af + vT \leq 0 \\
&\qquad\qquad f(\overrightarrow{uv}) \leq 0, \text{ for all saturated edges } \overrightarrow{uv} \text{ in 18.18} \\
&\qquad\qquad -f(\overrightarrow{uv}) \leq 0, \text{ for all zero flow edges } \overrightarrow{uv} \text{ in 18.18} \\
&\qquad\qquad\qquad f \leq \overrightarrow{1} \\
&\qquad\qquad\qquad v \leq \overrightarrow{1}
\end{aligned} \tag{18.19}$$

It is easy to solve the dual-restricted primal in 18.19. We wish to maximize v and so we can try $v = 1$. Thus we must choose a flow f such that

$$[\mathit{Row}_s(A)] \cdot f = 1$$

Concerning the edges incident to s, the inequalities in 18.19 tell us that we can set $f(\overrightarrow{su}) = 1$ if the edge $\overrightarrow{su}$ has zero flow or is unsaturated and we can set $f(\overrightarrow{us}) = -1$ if the edge $\overrightarrow{us}$ is saturated and/or does not have zero flow. Let S_1 be the set of vertices u incident to s satisfying

if $u \longrightarrow s$, then $\overrightarrow{su}$ has zero flow or is unsaturated;

if $u \longrightarrow s$, then $\overrightarrow{us}$ is saturated or does not have zero flow.

Hence we choose $v_1 \in S_1$ and set the flow on the associated edge to 1 if it leaves s and to -1 if it enters s. Now we must consider $[Row_{v_1}](A)$ and consider the edges incident to u_1. Let S_2 be the set of vertices u incident to some v in S_1 satisfying

$$\left.\begin{array}{l}\text{if } u \longrightarrow s, \text{ then } \overrightarrow{su} \text{ has zero flow or is unsaturated;}\\ \text{if } u \longrightarrow s, \text{ then } \overrightarrow{us} \text{ is saturated or does not have zero flow.}\end{array}\right\} \tag{18.20}$$

In general let S_{k+1} be the set of vertices u incident to some v in S_k satisfying conditions in 18.20 and continue until some S_k contains the target t. When and if it does, we can choose vertices $v_i \in S_i$, $i = 1, 2, \ldots, k-1$ such that $sv_1v_2v_3 \cdots v_{k-1}t$ is a st-path P. We obtain an optimal solution f_{OPT} to the dual-restricted primal in 18.19 as follows:

$$f_{\text{OPT}}(\overrightarrow{uv}) = \begin{cases} +1, & \text{if } \overrightarrow{uv} \text{ is a forward edge of } P\\ -1, & \text{if } \overrightarrow{uv} \text{ is a backward edge of } P\\ 0, & \text{if } \overrightarrow{uv} \text{ is not an edge of } P.\end{cases}$$

We then calculate

$$\begin{aligned}\theta_\star &= \text{MIN}_{uv\in E(P)} \begin{cases} (c(\overrightarrow{uv}) - f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = 1\\ (0 - (-f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = -1\end{cases}\\ &= \text{MIN}_{uv\in E(P)} \begin{cases} (\text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv})), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = 1\\ f(\overrightarrow{uv}), & \text{if } f_{\text{OPT}}(\overrightarrow{uv}) = -1\end{cases}\\ &= \text{MIN}\{\text{RESCAP}(uv) : uv \in E(P)\},\end{aligned}$$

where

$$\text{RESCAP}(uv) = \begin{cases} \text{CAP}(\overrightarrow{uv}) - f(\overrightarrow{uv}), & \text{if } uv \text{ is a forward edge,}\\ f(\overrightarrow{vu}), & \text{if } uv \text{ is a backward edge.}\end{cases}$$

We update the flow by

$$f \leftarrow f + \theta_\star f_{\text{OPT}},$$

recompute the set of admissible edges J to get the new dual-restricted primal, and repeat until J is empty at which point the flow f will be maximum.

It is now easy to see the similarity of this method to that in Section 10.2 and realize that the primal-dual algorithm for network flow is exactly the Ford-Fulkerson algorithm.

Exercises

18.6.1 Consider Problem 18.2 the Weighted Matching problem.

Problem 18.2: Weighted Matching
Instance: undirected graph G and weight $w_e \geq 0$ for each edge e of G.
Find: a matching M of G with maximal possible weight

$$\text{WT}(M) = \sum_{e \in E(M)} w_e.$$

Formulate a primal-dual algorithm for solving Problem 18.2 and give an interpretation for the restricted primal.

18.6.2 Use the primal-dual algorithm as discussed in Section 18.6.2 to find a shortest path between each pair of nodes in the graph given in Figure 18.5.

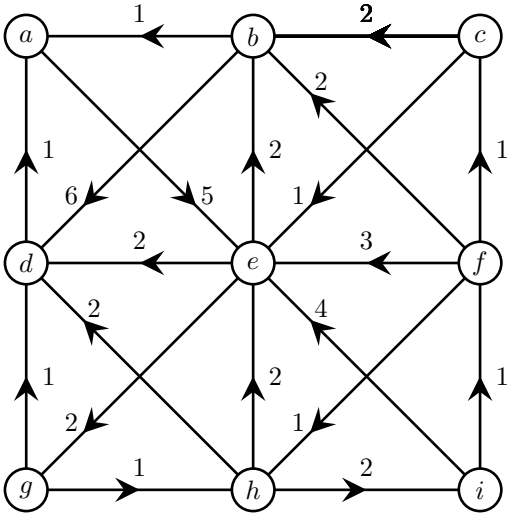


FIGURE 18.5
An instance of the shortest-path problem

18.7 Notes

The primal-dual algorithm was first described in 1956 by DANTZIG; see [42]. Our treatment is similar to that of PAPADIMITRIOU and STIEGLITZ; see [134].

19

Discrete Linear Programming

19.1 Introduction

An *integer linear program* (ILP) is a linear program in which the variables have been constrained to be integers.

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X & & \\ \text{Subject to:} & AX & \leq & b \\ & X & \geq & 0 \\ & X & & \text{integral.} \end{array} \tag{19.1}$$

If all of the variables are each constrained to a finite set of values, we say that the integer linear program is discrete. Notice that frequently the equality constraints force the variables to be discrete, for if $b_i/a_{i,j} > 0$ for all j in the constraint

$$\sum_{i=1}^{n} a_{i,j} x_j = b_i,$$

then x_j cannot exceed $m_j = \lfloor b_i/a_i, j \rfloor$. Hence $x_j \in \{0, 1, 2, \ldots, m_j\}$ for all j.

DEFINITION 19.1: A *discrete linear program* (DLP) is an integer linear program in which the variables are a bounded

$$\begin{array}{ll} \text{Minimize:} & Z = c^T X \\ \text{Subject to:} & AX = b \\ & 0 \leq x_j \leq m_j, j = 1, 2, \ldots, n \\ & X \text{ integral.} \end{array} \tag{19.2}$$

Consider Problem 19.1, the Knapsack problem. This problem is motivated by what to carry on a proposed hiking trip. The weight limit on how much can be carried is the capacity M. Each of the n objects under consideration have a certain weight w_i and each has a certain value or profit p_i, $i = 1, 2, \ldots, n-1$. Furthermore each object can be either carried or left behind. We cannot choose to carry a fraction of an object.

Problem 19.1: Knapsack

Instance: *profits* $p_1, p_1, p_2, \ldots, p_n$;
weights $w_1, w_1, w_2, \ldots, w_n$; and
capacity M;

Find: the maximum value of

$$P = \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$

and $[x_1, \ldots, x_n] \in \{0, 1\}^n$.

This problem can be formulated as the discrete linear program:

$$\begin{array}{ll} \text{Minimize:} & Z = -(p_1 x_1 + p_1 x_1 + \cdots + p_n x_n) \\ \text{Subject to:} & w_1 x_1 + w_1 x_1 + \cdots + w_n x_n \leq M \\ & 0 \leq x_j \leq 1, j = 1, 2, \ldots, n \\ & X \text{ integral.} \end{array}$$

19.2 Backtracking

A discrete integer linear program can be solved with a backtracking search. For example, the Knapsack problem can be solved with Algorithm 19.2.1.

Algorithm 19.2.1: KNAPSACK1(ℓ)

if $\ell > n$

then
- **if** $\sum_{i=1}^{n} w_i x_i \leq M$
  - **then**
    - $\mathit{CurrentProfit} \leftarrow \sum_{i=1}^{n} p_i x_i$
    - **if** $\mathit{CurrentProfit} > \mathit{OptimalProfit}$
      - **then**
        - $\mathit{OptimalProfit} \leftarrow \mathit{CurrentProfit}$
        - $\mathit{OptimalX} \leftarrow [x_1, \ldots, x_n]$

else
- $x_\ell \leftarrow 1$
- KNAPSACK1($\ell + 1$)
- $x_\ell \leftarrow 0$
- KNAPSACK1($\ell + 1$)

Initially Algorithm 19.2.1 is started with $\ell = 1$.

In general the backtracking method to solve a discrete integer linear program is performed by computing for a partial solution, in which values for $x_1, x_2, \ldots, x_{\ell-1}$ have been assigned, the set of possible values $\mathcal{C}_\ell$ for x_ℓ. Each possible value is examined to see whether the partial solution can be extended with it. The general backtracking algorithm is provided in Algorithm 19.2.2

Algorithm 19.2.2: BACKTRACK(ℓ)

if $[x_1, x_1, \ldots, x_\ell]$ is a feasible solution
  then process it
Compute $\mathcal{C}_\ell$
for each $x \in \mathcal{C}_\ell$
  do $\begin{cases} x_\ell \leftarrow x \\ \text{BACKTRACK}(\ell + 1) \end{cases}$

Algorithm 19.2.1, is an application of Algorithm 19.2.2 with $\mathcal{C}_\ell = \{1, 0\}$. We can improve the running time of a backtracking algorithm if we can find efficient ways of reducing the size of the choice sets $\mathcal{C}_\ell$. This process is called *pruning*.

For the Knapsack problem, one simple method is to observe that we must have

$$\sum_{i=1}^{\ell} w_i x_i \leq M$$

for any partial solution $[x_1, x_2, \ldots, x_{\ell-1}]$. In other words, we can check partial solutions to see if the feasibility condition is satisfied. Consequently, if $\ell \leq n$ and we set

$$\mathit{CurrentWt} = \sum_{i=1}^{\ell-1} w_i x_i,$$

then we have

$$\mathcal{C}_\ell = \begin{cases} \{1, 0\}, & \text{if } \mathit{CurrentWt} + w_\ell \leq M; \\ \{0\}, & \text{otherwise.} \end{cases}$$

Using Algorithm 19.2.2 as a template, we obtain Algorithm 19.2.3, which is invoked with $\ell = 1$ and $\mathit{CurrentWt} = 0$.

Algorithm 19.2.3: KNAPSACK2(ℓ)

if $\ell > n$

then { **if** $\sum_{i=1}^{n} p_i x_i > \mathit{OptimalProfit}$ **then** { $\mathit{OptimalProfit} \leftarrow \sum_{i=1}^{n} p_i x_i$; $\mathit{OptimalX} \leftarrow [x_1, \ldots, x_n]$

if $\ell = n$

then $C_\ell \leftarrow \emptyset$

else { **if** $\mathit{CurrentWt} + w_\ell \leq M$ **then** $C_\ell \leftarrow \{1, 0\}$ **else** $C_\ell \leftarrow \{0\}$

for each $x \in \mathcal{C}_\ell$

do { $x_\ell \leftarrow x$; $\mathit{CurrentWt} = \mathit{CurrentWt} + w_\ell x_\ell$; KNAPSACK2($\ell + 1$) ; $\mathit{CurrentWt} = \mathit{CurrentWt} - w_\ell x_\ell$

A backtracking algorithm with simple pruning for solving the discrete integer linear program 19.2 in which the coefficient matrix $A = [A_i, A_2, \ldots, A_n]$ and b consists of only non-negative entries is given as Algorithm 19.2.4.

Algorithm 19.2.4: BACKTRACK2(ℓ)

if $\ell > n$

then { **if** $\sum_{i=1}^{n} c_i x_i < \mathit{OptimalZ}$ **then** { $\mathit{OptimalZ} \leftarrow \sum_{i=1}^{n} c_i x_i$; $\mathit{OptimalX} \leftarrow [x_1, \ldots, x_n]$

comment: Compute C_ℓ

if $\ell = n$

then $C_\ell \leftarrow \emptyset$

else { $C_\ell \leftarrow \{\}$; **for** $x = 0$ **to** m_j **do** { **if** $\mathit{CurrentWt[i]} + A_i x \leq b$ **then** $C_\ell \leftarrow C_\ell \cup \{x\}$

for each $x \in \mathcal{C}_\ell$

do { $x_\ell \leftarrow x$; $\mathit{CurrentWt} = \mathit{CurrentWt} + A_i x_\ell$; BACKTRACK2($\ell + 1$) ; $\mathit{CurrentWt} = \mathit{CurrentWt} - A_i x_\ell$

19.3 Branch and bound

Another strategy for solving an integer linear program is to first ignore the integer constraints and solve the corresponding linear program. This linear program is called the *relaxation* of the integer linear program. If a solution is found to the relaxed integer linear program, it can be rounded to the nearest integer solution. Although this may seem plausible, it generally leads to solutions that are either infeasible or far from the optimal solution. We illustrate this difficulty with the following integer linear program:

$$\begin{array}{lrcl} \text{Minimize:} & Z = -x_1 - x_2 & & \\ \text{Subject to:} & -2x_1 + 7x_2 & \leq & 14 \\ & 2x_1 - 2x_2 & \leq & 1 \\ & x_1, x_2 & \geq & 0 \\ & x_1, x_2 \text{ integral.} & & \end{array} \tag{19.3}$$

This problem is solved graphically in Figure 19.1, and we see that there are six feasible solutions to this integer linear program: $(0,0)$, $(0,1)$, $(1,1)$, $(0,2)$, $(1,2)$, and $(2,2)$. The value of the object function, respectively, at these points is: $0, -1, -2, -2, -3$, and -4. Hence $(2,2)$ is the optimal solution to this integer linear program. On the other hand, the optimal solution to the relaxed integer linear program is $(3.5, 3)$ and the value of the objective function at this point is -6.5. The nearest integer points are $(3,3)$ and $(4,3)$ having values -6 and -7. Neither of these points are feasible and the value of the objective function at them is far from the value at the true optimal solution.

Thus in practice the relaxation strategy can result in misleading information. However, not all is lost. It is not to difficult to see that a bound on the value of the objective function is obtained.

Theorem 19.1. *Let $\widehat{X}$ be an optimal solution to relaxation of the integer linear program*

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X & & \\ \text{Subject to:} & AX & = & b \\ & X & \geq & 0 \\ & X & & \text{integral} \end{array} \tag{19.4}$$

Then any solution X to the integer linear program 19.4 satisfies

$$c^T X \leq c^T \widehat{X}.$$

Proof. If X is a feasible solution to the integer linear program, it is also a feasible solution to the linear program, in which the integer constraints have been removed. □

If values have been assigned to each of $x_1, x_2, \ldots, x_{\ell-1}$, then the remaining

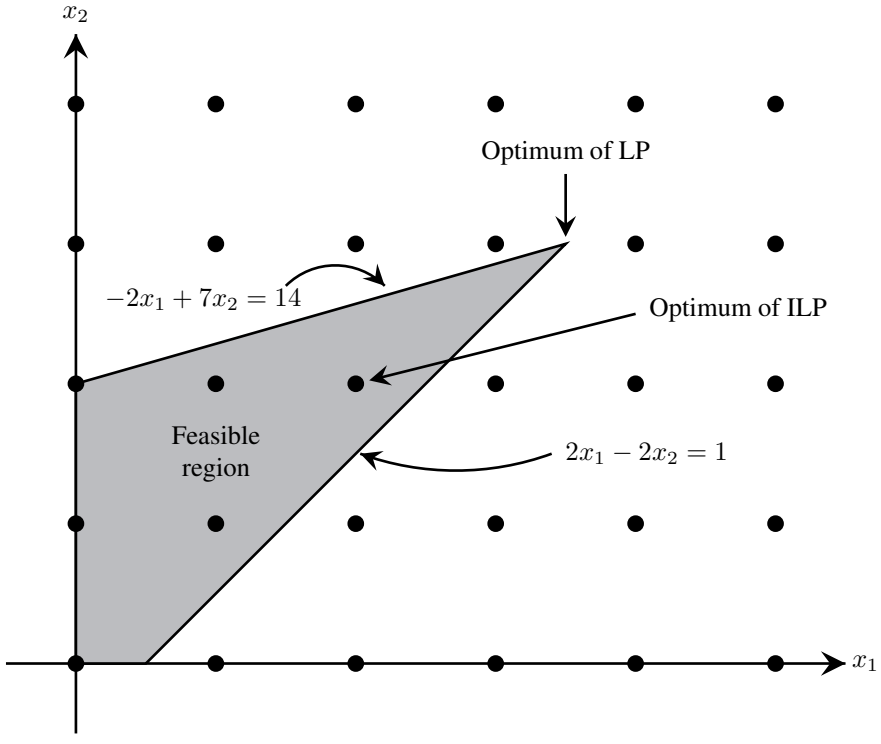


FIGURE 19.1
The integer points closest to the optimal solution to the standard linear program are infeasible in the integer linear program.

variables $x_\ell, x_{\ell+1}, \ldots, x_n$ must satisfy

$$\begin{array}{lrcl} \text{Minimize:} & \widehat{Z} = \widehat{c}^T \widehat{X} & & \\ \text{Subject to:} & \widehat{A}\widehat{X} & \leq & \widehat{b} \\ & \widehat{X} & \geq & 0 \\ & \widehat{X} & \text{integral,} & \end{array}$$

where

$$\begin{array}{rcl} \widehat{A} & = & [A_\ell, A_{\ell+1}, \ldots, A_n] \\ \widehat{X} & = & [x_\ell, x_{\ell+1}, \ldots, x_n] \\ \widehat{c} & = & [c_\ell, c_{\ell+1}, \ldots, c_n] \\ \widehat{b} & = & b - \mathit{CurrentWt} \end{array}$$

and

$$\mathit{CurrentWt} = \sum_{j=1}^{\ell-1} x_j A_j.$$

(Here A_j is the j^{th} column of the coefficient matrix A.) Thus according to Theorem 19.1, any feasible solution X that extends $x_1, x_2, \ldots, x_{\ell-1}$ has value $Z = c^T X$ no larger than

$$B = \mathit{CurrentZ} + \lfloor \widehat{Z}_{\text{OPT}} \rfloor,$$

where $\widehat{Z}_{\text{OPT}}$ is value of the objective function $\widehat{Z} = \widehat{c}^T \widehat{X}$ for the linear program

$$\begin{array}{lrcl} \text{Minimize:} & \widehat{Z} = \widehat{c}^T \widehat{X} & & \\ \text{Subject to:} & \widehat{A}\widehat{X} & \leq & \widehat{b} \\ & \widehat{X} & \geq & 0. \end{array}$$

Consequently, if a feasible solution X_0 has already been obtained that has value $Z = c^T X_0 \leq B$, then no extension of the given assignment to the variables $x_1, x_2, \ldots, x_{\ell-1}$ will lead to an improved Z value. Hence the search for such extensions can be aborted. This method is known as *branch and bound* and is recorded as Algorithm 19.3.1, which we start with $\ell = 1$, $\mathit{CurrentWt} = 0$, $\mathit{CurrentZ} = 0$, and $\mathit{OptimalZ} = \infty$.

Algorithm 19.3.1: BACKTRACK3(ℓ)

if $\ell > n$

then {
- **if** *CurrentWt* $= b$
  - **then** {
    - **if** *CurrentZ* $<$ *OptimalZ*
      - **then** {
        - **for** $j = 1$ **to** n **do** *OptimalX*$[j] \leftarrow X[j]$
        - *OptimalZ* $\leftarrow$ *CurrentZ*
      }
  - **return**

}

Compute C_ℓ

Use the simplex algorithm to solve the linear program:

$$\begin{array}{lrcl} \text{Minimize:} & \widehat{Z} = \widehat{c}^T \widehat{X} & & \\ \text{Subject to:} & \widehat{A}\widehat{X} & = & \widehat{b}, \text{ where} \\ & \widehat{X} & \geq & 0 \end{array} \quad \begin{cases} \widehat{A} = [A_\ell, A_{\ell+1}, \ldots, A_n] \\ \widehat{X} = [x_\ell, x_{\ell+1}, \ldots, x_n] \\ \widehat{c} = [c_\ell, c_{\ell+1}, \ldots, c_n] \\ \widehat{b} = b - \textit{CurrentWt} \end{cases}$$

if the linear program has optimal value $\widehat{Z}_{\text{OPT}}$ at $\widehat{X}_{\text{OPT}}$

then {
- $B \leftarrow$ *CurrentZ* $+ \lfloor \widehat{Z}_{\text{OPT}} \rfloor$
- **if** $\widehat{X}_{\text{OPT}}$ is integer valued
  - **then** {
    - **if** $B <$ *OptimalZ*
      - **then** {
        - **for** $i \leftarrow 1$ **to** $(\ell - 1)$ **do** *OptimalX*$[i] \leftarrow X[i]$
        - **for** $i \leftarrow \ell$ **to** n **do** *OptimalX*$[i] \leftarrow \widehat{X}_{\text{OPT}}[i]$
        - *OptimalZ* $\leftarrow B$
      }
  - **else** {
    - **for each** $x \in C_\ell$
      - **do** {
        - **if** $B \geq$ *OptimalZ* **then return**
        - $x_\ell \leftarrow x$
        - *CurrentWt* $=$ *CurrentWt* $+ A_\ell x_\ell$
        - *CurrentZ* $=$ *CurrentZ* $+ c_\ell x_\ell$
        - BACKTRACK3$(\ell + 1)$
        - *CurrentWt* $=$ *CurrentWt* $- A_\ell x_\ell$
        - *CurrentZ* $=$ *CurrentZ* $- c_\ell x_\ell$
      }
  }

}

if the linear program is unbounded

then {
- **for each** $x \in C_\ell$
  - **do** {
    - $x_\ell \leftarrow x$
    - *CurrentWt* $=$ *CurrentWt* $+ A_\ell x_\ell$
    - *CurrentZ* $=$ *CurrentZ* $+ c_\ell x_\ell$
    - BACKTRACK3$(\ell + 1)$
    - *CurrentWt* $=$ *CurrentWt* $- A_\ell x_\ell$
    - *CurrentZ* $=$ *CurrentZ* $- c_\ell x_\ell$
  }

}

if the linear program is infeasible **then return**

By way of example we provide the backtracking state space search tree in Figure 19.2 that results when Algorithm 19.3.1 is applied to the integer linear program 19.3. Observe that adding the two constraints of linear program 19.3 we see

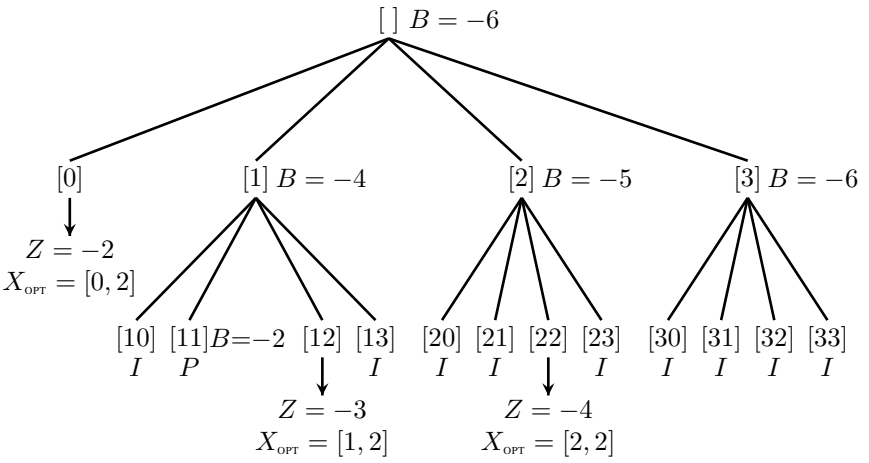


FIGURE 19.2
The backtracking state space search tree that results when Algorithm 19.3.1 is applied to the integer linear program 19.3.

that $x_2 \leq 3 = m_2$ and thus $x_1 \leq 3 = m_1$ as well. These bounds means that the linear program is a discrete linear program and indeed we can apply Algorithm 19.3.1. Each leaf-node in Figure 19.2 is decorated with I, P, or an arrow. The I indicates that the corresponding reduced integer linear program is infeasible, the P indicates pruning by a previously obtained feasible solution and an arrow indicates that a new optimal solution X_{OPT} has been found. Its value $Z = c^T X_{\text{OPT}}$ is also given. For this example the final optimal solution is $X_{\text{OPT}} = [2, 2]$ with value $Z = -4$. This agrees with our previous analysis.

As a second example consider the linear program

$$\begin{array}{lrcl} \text{Minimize:} & Z = -2x_1 - 3x_2 - x_3 - 2x_4 & & \\ \text{Subject to:} & x_1 + 2x_2 + 3x_3 + 2x_4 & \leq & 8 \\ & -3x_1 + 4x_2 + x_3 + 3x_4 & \leq & 8 \\ & 3x_1 - 4x_2 - 6x_3 - 10x_4 & \leq & -20 \\ & x_1, x_2, x_3, x_4 & \geq & 0 \\ & x_1, x_2, x_3, x_4 \text{ integral.} & & \end{array} \tag{19.5}$$

The constraint $x_1 + 2x_2 + 3x_3 + 2x_4 \leq 8$ provides the upper bounds $m_1 = 8, m_2 = 4, m_3 = 2$, and $m_4 = 4$, on the variables x_1, x_2, x_3, and x_4, respectively. Thus we can take $C_1 = \{0, 1, \ldots, 8\}$, $C_2 = C_4 = \{0, 1, \ldots, 4\}$, and $C_3 = \{0, 1, 2\}$. The backtracking state space search tree is given in Figures 19.3 and 19.4.

Notice in this second example, that if we were to process the subtree with root [2] prior to the subtree [0] and [1], pruning would be dramatic and the search would be quite short. This is because the bound obtained when the relaxed linear program with $x_1 = 2$ leads to a bound of -10, and there is an integer solution that achieves this bound, namely, $X = [2, 0, 0, 3]$ in the subtree with root [2]. The bound obtained when the root is [0] and [1] is -7 and -10, respectively, so these subtrees will be

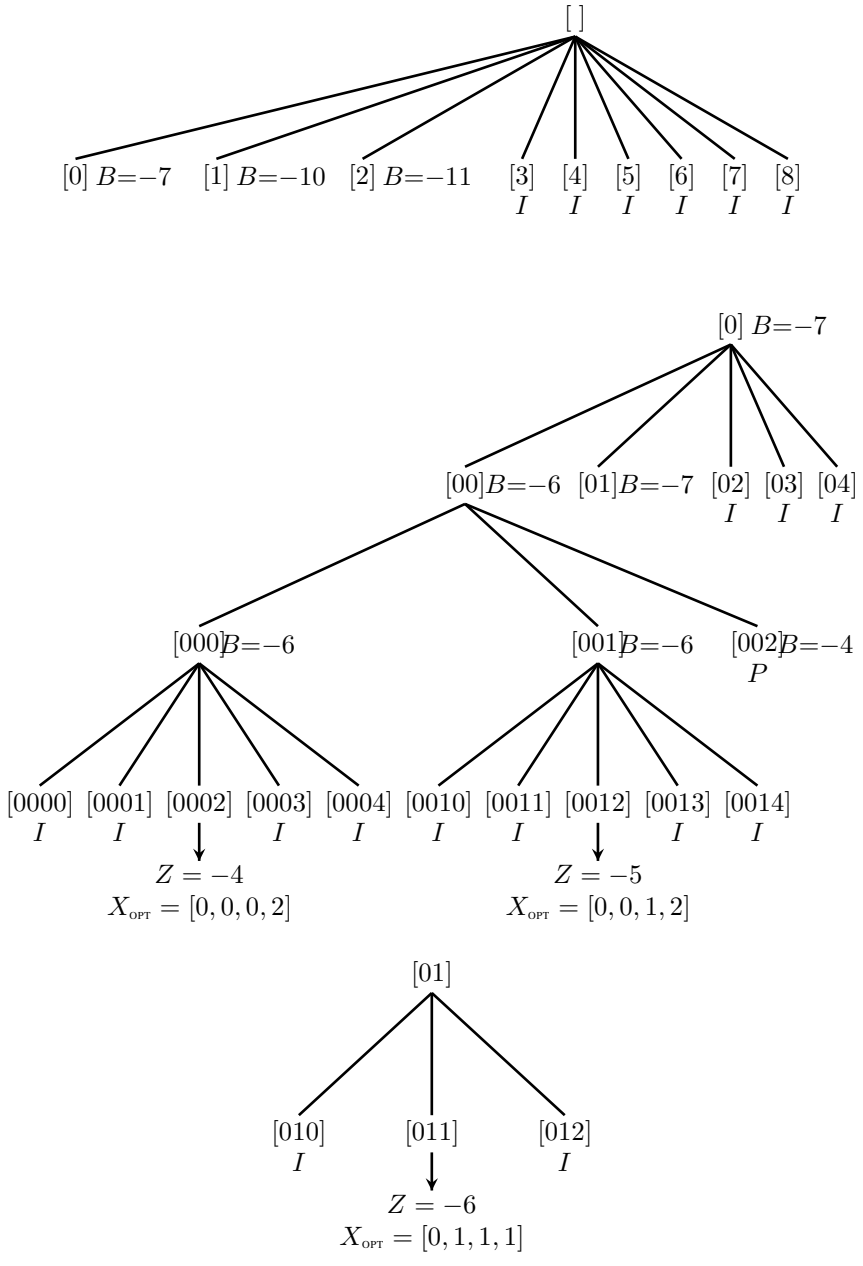


FIGURE 19.3
The first part of the backtracking state space search tree that results when Algorithm 19.3.1 is applied to the integer linear program 19.5. So far we see that $\boldsymbol{X}_{\text{OPT}} = [\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$ gives $\boldsymbol{z} = \mathbf{-6}$.

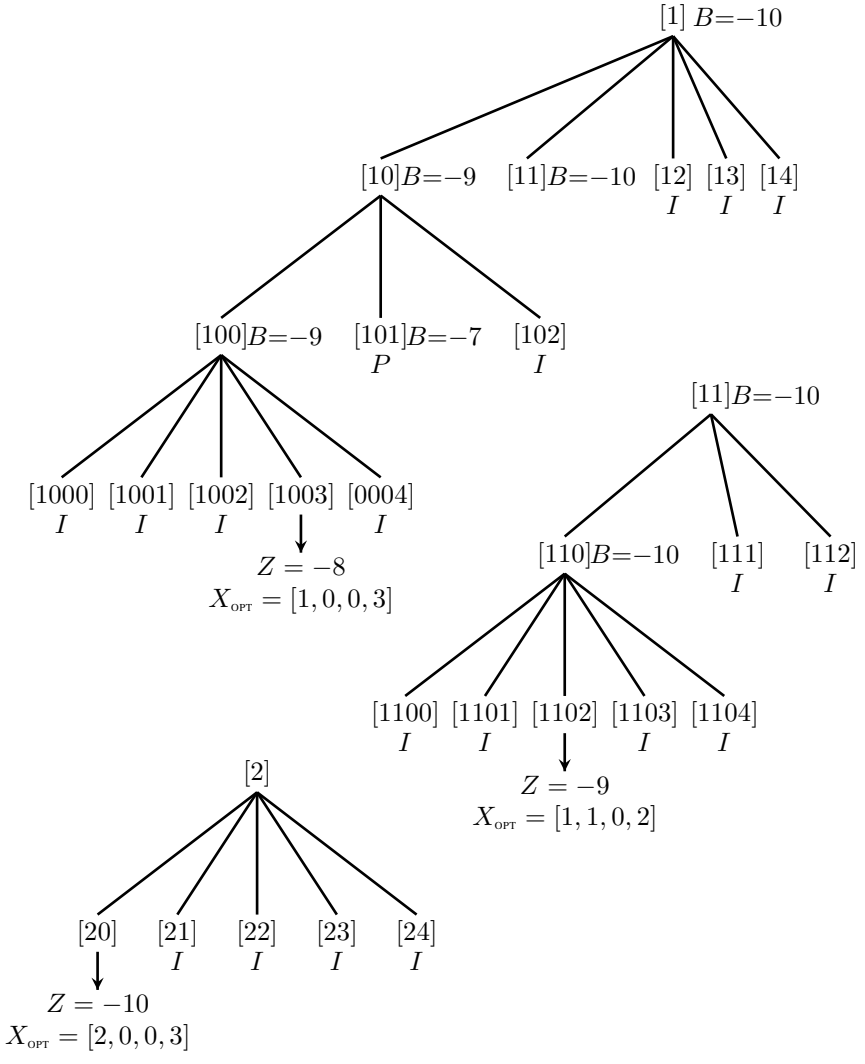


FIGURE 19.4
Continuation of Figure 19.3. Notice the subtree with root $[\mathbf{101}]$ is pruned because $[\mathbf{1003}]$ gave $\boldsymbol{z = -8}$. The final solution appears at $[\mathbf{20}]$ and is $\boldsymbol{X_{\text{OPT}} = [2, 0, 0, 3]}$ with objective value $\boldsymbol{z = -10}$.

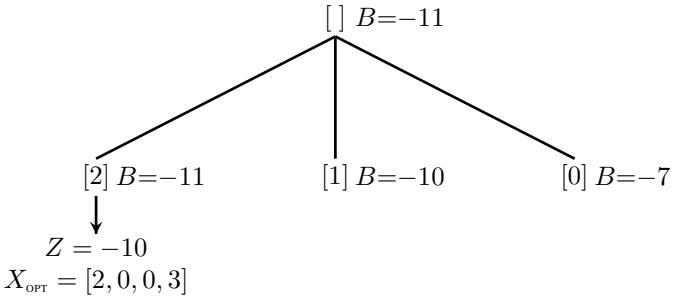


FIGURE 19.5
Dramatic pruning when the subtrees are processed in order of the bounds

pruned if [2] is examined first. The search tree with this type of pruning is given in Figure 19.5.

One way to implement this type of pruning is as follows. After values for $x_1, x_2, \ldots, x_{\ell-1}$ have been assigned, precompute the bounds obtained for each possible assignment for x_ℓ and then process the assignment in order of increasing bound. Algorithm 19.3.2 describes this method. The danger in using this method is that at each node we are required to do sorting. If there are m_ℓ choices for x_ℓ, this will require $O(m_\ell \log(m_\ell))$ additional steps at each node that occurs at level $\ell - 1$ in the search tree. This computation may be prohibitive. (There no additional calls to the simplex algorithm.)

The relaxed linear program for the Knapsack problem is easy to solve. A straightforward method which uses a *greedy strategy*, to solve the relaxed Knapsack problem is given in Algorithm 19.3.3. (See Exercises 19.3.4, 19.3.5, and 19.3.6.) It returns the optimal profit for the Knapsack problem, in which the integer constraint has been relaxed to allow non-integer (i.e., rational) solutions.

Algorithm 19.3.2: BACKTRACK4(ℓ)

if $\ell > n$
then { **if** *CurrentWt* $= b$
then { **if** *CurrentZ* $<$ *OptimalZ*
then { **for** $j = 1$ **to** n **do** *OptimalX*$[j] \leftarrow X[j]$
OptimalZ $\leftarrow$ *CurrentZ* }
return }
$N_\ell \leftarrow 0$
if $\ell \leq n$ **then**
{ **for** $x \leftarrow 0$ **to** m_ℓ
do { Use the simplex algorithm to solve the linear program:

Minimize: $\widehat{Z} = \widehat{c}^T \widehat{X}$
Subject to: $\widehat{A}\widehat{X} = \widehat{b}$, $\widehat{X} \geq 0$, where $\widehat{A} = [A_{\ell+1}, \ldots, A_n]$, $\widehat{X} = [x_{\ell+1}, \ldots, x_n]$, $\widehat{c} = [c_{\ell+1}, \ldots, c_n]$, $\widehat{b} = b - \textit{CurrentWt} - x * A_\ell$

if the linear program has optimal value $\widehat{Z}_{\text{OPT}}$ at $\widehat{X}_{\text{OPT}}$
then { $B \leftarrow$ *CurrentZ* $+ \lfloor \widehat{Z}_{\text{OPT}} \rfloor$
if $\widehat{X}_{\text{OPT}}$ is integer valued
then { **if** $B <$ *OptimalZ* **then**
{ **for** $i \leftarrow 1$ **to** $\ell - 1$ **do** *OptimalX*$[i] \leftarrow X[i]$
OptimalX$[\ell] \leftarrow x$
for $i \leftarrow \ell + 1$ **to** n **do** *OptimalX*$[i] \leftarrow \widehat{X}_{\text{OPT}}[i]$
OptimalZ $\leftarrow B$ }
else if $B <$ *OptimalZ* **then**
{ $N_\ell \leftarrow N_\ell + 1$
$C_\ell[N_\ell] \leftarrow (x, B)$ }
if the linear program is unbounded
then { $N_\ell \leftarrow N_\ell + 1$
$C_\ell[N_\ell] \leftarrow (x, -\infty)$ }
if the linear program is infeasible **then return**
Sort the ordered pairs in C_ℓ in order of increasing second coordinate
for $h \leftarrow 1$ **to** N_ℓ
do { **if** $Bound_\ell[h] \geq$ *OptimalZ* **then return**
$x_\ell \leftarrow$ first coordinate of the ordered pair $C_\ell[h]$
CurrentWt $\leftarrow$ *CurrentWt* $+ A_\ell x_\ell$
CurrentZ $\leftarrow$ *CurrentZ* $+ c_\ell x_\ell$
BACKTRACK3($\ell + 1$)
CurrentWt $\leftarrow$ *CurrentWt* $- A_\ell x_\ell$
CurrentZ $\leftarrow$ *CurrentZ* $- c_\ell x_\ell$ }

Algorithm 19.3.3: RELAXEDKNAPSACK$(p_1, p_2, \ldots, p_n, w_1, \ldots, w_n, M)$

permute the indices so that $p_1/w_1 \geq p_2/w_2 \geq p_n/w_n$
$i \leftarrow 1; \quad P \leftarrow 0; \quad W \leftarrow 0$
for $j \leftarrow 1$ **to** n **do** $x_j \leftarrow 0$
while $W < M$ **and** $i \leq n$
do **if** $W + w_i \leq M$
  then $x_i \leftarrow 1; \quad W \leftarrow W + w_i; \quad P \leftarrow P + p_i$
    $i \leftarrow i + 1$
  else $x_i \leftarrow (M - W)/w_i; \quad W \leftarrow M; \quad P \leftarrow P + x_i\, p_i$
    $i \leftarrow i + 1$
return (P)

To solve an instance of the Knapsack problem, it will be useful to presort the objects in non-decreasing order of the profit/weight ratio, before we begin the backtracking algorithm. Then, when we apply Algorithm 19.3.3, the first step will be unnecessary, and consequently RELAXEDKNAPSACK will run faster. Thus, we will assume that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}.$$

The improved Knapsack algorithm is given as Algorithm 19.3.4.

Algorithm 19.3.4: KNAPSACK3(ℓ)

if $\ell = n$
then **if** *CurrentProfit* $>$ *OptimalProfit*
  then *OptimalProfit* $\leftarrow$ *CurrentProfit*
    OptimalX $\leftarrow [x_1, \ldots, x_n]$
if $\ell = n$
then $C_\ell \leftarrow \emptyset$
else **if** *CurrentWt* $+ w_\ell \leq M$
  then $C_\ell \leftarrow \{1, 0\}$
  else $C_\ell \leftarrow \{0\}$
$B \leftarrow$ *CurrentProfit*
  $+$RELAXEDKNAPSACK$(p_\ell, \ldots, p_n, w_\ell, \ldots, w_n, M -$ *CurrentWt*$)$
for each $x \in C_\ell$
do **if** $B \leq$ *OptimalProfit* **then return**
  $x_\ell \leftarrow x$
  CurrentWt $=$ *CurrentWt* $+ w_\ell x_\ell$
  CurrentProfit $=$ *CurrentProfit* $+ p_\ell x_\ell$
  KNAPSACK3$(\ell + 1)$
  CurrentWt $=$ *CurrentWt* $- w_\ell x_\ell$
  CurrentProfit $=$ *CurrentProfit* $- p_\ell x_\ell$

Exercises

19.3.1 Solve graphically the following integer linear programs:

(a)
$$\begin{cases} \text{Minimize:} & Z = 5x_1 - 3x_2 \\ \text{Subject to:} & x_1 + x_2 \leq 6 \\ & 2x_1 + 4x_2 \leq 15 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{ integral.} \end{cases}$$

(b)
$$\begin{cases} \text{Minimize:} & Z = x_1 - x_2 \\ \text{Subject to:} & x_1 + x_2 \leq 6 \\ & -8x_1 - 5x_2 \leq -40 \\ & -5x_1 + 6x_2 \leq 30 \\ & 6x_1 + 7x_2 \leq 88 \\ & 9x_1 - 5x_2 \leq 18 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{ integral.} \end{cases}$$

Use Algorithm 19.3.4 to solve the following instances of the Knapsack problem.

(a) *Profits* 122 2 144 133 52 172 169 50 11 87 127 31 10 132 59
Weights 63 1 71 73 24 79 82 23 6 43 66 17 5 65 29
Capacity 323

(b) *Profits* 143 440 120 146 266 574 386 512 106 418 376 124 48 535 55
Weights 72 202 56 73 144 277 182 240 54 192 183 67 23 244 29
Capacity 1019

(c) *Profits* 818 460 267 75 621 280 555 214 721 427 78 754 704 44 371
Weights 380 213 138 35 321 138 280 118 361 223 37 389 387 23 191
Capacity 1617

19.3.2 Algorithm 19.3.4 does not take advantage of the fact that given a partial solution X', if the optimal solution to the corresponding relaxed knapsack problem is integer-valued it gives that best solution X that extends X'. Hence there is no need to pursue further extensions, and the search can be pruned. This type of pruning was done for the general problem in Algorithm 19.3.1. Construct a new algorithm that takes advantage of this pruning for the Knapsack problem. Test your algorithm on the data in Exercise 1. How does it compare with Algorithm 19.3.4?

19.3.3 Program Algorithm 19.3.1 and use it to solve the following integer linear

program.

$$\begin{array}{lrcl} \text{Minimize:} & Z = x_1 - 3x_2 + x5 & & \\ \text{Subject to:} & x_1 + 2x_2 + 3x_4 & \leq & 6 \\ & 5x_1 + 4x_2 + 9x_5 & \leq & 20 \\ & X & \geq & 0 \\ & X & & \text{integral.} \end{array}$$

19.3.4 Prove that Algorithm 19.3.3 does indeed solve

Problem 19.2: Relaxed Knapsack
Instance: *profits* $p_1, p_1, p_2, \ldots, p_n$;
weights $w_1, w_1, w_2, \ldots, w_n$; and
capacity M;
Find: the maximum value of

$$P = \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$

and $x_1, x_2 \ldots, x_n$ are rational

as was claimed.

19.3.5 Verify that the simplex algorithm when applied to Problem 19.2 gives exactly the same result as Algorithm 19.3.3.

19.3.6 Determine the running time complexity of Algorithm 19.3.3.

19.3.7 In Section 11.6 we studied the traveling salesman problem. Let $W(v_i v_j)$ be the non-negative weight on the edge $v_i v_j$ of the complete graph with vertices $V = \{v_1, v_2, \ldots, v_n\}$. The traveling salesman problem is to find a hamilton cycle C that has minimum weight

$$\sum_{uv \in E(C)} W(uv).$$

Let $x_{uv} \in \{0, 1\}$ be a variable that denotes an edge uv in the hamilton cycle if $x_{uv} = 1$ and an edge not in the cycle if $x_{uv} = 0$. Show that the optimal solution to the discrete linear program

$$\begin{array}{lrcll} \text{Minimize:} & Z = \sum_{uv} x_{uv} W(uv) & & & \\ \text{Subject to:} & \sum_{u \in V \setminus \{v\}} x_{uv} W(uv) & = & 2, & v \in V \\ & \sum_{uv \in [S, \widehat{S}]} x_{uv} W(uv) & \geq & 1, & \emptyset \neq S \subset V \\ & x_{uv} & \in & \{0, 1\}, & \text{for each edge } uv \end{array}$$

solves the traveling salesman problem.

19.3.8 Prove that the problem

Problem 19.3: ILP decision
Instance: an integer linear program.
Question: does the given integer linear program have a feasible solution?

is NP-complete. (*Hint*: Transform from 3-Sat.)

19.3.9 Use Algorithm 19.3.1 to determine the maximum number of edge disjoint triangles in the complete graph K_n, for $n = 7, 8, 9, \ldots, 13$. *Hint*: Use the $\binom{n}{2}$ by $\binom{n}{3}$ matrix A whose rows are labeled by the edges, whose columns are labeled by the triangles and whose $[e, t]$-entry is 1 if e is an edge on triangle t and is 0 otherwise. When $n \equiv 1, 3 (\mathrm{mod}\ 6)$, then the maximum number edge disjoint triangles is $n(n-1)/6$. The corresponding collection of edge disjoint triangles is called a *Steiner triple system*.

19.4 Totally unimodular matrices

In Sections 18.6.2 and 18.6.3 we studied primal-dual algorithms for the Shortest Path and Max-Flow problems, respectively. Surprisingly we found that their optimal solutions were always integral although we never required that the variables be constrained to be integers. The reason for this is that the node-edge incidence matrix of any digraph is *totally unimodular* (*TUM*).

DEFINITION 19.2: An m by n integer valued matrix is *totally unimodular* if the determinant of each square submatrix is equal to 0, 1, or -1.

Theorem 19.2. *Every basis feasible solution to the linear program*

$$\begin{array}{lrcl} \textit{Minimize:} & Z = c^T X & & \\ \textit{Subject to:} & AX & = & b \\ & X & \geq & 0, \end{array}$$

where A is a totally unimodular m by n matrix, and b is integer-valued, is integer-valued.

Proof. If X is the basis feasible solution corresponding to the submatrix B composed of m linearly independent columns $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$, then

$$X_B = B^{-1}b = \frac{\textsc{Adj}(B)}{\det(B)}b,$$

where $\textsc{Adj}(B)$ is the adjoint of B. Hence X_B has integer values, because the total unimodularity of A implies that the $\det(B) = \pm 1$. Finally

$$X[j] = \begin{cases} X_B[j_\ell], & \text{if } j_\ell = j \\ 0, & \text{otherwise,} \end{cases}$$

and so the entries of X are integers. □

Theorem 19.3. *Every basis feasible solution to the linear program*

$$\begin{array}{lrcl} \textit{Minimize:} & Z = c^T X & & \\ \textit{Subject to:} & AX & \leq & b \\ & X & \geq & 0, \end{array}$$

where A is a totally unimodular m by n matrix, and b is integer-valued, is integer-valued.

Proof. Adding slack variables Y we obtain the following equivalent linear program:

$$\begin{array}{lrcl} \text{Minimize:} & Z = c^T X & & \\ \text{Subject to:} & [A, I_m] \begin{bmatrix} X \\ Y \end{bmatrix} & = & b \\ & X, Y & \geq & 0. \end{array}$$

Thus we need only show that if A is a totally unimodular, then $[A, I_m]$ is totally unimodular, where I_m is the m by m identity matrix. Then the result follows from Theorem 19.2. Let M be a square nonsingular submatrix of $[A, I_m]$; then after a suitable permutation of the rows and columns we see that M has the form

$$\left[\begin{array}{c|c} B & 0 \\ \hline N & I_{m-k} \end{array}\right]$$

where B is a square k by k submatrix of A, and I_ℓ is the ℓ by ℓ identity matrix, for some k and ℓ. The determinant of B is ± 1, because A is totally unimodular and permutations of the rows and columns of M only change the determinant of M by a factor of ± 1. Thus

$$\det(M) = \pm \det(B) \det(I_{m-k}) = \pm 1.$$

□

Theorem 19.4. *The node-edge incidence matrix of a directed graph is totally unimodular.*

Proof. Let $G = (V, E)$ be a directed graph and let A be its node-edge incidence matrix. Then

$$A[v, e] = \begin{cases} +1, & \text{if } e \text{ leaves } v, \\ -1, & \text{if } e \text{ enters } v, \\ 0, & \text{otherwise.} \end{cases}$$

In particular, A has exactly two non-zero entries in each column, one is a -1 and the other is $+1$. Let M be any k by k submatrix of A. If $k = 1$, then clearly $\det(M) = 0$, $+1$, or -1. So suppose $k > 1$ and proceed by induction. If M contains a column of

zeros, then $\det(M) = 0$. If M contains a column j with a single non-zero entry $a = \pm 1$ say in row i, then $\det(M) = \pm a \det(N)$ where N is the $k-1$ by $k-1$ submatrix obtained by removing column j and row i. By induction we have $\det(N) = 0$, $+1$ or -1, and so $\det(M) = 0$, $+1$ or -1. Finally we have the case when each column has two non-zero entries in each column. One is a -1 and the other is a $+1$; hence, each column sums to zero and therefore M is singular and hence has determinant zero. □

Exercises

19.4.1 Show that the following statements are all equivalent:

(a) A is totally unimodular.
(b) The transpose of A is totally unimodular.
(c) $[A, I_m]$ is totally unimodular.
(d) A matrix obtained by deleting a row or column of A is totally unimodular.
(e) A matrix obtained by multiplying a row or column of A by -1 is totally unimodular.
(f) A matrix obtained by duplicating a row or column of A is totally unimodular.
(g) A matrix obtained by pivoting on an entry of A is totally unimodular.

19.4.2 Show that the matrix

$$\begin{bmatrix} 1 & -1 & 0 & 0 & -1 \\ -1 & 1 & -1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ -1 & 0 & 0 & -1 & 1 \end{bmatrix}$$

is totally unimodular.

19.4.3 Let G be an undirected bipartite graph with bipartition (X, Y). Show that the vertex-edge incidence matrix M of G is totally unimodular.

$$M[v, e] = \begin{cases} 1, & \text{if } v \text{ is incident to } e \\ 0, & \text{otherwise.} \end{cases}$$

19.5 Notes

An excellent treatment of backtracking algorithms is given in the book by KREHER and STINSON [111]. The treatment of the Knapsack problem and exercise 19.3.1 is taken from this book. Two other good books that discuss general integer linear programming are PAPIDIMITRIOU and STEIGLITZ [134] and NEMHAUSER and WOLSEY [132].

Bibliography

1. A.V. AHO, J.E. HOPCROFT, AND J.D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1974.
2. M. AIGNER, *Graph Theory, a Development from the 4-Color Problem*, BCS Associates, Moscow, Idaho, 1987.
3. F. ALLAIRE, Another proof of the four colour theorem, Proceedings of the Seventh Manitoba Conference on Numerical Mathematics and Computing (Univ. Manitoba, Winnipeg, Man., 1977), pp. 3–72, *Congressus Numerantium*, XX, Utilitas Mathematica, Winnipeg, Man.,1978.
4. F. APÉRY, *Models of the Real Projective Plane*, Friedr. Vieweg und Sohn Verlagsgesellschaft, Braunschweig, 1987.
5. K. APPEL AND W. HAKEN, Every planar map is four colorable. part I: discharging, *Illinois Journal of Mathematics* 21 (1977), pp. 429-490.
6. K. APPEL AND W. HAKEN, Every planar map is four colorable. part II: reducibility, *Illinois Journal of Mathematics* 21 (1977), pp. 491-456.
7. D. ARCHDEACON, A Kuratowski theorem for the projective plane, *Journal of Graph Theory* 5 (1981), pp. 243-246.
8. E. ARJOMANDI, An efficient algorithm for colouring the edges of a graph with $\Delta + 1$ colours, *Discrete Mathematical Analysis and Combinatorial Computation*, University of New Brunswick, 1980, pp. 108-132.
9. J. BANG-JENSEN AND G. GUTIN, *Digraphs*, Springer-Verlag, New York, 2002.
10. M. BEHZAD AND G. CHARTRAND, *Introduction to the Theory of Graphs*, Allyn & Bacon, Boston, 1971.
11. L.W. BEINEKE AND R.J. WILSON, *Selected Topics in Graph Theory*, Academic Press, London, 1978.
12. L.W. BEINEKE AND R.J. WILSON, *Selected Topics in Graph Theory 2*, Academic Press, London, 1983.
13. L.W. BEINEKE AND R.J. WILSON, *Selected Topics in Graph Theory 3*, Academic Press, London, 1988.
14. CLAUDE BERGE, *Graphs and Hypergraphs*, North-Holland Publishing Co., Amsterdam, 1979.

15. CLAUDE BERGE, *Principles of Combinatorics*, Academic Press, New York, 1971.

16. J.-C. BERMOND, Hamiltonian graphs, in [11], pp. 127-168.

17. N. BIGGS, *Algebraic Graph Theory*, second edition, Cambridge University Press, Cambridge, 1993.

18. R.G. BLAND, New finite pivoting rules for the simplex method. *Mathematics of Operations Research* 2 (1977), 103-107.

19. B. BOLLOBÁS, *Graph Theory, An Introductory Course*, Springer-Verlag, New York, 1979.

20. B. BOLLOBÁS, *Modern Graph Theory*, Springer-Verlag, New York, 2002.

21. J.A. BONDY AND R.L. HEMMINGER, Graph reconstruction – a survey, *J. of Graph Theory* 1 (1977), 227-268.

22. J.A. BONDY, A Graph Reconstructor's Manual *Surveys in Combinatorics*, Ed. A.D. Keedwell, London Mathematical Society Lecture Notes 166, Cambridge University Press, 1991.

23. J.A. BONDY AND U.S.R. MURTY, *Graph Theory with Applications*, American Elsevier Publishing Co., New York, 1976.

24. J.A. BONDY AND U.S.R. MURTY, *Graph Theory*, Springer Verlag, 2008.

25. D. BRELAZ, New methods to color the vertices of a graph, *Communications ACM* 22 (1970), pp. 251-256.

26. HERMANN BUER AND ROLF H. MÖHRING, A fast algorithm for the decomposition of graphs and posets, *Mathematics of Operations Research* 8 (1983), pp. 170-184.

27. G. BUTLER, *Fundamental Algorithms for Permutation Groups*, Lecture Notes in Computer Science 559, Springer-Verlag, Berlin, 1991.

28. M. CAPOBIANCO AND J.C. MOLLUZZO, *Examples and Counterexamples in Graph Theory*, Elsevier North-Holland, New York, 1978.

29. D. CHERITON AND R.E. TARJAN, Finding minimum spanning trees, *SIAM Journal of Computing* 5 (1976), pp. 724-742.

30. G. CHARTRAND AND F. HARARY, Graphs with prescribed connectivities, in, *Theory of Graphs, Proceedings Tihany, 1966*, Ed. P. Erdös and G. Katona, Academic Press, 1968, pp. 61-63.

31. G. CHARTRAND AND L. LESNIAK, *Graphs and Digraphs*, Wadsworth & Brooks/Cole, Monterey, California, 1986.

32. G. CHARTRAND AND O. OELLERMANN, *Applied and Algorithmic Graph Theory*, McGraw-Hill, Inc., New York, 1993.

33. S.A. CHOUDUM, A simple proof of the Erdös-Gallai theorem on graph sequences, *Bulletin of the Australian Math. Soc.* 33 (1986), pp. 67-70.

34. N. CHRISTOFIDES, *Graphs Theory, An Algorithmic Approach*, Academic Press, London, 1975.

35. V. CHVÁTAL, *Linear Programming*, W.H. Freeman and Co., 1983.

36. S.A. COOK, The complexity of theorem proving procedures, *Proc. Third ACM Symposium on the Theory of Computing, ACM* (1971), pp. 151-158.

37. D.G. CORNEIL, H. LERCHS, AND L. STEWART BURLINGHAM, Complement reducible graphs, *Discrete Applied Mathematics* 3 (1981), pp. 163-174.

38. H.S.M. COXETER, *Regular Polytopes*, Dover Publications, New York, 1973.

39. D. CVETKOVIC, M.D. DOOB, AND H. SACHS, *Spectra of Graphs: Theory and Applications*, John Wiley & Sons, 1998.

40. G.B. DANTZIG, Programming of independent activities, II, mathematical model. *Econometrics* 17 (1949), pp. 200-211.

41. G.B. DANTZIG, Programming of independent activities, II, mathematical model, in *Activative of Production and Allocations*, ed. T.C. Koopermans, John Wiley & Sons, New York, 1951, pp. 19-32.

42. G.B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, 1963.

43. N. DEO, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

44. R. DIESTEL, *Graph Theory*, Graduate Texts in Mathematics 173, Springer-Verlag, New York, Berlin, Heidelberg, 1997.

45. G.A. DIRAC, Some theorems on abstract graphs, *Proceedings London Mathematical Society* 2 (1952), pp. 69-81.

46. J.R. EDMONDS, A combinatorial representation for polyhedral surfaces, *Notices American Mathematical Society* 7 (1960), pp. 646.

47. J.R. EDMONDS, Paths, trees, and flowers, *Canadian Journal of Mathematics* 17 (1965), pp. 449-467.

48. J.R. EDMONDS AND R.M. KARP, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the Association of Computing Machinery* 19 (1972), pp. 248-264.

49. M.C. ESCHER, *The Graphic Work of M.C. Escher*, Gramercy Publishing Co., New York, 1984.

50. S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979.

51. J.R. FIEDLER, J.P. HUNEKE, R.B. RICHTER, AND N. ROBERTSON, Computing the orientable genus of projective graphs, *Journal of Graph Theory* 20 (1995), pp. 297-308.

52. J-C. FOURNIER, Colorations des aretes d'un graphe, *Cahiers du CERO* 15 (1973), pp. 311-314.

53. L.R. FORD, JR. AND D.R. FULKERSON, Maximal flow through a network, *Canadian Journal of Mathematics* 8 (1956), pp. 399-404.

54. M. FRÉCHET AND KY FAN, *Initiation to Combinatorial Topology*, Prindle, Weber, & Schmidt, Inc., Boston, 1967.

55. C. FREMUTH-PAEGER AND D. JUNGNICKEL, An introduction to balanced network flows, in *Codes and Designs*, Ohio State University, Math. Res. Inst. Publ. 10 (2000), pp. 125-144.

56. RUDOLF FRITSCH AND GERDA FRITSCH, *The Four Color Theorem*, Springer Verlag, New York, 1998.

57. D.R. FULKERSON, ED., *Studies in Graph Theory, Parts I and II*, Mathematical Association of America, Washington, D.C., 1975.

58. D.R. FULKERSON, Flow networks and combinatorial operations research, in [57], pp. 139-171.

59. A. GAGARIN, *Graph Embedding Algorithms*, Ph.D. thesis, University of Manitoba, 2003.

60. A. GAGARIN AND W. KOCAY, Embedding graphs contianing K_5 subdivisions, *Ars Combinatoria* 64 (2002), pp. 33-49.

61. A. GAGARIN, W. MYRVOLD, AND J. CHAMBERS, The obstructions for toroidal graphs with no $K_{3,3}$'s, *Discrete Math.* 309 (2009), no. 11, pp. 3625-3631.

62. T. GALLAI, Transitiv orientierbare Graphen, *Acta Math. Acad. Sci. Hung.* 18 (1967), pp. 25-66.

63. CYRIL F. GARDINER, *A First Course in Group Theory*, Springer-Verlag, New York, 1980.

64. M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, California, 1979.

65. C.F. GAUSS, Die Kugel, *in Werke* 8 (c. 1819), pp. 351-356.

66. A. GIBBONS, *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, 1985.

67. H.H. GLOVER, J.P. HUNEKE, AND C.S. WANG, 103 graphs that are irreducible for the projective plane, *J. Combin. Theory Ser. B* 27 (1979), pp. 332-370.

68. C.D. GODSIL, On the full automorphism group of a graph, *Combinatorica* 1 (1981), pp. 243-256.

69. C.D. GODSIL AND W.L. KOCAY, Constructing graphs with pairs of pseudo-similar vertices, *J. Combinatorial Th. B* 32 (1981), pp. 146-155.

70. C. GODSIL AND G. ROYLE, *Algebraic Graph Theory*, Springer-Verlag, New York, 2001.

71. MARK GOLDBERG, A non-factorial algorithm for testing isomorphism of two graphs, *Discrete Applied Mathematics* 6 (1983), pp. 229-236.

72. A.W. GOODMAN, On the number of acquaintances and strangers at a party, *Amer. Math. Monthly* 66 (1959), pp. 778-783.

73. R. GOULD, *Graph Theory*, Benjamin/Cummings Publishing, Menlo Park, California, 1988.

74. J.L. GROSS AND T.W. TUCKER, *Topological Graph Theory*, John Wiley & Sons, New York, 1987.

75. J. GROSS AND J. YELLEN, *Graph Theory and Its Applications*, CRC Press, Boca Raton, Florida, 1999.

76. B. GRÜNBAUM, *Convex Polytopes*, Springer-Verlag, New York, 2003.

77. CARSTEN GUTWENGER AND PETRA MUTZEL, A linear time implementation of SPQR trees, *Proceedings of the Eight International Symposium on Graph Drawing*, *Lecture Notes in Computer Science*, (2001), pp. 77-90.

78. G. HADLEY, *Linear Programming*, Addison-Wesley Publishing Co., 1962.

79. P. HALL, On representatives of subsets, *Journal london Mathematical Society* 10 (1935), pp. 26-30.

80. F. HARARY, *Graph Theory*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1972.

81. L. HEFFTER, Über das Problem der Nachbargebiete, *Mathematisce Annalen* 38 (1891), pp. 477-508.

82. P.HELL AND J. NEŠETŘIL, *Graphs and Homomorphisms*, Oxford University Press, New York, 2004.

83. D. HILBERT AND S. COHN-VOSSEN, *Geometry and the Imagination*, Chelsea Publishing Co., New York, 1983.

84. A.J. HOFFMAN, Eigenvalues of graphs, in [57], pp. 225-245.

85. A.J. HOFFMAN AND R.R. SINGLETON, On Moore graphs with diameters 2 and 3, *IBM Journal of Research and Development* 4 (1960), pp. 497-504.

86. I. HOLYER, The NP-completeness of edge-coloring, *SIAM Journal of Computing* 10 (1981), pp. 718-720.

87. J.E. HOPCROFT AND R.M. KARP, An $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM Journal of Computing* 2 (1973), pp. 225-231.

88. J. HOPCROFT AND R.E. TARJAN, Efficient planarity testing, *Journal of the Association of Computing Machinery* 21 (1974), pp. 449-568.

89. J. HOPCROFT AND R.E. TARJAN, Algorithm 447: efficient algorithms for graph manipulation, *CACM* 16 (1973), pp. 372-378.

90. J. HOPCROFT AND R.E. TARJAN, Dividing a graph into triconnected components, *SIAM Journal of Computing* 2 (1973), pp. 135-158.

91. PIERRE ILLE, Indecomposable graphs, *Discrete Mathematics* 173 (1997), pp. 71-78.

92. W. IMRICH, Graphical regular representations of groups of odd order, *Combinatorics* (Proc. Hungarian Colloq., Keszthely, 1976) Vol II, pp. 611-621.

93. B. IVERSEN, *Hyperbolic Geometry*, London Mathematical Society, Cambridge University Press, 1992.

94. FRANCOIS JAEGER, Nowhere Zero Flow Problems, *Selected Topics in Graph Theory 3*, pp. 71-96. Ed. Lowell Beineke and Robin Wilson. Academic Press, San Diego, CA, 1988.

95. D.S. JOHNSON, Worst case behavior of graph coloring algorithms, *Proceedings of the Fifth Southeastern Conference on Combinatorics, Graph Theory, and Computing*, *Congressus Numerantium* 10 (1974), pp. 513-527.

96. R.M. KARP, Reducibility among combinatorial problems, in *Complexity of Computer Computations*, eds. R.E. Miller and J.W. Thatcher, Plenum Press, New York, 1972, pp. 85-103.

97. L.G. KHACHIAN, A polynomial algorithm in linear programing, *Doklady Adademiia Nauk SSR* 224 (1979), 1093-1096. (English translation: *Soviet Mathematics Doklady* 20 (1979), 191-194.)

98. KIMBLE, SCHWENK, AND STOCKMEYER, Pseudosimilar vertices in a graph, *J. Graph Th.* 5 (1981), 171-181.

99. V. KLEE AND G.J. MINTY, How good is the simplex algorithm?, in *Inequalities-III*, ed. O. Shisha, Acdemic Press, New York, pp. 159-175, 1972.

100. F. KLEIN, Vergleichende Betrachtungen über neuere geometrische Forschungen (Erlanger Programm), Ges. Math. Abhandl. 1 (1872), 460–497.

101. W. KLOTZ, A constructive proof of Kuratowski's theorem, *Ars Combinatoria*, 28 (1989), pp. 51-54.

102. D.E. KNUTH, *The Art of Computer Programming*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1973.

103. D.E. KNUTH, *Searching and Sorting*, Addison Wesley Publishing Co., Reading, Massachusetts, 1973.

104. W. KOCAY, D. NEILSON, AND R. SZYPOWSKI, Drawing graphs on the torus, *Ars Combinatoria* 59 (2001), 259-277.

105. W. KOCAY, An extension of the multi-path algorithm for finding hamilton cycles, *Discrete Mathematics* 101, (1992), pp. 171-188.

106. W. KOCAY, Some new methods in reconstruction theory, *Combinatorics IX*, Lecture Notes in Mathematics 952, pp. 89-114, Springer, 1982.

107. W. KOCAY, On writing isomorphism programs, in *Computational and Constructive Design Theory*, pp. 135-175 Editor: W.D. Wallis, Kluwer Academic Publishers, 1996.

108. W. KOCAY AND PAK-CHING LI, An algorithm for finding a long path in a graph, *Utilitas Mathematica* 45 (1994) pp. 169-185.

109. W. KOCAY AND C. PANTEL, An algorithm for constructing a planar layout of a graph with a regular polygon as outer face, *Utilitas Mathematica* 48 (1995), pp. 161-178.

110. W. KOCAY AND D. STONE, Balanced network flows, *Bulletin of the Institute of Combinatorics and Its Applications* 7 (1993), pp. 17-32.

111. D.L. KREHER AND D.R. STINSON, *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press, Boca Raton, Florida, 2000.

112. C. KURATOWSKI, Sur le problème des curbes gauche en topologie, *Fund. Math.* 15 (1930), pp. 271-283.

113. J. LAURI AND R. SCAPELLATO, *Topics in Graph Automorphisms and Reconstruction*, London Mathematical Society Student Texts 54, Cambridge University Press, 2003.

114. E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS, EDS., *The Traveling Salesman Problem*, John Wiley & Sons, Essex, U.K., 1990.

115. R.M.R. LEWIS, *A Guide to Graph Coloring, Algorithms, and Applications*, Springer Verlag, 2016.

116. C.C. LINDNER AND C.A. RODGER, *Design Theory*, CRC Press, Boca Raton, Florida, 1997.

117. LOBACHEVSKY, *Geometric Researches on the Theory of Parallels*, George B. Halstead, translator, Open Court Publishing, Chicago, 1914.

118. G. LOPEZ, Deux resultats concernant la determination dune relation par les types d'isomorphie de ses restrictions, *C.R.A.S. Srie A* 274 (1972) pp. 1525-1528.

119. L. LOVÁSZ, Three short proofs in graph theory, *Journal of Combinatorial Theory (B)* 19 (1975), pp. 111-113.

120. L. LOVÁSZ AND M.D. PLUMMER, *Matching Theory*, Elsevier Science, 1986.

121. WILHELM MAGNUS, *Non-Euclidean Tesselations and Their Groups*, Academic Press, New York, 1974.

122. D. McCarthy and R.G. Stanton, eds., *Selected Papers of W.T. Tutte*, Charles Babbage Research Centre, St. Pierre, Manitoba, 1979.

123. B.D. McKay, Isomorph-free exhaustive generation, *Journal of Algorithms* 26 (1998), pp. 306-324.

124. B.D. McKay and A. Piperno, *Practical graph isomorphism II*, J. Symbolic Computing 60 (2014), pp. 94-112.

125. B. Mohar, Projective planarity in linear time, *Journal of Algorithms* 15 (1993), pp. 482-502.

126. B. Mohar and C. Thomassen, *Graphs on Surfaces*, Johns Hopkins University Press, Baltimore, 2001.

127. J.W. Moon, *Topics on Tournaments*, Holt, Rinehart, and Winston, New York, 1968.

128. W. Myrvold, personal communication, 2004.

129. W. Myrvold and W. Kocay, Errors in graph embedding algorithms, *Journal of Computer and System Sciences* 77(2), (2011) pp. 430-438.

130. W. Myrvold and J. Roth, Simpler projective planar embedding, *Ars Combinatoria* 75, 2005.

131. C.St.J.A. Nash-Williams, The reconstruction problem, *Selected Topics in Graph Theory*, Ed. L.W. Beineke and R.J. Wilson, Academic Press, London, 1978.

132. G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.

133. O. Ore, *The Four-Colour Problem*, Academic Press, New York, 1967.

134. C.H. Papadimitriou and K.Steiglitz, *Combinatorial Optimization*, Dover Publications Inc., Mineola, New York, 1998.

135. Dan Pedoe, *Geometry, a Comprehensive Course*, Dover Publications Inc., Mineola, New York, 1988.

136. H. Poincaré, Théorie des groupes Fuchsien, *Acta Math.* 1 (1882), pp. 1-62.

137. H. Prüfer, Neuer Beweis eines Satzes über Permutationen, *Arch. Math. Phys.* 27 (1918), pp. 742-744.

138. P.W. Purdom, Jr. and C.A. Brown, *The Analysis of Algorithms*, Holt, Rinehart, Winston, New York, 1985.

139. S. Ramachandran, Graph reconstruction – some new developments, *AKCE J. Graphs. Combin.* 1 (2004), pp. 51-61.

140. A. Ramsay and R. Richtmyer, *Introduction to Hyperbolic Geometry*, Springer, New York, 1995.

141. R.C. Read, ed., *Graph Theory and Computing*, Academic Press, New York, 1972.

142. R.C. READ, The coding of various kinds of unlabeled trees, in [141].

143. R.C. READ, Graph theory algorithms, in *Graph Theory and its Applications*, ed. Bernard Harris, Academic Press, New York, 1970, pp. 51-78.

144. R.C. READ, A new method for drawing a planar graph given the cyclic order of the edges at each vertex, *Congressus Numerantium* 56 (1987), pp. 31-44.

145. R.C. READ AND W.T. TUTTE, Chromatic polynomials, in [13], pp. 15-42.

146. K.B. REID AND L.W. BEINEKE, Tournaments, in [11], pp. 83-102.

147. GERHARD RINGEL, Sellbstkomplementäre Graphen, *Arch. Math (Basel)* 14 (1963) pp. 354-358.

148. N. ROBERTSON AND P.D. SEYMOUR, Graph minors – a survey, in *Surveys in Combinatorics 1985*, Proceedings of the Tenth British Combinatorial Conference, (I. Anderson, ed.), London Math. Society Lecture Notes 103, Cambridge, 1985, pp. 153-171.

149. N. ROBERTSON, D.P. SANDERS, P. SEYMOUR, AND R. THOMAS, The four-colour theorem, *Journal of Combinatorial Theory (B)* 70 (1997), pp. 2-44.

150. FIONA ROSS AND WILLIAM T. ROSS, The Jordan curve theorem is non-trivial, *J. Mathematics and the Arts* 5:4 (2011), pp. 213-219.

151. JOSEPH J. ROTMAN, *The Theory of Groups, an Introduction*, Allyn and Bacon, Inc., Boston, 1973.

152. F. RUBIN, A search procedure for hamilton paths and circuits, *JACM* 21 (1974), pp. 576-580.

153. T.L. SAATY AND P.C. KAINEN, *The Four-Color Problem, Assaults and Conquest*, Dover Publications, New York, 1977.

154. P.D. SACHS, Über sellbstkomplementäre Graphen, *Publ. Math. Debrecen* 9 (1962) pp. 270-288.

155. JAMES H. SCHMERL AND WILLIAM T. TROTTER, *Critically indecomposable partially ordered sets, graphs, tournaments, and other binary relational structures*, *Discrctc Mathematics* 113 (1993), pp. 191-205.

156. A.J. SCHWENK AND R.J. WILSON, Eigenvalues of graphs, in [11], pp. 307-336.

157. R. SEDGEWICK, *Algorithms in C++*, Addison-Wesley Publishing Co., Boston, 1998.

158. ÁKOS SERESS, *Permutation Group Algorithms*, Cambridge University Press, New York, 2003.

159. P.D. SEYMOUR, Nowhere-zero 6-flows, *J. Comb. Theory B* 30 (1981) pp. 130-135.

160. S. STAHL, *The Poincaré Half Plane*, Jones and Bartlett Publishers, Boston, MA, 1993.

161. J. STILLWELL, *Classical Topology and Combinatorial Group Theory*, Springer-Verlag, New York, 1980.

162. J. STILLWELL, *Geometry of Surfaces*, Springer-Verlag, New York, 1992.

163. J. STILLWELL, *Mathematics and its History, Third Edition*, Springer-Verlag, New York, 2010.

164. J. STOCKMEYER, The falsity of the reconstruction conjecture for tournaments, *J. Graph Th.* 1 (1977), pp. 19-25.

165. H.A. TAHA, *Operations Research: An Introduction*, Prentice Hall, Englewood Cliffs, New Jersey, 2003.

166. R.E. TARJAN, Depth-first search and linear graph algorithms, *SIAM Journal of Computing* 1 (1972) pp. 146-160.

167. C. THOMASSEN, Kuratowski's Theorem, *Journal of Graph Theory* 5 (1981), pp. 225-241.

168. C. THOMASSEN, The graph genus problem is NP-complete, *Journal of Algorithms* 10 (1989), pp. 568-576.

169. C. THOMASSEN, Planarity and duality of finite and infinite graphs, *Journal of Combinatorial Theory (B)* 29 (1980), pp. 244-271.

170. C. THOMASSEN, Whitney's 2-switching problem, cycle spaces, and arc mappings of directed graphs, *J. Combin. Th. B* 46 (1989), pp. 257-291.

171. WILLIAM P. THURSTON, *Three Dimensional Geometry and Topology*, Princeton University Press, New Jersey, 1997.

172. R. TRUDEAU, *Introduction to Graph Theory*, Dover Publications, Mineola, 1994.

173. W.T.TUTTE, How to draw a graph, in [122], pp. 360-388.

174. W.T.TUTTE, A short proof of the factor theorem for finite graphs, in [122], pp. 169-175.

175. W.T.TUTTE, The factorization of linear graphs, in [122], pp. 89-97.

176. W.T.TUTTE, Chromials, in [57], pp. 361-377.

177. W.T.TUTTE, *Connectivity in Graphs*, University of Toronto Press, 1966.

178. W.T.TUTTE, *Graph Theory*, *Encyclopedia of Mathematics and its Applications, vol. 21*, Addison Wesley, 1984.

179. W.T.TUTTE, A contribution to the theory of chromatic polynomials, *Canadian J. Math.* 6 (1954), pp. 80-91.

180. W.T.TUTTE, On the algebraic theory of graph colorings, *J. Comb. Theory* 1 (1966), pp. 15-50.

181. W.T.TUTTE, Unsolved problem 48, in [23].

182. S.M. ULAM, *A Collection of Mathematical Problems*, Wiley, New York, 1960.

183. J.H. VAN LINT AND R.M. WILSON, *A Course in Combinatorics*, Cambridge University Press, Cambridge, 1992.

184. V.G. VIZING, Critical graphs with given chromatic index, *Metody Diskret. Analiz.* 5 (1965), pp. 9-17.

185. R.C. WALKER, *Introduction to Mathematical Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1999.

186. WALTER WALLIS, *A Beginner's Guide to Graph Theory*, Birkhaüser Boston, Boston, MA, 2007.

187. MARK E. WATKINS, On the action of non-Abelian groups on graphs, *J. Combinatorial Th. (B)* 11 (1971), pp. 95-104.

188. M.A. WEISS, *Data Structures and Algorithm Analysis*, Benjamin Cummings Publishing Co., Redwood City, California, 1992.

189. D.B. WEST, *Graph Theory*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

190. A.T. WHITE, The Proof of the Heawood Conjecture, in [11].

191. H. WHITNEY, 2-Isomorphic Graphs, *American Journal of Mathematics* 55 (1933), pp. 245-254.

192. S.G. WILLIAMSON, Embedding graphs in the plane algorithmic aspects, *Annals of Discrete Mathematics* 6 (1980), pp. 349-384.

193. R.J. WILSON, *Four Colours Suffice*, Penguin Books, London, 2002.

194. D.R. WOODALL AND R.J. WILSON, The Appel-Haken proof of the four-colour theorem, in [11], pp. 83-102.

195. D.H. YOUNGER, Integer Flows, *J. Graph Theory* 7 (1983), pp. 349-357.

196. G.M. ZIEGLER, *Lectures on Polytopes*, Springer-Verlag, New York, 1998.

Index