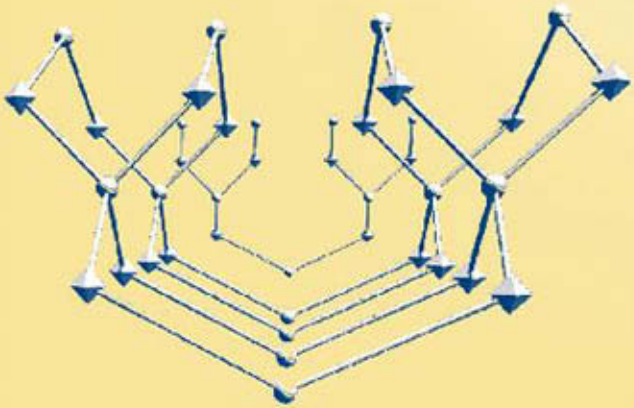
GRAPH ALGORITHMS

2ND EDITION



SHIMON EVEN
EDITED BY GUY EVEN

FOREWORD BY RICHARD M. KARP

CAMBRIDGE

Graph Algorithms, 2nd Edition

Shimon Even's *Graph Algorithms*, published in 1979, was a seminal introductory book on algorithms read by everyone engaged in the field. This thoroughly revised second edition, with a foreword by Richard M. Karp and notes by Andrew V. Goldberg, continues the exceptional presentation from the first edition and explains algorithms in formal but simple language with a direct and intuitive presentation.

The material covered by the book begins with basic material, including graphs and shortest paths, trees, depth-first search, and breadth-first search. The main part of the book is devoted to network flows and applications of network flows. The book ends with two chapters on planar graphs and on testing graph planarity.

SHIMON EVEN (1935–2004) was a pioneering researcher on graph algorithms and cryptography. He was a highly influential educator who played a major role in establishing computer science education in Israel at the Weizmann Institute and the Technion. He served as a source of professional inspiration and as a role model for generations of students and researchers. He is the author of *Algorithmic Combinatorics* (1973) and *Graph Algorithms* (1979).

Graph Algorithms

2nd Edition

SHIMON EVEN

Edited by

GUY EVEN
Tel-Aviv University

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Tokyo, Mexico City

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521736534

First edition published 1979 by Computer Science Press
Second edition published 2012

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data

ISBN 978-0-521-51718-8 Hardback
ISBN 978-0-521-73653-4 Paperback

Contents

Foreword

In Appreciation of Shimon Even

Shimon was a great computer scientist who inspired generations of Israeli stutents and young researchers, including many future leaders of theoretical computer science.

He was a master at creating combinatorial algorithms, constructions, and proofs. He always sought the simplest and most lucid solutions. Because he never allowed himself to use a known theorem unless he understood its proof, his discoveries were often based on original methods. His lectures were legendary for their clarity.

Shimon was devoted to his family, generous to his colleagues, and freely available to the students in his classes.

He expressed his views forcefully and with complete honesty. He expected honesty in return, and reserved his disapproval for those who tried to obfuscate or mislead.

Shimon had an unending supply of interesting anecdotes, and would laugh uproariously at good jokes, including his own.

In sum, he was a great and unforgettable man and a great scientist, and his name has a permanent place in the annals of theoretical computer science.

Richard M. Karp
Berkeley, April 2011

Preface to the Second Edition

My father, Shimon Even, died on May 1, 2004. In the year prior to his illness, he began revising this book. He used to tell me with great satisfaction whenever he completed the revision of a chapter. To his surprise, he often discovered that, after twenty-five years, he preferred to present the material differently (the first edition was published in 1979). Unfortunately, he only managed to revise Chapters 1, 2, 3, and 5. These revised chapters appear in this edition. However, since the material in Chapters 9 and 10 on NP-completeness is well covered in a few other books, we decided to omit these chapters from the second edition. Therefore, the second edition contains only the first eight chapters.

As I was reading the manuscript for the second edition, my father's deep voice resonated clearly in my mind. Not only his voice, but also his passion for teaching, for elegant explanations, and, most importantly, for distilling the essence. As an exceptional teacher, he used his voice and his physique to reinforce his arguments. His smile revealed how happy he was to have the opportunity to tell newcomers about this wonderful topic. One cannot overvalue the power of such enthusiasm. Luckily, this enthusiasm is conveyed in this book.

Many people tell me (with a smile) about being introduced to the topic of algorithms through this book. I believe the source of their smiles is its outstanding balance between clarity and preciseness. When one writes mathematical text, it is very easy to get carried away with the desire to be precise. The written letter is long lasting, and being aware that one's text leaves certain gaps requires boldness. For my father this task was trivial. The audience he had in mind consisted simply of himself. He wrote as he would have wanted the material to be presented to him. This meant that he elaborated where he needed to, and he did not hesitate to skim over details he felt comfortable with. As a child, I recall seeing him prepare for class by reading a chapter from his book. I asked him:

"Why are you reading your *own* book? Presumably, you know what is there." "True," he replied, "but I don't remember!"

This second edition would have never been completed without Oded Goldreich. Oded Goldreich began by convincing me to prepare the second edition. Then he put me in touch with Lauren Cowles from Cambridge University Press. Finally, he continuously encouraged me to complete this project. It took almost seven years! There is no good excuse for it. We all know how such a task can be pushed aside by more "urgent" and "demanding" tasks. Apparently, it took me some time to realize how important this task was, and that it could not be completed without a coordinated effort. Only after I recruited Lotem Kaplan to do the typesetting of the unrevised chapters and complete the missing figure and index terms did this project begin to progress seriously. I am truly grateful to Oded for his insistence, to Lotem for her assistance, and to Lauren for her kind patience.

Finally, I wish to thank Richard M. Karp, an old friend of my father's, for his foreword. I also wish to thank Andrew Goldberg, the expert in network flow algorithms, for the notes he contributed in Chapter 5. These notes outline the major developments in the algorithms for maximum flow that have taken place since the first edition of this book was published.

Guy Even
Tel-Aviv, March 2011

Preface to the First Edition

Graph theory has long been recognized as one of the more useful mathematical subjects for the computer science student to master. The approach that is natural to computer science is the algorithmic one; our interest is not so much in the existence proofs or enumeration techniques as it is in finding efficient algorithms for solving relevant problems or, alternatively, in showing evidence that no such algorithm exists. Although algorithmic graph theory was started by Euler, if not earlier, its development in the last ten years has been dramatic and revolutionary. Much of the material in Chapters 3, 5, 6, 8, 9, and 10 is less than ten years old.

This book is meant to be a textbook for an upper-level undergraduate, or a graduate course. It is the result of my experience in teaching such a course numerous times, since 1967, at Harvard, the Weizmann Institute of Science, Tel-Aviv University, the University of California at Berkeley, and the Technion. There is more than enough material for a one-semester course; I am sure that most teachers will have to omit parts of the book. If the course is for undergraduates, Chapters 1 to 5 provide enough material, and even then, the teacher may choose to omit a few sections, such as 2.6, 2.7, 3.3, and 3.4.[1] Chapter 7 consists of classical nonalgorithmic studies of planar graphs, which are necessary in order to understand the tests of planarity, described in Chapter 8; it may be assigned as a preparatory reading assignment. The mathematical background needed for understanding Chapters 1 to 8 includes some knowledge of set theory, combinatorics, and algebra, which the computer science student usually masters during his freshman year through courses on discrete mathematics and on linear algebra. However, the student also needs to know something about data structures and programming techniques, or he may not

The first edition was published in 1979 (G.E.).

[1] Sections 2.6 and 2.7 were removed from the second edition by Shimon Even.

appreciate the algorithmic side or may miss the complexity considerations. It is my experience that after two courses in programming, students have the necessary knowledge. However, in order to follow Chapters 9 and 10,[2] additional background is necessary, namely, in theory of computation. Specifically, the student should know about Turing machines and Church's thesis.

The book is self-contained. No previous knowledge is needed beyond the general background just described. No comments such as "the rest of the proof is left to the reader" or "this is beyond the scope of this book" are ever made. Some unproved results are mentioned, with a reference, but are not used later in the book.

At the end of each chapter, there are a few problems teachers can use for homework assignments. The teacher is advised to use them discriminately, since some of them may be too hard for his students.

I would like to thank some of my past colleagues for our joint work and for the influence they have had on my work, and therefore on this book: I. Cederbaum, M. R. Garey, J. E. Hopcroft, R. M. Karp, A. Lempel, A. Pnuely, A. Shamir, and R. E. Tarjan. Also, I would like to thank some of my former Ph.D. students for all that I have learned from them: O. Kariv, A. Itai, Y. Perl, M. Rodeh, and Y. Shiloach. Finally, I would like to thank E. Horowitz for his continuing encouragement.

S.E., Techinion, Haifa, Israel

[2] Chapters 9 and 10 are not included in the second edition.

1
Paths in Graphs

1.1 Introduction to Graph Theory

A *graph* $G(V,E)$ is a structure consisting of a set of *vertices* $V=\{v_1,v_2,\ldots\}$ and a set of *edges* $E=\{e_1,e_2,\ldots\}$; each edge e has two *endpoints*, which are vertices, and they are not necessarily distinct.

Unless otherwise stated, both V and E are assumed to be finite. In this case we say that G is finite.

For example, consider the graph in Figure 1.1. Here, $V=\{v_1,v_2,v_3,v_4,v_5\}$, $E=\{e_1,e_2,e_3,e_4,e_5\}$. The endpoints of e_2 are v_1 and v_2. Alternatively, we say that e_2 is *incident* on v_1 and v_2. The edges e_4 and e_5 have the same endpoints and are therefore called *parallel*. Both endpoints of e_1 are the same – v_1; such an edge is called a *self-loop*.

The *degree* of a vertex v, $d(v)$, is the number of times v is used as an endpoint. Clearly, a self-loop uses its endpoint twice. Thus, in our example, $d(v_1)=4$, $d(v_2)=3$. Also, a vertex whose degree is zero is called *isolated*. In our example, v_3 is isolated since $d(v_3)=0$.

Lemma 1.1 *In a finite graph the number of vertices of odd degree is even.*

Proof: Let $|V|$ and $|E|$ be the number of vertices and edges, respectively. It is easy to see that

$$\sum_{i=1}^{|V|} d(v_i) = 2\cdot|E|,$$

since each edge contributes two to the left-hand side: one to the degree of each of its two endpoints if they are distinct; and two to the degree of its endpoint if the edge is a self-loop. For the left-hand side to sum up to an even number, the number of odd terms must be even. ■

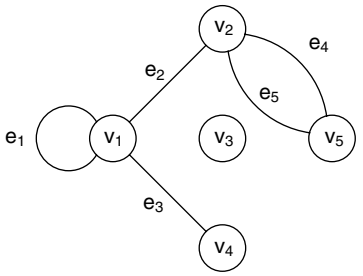


Figure 1.1: Example of a graph.

The notation $u \overset{e}{\text{---}} v$ means that the edge e is incident on vertices u and v. In this case we also say that e *connects* vertices u and v, or that vertices u and v are *adjacent*.

A *path*, P, is a sequence of vertices and edges, interweaved in the following way: P starts with a vertex, say v_0, followed by an edge e_1 incident to v_0, followed by the other endpoint v_1 of e_1, and so on. We write

$$P\colon\ v_0 \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \cdots$$

If P is finite, it ends with a vertex, say v_l. We call v_0 the *start*-vertex of P and v_l the *end*-vertex of P. The number of edge appearances in P, l, is called the *length* of P. If $l = 0$, then P is said to be *empty*, but it has a start-vertex and an end-vertex, which are identical. (We shall not use the term "path" unless a start-vertex exists.)

In a path, edges and vertices may appear more than once, unless otherwise stated. If no vertex appears more than once, and therefore no edge can appear more than once, the path is called *simple*.

A *circuit*, C, is a finite path in which the start and end vertices are identical. However, an empty path is not considered a circuit. By definition, the start and end vertices of a circuit are the same, and if there is no other repetition of a vertex, the circuit is called *simple*. However, a circuit of length two, $a \overset{e}{\text{---}} b \overset{e}{\text{---}} a$, where the same edge, e, appears twice, is not considered simple. (For a longer circuit, it is superfluous to state that if it is simple, then no edge appears more than once.) A self-loop is a simple circuit of length one.

If for every two vertices u and v of a graph G, there is a (finite) path that starts in u and ends in v, then G is said to be *connected*.

A *digraph* or *directed graph* $G(V, E)$ is defined similarly to a graph, except that the pair of endpoints of every edge is now ordered. If the ordered pair of

endpoints of a (directed) edge e is (u,v), we write

$$u \xrightarrow{e} v.$$

The vertex u is called the *start-vertex* of e; and the vertex, v, the *end-vertex* of e. The edge e is said to be directed from u to v. Edges with the same start-vertex and the same end-vertex are called *parallel*. If $u \neq v$, $u \xrightarrow{e_1} v$ and $v \xrightarrow{e_2} u$, then e_1 and e_2 are *antiparallel*. An edge $u \longrightarrow u$ is called a *self-loop*.

The *out-degree* $d_{out}(v)$ of vertex v is the number of (directed) edges having v as their start-vertex; *in-degree* $d_{in}(v)$ is similarly defined. Clearly, for every finite digraph $G(V,E)$,

$$\sum_{v \in V} d_{in}(v) = \sum_{v \in V} d_{out}(v).$$

A *directed path* is similar to a path in an undirected graph; if the sequence of edges is $e_1, e_2, \cdots$ then for every $i \geqslant 1$, the end-vertex of e_i is the start-vertex of e_{i+1}. The directed path is *simple* if no vertex appears on it more than once. A finite directed path C is a *directed circuit* if the start-vertex and end-vertex of C are the same. If C consists of one edge, it is a *self-loop*. As stated, the start and end vertices of C are identical, but if there is no other repetition of a vertex, C is *simple*. A digraph is said to be *strongly connected* if, for every ordered pair of vertices (u,v) there is a directed path which starts at u and ends in v.

1.2 Computer Representation of Graphs

To understand the time and space complexities of graph algorithms one needs to know how graphs are represented in the computers memory. In this section two of the most common methods of graph representation are briefly described.

Let us consider graphs and digraphs that have no parallel edges. For such graphs, the specification of the two endpoints is sufficient to specify the edge; for digraphs, the specification of the start-vertex and the end-vertex is sufficient. Thus, we can represent such a graph or digraph of n vertices by an $n \times n$ matrix M, where $M_{ij} = 1$ if there is an edge connecting vertex v_i to v_j, and $M_{ij} = 0$, if not. Clearly, in the case of (undirected) graphs, $M_{ij} = 1$ implies that $M_{ji} = 1$; or in other words, M is symmetric. But in the case of digraphs, any $n \times n$ matrix of zeros and ones is possible. This matrix is called the *adjacency matrix*.

Given the adjacency matrix M of a graph, one can compute $d(v_i)$ by counting the number of ones in the i-th row, except that a one on the main diagonal represents a self-loop and contributes two to the count. For a digraph, the number

of ones in the i-th row is equal to $d_{out}(v_i)$, and the number of ones in the j-th column is equal to $d_{in}(v_j)$.

The adjacency matrix is not an efficient representation of the graph if the graph is *sparse*; namely, the number of edges is significantly smaller than n^2. In these cases, it is more efficient to use the *incidence lists* representation, described later. We use this representation, which also allows parallel edges, in this book unless stated otherwise.

A *vertex array* is used. For each vertex v, it lists v's incident edges and a pointer indicating the *current* edge. The *incidence list* may simply be an array or may be a linked list. Initially, the pointer points to the first edge on the list. Also, we use an *edge array*, which tells us for each edge its two endpoints (or start-vertex and end-vertex, in the case of a digraph).

Assume we want an algorithm TRACE(s, P), such that given a finite graph $G(V, E)$ and a start-vertex $s \in V$ traces a maximal path P that starts at s and does not use any edge more than once. Note that by "maximal" we do not mean that the resulting path, P, will be the longest possible; we only mean that P cannot be extended, that is, there are no unused incident edges at the end-vertex.

We can trace a path starting at s by taking the first edge e_1 on the incidence list of s, marking e_1 as "used" in the edge array, and looking up its other endpoint v_1 (which is s if e_1 is a self-loop). Next, use the vertex array to find the pointer to the current edge on the list of v_1. Scan the incidence list of v_1 for the first unused edge, take it, and so on. If the scanning hits the last edge and it is used, TRACE(s, P) halts. A PASCAL-like description of TRACE(s, P) is presented in Algorithm 1.1. Here is a list of the data structures it uses:

(i) A *vertex table* such that, for every $v \in V$, it includes the following:
  – A list of the edges incident on v, which ends with *NIL*
  – A pointer $N(v)$ to the current item in this list. Initially, $N(v)$ points to the first edge on the list (or to *NIL*, if the list is empty).

(ii) An *edge table* such that every $e \in E$ consists of the following:
  – The two endpoints of e
  – A flag that indicates whether e is *used* or *unused*. Initially, all edges are *unused*.

(iii) An array (or linked list) P of edges that is initially empty, and when TRACE(s, P) halts, will contain the resulting path.

Notice that in each application of the "while" loop of TRACE (lines 2–10 in Algorithm 1.1), either $N(v)$ is moved to the next item on the incidence list of v (line 4), or lines 6–10 are applied, but not both. The number of times line

```
Procedure TRACE(s, P)

1   v ← s
2   while N(v) points to an edge (and not to NIL) do
3       if N(v) points to a used edge do
4           change N(v) to point to the next item on the list
5       else do
6           e ← N(v)
7           change the flag of e to used
8           add e to the end of P
9           use the edge table to find the second endpoint of e, say u
10          v ← u
```

Algorithm 1.1: The TRACE algorithm.

4 is applied is clearly $O(|E|)$. The number of times lines 6–10 are applied is also $O(|E|)$, since the flag of an unused edge changes to used, and each of these lines takes time bounded by a constant to run. Thus, the time complexity of TRACE is $O(|E|)$.[1] (In fact, if the length of the resulting P is L then the time complexity is $O(l)$; this follows from the fact that each edge that joins P can "cause a waste" of computing time only twice: once when it joins P and, at most, once again by its appearance on the incidence list of the adjacent vertex.)

If one uses the adjacency matrix representation, in the worst case, the tracing algorithm takes time (and space) complexity $\Omega(|V|^2)$.[2] And if $|E| << |V|^2$, as is the case for *sparse* graphs, the complexity is reduced by using the incidence-list representation. Since in most applications, the graphs are sparse, we prefer to use the incidence-list representation.

Note that in our discussions of complexity, we assume that the word length of our computer is sufficient to store the names of our atomic components: vertices and edges. If one does not make this assumption, then one may have to allow $\Omega(\log(|E|+|V|))$ bits to represent the atomic components, and to multiply the complexities by this factor.

[1] $f(x)$ is $O(g(x))$ if there are two constants k_1 and k_2, such that for every x, $f(x) \leqslant k_1 \cdot g(x) + k_2$.

[2] $f(x)$ is $\Omega(g(x))$ if there are two constants k_3 and k_4, such that for every x, $f(x) \leqslant k_3 \cdot g(x) + k_4$.

1.3 Euler Graphs

An *Euler path* of a finite undirected graph $G(V, E)$ is a path such that every edge of G appears on it once. Therefore, the length of an Euler path is $|E|$. If G has an Euler path, then it is called an *Euler graph*.

Theorem 1.1 *A finite (undirected) connected graph is an Euler graph if and only if exactly two vertices are of odd degree or all vertices are of even degree. In the latter case, every Euler path of the graph is a circuit, and in the former case, none is.*

As an immediate conclusion of Theorem 1.1 we observe that none of the graphs in Figure 1.2 is an Euler graph because both have four vertices of odd degree. The graph shown in Figure 1.2(a) is the famous Königsberg bridge problem solved by Euler in 1736. The graph shown in Figure 1.2(b) is a common misleading puzzle of the type "draw without lifting your pen from the paper."

Proof: It is clear that if a graph has an Euler path that is not a circuit, then the start-vertex and the end-vertex of the path are of odd degree, while all the other vertices are of even degree. Also, if a graph has an Euler circuit, then all vertices are of even degree.

Assume now that G is a finite graph with exactly two vertices of odd degree, a and b. We now describe an algorithm (A), which will find an Euler path from a to b.

First, trace a maximal path P, as in the previous section, starting at vertex a. Since G is finite, the algorithm halts, producing a path. But as soon as the path emanates from a, one of the edges incident to a is used, and a's residual degree becomes even. Thus, every time a is reentered, there is an unused edge to leave

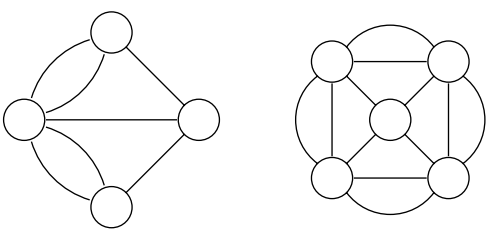


Figure 1.2: Non-Eulerian graphs.

a by. This proves that P cannot end in a. Similarly, if vertex $v \in V \setminus \{a, b\}$, then P cannot end in v. It follows that P ends in b.

If P contains all the edges of G, we are done. If not, we make the following observations:

- The residual degree of every vertex is even.
- There is an unused edge incident on some vertex v that is on P. To see that this must be so, let $u \overset{e}{\text{---}} w$ be an unused edge. If either u or w is on P, we are done. If not, since G is connected, there is a path Q from a to u. There must be unused edges on Q. Going from a on Q, the first unused edge we encounter fits the bill.

Now, trace a maximal path P' in the residual graph, which consists of the set V of vertices and all edges of E that are not in P. Start P' at v. Since all vertices of the residual graph are of even degree, P' ends in v (and is therefore a circuit). Next, combine P and P' to form one path from a to b as follows: Follow P until it enters v. Now, incorporate P', and then follow the remainder of P.

Repeat, incorporating additional circuits into the present path as long as there are unused edges. Since the graph is finite, this process will terminate, yielding an Euler path.

If all vertices of the graph are of even degree, the first traced path can start at any vertex, and will be a circuit. The remainder of the algorithm is similar to to this process. ■

In the case of digraphs, a *directed Euler path* is a directed path in which every edge appears once. A *directed Euler circuit* is a directed Euler path for which the start and end vertices are identical. In addition, digraph is called *Euler* if it has a directed Euler path (or circuit).

The *underlying (undirected) graph* of a digraph is the graph resulting from the digraph if the direction of the edges is ignored. Thus, the underlying graph of the digraph in Figure 1.3(a) is shown in Figure 1.3(b).

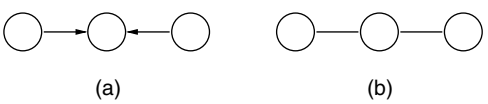


Figure 1.3: A digraph and its underlying graph.

Theorem 1.2 *A finite digraph is an Euler digraph if and only if its underlying graph is connected and one of the following two conditions holds:*

(i) There is one vertex a *such that* $d_{out}(a) = d_{in}(a)+1$*, and another vertex* b *such that* $d_{out}(b)+1 = d_{in}(b)$*, while for every other vertex* v*,* $d_{out}(v) = d_{in}(v)$.
(ii) For every vertex v*,* $d_{out}(v) = d_{in}(v)$.

In the former case, every directed Euler path starts at a *and ends in* b*. In the latter, every directed Euler path is a directed Euler circuit.*

The proof of Theorem 1.2 is along the same lines as the proof of Theorem 1.1, and is therefore not repeated here.

Let us make now a few comments about the complexity of the algorithm $\mathcal{A}$ for finding an Euler path, as described in the proof of Theorem 1.1. Our purpose is to show that the time complexity of the algorithm is $O(|E|)$.

Assume $G(V,E)$ is presented in the incidence list's data structure. The main path P and the detour P' will be represented by linked lists, where each item on the list represents an edge.

In the vertex table, we add for each vertex v the following two items:

(i) A flag that indicates whether v is already on the main path P or the detour P'. Initially, this flag is "unvisited."
(ii) For every visited vertex v, there is a pointer $E(v)$ to the location on the path of the edge through which v was first encountered. Initially, for every v, $E(v) =$*NIL*.

We shall also use a list L of visited vertices. Each vertex enters L once, when its flag is changed from "unvisited" to "visited."

$\mathcal{A}$ starts by running TRACE(a,P), updating the vertices' flags, and $E(v)$ for each newly visited vertex v. Next, the following loop is applied:

If L is empty, $\mathcal{A}$ halts. If not, take a vertex v from L, and remove v from L. Use TRACE(v,P') to produce P'. Look up edge $E(v)$, recording the location of the edge e it is linked to. Change this link to point to the first edge on P'. Now, let the last edge of P' point to e.

Note that when TRACE(v,P') terminates, v has no unused incident edges. This explains why we can remove v from L.

Now that P' has been incorporated into P, the loop is repeated.

It is not hard to see that both the time and space complexities of $\mathcal{A}$ are $O(|E|)$.

1.4 De Bruijn Sequences

Let $\Sigma = \{0, 1, ..., \sigma - 1\}$ be an alphabet of σ letters. Clearly, there are $L = \sigma^n$ different words of length n over Σ. A *de Bruijn sequence* is a (circular) sequence $a_0 a_1 \cdots a_{L-1}$ over Σ such that for every word w of length n over Σ there exists a (unique) $0 \leqslant j < L$ such that

$$a_j a_{j+1} \cdots a_{j+n-1} = w,$$

where the computation of the indexes is modulo L.

The most important case is that of $\sigma = 2$. Binary de Bruijn sequences are of great importance in coding theory and can be generated by shift registers. (See Golomb, 1967, on the subject.) In this section we discuss the existence of de Bruijn sequences for every σ and every n.

For that purpose let us define the *de Bruijn digraph* $G_{\sigma,n}(V, E)$ as follows:

(i) $V = \Sigma^{n-1}$; i.e., the set of all σ^{n-1} words of length $n-1$ over Σ.
(ii) $E = \Sigma^n$.
(iii) The directed edge $b_1 b_2 \cdots b_n$ starts at vertex $b_1 b_2 \cdots b_{n-1}$ and ends in vertex $b_2 b_3 \cdots b_n$.

Digraphs $G_{2,3}$ and $G_{2,4}$ are shown in Figure 1.4. $G_{3,2}$ is shown in Figure 1.5.

Observe that if $w_1, w_2 \in \Sigma^n$, then w_2 can follow w_1 in a de Bruijn sequence only if in $G_{\sigma,n}$ the edge w_2 starts at the vertex in which w_1 ends. It follows that there is a de Bruijn sequence for σ and n if and only if there is a directed Euler circuit in $G_{\sigma,n}$.

For example, consider the directed Euler circuit of $G_{2,3}$, which consists of the following sequence of directed edges:

000, 001, 011, 111, 110, 101, 010, 100.

The corresponding de Bruijn sequence, 00011101, follows by reading the first letter of each word (edge) in the circuit.

Theorem 1.3 *For every σ and n, $G_{\sigma,n}$ has a directed Euler circuit.*

Proof: To use Theorem 1.2 we have to show that the underlying undirected graph of $G_{\sigma,n}$ is connected and that for every vertex v, $d_{out}(v) = d_{in}(v)$.

Let us show that $G_{\sigma,n}$ is strongly connected. This implies that its underlying undirected graph is connected.

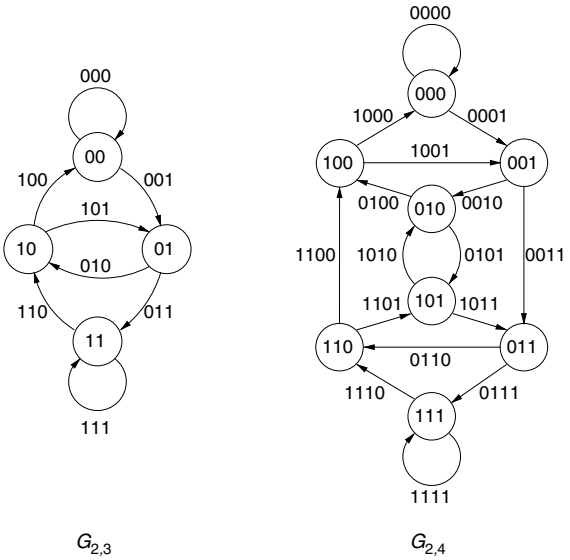


Figure 1.4: Examples of de Bruijn digraphs for $\sigma = 2$.

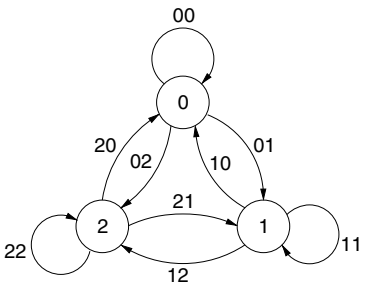


Figure 1.5: $G_{3,2}$

Let $b_1 b_2 \cdots b_{n-1}$ and $c_1 c_2 \cdots c_{n-1}$ be any two vertices. The directed path

$$b_1 b_2 \cdots b_{n-1} c_1,\ b_2 b_3 \cdots b_{n-1} c_1 c_2,\ \ldots, b_{n-1} c_1 \cdots c_{n-1}$$

is of length $n-1$, it starts at vertex $b_1 b_2 \cdots b_{n-1}$ and ends in vertex $c_1 c_2 \cdots c_{n-1}$, showing that $G_{\sigma,n}$ is strongly connected. (Observe that this directed path is not necessarily simple; it may use vertices and edges more than once.)

Now, observe that for each vertex $v = b_1 b_2 \cdots b_{n-1}$, every outgoing edge is of the form $b_1 b_2 \cdots b_{n-1} c$, where c can be any of the σ letters. Thus, $d_{out}(v) = \sigma$.

A similar statement holds for $d_{in}(v)$. Thus, the condition that $d_{out}(v) = d_{in}(v)$ holds, uniformly. ■

Corollary 1.1 *For every σ and n there exists a de Bruijn sequence.*

1.5 Shortest-Path Algorithms

In general, the shortest-path problem is concerned with finding shortest paths between vertices. Many interesting problems arise, and their variety depends on the type of graph in our application and the exact question we want to answer. Some of the characteristics that may help in defining the exact problem are as follows:

(i) The graph is finite or infinite.
(ii) The graph is undirected or directed.
(iii) The edges are all of length one, or all lengths are nonnegative, or negative lengths are allowed.
(iv) We may be interested in shortest paths from a given vertex to another, or from a given vertex to all the other vertices, or from each vertex to all the other vertices.
(v) We may be interested in finding just one path, or in all paths, or in counting the number of shortest paths.

Clearly, this section will deal only with a few of all possible problems. We attempt to describe the most important techniques.

It is assumed that we are given a graph (or digraph) $G(V, E)$ and a *length* function $l : E \mapsto R$. Namely, if e is an edge, then $l(e)$ is the length of e. The length of a path is the sum of the lengths of edges on it. The *distance* from vertex u to vertex v, $d(u, v)$, is the length of a shortest path from u to v if there are paths from u to v, and is infinite if no such path exists.

In most algorithms described in this section, the scenario is as follows: We assume that there is a designated vertex $s \in V$, called the *source*. We denote by $\delta(v)$ the value $d(s, v)$. The algorithm will assign a label $\lambda(v)$ to some (or all) vertices v. Thus prove that when the algorithm halts, $\lambda(v) = \delta(v)$.

1.5.1 Breadth-First Search

Let us start with the case that G is finite and undirected, $l(e) = 1$ for every edge e, and $s \in V$ is a designated source. Our goal is to compute $\delta(v)$ for every vertex v.

```
Procedure BFS(G, s;λ)

1   empty Q and put s in Q
2   λ(s) ← 0
3   T ← {s}
4   while Q ≠ ∅ do
5       remove the first vertex, u, from Q
6       for every edge u —— v, if v ∉ T, do
7           T ← T ∪ {v}
8           λ(v) ← λ(u) + 1
9           put v in Q
```

Algorithm 1.2: The BFS algorithm.

The natural and simple algorithm that follows was first suggested by Moore 1957, and was later called *Breadth-First Search*, or BFS. Intuitively, all it does is the following:

We start by assigning $\lambda(v) \leftarrow \infty$ for every $v \in V$. We then proceed in *waves*. In wave $i \geqslant 0$ a set $W(i)$ of vertices is assigned a finite label $\lambda(v) \leftarrow i$, and no vertex with a finite label will ever be relabeled. In wave 0, $\lambda(s) \leftarrow 0$, and therefore $W(0) = \{s\}$. As long as $W(i) \neq \emptyset$, i is incremented, for every edge u —— v, such that $\lambda(u) = i-1$ and $\lambda(v) = \infty$, assign $\lambda(v) \leftarrow i$ and put $v \in W(i)$.

In the Pascal-like algorithm, described in Algorithm 1.2, we use a slightly different presentation; our reasons for doing so are discussed later. In this presentation we do not use the sets of waves, nor is there a running index i. Instead, we use a queue Q of vertices and a set T of vertices. The first vertex to be removed from Q is s, and for every edge incident on s, the adjacent vertex, if "new,"[3] is labeled 1, joins T, and is put on Q. (These are the vertices of $W(1)$.) Next, vertices (that would have been in $W(1)$) are removed from Q, and their new neighbors[4] are labeled 2, join T, are put in Q, and so on.

Let us say that a vertex v is *accessible* from u if there is a path from u to v.

Theorem 1.4 *Algorithm BFS assigns every vertex v, which is accessible from s, a label $\lambda(v)$ and $\lambda(v) = \delta(v)$.*

[3] Not in T.
[4] Adjacent vertices.

Proof: First, let us prove that if v is accessible from s, then it gets a (finite) label $\lambda(v)$, and $\lambda(v) \leqslant \delta(v)$. This is proved by induction on the value of $\delta(v)$. The basis is established by Line 2, where s is labeled 0.

Now, assume the claim holds for vertices whose distance from s is less than i, and let us prove it for vertex v for which $\delta(v) = i$.

There is a path P from s to v of length i. The vertex u that precedes v on P, satisfies $\delta(u) = i - 1$.[5] Thus, by the inductive hypothesis, u is labeled, and $\lambda(u) \leqslant i - 1$. When u is removed from Q, the edge between u and v on P is examined. If at that time v is not yet labeled, it gets a label $\lambda(v) \leqslant i$, proving the claim. If v is already labeled, by the fact that waves exit Q according to nondecreasing order, $\lambda(v)$ cannot be higher than i.

It is easy to see that if a vertex v gets a label $\lambda(v)$, then there is a path of length $\lambda(v)$ from s to v. Such a path can be traced back from v to s by the edges through which vertices have been labeled. This proves that $\lambda(v) \geqslant \delta(v)$. Putting the two inequalities together, the theorem follows. ■

The foregoing discussion holds for digraphs as well. The only change one has to make in Algorithm 1.2 is to replace "u —— v" by "$u \longrightarrow v$."

In the case of infinite graphs (digraphs), one can still use a modification of BFS to solve the problem, provided the following conditions hold:

(i) Since it is impossible to store the entire input graph, there should be an algorithm that provides the data of the vertex table of a specified vertex, when such is requested. Clearly, the degrees (out-degrees) of the vertices must be finite.

(ii) There is a designated-target vertex t, which is accessible from s, and all we want to do is to find $\delta(t)$ (and $\delta(v)$ for every v, such that $\delta(v) < \delta(t)$).

All one needs to change in Algorithm 1.2 is Line 4, as follows: "*while* $Q \neq \emptyset$ *and* $t \notin T$ *do*." Note that the description of BFS as in Algorithm 1.2 does not require the prelabeling of all vertices by ∞, which is an impossible task if G has infinitely many vertices.

Let us discuss the time complexity of BFS for finite graphs (digraphs). We can represent T by an array of length $|V|$ in which the i'th position is 0 if $v_i \notin T$ and one if $v_i \in T$. Looking up whether $v \in T$ or changing the i'th position takes constant time, but the performance of Line 3 takes $\Omega(|V|)$ time. In addition, note that each edge is examined at most twice once from each of its endpoints,

[5] It is easy to prove that every subpath of a shortest path is itself a shortest path between its two ends. This is sometimes referred to as the "principle of dynamic programming."

and when the edge is examined it may cause a computation that takes constant time. Thus, the time complexity of BFS is $O(|V|+|E|)$.

Finally, let us comment on how one can use the data generated by BFS to trace a shortest path from s to a vertex $v \in T$. This can be done as described in the last paragraph of the proof of Theorem 1.4. To find the edge by which a vertex has been labeled, one can add to the vertex table an item that carries this information; initially, the value of this item is *NIL* for every vertex, and when a vertex joins T, the item is updated. This shows that the time complexity remains essentially unchanged.

1.5.2 Dijkstra's Algorithm

In this subsection, we shall first assume that we are given a finite digraph $G(V,E)$; there is a source vertex s, and every (directed) edge e has a nonnegative length $l(e) \geqslant 0$. Our task is to compute $\delta(v)$ for every vertex v. Later, we shall discuss other cases in which similar algorithms apply.

If all edge lengths are positive integers, it may seem that by replacing each edge e by a directed path that goes through $l(e)$ new edges, all of length 1, and $l(e)-1$ new (intermediate) vertices, the problem is mapped to the case described in the previous subsection, and BFS can be used to solve it. Although, this is a valid statement, from a computer-science point of view, it is a bad idea. The reason is that it takes $\log l(e)$ bits to represent $l(e)$, and the foregoing suggested transformation introduces $l(e)-1$ new vertices and edges. This blows up the length of the input data exponentially. The *Dijkstra Algorithm* (Dijkstra 1959, vol.1) avoids this blowup and keeps the complexity bounded polynomially in terms of the length of the input data.

The Dijkstra algorithm is presented in Algorithm 1.3 in Pascal-like style. Two sets of vertices are used: T is the set of temporarily labeled vertices; that is, vertices for which λ has been assigned but its value is still subject to change. P is the set of permanently labeled vertices. Vertices neither in T, nor in P, have not been labeled yet. A vertex v, for which $\lambda(v)$ is minimum, among the vertices in T, is chosen in Line 5. This vertex moves from T to P, and every edge e outgoing from v is examined. If the end-vertex u of e is in T, then its label is lowered in Line 10, if e enables such an improvement. If u is new, it gets a label in Line 12, and joins T in Line 13.

Lemma 1.2 *When procedure DIJKSTRA is applied to any finite digraph* G *and start-vertex* s*, it halts.*

```
Procedure DIJKSTRA(G, s, l; λ)

1   λ(s) ← 0
2   T ← {s}
3   P ← ∅
4   while T ≠ ∅ do
5       choose a vertex v ∈ T for which λ(v) is minimum
6       T ← T \ {v}
7       P ← P ∪ {v}
8       for every v --e--> u do
9           if u ∈ T then do
10              λ(u) ← min{λ(u), λ(v) + l(e)}
11          else, if u ∉ P then do
12              λ(u) ← λ(v) + l(e)
13              T ← T ∪ {u}
```

Algorithm 1.3: The Dijkstra algorithm.

Proof: Every vertex enters T at most once, and every vertex chosen in Line 5 leaves T. The performance of Line 5 takes at most $O(|V|)$ time, and the sum total time to perform Lines 8–13, for all chosen vertices, is $O(|E|)$, since no edge is examined more than once. After performing Line 5 $|V|$ times, T must be empty, and the procedure halts. ■

It follows that the time complexity of Dijkstra's algorithm is $O(|V|^2 + |E|)$. We shall return to this issue.

Lemma 1.3 *During the computation of Dijkstra's algorithm, every vertex accessible from* s *gets a label.*

Proof: Let $\mathcal{P}$ be a directed path from s to v. If v does not get a label, let u be the first unlabeled vertex along $\mathcal{P}$ when the algorithm halts. Consider the edge $w \xrightarrow{e} u$ on $\mathcal{P}$. Since w has been labeled, eventually, it is chosen to leave T, and by the edge e, u gets labeled. A contradiction. ■

Lemma 1.4 *At any time during the computation of Dijkstra's algorithm, if a vertex* v *is labeled* $\lambda(v)$*, then there is a directed path from* s *to* v *whose length is* $\lambda(v)$.

Proof: By induction on the time during which an assignment or reassignment of a label takes place. The first assignment is in Line 1, and indeed, there is a path of length 0 from s to itself, that is, the empty path.

Now, look at an assignment, or reassignment, that occurs at time τ, and assume all previous assignments satisfy the claim. Assume vertex u is assigned a label at time τ (for the first time) in Line 12. Thus, the label of v at time τ was assigned earlier. By the inductive hypothesis, there is a directed path from vertex s to v of length $\lambda(v)$, and this path, appended with e, is a path of length $\lambda(u)$ from s to u. A similar argument holds in case of a reassignment, per Line 10. ■

Note that Lemma 1.4 implies that for every labeled vertex v (and at any time),

$$\lambda(v) \geqslant \delta(v). \tag{1.1}$$

Theorem 1.5 *When Dijkstra's algorithm halts, for every vertex v accessible from* s*,*

$$\lambda(v) = \delta(v).$$

Proof: If v is accessible from s, then by Lemma 1.3, v has an assigned label $\lambda(v)$. By Equation 1.1, $\lambda(v) \geqslant \delta(v)$. It remains to show that $\lambda(v) \leqslant \delta(v)$.

The proof is by induction on the order in which vertices join the set P.

Let $\mathcal{P}$ be a shortest directed path from s to v. Let τ be the time when vertex v is chosen to join P. At that time, s is in P and v is not. Thus, there must be an edge $u \xrightarrow{e} w$ on $\mathcal{P}$ such that $u \in P$ and $w \notin P$. Consider the following sequence of claims:

- Since the length of edges is nonnegative, the subpath of $\mathcal{P}$, from w to v, is of a nonnegative length. Thus,

$$\delta(v) \geqslant \delta(w).$$

- Since every subpath of a shortest path is shortest from its start-vertex to its end-vertex, the subpath of $\mathcal{P}$ from s to w is shortest, and its length consists of the length of a shortest path from s to u plus $l(e)$, that is,

$$\delta(w) = \delta(u) + l(e).$$

- Since u has joined P before time τ, by the inductive hypothesis, $\delta(u) \geqslant \lambda(u)$. Thus,

$$\delta(u) + l(e) \geqslant \lambda(u) + l(e).$$

- Since u has joined P before time τ, all its outgoing edges, including e, have been examined, as per Lines 8–13. Thus, at time τ, $\lambda(w)$ has been assigned, and
$$\lambda(u) + l(e) \geqslant \lambda(w).$$
- However, at time τ, v has been chosen to join P, not w. Thus,
$$\lambda(w) \geqslant \lambda(v).$$

These foregoing five relations imply that, at time τ,

$$\delta(v) \geqslant \lambda(v).$$

■

As we have seen, a simple implementation of the Dijkstra algorithm is of time complexity $O(|V|^2 + |E|)$. For sparse graphs the bulk of the computation is in the $\Omega(|V|)$ applications of Line 5, where many of them may take $\Omega(|V|)$ time each. If one uses a heap to store the vertices of T (see, e.g., Cormen et al. [4]), then the complexity is reduced to $O(|E| \cdot \log|V|)$. If one uses Fibonacci heaps (see, e.g., [5] or [4]), then the complexity is reduced further to $O(|V| \cdot \log|V| + |E|)$.

In case of undirected graphs, the Dijkstra algorithm is applicable; the only change one should make is to replace "$v \xrightarrow{e} u$" by "$v \overset{e}{\text{———}} u$" in Line 8. Alternatively, transform the given graph to a digraph by replacing each undirected edge with a pair of antiparallel directed edges of the same length, and apply Dijkstra's algorithm as is.

If there is a target vertex t, and one does not want to continue the computation after t has been put in P, then one should replace Line 4 with "*while* $T \neq \emptyset$ *and* $t \notin P$ *do*." This change makes the algorithm applicable for infinite graphs, provided t is accessible from s, the out-degrees of the vertices are finite, and the number of vertices v, for which $\delta(v) \leqslant \delta(t)$, is finite.

Note that Dijkstra's algorithm may not be applicable if there are edges of negative length. This is the case even if the graph is directed and there are no directed circuits whose length is negative.

1.5.3 The Ford Algorithm

In this subsection we assume that the given digraph, $G(V, E)$, is finite, every (directed) edge e has a length $l(e)$, which may be negative. We are also given a source vertex s. Our task is to compute $\delta(v)$ for every vertex v.

Procedure gen-FORD$(G, s, l; \lambda)$

1 *for every* $v \in V$ *do*
2 $\quad \lambda(v) \leftarrow \infty$
3 $\lambda(s) \leftarrow 0$
4 *while* there is an edge $u \xrightarrow{e} v$ such that $\lambda(u)$ is finite and $\lambda(v) > \lambda(u) + l(e)$ *do*
5 $\quad \lambda(v) \leftarrow \lambda(u) + l(e)$

Algorithm 1.4: The generic Ford algorithm.

Let us call a directed circuit *negative* if its length is negative. Notice that if there is a negative circuit C, and it is accessible from s, then the distance from s to the vertices on C is not defined; for every real number r, one may take a path from s to one of C's vertices and go around C sufficiently many times, to build up a path of length less than r. But, if there are no negative circuits accessible from s, then either v is not accessible from s, and then $\delta(v) = \infty$, or only simple paths from s to v need to be considered. The number of such paths is finite, and therefore, $\delta(v)$ is well defined. Thus, we conclude that $\delta(\cdot)$ is well defined.

The generic Ford algorithm [6, 7],[6] described in Algorithm 1.4, computes for every vertex v, a value $\lambda(v)$. As we shall see, if there are no negative circuits accessible from s, the procedure will terminate, and upon termination, for every vertex v, $\lambda(v) = \delta(v)$.

Lemma 1.5 *While running gen-FORD, if* $\lambda(v)$ *is finite then there is a directed path from* s *to* v *whose length is* $\lambda(v)$.

The proof is similar to that of Lemma 1.4.

Lemma 1.5 holds even if there are negative circuits. However, if there are no such circuits, the path traced in the proof cannot return to a vertex visited earlier. For if it does, then by going around the directed circuit, a vertex has improved its own label. This implies that the sum of the edge lengths of the circuit is negative. Therefore, we have:

[6] Sometimes called the Bellman-Ford algorithm.

Lemma 1.6 *If the digraph has no accessible negative circuits and if, while running, gen-FORD $\lambda(v)$ is finite, then there is a simple directed path from s to v whose length is $\lambda(v)$.*

Under the conditions of Lemma 1.6, since each new assignment of $\lambda(\cdot)$ corresponds to a new simple directed path from s, and since the number of simple directed paths (from s) is finite, we conclude that under these conditions procedure gen-FORD must terminate, and is therefore, an algorithm.

Lemma 1.7 *For a digraph with no accessible negative circuits, upon termination of the Ford algorithm, $\lambda(v) = \delta(v)$ for every vertex v.*

Proof: If v is not accessible from s, then both $\lambda(v)$ and $\delta(v)$ are equal to ∞ and the claim holds.

If v is accessible from s, by Lemma 1.5, $\lambda(v) \geqslant \delta(v)$. It remains to be shown that $\lambda(v) \leqslant \delta(v)$.

Let P be a shortest path from s to v, where

$$P : s = u_0 \xrightarrow{e_1} u_1 \xrightarrow{e_2} u_2 \cdots u_{l-1} \xrightarrow{e_l} u_l = v.$$

We prove, by induction on $0 \leqslant i \leqslant l$, that $\lambda(u_i) \leqslant \delta(u_i)$. The claim clearly holds for u_0, by Line 3 and the fact that labels never increase.

Let i be the least value for which $\lambda(u_i) > \delta(u_i)$. Since every section of a shortest path is shortest between its two endpoints, the length of the subpath of P, from u_0 to u_i is $\delta(u_i)$ and $\delta(u_i) = \delta(u_{i-1}) + l(e_i)$. By the choice of i, $\lambda(u_{i-1}) \leqslant \delta(u_{i-1})$. It follows that

$$\lambda(u_i) > \delta(u_i) = \delta(u_{i-1}) + l(e_i) \geqslant \lambda(u_{i-1}) + l(e_i).$$

Thus, the algorithm should not have terminated. ■

In spite of the fact that gen-FORD is a valid algorithm, the lack of determinism in the choice of the order in which the edges are observed in Line 4 may be abused to cause the algorithm to take exponential time. (See Johnson [8].) There is a simple remedy: Order the edges of the digraph – any order will do – and perform Line 4 by scanning the edges in this order. Once the scan is complete, repeat it until there is no improvement in a complete scan. This procedure, adv-FORD, is described in Algorithm 1.5, where it is assumed that

$$E = \{e_1, e_2, \ldots, e_m\}.$$

```
Procedure adv-FORD(G(V,E),s,l;λ)

1   for every v ∈ V do
2       λ(v) ← ∞
3   λ(s) ← 0
4   repeat
5       Flag ← False
6       for every 1 ⩽ i ⩽ m do
7           let u --e_i--> v
8           if λ(u) is finite and λ(v) > λ(u) + l(e_i) then do
9               λ(v) ← λ(u) + l(e_i)
10              Flag ← True
11  until Flag = False
```

Algorithm 1.5: The advanced Ford algorithm.

Theorem 1.6 *If the digraph has no accessible negative circuits, then procedure adv-FORD terminates in* $O(|V| \cdot |E|)$ *time, and when it terminates,* $\lambda(v) = \delta(v)$ *for every vertex* v.

Proof: Let us prove by induction on k that for every vertex v, if there is a shortest path from s to v, which consists of k edges, then after the k-th application of the loop (Lines 4-11), or if procedure adv-FORD halts earlier, $\lambda(v) = \delta(v)$.

For $k = 0$, the only applicable vertex is s, and Line 3 establishes the claim. Assume now that the claim holds for $0 \leqslant k \leqslant j$ and show that it holds for $k = j + 1$.

If procedure adv-FORD terminates before the $j+1$-st application of the loop, the claim follows from Lemma 1.7. Let v be a vertex such that there is a shortest path, P, from s to v that consists of $j+1$ edges. (If there is also a shortest path from s to v which consists of less edges, then there is nothing to prove.) Let $u \xrightarrow{e} v$ be the last edge in P. Since the subpath of P from s to u is a shortest path to u, and since it consists of j edges, by the inductive hypothesis, after the j-th application of the loop, $\lambda(u) = \delta(u)$. In the $j+1$-st application of the loop, when e is checked, $\lambda(v)$ gets the value $\delta(v)$, if it has not had that value already. That proves the claim.

If v is accessible from s, and since there are no accessible negative circuits, a shortest path from s to v is simple and consists $|V| - 1$ edges, or fewer. Thus,

during the $|V|$-th application of the loop no vertex improves its label, and the procedure halts. Since the time complexity of the loop is $O(|E|)$, the whole procedure takes $O(|V| \cdot |E|)$ time. ■

A simple conclusion of the proof of Theorem 1.6 is that, if adv-FORD does not halt after the V-th application of the loop, then there must be an accessible negative circuit. The algorithm can easily be modified to detect the existence of negative circuits in digraphs in time $O(|V| \cdot |E|)$.

1.5.4 The Floyd Algorithm

As in the previous subsection, assume we are given a finite digraph $G(V, E)$ and a length function $l : E \mapsto \mathcal{R}$.[7] We shall assume that there are no negative circuits in G. Our aim is to compute a complete distance table; that is, to compute the distance $\delta(u, v)$, from vertex u to vertex v, for every (ordered) pair of u and v.

We shall assume that there are no parallel edges; if there is more than one edge from u to v, then we can remove them all, except for one of the shortest. Self-loops are also superfluous, but we shall allow them, and as we shall see, the algorithm can be used to check if there are negative circuits, including the case of negative self-loops.

If all edges were of nonnegative lengths, we could have used Dijkstra's algorithm from every vertex, and the complexity, in the simple implementation, would have been $O(|V|^3)$. If there are negative edges, however, this cannot be done. If we use Ford's algorithm from every vertex, the complexity is $O(|V|^2 \cdot |E|)$, and for dense graphs, that can take $\Omega(|V|^4)$ time.

Floyd's algorithm, [9], presented bellow, achieves the goal in time complexity $O(|V|^3)$.

Let us assume that $V = \{1, 2, \ldots, n\}$ and for every $0 \leqslant k \leqslant n$ let δ^k be an $n \times n$ matrix.[8] $\delta^k(i, j)$ stands for the length of a shortest path from vertex i to vertex j, among all paths which do not go through vertices $k+1, k+2, \ldots, n$ as intermediate vertices.

The Floyd algorithm is described in Algorithm 1.6.

It is easy to see that the time complexity of the Floyd algorithm is $O(n^3)$; there are n applications of the outer loop (Lines 9–11), and in each of its applications, there are n^2 applications of Line 11.

The proof of validity is also easy, by induction on k. A shortest path from i to j, among paths that do not go through vertices higher than k, either does not

[7] $\mathcal{R}$ denotes the set of real numbers.

[8] We will show that only one matrix is necessary, but for didactic reasons, let us start with $n + 1$ matrices.

Procedure FLOYD$(G(V,E),l;\delta^n)$

1 *for every* $1 \leqslant i \leqslant n$ *do*
2 *if* there is a self-loop $i \xrightarrow{e} i$ *and* $l(e) < 0$ *then do*
3 $\delta^0(i,i) \leftarrow l(e)$
4 *else* $\delta^0(i,i) \leftarrow 0$
5 *for every* $1 \leqslant i,j \leqslant n$ *such that* $i \neq j$ *do*
6 *if* there is an edge $i \xrightarrow{e} j$ *then do*
7 $\delta^0(i,j) \leftarrow l(e)$
8 *else* $\delta^0(i,j) \leftarrow \infty$
9 *for every* k, starting with $k = 1$ and ending with $k = n$ *do*
10 *for every* $1 \leqslant i,j \leqslant n$ *do*
11 $\delta^k(i,j) \leftarrow \min\{\delta^{k-1}(i,j),\ \delta^{k-1}(i,k) + \delta^{k-1}(k,j)\}$

Algorithm 1.6: The Floyd algorithm.

go through vertex k, and is therefore equal to $\delta^{k-1}(i,j)$, or does go through k and is therefore equal to $\delta^{k-1}(i,k) + \delta^{k-1}(k,j)$.

However, the space complexity of the algorithm, as stated in Algorithm 1.6, is $\Omega(n^3)$, since there are n matrices of size $n \times n$ each. It is easy to see that there is no need to keep previous matrices. Two matrices suffice: The previous one and the one being computed. In fact, one matrix δ will do, where some of the entries are as in δ^{k-1}; and some, as in δ^k. (Observe that a shortest path from i to k, or from k to j, never needs to go through k, since there are no negative circuits.) Thus, in fact, the space complexity can be reduced to $O(n^2)$, by dropping the superscript indexes of δ.

Finally, one can use Floyd's algorithm to check whether there are negative circuits in the digraph. Simply apply the algorithm, and in the end, check whether there is an i for which $\delta(i,i) < 0$. If so, there is a negative circuit.

1.6 Problems

Problem 1.1 Prove that if a connected (undirected) finite graph has exactly $2k$ vertices of odd degree, then the set of edges can be partitioned into k paths such that every edge is used exactly once. Is the condition of connectivity necessary or can it be replaced by a weaker condition?

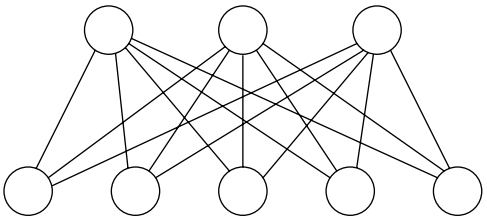


Figure 1.6: A graph for Problem 1.4.

Problem 1.2 Let $G(V, E)$ be an undirected finite circular Euler graph; that is, G is connected and for every $v \in V$, $d(v)$ is even. A vertex s is called *universal* if every application of TRACE(s, G), no matter how the edges are ordered in the incidence lists, produces an Euler circuit.

Prove that s is universal if and only if s appears in every simple circuit of G.[9]

Problem 1.3 Let $G(V, E)$ be a finite digraph such that for every $v \in V$, $d_{in}(v) = d_{out}(v)$. Also, assume that the exits from v are labeled 1, 2, ..., $d_{out}(v)$.

Consider a tour in G, which starts at a given vertex s. Every time a vertex v is visited, the next exit is chosen to leave, starting with exit number 1 and continuing cyclically. However, the tour stops if s is reached and all its exits have been taken.

Prove that the tour stops and that every edge has been used at most once.[10]

Problem 1.4 A *Hamilton* path (circuit) is a simple path (circuit) in which every vertex of the graph appears exactly once.

Prove that the graph shown in Figure 1.6 has no Hamilton path or circuit.

Problem 1.5 Prove that in every completely connected digraph (a digraph in which every two vertices are connected by exactly one directed edge in one of the two possible directions), there is always a directed Hamilton path. (Hint: Prove by induction on the number of vertices.)

Problem 1.6 Prove that a directed Hamilton circuit of the de Bruijn digraph, $G_{\sigma,n}$, corresponds to a directed Euler circuit of $G_{\sigma,n-1}$. Is it true that $G_{\sigma,n}$ always has a directed Hamilton circuit?

9 See[10].

10 More about such tours and finding Euler circuits can be found in [11].

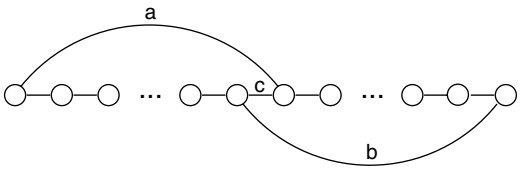


Figure 1.7: A switch for Problem 1.7.

Problem 1.7 In the following, assume that $G(V, E)$ is a finite undirected graph, with no parallel edges and no self-loops.

(i) Describe an algorithm which attempts to find a Hamilton circuit in G by working with a partial simple path. If the path cannot be extended in either direction, then try to close it into a simple circuit by the edge between its endpoints, if it exists, or by a switch, as suggested by Figure 1.7, where edges a and b are added and c is deleted. Once a circuit is formed, look for an edge from one of its vertices to a new vertex, and open the circuit to a now longer simple path, and so on.

(ii) Prove that if for every two vertices u and v, $d(u) + d(v) \geqslant n$, where $n = |V|$, then the algorithm never fails to produce a Hamilton circuit.

(iii) Deduce Dirac's Theorem [12]: "If for every vertex v, $d(v) \geqslant \frac{n}{2}$, then G has a Hamilton circuit."

Problem 1.8 Describe an algorithm for finding the number of shortest paths from s to t, after the BFS algorithm has been performed.

Problem 1.9 A digraph is called *acyclic* if there are no directed circuits.[11] Let $G(V, E)$ be a finite acyclic digraph. A bijection $f : V \mapsto \{1, 2, \ldots, n\}$, where $n = |V|$, is called a *topological sorting* if for every edge $u \longrightarrow v$, $f(u) < f(v)$.

Consider the procedure described in Algorithm 1.7. A queue Q of vertices is used, which is initially empty.

Prove that this is an algorithm, that it computes a topological sorting and that its time complexity is $O(|V| + |E|)$.

Problem 1.10 Show that the Dijkstra algorithm is not applicable if there are negative edges, even if the digraph is acyclic.

[11] Sometimes called DAG, for directed acyclic graph.

```
Procedure TOPO.SORT(G;f)

1   for every v ∈ V compute d_in(v)
2   for every v ∈ V do
3       if d_in(v) = 0 then put v in Q
4   i ← 1
5   while Q ≠ ∅ do
6       remove the first vertex u from Q
7       f(u) ← i
8       i ← i + 1
9       for every edge u ⟶ v do
10          d_in(v) ← d_in(v) − 1
11          if d_in(v) = 0 then put v in Q
```

Algorithm 1.7: Topological sorting.

Problem 1.11 In the Dijkstra algorithm, assume the sequence of vertices which join P, in this order, is $s = v_1, v_2, \ldots$. Prove that the sequence $\lambda(v_1), \lambda(v_2), \ldots$ is nondecreasing.

Problem 1.12 Assume $G(V, E)$ is a finite digraph, $l : E \mapsto \mathcal{R}$ a length function, and assume the length of every directed circuit is positive. Also, assume $s \in V$ is the source, V' is the set of vertices accessible from, s and $\delta : V' \mapsto \mathcal{R}$ is the distance function.

We want to compute the function $\nu : V' \mapsto \mathcal{Z}$, where $\nu(v)$ is the number of shortest paths from s to v.

(i) Let $H(V', E')$ be a subgraph of G, where E' is the set of edges $u \xrightarrow{e} v$ such that $\delta(v) = \delta(u) + l(e)$. Prove that H is acyclic.
(ii) Show how a modification of the topological sorting algorithm, applied to H, can compute ν. What is the complexity of this algorithm?

Problem 1.13 Prove that a connected undirected graph G is orientable (by giving each edge some direction) into a strongly connected digraph if and only if each edge of G is in some simple circuit in G.

Problem 1.14 Prove that if the digraph $G(V, E)$, with edge lengths $l : E \mapsto R$, has a negative circuit, then there is a vertex i on this circuit such that in the matrix δ computed by Floyd's algorithm, $\delta(i, i) < 0$.

```
Procedure WARSHALL(G(V,E);T)

1   for every 1 ⩽ i,j ⩽ n do
2       if there is an edge i ⟶ j in G then do
3           T(i,j) ← 1
4       else T(i,j) ← 0
5   for every k, starting with k = 1 and ending with k = n do
6       for every 1 ⩽ i,j ⩽ n do
7           T(i,j) ← max{T(i,j), T(i,k) · T(k,j)}
```

Algorithm 1.8: The Warshall algorithm.

Problem 1.15 The *transitive closure* of a digraph $G(V,E)$ is a digraph $T(V,E_T)$ such that there is an edge $u \longrightarrow v$ in E_T if and only if there is a nonempty directed path from u to v in G.

Show how BFS can be used to construct T in time $O(|V| \cdot |E|)$.

Problem 1.16 This problem is about Warshall's algorithm [13] for the computation of the transitive closure.

Given a digraph $G(V,E)$, where $V = \{1, 2, \ldots, n\}$, we want to compute an $n \times n$ matrix T, such that

$$T(i,j) = \begin{cases} 1 & \text{if there is a nonempty directed path in } G \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Warshall's algorithm is described in Algorithm 1.8.

(i) What is the complexity of Warshall's algorithm? Compare it with the repeated BFS above.
(ii) Prove the validity of Warshall's algorithm. (Hint of one possible proof: Consider a (simple) path from i to j and the order in which the intermediate vertices on it are processed in the loop of Lines 5–7.)
(iii) Show that there is a close relationship between Floyd's algorithm and Warshall's: $\delta(i,j)$ is finite if and only if $T(i,j) = 1$.

Problem 1.17 This problem is about Dantzig's algorithm [14] for computing all distances in a finite digraph, like Floyd's algorithm.

Procedure Dantzig$(G(V,E),l;\delta^n)$

1 *for every* $1 \leqslant i,j,k \leqslant n$ *do*
2 $\quad\delta^k(i,j) \leftarrow \infty$
3 $\delta^1(1,1) \leftarrow l(1,1)$
4 *for every* k, starting with $k=2$ and ending with $k=n$ *do*
5 $\quad$*for every* $1 \leqslant i < k$ *do*
6 $\quad\quad\delta^k(i,k) \leftarrow \min_{1\leqslant j<k}\{\delta^{k-1}(i,j)+l(j,k)\}$
7 $\quad\quad\delta^k(k,i) \leftarrow \min_{1\leqslant j<k}\{l(k,j)+\delta^{k-1}(j,i)\}$
8 $\quad$*for every* $1 \leqslant i,j < k$ *do*
9 $\quad\quad\delta^k(i,j) \leftarrow \min\{\delta^{k-1}(i,j),\ \delta^k(i,k)+\delta^k(k,j)\}$

Algorithm 1.9: The Dantzig algorithm.

For $1 \leqslant i,j \leqslant k \leqslant n$, let $\delta^k(i,j)$ denote the length of a shortest path from i to j among paths that do not use vertices higher than k. The algorithm computes $\delta^k(i,j)$ for all i, j and k.

Set $l(i,j)$ as follows:

$$l(i,j) = \begin{cases} l(e) & \text{if } i \neq j \text{ and there is an edge } i \xrightarrow{e} j \\ \infty & \text{if } i \neq j \text{ and there is no edge } i \longrightarrow j \\ \min\{0,\ l(e)\} & \text{if } i = j \text{ and there is an edge } i \xrightarrow{e} i \\ 0 & \text{if } i = j \text{ and there is no edge } i \longrightarrow i \end{cases}$$

The Dantzig algorithm is described in Algorithm 1.9.

(i) Show that Dantzig's algorithm is valid.
(ii) How can negative circuits be detected?
(iii) What is the time complexity of this algorithm?
(iv) What is the space complexity of the algorithm? Can it be reduced?

Problem 1.18 Describe an algorithm whose input is a finite, strongly connected digraph $G(V,E)$, and it determines whether there is a directed circuit whose length (in terms of the number of edges in it) is odd. The algorithm should be of time complexity $O(|E|)$. (Hint: Does the fact that for some pair of vertices, a and b, there are two directed paths from a to b, one of odd length and one of even length, imply the existence of an odd circuit?)

Problem 1.19 Let an undirected connected finite graph $G(V, E)$ be the road map of a country, where each edge represents a road, and each vertex represents an intersection. Let $h : E \mapsto \mathcal{R}^+$ be a function such that $h(e)$ specifies the maximum height of vehicles allowed on the road represented by e. Also, let $c \in V$ be a specified vertex (such as the capital of the country).

Write an algorithm that, when given the data above, computes for each vertex v the maximum height of vehicles which can travel from c to v.

Make sure the time complexity of your algorithm is $O(|V|^2)$.

(Hint: A proper modification of Dijkstra's algorithm can solve the problem.)

Bibliography

[1] Golomb, S. W., *Shift Register Sequences*, Holden-Day, 1967.

[2] Moore, E. F., "The Shortest Path Through a Maze," *Proc. Iternat. Symp. Switching Th.*, 1957, Part II, Harvard Univ. Press, 1959, pp. 285–292.

[3] Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," *Numerische Math.*, Vol. 1,1959, pp. 269–271.

[4] Cormen, T. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[5] Fredman, M. L., and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *J. of the ACM*, Vol. 34, No. 3, 1987, pp. 596–615.

[6] Ford, L. R., Jr., "Network Flow Theory," The Rand Corporation, P-923, August, 1956.

[7] Ford, L. R., Jr. and D. R. Fulkerson, D. R., *Flows in Networks*, Princeton University Press, 1962, Chap. III, Sec. 5.

[8] Johnson, D. B., "A Note on Dijkstra's Shortest Path Algorithm," *J. of the ACM*, Vol. 20, No. 3, 1973, pp. 385–388.

[9] Floyd, R. W., "Algorithm 97: Shortest Path," *Comm. of the ACM*, Vol. 5, 1962, p. 345.

[10] Ore, O., "A Problem Regarding the Tracing of Graphs," *Elem. Math.*, Vol. 6, 1951, pp. 49–53.

[11] Bhatt, S., D. Greenberg, S. Even, and R. Tayar, "Traversing Directed Eulerian Mazes," *J. of Graph Algorithms and Applications*, Vol. 6, No. 2, 2002, pp. 157–173.

[12] Dirac, G. A., "Connectivity Theorems for Graphs," *Quart. J. of Math.*, Ser. (2), Vol. 3, 1952, pp. 171–174.

[13] Warshall, S., "A Theorem on Boolean Matrices," *J. of the ACM*, Vol. 9, No. 1, 1962, pp. 11–12.

[14] Dantzig, G. B., "All Shortest Routes in a Graph," *Oper. Res. House*, Stanford University Tech. Rep. 66–3, November 1966.

2

Trees

2.1 Tree Definitions

Let $G(V, E)$ be an undirected, finite, or infinite graph. We say that G is *circuit-free* if there are no simple circuits in G. G is called a *tree* if it is connected and circuit-free.

Theorem 2.1 *The following four conditions are equivalent:*

(a) G *is a tree.*
(b) G *is circuit-free, but if any new edge is added to* G*, a simple circuit is formed.*
(c) G *has no self-loops, and for every two vertices, there is a unique simple path connecting them.*
(d) G *is connected, but if any edge is deleted, the connectivity of* G *is interrupted.*

Proof: We shall prove that (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (d) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): We assume that G is connected and circuit-free. Let $a \overset{e}{\text{---}} b$ be a new edge. If $a = b$ then e forms a self-loop, and therefore a simple circuit exists. If $a \neq b$, there is a simple path in G (without e) connecting a and b. The addition of e creates a simple circuit.

(b) $\Rightarrow$ (c): We assume that G is circuit-free and that no edge can be added to G without creating a simple circuit. Clearly, G has no self-loops.

Let a and b be any two vertices of G. If there is no path connecting them, then we can add an edge between a and b without creating a simple circuit. Thus, G must be connected.

Moreover, if there are two simple paths, P and P', connecting a and b, then there must be a simple circuit in G. To see that, assume that

$$P : a = v_0 \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \cdots v_{l-1} \overset{e_l}{\text{---}} v_l = b,$$

and

$$P' : a = v'_0 \xrightarrow{e'_1} v'_1 \xrightarrow{e'_2} v'_2 \cdots v'_{l'-1} \xrightarrow{e'_{l'}} v'_{l'} = b.$$

Since both paths are simple, one cannot be the beginning of the other. Let i be the first index for which $e_i \neq e'_i$. That is, the two paths split at $v_{i-1} = v'_{i-1}$. Let v be the first vertex on P, after the split, which is also on P'. Thus, for some $j, k \geqslant i$, $v = v_j = v'_k$. The subpath of P, between v_{i-1} and v_j, and the subpath of P', between v'_{i-1} and v'_k, form a simple circuit.

(c) $\Rightarrow$ (d): We assume that G has no self-loops and that for every two vertices, there is a unique simple path connecting them. Thus, G is connected.

Assume now that we delete an edge $a \xrightarrow{e} b$ from G. Since G has no self-loops, $a \neq b$. If there is still a (simple) path in $(V, E \setminus \{e\})$ connecting a and b, then in (V, E), there are two simple paths connecting a and b. A contradiction.

(d) $\Rightarrow$ (a): We assume that G is connected and that no edge can be deleted without interrupting the connectivity.

If G has a simple circuit, any edge on this circuit can be deleted without interrupting the connectivity. Thus, G is circuit-free. ■

There are two, more common ways to define a finite tree. These are presented in the following theorem:

Theorem 2.2 *Let* $G(V, E)$ *be a finite graph and* $n = |V|$. *The following three conditions are equivalent:*

(a) G *is a tree.*
(b) G *is circuit-free and* $|E| = n - 1$.
(c) G *is connected and* $|E| = n - 1$.

Proof: For $n = 1$ the theorem is trivial. Assume $n \geqslant 2$. We shall prove that (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): Let us prove, by induction on n, that if $G(V, E)$ is a tree, then $|E| = n - 1$. This statement clearly holds for $n = 1$. Assume that it is true for all $n < m$, and let G be a tree with m vertices.

Delete from G any edge e. By condition (d) of Theorem 2.1, the resulting graph is not connected any more, and has two disjoint connected components. Each of these components is circuit-free and is therefore a tree. By the inductive hypothesis, each component has one edge less than its number of vertices. Thus, together they have $m - 2$ edges. Add e back, and the number of edges is $m - 1$.

(b) $\Rightarrow$ (c): We assume that G is circuit-free and has $n - 1$ edges. Let us first show that G has at least two vertices of degree 1.

Choose any edge $u \overset{e}{\text{---}} v$; there is at least one edge, since the number of edges is $n-1$ and $n \geqslant 2$. Also, $u \neq v$ since G is circuit-free. As in the TRACE procedure, walk on new edges, starting from u. Since the number of edges is finite, this walk must end, say, at vertex t_1. Also, the traced path is simple, or a simple circuit would have been formed. Thus, $d(t_1) = 1$. A similar walk, starting from v, yields another vertex, t_2, and $d(t_2) = 1$, as well. Thus, there are two vertices of degree 1.

Now, the proof that G is connected proceeds by induction on the number of vertices, n. Obviously, the statement holds for $n = 1$. Assume that it holds for $n = m-1$, and let G be a circuit-free graph with m vertices and $m-1$ edges. Eliminate from G a vertex v of degree 1 and its incident edge. The resulting graph is still circuit-free and has $m-1$ vertices and $m-2$ edges. Thus, by the inductive hypothesis it is connected. Therefore, G is connected, as well.

(c) $\Rightarrow$ (a): Assume that G is connected and has $n-1$ edges. As long as G has simple circuits, we can eliminate edges (without eliminating vertices) and maintain the connectivity. When this process terminates, the resulting graph is a tree, and, by (a) $\Rightarrow$ (b), it has $n-1$ edges. This, however, is the number of edges we started with. Thus, no edge has been eliminated, and therefore G is circuit-free.

■

Let us call a vertex whose degree is 1, a *leaf*. A corollary of Theorem 2.2 and the statement proved in the (b) $\Rightarrow$ (c) part of its proof is the following:

Corollary 2.1 *A finite tree with more than one vertex has at least two leaves.*

2.2 Minimum Spanning Tree

A graph $G'(V', E')$ is called a *subgraph* of a graph $G(V, E)$, if $V' \subset V$ and $E' \subset E$.[1]

Assume $G(V, E)$ is a finite, connected (undirected) graph and each edge $e \in E$ has a known length $l(e) > 0$. Assume that we want to find a connected subgraph $G'(V, E')$, whose total length, $\sum_{e \in E} l(e)$, is minimum; or, in other words, we want to remove from G a subset of edges whose total length is maximum, and yet, the resulting subgraph remains connected. Such a subgraph is a tree. For G' is assumed to be connected, and since its total length is minimum, none of its edges can be removed without destroying its connectivity. By Theorem 2.1 (see part (d)) G' is a tree. A subgraph of G that contains all of its vertices and

[1] Note that an arbitrary choice of $V' \subset V$ and $E' \subset E$ may not yield a subgraph, simply because it may not be a graph; that is, some of the endpoints of edges in E' may not be in V'.

```
Procedure PRIM(G, l; T)

1   for every v ∈ V do
2       λ(v) ← ∞
3   choose a vertex s ∈ V
4   λ(s) ← 0
5   ε(s) ← ∅
6   TEMP ← V
7   T ← ∅
8   while TEMP ≠ ∅ do
9       choose a vertex v ∈ TEMP for which λ(v) is minimum
10      TEMP ← TEMP \ {v}
11      T ← T ∪ ε(v)
12      for every v —e— u do
13          if u ∈ TEMP and λ(u) > l(e) then do
14              λ(u) ← l(e)
15              ε(u) ← {e}
```

Algorithm 2.1: The Prim algorithm.

is a tree is called a *spanning tree* of G. Thus, our problem is that of finding a minimum (length) spanning tree (MST) of G.

There are several known algorithms for constructing an MST. We describe here the Prim algorithm [1]. Additional algorithms are discussed in the problems section in the end of the chapter.

The Prim algorithm is described in Algorithm 2.1.[2]

In the following discussion, we shall use T to denote a subgraph of G, which is a tree, and the set of edges in this tree. *TEMP* denotes the set of vertices, not yet in T.

The idea of the algorithm is to "grow" T, by starting with a tree with one vertex, s, and in each step adding a new leaf, v, to T, until all vertices have joined it. Thus, v is chosen in such a way that the edge connecting it to T (the single edge in the set $\varepsilon(v)$) is of minimum length among edges which connect a vertex of T to a vertex of *TEMP*.

[2] The structure of the algorithm is similar to that of Dijkstra, but historically, Prim's algorithm precedes it.

When a new vertex, v, joins T, all its incident edges, $v \overset{e}{\text{---}} u$ are checked. If u is not a vertex of T yet (this is equivalent to the fact that $u \in \mathit{TEMP}$), and if $l(e)$ is shorter than the presently known shortest edge (if any) that connects u to a vertex of T, then the value of $\lambda(u)$ is updated to be $l(e)$, and the edge e is recorded in (the set) $\varepsilon(u)$.

Since G is connected, when PRIM halts, the set of edges in T constitutes a spanning tree. The complexity of the algorithm is similar to that of Dijkstra's algorithm; see Section 1.5.2.

Before we discuss the validity of the algorithm, let us define the concept of a cut.

Let $S \subset V$ and $\bar{S} = V \setminus S$. The *cut* $(S;\bar{S})$ is the set of edges with one endpoint in S and the other in $\bar{S}$.

Theorem 2.3 *The tree T produced by Prim's algorithm is a Minimum Spanning Tree of* G.

Proof: Let us prove that for every T produced while the algorithm is running, there is an MST, T_{opt}, of G, such that T is a subgraph of T_{opt}. This implies that the final T is an MST. The proof is by induction on v, the number of vertices in T.

Initially, $v = 1$ and there are no edges in T. Thus, the claim holds trivially. Assume now that the claim holds for $v < |V|$, and let us prove it for $v + 1$.

By the inductive hypothesis, there is an MST, T_{opt}, such that T is a subgraph of T_{opt}. If the newly added edge, e, is such that $e \in T_{opt}$, then the claim holds for $T \cup \{e\}$ as well, and we are done.

If $e \notin T_{opt}$, consider the graph $H(V, T_{opt} \cup \{e\})$. By Theorem 2.1 part (b), there is a simple circuit C in H. Also, let S be the set of vertices in T and consider the cut $(S;\bar{S})$ of G. There are at least two edges of C in $(S;\bar{S})$. One of them is e, and assume another is e'. Now consider the graph $H'(V, (T_{opt} \cup \{e\}) \setminus \{e'\})$. Since an edge of a simple circuit has been removed, H' remains connected, and the number of edges is $|V| - 1$. By Theorem 2.2 part (c), H' is a spanning tree. By the choice of v, $\lambda(v)$ is minimum (among values of λ for vertices in *TEMP*), and the edge $e \in \varepsilon(v)$ satisfies $l(e) = \lambda(v) \leqslant l(e')$. Thus, the total length of H' is less than or equal to that of T_{opt}, and is therefore an MST.

■

The analogous problem for digraphs, namely, that of finding a subset of the edges E' whose total length is minimum among those for which (V, E') is a strongly connected subgraph, is much harder. In fact, even the case where $l(e) = 1$ for all edges is hard. This is discussed in Chapter 10.

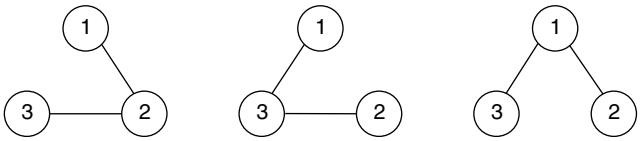


Figure 2.1: The spanning trees on three named vertices.

2.3 Cayley's Theorem

In a later section we shall consider the question of the number of spanning trees of a given graph. Here we consider the more restricted, and yet interesting, problem of the number of trees one can construct on a given set of vertices, $V = \{1, 2, \ldots, n\}$.

For $n = 2$, there is only one tree that one can construct, consisting of an edge between the two vertices. For $n = 3$, there are three possible trees, as shown in Figure 2.1. The reader can verify, by exhausting all cases, that for $n = 4$ the number of trees is 16. The following theorem is due to Cayley [3]. Warning: We shall use the integers $1, 2, \ldots, n$ in three different meanings. As names of vertices, as integers, and as letters of an alphabet.

Theorem 2.4 *The number of spanning trees for* n *distinct vertices is* n^{n-2}.

The remainder of this section describes a proof due to Prüfer [4]. (For a survey of various proofs see Moon [5].)

Assume $V = \{1, 2, \ldots, n\}$. Let us display a bijection between the set of the spanning trees and the n^{n-2} words of length $n-2$ over the alphabet $\{1, 2, \ldots, n\}$. The mapping from the set of spanning trees to the corresponding set of words is defined by an algorithm which is described in Algorithm 2.2.

For example, assume that $n = 6$ and T is as shown in Figure 2.2. We now apply TREEtoWORD. We start with the given T and an empty template for a word w of 4 letters. This is depicted in the first line of Figure 2.3. T has 3 leaves, and vertex 2 has the least name. Therefore, vertex 2 and its incident edge, $2 \text{——} 4$, are removed. The first letter in the word w is 4, as shown in the second line of Figure 2.3. The next leaf to be removed is 3, and $a_2 = 1$, and so on. After four steps, the tree consists of two vertices (4 and 6) and an edge between them, while $w = 4164$.

By Corollary 2.1, Line (2) of TREEtoWORD algorithm can always be performed. Note that when a leaf is removed, the remaining graph is still a tree. It follows that for every tree of n vertices, a word w of length $n-2$ is produced. Since algorithm TREEtoWORD is deterministic, it defines a mapping f from

Procedure TREEtoWORD($T(V,E)$; $w = a_1 a_2 \cdots a_{n-2}$)

1 *for every* i starting with 1 and ending with $n-2$ *do*
2 choose a leaf v whose name is minimum among the leaves of T
3 let $v \overset{e}{\text{---}} u$ be its incident edge
4 $a_i \leftarrow u$
5 remove e and v from T

Algorithm 2.2: The TREEtoWORD algorithm. Mapping a spanning tree T to a word w.

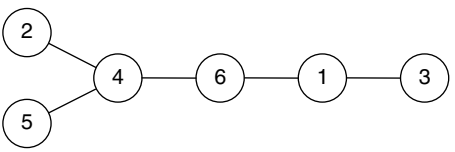


Figure 2.2: T: An example of a spanning tree with six vertices.

trees to words. It remains to be shown that no word is produced by two different trees, and that for every word w there is a tree T such that $f(T) = w$.

Notice that TREEtoWORD is insensitive to the nature of the set of vertices, V, of T, as long as the names of the vertices are distinct and an order is defined on these names. We shall assume that V is a set of integers, not necessarily $\{1, 2, \ldots, n\}$.

Lemma 2.1 *If in* T *a vertex* v *has a degree* $d(v)$*, then in* $w = f(T)$ *the letter* v *appears* $d(v) - 1$ *times.*

Proof: When each of the edges incident on v are removed, except the last one, one writes the letter v in w. Thus, v appears $d(v) - 1$ times in w. The last edge may not be removed at all, if v is one of the two vertices which remain when TREEtoWORD halts. And if v's last edge is removed, then v is the removed leaf, and its neighbor, not v, is written in w. ■

Lemma 2.2 *For every word* $w = a_1 a_2 \cdots a_{n-2}$ *over an alphabet* V*, where* $|V| = n$*, there is a unique tree* T *whose vertices are* V *and* $f(T) = w$*.*

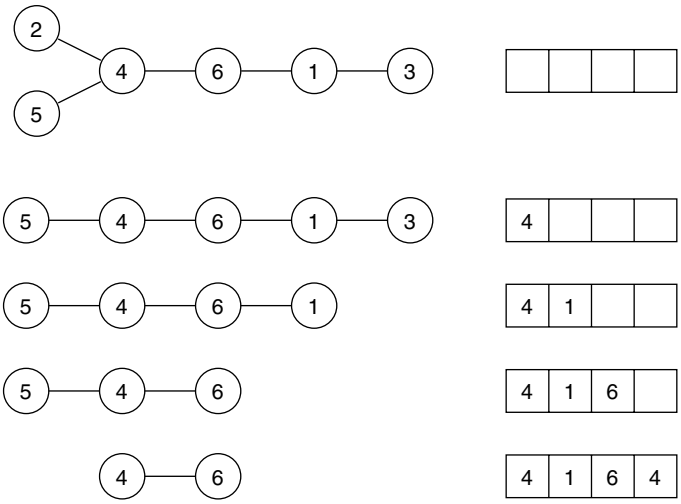


Figure 2.3: Applying TREEtoWORD to the the example tree T.

Proof: By induction on $n \geqslant 2$. For $n = 2$, w is the empty word and if $V = \{u, v\}$, there is only one tree whose set of vertices is V. TREEtoWORD does nothing (leaving T intact, and w empty).

Now assume the claim holds for $n-1$ and let us prove it for n. Let $w = a_1 a_2 \cdots a_{n-2}$ be a word over an alphabet V, where $|V| = n$.

Since the alphabet has n letters, while there are $n-2$ appearances of letters in w, there are at least two letters missing from w. Let v be the least letter that does not appear in w. Now consider the word $w' = a_2 a_3 \cdots a_{n-2}$, with the alphabet $V' = V \setminus \{v\}$. By the inductive hypothesis, there is a unique tree T' whose set of vertices is V', such that $f(T') = w'$. Notice that a_1 appears in w, and therefore $a_1 \neq v$, and thus $a_1 \in V'$.

By Lemma 2.1, the set of leaves of T' is equal to the set of elements of V' that do not appear as letters in w'. Thus, v is less than every leaf of T', with the possible exception of a_1.

Now define T to be the tree of n vertices that results from T' by adding to it the vertex v and an edge connecting v to a_1. Clearly T is a tree, and a_1 is not a leaf of T, even if it has been a leaf of T'. It follows that v is the least leaf of T. Thus, if one applies TREEtoWORD to T, the first letter assigned to the word TREEtoWORD builds is indeed a_1, v and its incident edge are removed, and the tree on which TREEtoWORD continues is T'. By the inductive hypothesis, T' is the unique tree that produces w'. We conclude that $f(T) = w$.

Since v is the least letter missing from w, and since a_1 is the first letter of w, every tree that produces w must have an edge v — a_1. Thus, T is the unique tree that produces w.

■

2.4 Directed Tree Definitions

A digraph $G(V, E)$ is said to have a *root* r if $r \in V$ and every vertex $v \in V$ is reachable from r; that is, there is a directed path that starts in r and ends in v.

A digraph (finite or infinite) is called a *directed tree* if it has a root and its underlying undirected graph is a tree.

Theorem 2.5 *Assume* G *is a digraph. The following five conditions are equivalent:*

(a) G *is a directed tree.*
(b) G *has a root from which there is a unique directed path to every vertex.*
(c) G *has a root* r *for which* $d_{in}(r) = 0$*, and for every other vertex* v*,* $d_{in}(v)=1$.
(d) G *has a root and the deletion of any edge (but no vertices) interrupts this condition.*
(e) The underlying undirected graph of G *is connected and* G *has one vertex* r *for which* $d_{in}(r) = 0$*, while for every other vertex* v*,* $d_{in}(v) = 1$.

Proof: We prove that (a) ⇒ (b) ⇒ (c) ⇒ (d) ⇒ (e) ⇒ (a).

(a) ⇒ (b): We assume that G has a root, say r, and its underlying undirected graph G' is a tree. Thus, by Theorem 2.1 part (c), there is a unique simple path from r to every vertex in G'. Also, G' is circuit-free. Thus, a directed path from r to a vertex v, in G, must be simple and unique.

(b) ⇒ (c): Here, we assume that G has a root, say r, and a unique directed path from it to every vertex v. First, let us show that $d_{in}(r) = 0$.

Assume there is an edge $u \xrightarrow{e} r$. There is a directed path from r to u, and it can be continued, via e, back to r. Thus, in addition to the empty path from r to itself (containing no edges), there is one more path, in contradiction to the assumption of the path uniqueness.

Now, we have to show that if $v \neq r$, then $d_{in}(v) = 1$. Clearly, $d_{in}(v) > 0$, for it must be reachable from r. If $d_{in}(v) > 1$, then there are at least two edges, say, $v_1 \xrightarrow{e_1} v$ and $v_2 \xrightarrow{e_2} v$. Since there is a directed path, P_1, from r to v_1, and a directed path, P_2, from r to v_2, by adding e_1 to P_1 and e_2 to P_2, we get two different directed paths from r to v, contradicting the uniqueness assumption. (Note that the two paths are different, even if $v_1 = v_2$.)

(c) $\Rightarrow$ (d): This proof is trivial, for the deletion of any edge $u \xrightarrow{e} v$ will make v unreachable from r.

(d) $\Rightarrow$ (e): We assume that G has a root, say r, and the deletion of any edge interrupts this condition.

Clearly, $d_{in}(r) = 0$, for any edge entering r can be deleted without interrupting the condition that r is a root. Also, if $v \neq r$, then $d_{in}(v) > 0$, since v is reachable from r. If $d_{in}(v) > 1$, let $v_1 \xrightarrow{e_1} v$ and $v_2 \xrightarrow{e_2} v$ be two edges entering v. Let P be a simple directed path from r to v. At least one of the edges e_1 and e_2 is not used in P. This edge can be deleted without interrupting the fact that r is a root. Thus, $d_{in}(v) = 1$.

(e) $\Rightarrow$ (a): We assume that the underlying undirected graph of G, G', is connected, $d_{in}(r) = 0$, and for $v \neq r$, $d_{in}(v) = 1$. First let us prove that r is a root of G.

Let P' be a simple (undirected) path connecting r and v in G'. This must correspond to a directed path, P, from r to v in G, for if any of the edges points in the wrong direction, it would imply either that $d_{in}(r) > 0$ or that for some u, $d_{in}(u) > 1$.

Now, assume G' has a simple circuit, C'. In G, all corresponding edges must be directed in the same circular direction, or there would be a vertex v for which $d_{in}(v) \geqslant 2$. Thus, r is not one of the vertices of C, the directed circuit of G which corresponds to C'. Now, let P be a shortest directed path from r to a vertex of C, say it is u. Then, $d_{in}(u)$ must be at least 2. A contradiction. Thus, G' is circuit-free, and is therefore an undirected tree.

■

In case of finite digraphs one more useful definition of a directed tree is possible:

Theorem 2.6 *A finite digraph $G(V, E)$ is a directed tree if and only if its underlying undirected graph, G', is circuit-free, one of its vertices, r, satisfies $d_{in}(r) = 0$, and for all other vertices v, $d_{in}(v) = 1$.*

Proof: The "only if" part follows directly from the definition of a directed tree and Theorem 2.5 part (c).

To prove the "if" part we first observe that the number of edges is $|V| - 1$. Thus, by Theorem 2.2, (b) $\Rightarrow$ (c), G' is connected. Thus, by Theorem 2.5, (e) $\Rightarrow$ (a), G is a directed tree.

■

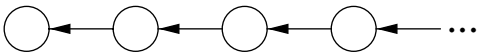


Figure 2.4: An example of an infinite arbitrated digraph with no root.

Let us say that a digraph is *arbitrated* (Berge [6] calls it quasi strongly connected) if for every two vertices, v_1 and v_2, there is a vertex v called an *arbiter* of v_1 and v_2, such that there are directed paths from v to v_1 and from v to v_2. There are infinite digraphs which are arbitrated but do not have a root. For example, see the digraph of Figure 2.4. However, for finite digraphs the following theorem holds:

Theorem 2.7 *If a finite digraph is arbitrated then it has a root.*

Proof: Let $G(V, E)$ be a finite arbitrated digraph, where $V = \{1, 2, \ldots, n\}$. Let us prove, by induction, that every set $\{1, 2, \ldots, m\}$, where $m \leqslant n$, has an arbiter; i.e., a vertex a_m such that every $1 \leqslant i \leqslant m$ is reachable from a_m. By definition, a_2 exists. Assume a_{m-1} exists. Let a_m, be the arbiter of a_{m-1} and m. Since a_{m-1} is reachable from a_m, and every $1 \leqslant i \leqslant m-1$ is reachable from a_{m-1}, every $1 \leqslant i \leqslant m-1$ is also reachable from a_m. ■

Thus, for finite digraphs, the condition that it has a root, as in Theorem 2.5 parts (a), (b), (c), and (d), can be replaced by it being arbitrated.

2.5 The Infinity Lemma

The following is known as König's Infinity Lemma [7]:

Theorem 2.8 *If* G *is an infinite digraph with a root* r*, and every vertex has a finite out-degree, then* G *has an infinite directed path starting in* r.

Before we present our proof, let us point out the necessity of the finiteness of the out-degrees of the vertices. For, if we allow a single vertex to be of infinite out-degree, the conclusion does not follow. Consider the digraph of Figure 2.5. The root r is connected to vertices v_1^1, v_1^2, v_1^3, ..., where v_1^k is the second vertex on a directed path of length k. It is clear this directed tree is infinite, and yet it has no infinite path. Furthermore, the replacement of the condition of finite degrees by the condition that for every k, the tree has a path of length k, does not work either, as the same example shows.

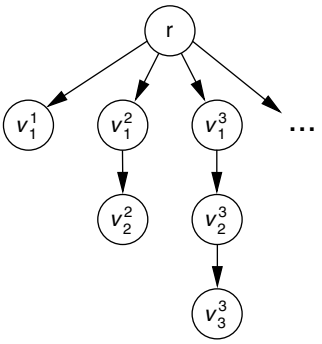


Figure 2.5: An example of an infinite digraph with no infinite path.

Proof: [3] First, let us restrict our attention to a directed tree T which is an infinite subgraph of G. T's root is r. All vertices of distance 1 away from r in G are also of distance 1 away from r in T. In general, if a vertex v is of distance δ away from r in G, it is also of distance δ away from r in T; all the edges entering v in G are now dropped, except one which connects a vertex of distance $\delta - 1$ to v. It is sufficient to show that in T there is an infinite directed path from r. Clearly, since T is a subgraph of G, all its vertices are of finite out-degrees too.

In T, r has infinitely many descendants (vertices reachable from r). Since r is of finite out-degree, at least one of its sons (the vertices reachable via one edge), say r_1, must have infinitely many descendants. One of r_1's sons has infinitely many descendants, too, and so we continue to construct an infinite directed path $r, r_1, r_2, \ldots$. ■

Despite the seeming simplicity of the theorem, it is useful. For example, imagine we conduct a search on a directed tree of finite degrees (but it is not known that there is a bound on the degrees). If it is known that the tree has no infinite directed path, then the theorem assures us that the tree is finite and our search will terminate.

An interesting application of Theorem 2.8 was made by Wang [8]. Consider the problem of tiling the plane with square tiles, all of the same size (Wang calls the tiles "dominoes"). There is a finite number of tile families. The sides of the tiles are labeled by letters of an alphabet, and all tiles of one family have the same labels and are indistinguishable. Tiles may not be rotated or reflected, and the labels are specified for their north side, south side, and so on. There

[3] We use a naive approach to set theory and thus ignore the applications of the axiom of choice.

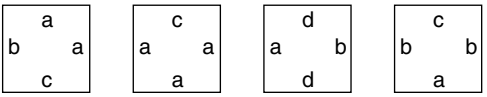


Figure 2.6: A set of families of tiles.

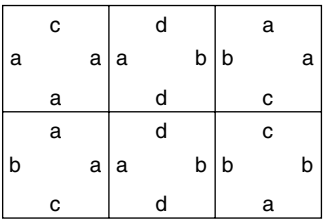


Figure 2.7: A torus constructed from the tile families depicted in Figure 2.6.

is an infinite supply of tiles of each family. Tiles may abut one another if the joining sides have the same labels. For example, if the tile families are as shown in Figure 2.6, then we can construct the "torus," as shown in Figure 2.7. Now, by repeating this torus infinitely many times horizontally and vertically, we can tile the whole plane.

Wang proved that if it is possible to tile the upper right quadrant of the plane with a given finite set of t tile families, then it is possible to tile the whole plane. The reader should realize that a southwest shift of the upper-right tiled quadrant cannot be used to cover the whole plane. In fact, if the number of tile families is not restricted to be finite, one can find sets of families for which the upper-right quadrant is tileable, whereas the whole plane is not. (See Problem 2.14.)

Consider the following directed tree T: The root r is connected to vertices, each representing a tile family, that is, a 1×1 square, tiled with the tile of that family. Thus, the out-degree of r is t. For every $k \geqslant 1$ and every legitimate way of tiling a $(2k+1) \times (2k+1)$ square, there is a vertex in T; its father is the vertex which represents the tiling of a $(2k-1) \times (2k-1)$ square, identical to the center part of the tiled square represented by the son.

Now, if the upper-right quadrant is tilable, then T has infinitely many vertices. However, a vertex representing a certain tiling of a $(2k-1) \times (2k-1)$ square has at most t^{8k} sons. Since the out-degree of each vertex is finite (although it may not be bounded), Theorem 2.8 implies that there is an infinite directed path in T. Such a path describes a way to tile the whole plane.

2.6 Problems

Problem 2.1 Let $T_1(V, E_1)$ and $T_2(V, E_2)$ be two spanning trees of $G(V, E)$. Prove that for every $\alpha \in E_1 \setminus E_2$ there is a $\beta \in E_2 \setminus E_1$ such that each of the sets

$$(E_1 \setminus \{\alpha\}) \cup \{\beta\}$$

$$(E_2 \setminus \{\beta\}) \cup \{\alpha\}$$

defines a spanning tree.

Problem 2.2 Let $G(V, E)$ be a finite connected graph and $l : E \mapsto \mathcal{R}$. Describe an algorithm for finding a maximum length spanning tree of G; explain why it is valid and analyze its complexity.

Problem 2.3 Algorithm 2.3 describes the Kruskal [2] algorithm for computing an MST of a finite connected undirected graph $G(V, E)$ with a length function $l : E \mapsto \mathcal{R}$. The set of edges in the resulting tree is T. Prove that the resulting T is indeed an MST and analyze the complexity of the algorithm.

Problem 2.4 Describe an algorithm that is similar to Kruskal's, but instead of adding edges, from light to heavy, as long as they do not create a simple circuit, it deletes edges, from heavy to light, as long as connectivity is maintained. Prove its validity and analyze its complexity.

Problem 2.5 Let $G(V, E)$ be a finite undirected graph, with a length function $l : E \mapsto \mathcal{R}$. Also, T is an MST of G.

```
Procedure KRUSKAL(G(V,E), l; T)

1   sort the set E into E = {e_1, e_2, ..., e_|E|}, so that if i < j, then l(e_i) ≤
      l(e_j)
2   T ← ∅
3   for i, starting with i = 1 and ending with i = |E| do
4       if (V, T ∪ {e_i}) is circuit-free then
5           T ← T ∪ {e_i}
```

Algorithm 2.3: The Kruskal algorithm.

A new edge, e, is added, of length $l(e)$. The following is a sketch of an algorithm for mending the tree to be an MST of the new graph: Add e to T. Let e' be an edge of maximum length in the simple circuit that forms. If $e' \neq e$, then remove e'. Otherwise, leave T intact.

Prove the validity of this algorithm and compare its time complexity with computing an MST anew.

Problem 2.6 Let $G(V, E)$ be a finite undirected graph, with a length function $l\colon E \mapsto \mathcal{R}$. Also, T is an MST of G.

An edge $e \in E$ is removed from G to form G'. The following is a sketch of an algorithm for mending the tree to be an MST of G': If $e \notin T$, do nothing. Otherwise, let S and $\bar{S}$ be the two sets of vertices in the two connected components of $T \setminus \{e\}$. If in G', the cut $(S;\bar{S}) = \emptyset$ then G' has no MST. Otherwise, let e' be of minimum length in $(S;\bar{S})$. The new MST is $(T \setminus \{e\}) \cup \{e'\}$.

Prove the validity of this algorithm and compare its time complexity with computing an MST anew.

Problem 2.7 Compute the number of trees that can be built on n, given labeled vertices with unlabeled edges, in such a way that one specified vertex is of degree k.

Problem 2.8 Let $V = \{1, 2, \ldots, n\}$ and for each $1 \leqslant i \leqslant n$ $d(i)$ is a positive integer. Prove that if

$$\sum_{i=1}^{n} d(i) = 2n - 2,$$

then there exists a tree $T(V, E)$ in which for every i, the degree of i is $d(i)$. How many such trees are there if the edges have no names?

Problem 2.9 What is the number of trees that one can build with n labeled vertices and $m = n - 1$ labeled edges? Prove that the number of trees that can be built with $m \geqslant 2$ labeled edges (and no labels on the vertices) is $(m+1)^{m-2}$. Explain why the condition that $m \geqslant 2$ is necessary.[4]

Problem 2.10 A digraph which has no directed circuits is called a DAG (directed acyclic graph). One wants an algorithm that checks whether a given finite DAG, $G(V, E)$, has a root.

One way to do this is first to check that there is only one vertex r, such that $d_{in}(r) = 0$, and then check if all vertices are reachable from r. Explain why this is valid, and prove that the time complexity of such an algorithm is $O(|E|)$.

[4] This problem was inspired by an unpublished report of S. Golomb and A. Lempel, and a comment made by A. Pnueli.

Problem 2.11 Given a finite digraph $G(V,E)$. Describe an algorithm which runs in time $O(|V|)$ and checks whether G is a directed tree.

Problem 2.12 Prove that, if G is an infinite undirected connected graph whose vertices are of finite degrees, then every vertex of G is the start vertex of some simple infinite path.

Problem 2.13 Show that, if rotation or flipping of tiles is allowed, then the question of tiling the plane becomes trivial.

Problem 2.14 Consider the following (infinite) set of tile families: For every ordered pair of positive integers (i,j), there is a tile family with a label $i-1$ in the West, i in the East, $j-1$ in the South, and j in the North. Prove that one can tile the upper-right quadrant with this set of families, but not the whole plane.

Problem 2.15 Let T be an undirected tree with n vertices. We want to invest $O(n)$ time, labeling the graph in a way that will allow one to find a (minimum) path between any two vertices that are of distance δ apart, in time $O(\delta)$. Describe both a preparatory algorithm and an algorithm for finding the path once the two vertices are specified.

Problem 2.16 A *clique* is a simple undirected graph such that for every two vertices there is an edge connecting them. Let $G(V,E)$ be a clique and $T(V,E')$ a spanning tree of G. Prove that the complement of T $(= (V,\overline{E'}))$ is either connected or consists of one isolated vertex, while the remaining vertices form a clique.

Problem 2.17 Let $G(V,E)$ be an undirected finite connected graph, where each edge e has a given length $l(e) > 0$ and s is a designated vertex. Also, let $\delta(v)$ denote the distance from s to v.

(a) Explain why every vertex $v \neq s$, has an incident edge $u \overset{e}{\text{---}} v$, such that $\delta(v) = \delta(u) + l(e)$.
(b) Show that if one such edge is chosen for each vertex $v \neq s$, then the set of these edges forms a spanning tree of G.
(c) Show that such a tree is not necessarily an MST.

Bibliography

[1] Prim, R.C., "Shortest Connection Networks and Some Generalizations," *Bell System Tech. J.*, Vol. 36, 1957, pp. 1389–1401.
[2] Kruskal, J.B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proc. of the Amer. Math. Society*, Vol. 7, 1956, pp. 48–50.

[3] Cayley, A., "A Theorem on Trees," *Quart. J. Math.*, Vol. 23, pp. 376–378. Also in *Collected Papers*, Vol. 13, Cambridge, 1897, pp. 26–28.
[4] Prüfer, H., "Neuer Beweise eines Satzes über Permutationen," *Arch. Math. Phys.*, Vol. 27, 1918, pp. 742–744.
[5] Moon, J.W., "Various Proofs of Cayley's Formula for Counting Trees," *A Seminar on Graph Theory*, F. Harary (ed.), Holt, Rinehart and Winston, 1967, pp. 70–78.
[6] Berge, C., and A. Ghouila-Houri, *Programming, Games and Transportation Networks*, Wiley, 1965, Sec. 7.4.
[7] König, D., *Theorie der endlichen und unendlichen Graphen*, Liepzig, 1936. Reprinted by Chelsea, 1950.
[8] Wang, H., "Proving Theorems by Pattern Recognition," *Bell System Tech. J.*, Vol. 40, 1961, pp. 1–41.

3
Depth-First Search

3.1 DFS of Undirected Graphs

The depth-first search (DFS) technique is a method of scanning a finite, undirected graph. Since the publication of the papers of Hopcroft and Tarjan [4, 6], DFS has been widely recognized as a powerful technique for solving various graph problems. However, the algorithm has been known since the nineteenth century as a technique for threading mazes. See, for example, Lucas' report of Trémaux's work [5]. Another algorithm, which was suggested later by Tarry [7], is just as good for threading mazes, and in fact, DFS is a special case of it. But the additional structure of DFS is what makes the technique so useful.

3.1.1 Trémaux's Algorithm

Assume one is given a finite, connected graph $G(V, E)$, which we will also refer to as the *maze*. Starting in one of the vertices, one wants to "walk" along the edges, from vertex to vertex, visit all vertices, and halt. We seek an algorithm that will guarantee that the whole graph will be scanned without wandering too long in the maze, and that the procedure will allow one to recognize when the task is done. However, before one starts walking in the maze, one does not know anything about its structure, and therefore, no preplanning is possible. So, decisions about where to go next must be made one by one as one goes along.

We will use "markers," which will be placed in the maze to help one to recognize that one has returned to a place visited earlier and to make later decisions on where to go next. Let us mark the *passages*, namely the connections of the edges to vertices. If the graph is presented by incidence lists, then we can think of each of an edge's two appearances in the incidence lists of its two endpoints as its two passages. It suffices to use two types of markers: F for the first passage used to enter the vertex, and E for any other passage used to leave

```
Procedure TRÉMAUX(G, s)

1   v ← s
2   while there is an unmarked passage in v or v has a passage marked F do
3       if there is an unmarked passage to edge v —e— u, then do
4           mark the passage of e at v by E
5           if u has no marked passages then do
6               mark the passage of e at u by F
7               v ← u
8           else mark the passage of e at u by E
9       else (there is a passage in v marked F) do
10          use the passage marked F to move to the neighboring vertex u
11          v ← u
```

Algorithm 3.1: The Trémaux algorithm.

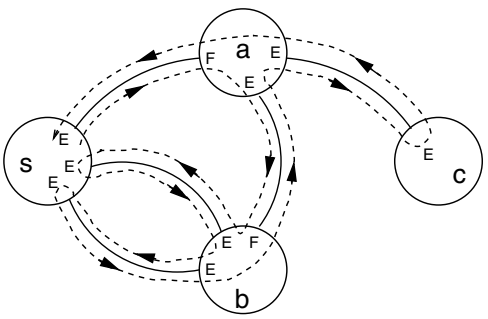


Figure 3.1: An example of running Trémaux's algorithm.

the vertex. No marker is ever erased or changed. There is no use of memory other than the markers on the passages and the stage of the algorithm one is has reached; in other words, the moves are controlled by a finite state automaton. As we shall prove later, the algorithm described in Algorithm 3.1 will terminate in the original starting vertex s, after scanning each edge once in each direction.

Let us demonstrate the algorithm on the graph shown in Figure 3.1. The initial value of v, the place where we are situated, or the center of activity, is s. All passages are unlabeled. We choose one, mark it E and traverse the edge. Its other endpoint is a ($u = a$). None of its passages are marked, therefore we mark the

passage through which a has been entered by F, the new center of activity is a ($v = a$), and we are back in Line 2. Since a has two unmarked passages, assume that we choose the one leading to b. The passage at a is marked E and the one at b is marked F since b is new, etc. The complete excursion is shown in Figure 3.1 by the dashed line.

Lemma 3.1 *Trémaux's algorithm never allows an edge to be traversed twice in the same direction.*

Proof:[1] If a passage is used as to exit a vertex and enter an edge), then either it is being marked E in the process, and thus the edge is never traversed again in this direction, or the passage is already marked F. It remains to be shown that no passage marked F is ever reused for entering the edge.

Let $u \xrightarrow{e} v$ be the first edge to be traversed twice in the same direction, from u to v. The passage of e, at u, must be labeled F. Since s has no passages marked F, $u \neq s$. Vertex u has been left $d(u) + 1$ times; once through each of the passages marked E and twice through e. Thus, u must have been entered $d(u) + 1$ times and some edge been used twice to enter u, before e is used for the second time. A contradiction. ■

An immediate corollary of Lemma 3.1 is that the process described by Trémaux's algorithm will always terminate. Clearly, it can only terminate in s, since every other visited vertex has an F passage. Therefore, all we need to prove is that, upon termination, the whole graph has been scanned.

Lemma 3.2 *Upon termination of Trémaux's algorithm, every edge of the graph has been traversed once in each direction.*

Proof: Let us state the proposition differently: For every vertex, all its incident edges have been traversed in both directions.

First, consider the start vertex s. Since the algorithm has terminated, all the incident edges of s have been traversed from s outward. Thus, s has been left $d(s)$ times, and since we end up in s, it has also been entered $d(s)$ times. However, by Lemma 3.1 no edge is traversed more than once in the same direction. Therefore, every edge incident to s has been traversed once in each direction. Let S be the set of vertices for which the statement that each of their incident edges has been traversed once in each direction holds. Since $s \in S$, $S \neq \emptyset$. Assume $V \neq S$. By the connectivity of the graph there must be edges

[1] The following terminology is used in the proof. A passage from a node u to an edge $v \xrightarrow{e} u$ is used as an *exit* if the algorithm exits v via the passage. In such a case, the passage is used to *enter* the edge. (G.E.)

connecting vertices of S with $V \setminus S$. All these edges have been traversed once in each direction. Let e be the first edge to be traversed from a vertex $v \in S$ to $u \in V \setminus S$. Clearly, the passage of e, at u, is marked F. Since this passage has been entered, all other passages of u must have been marked E. Thus, each of u's incident edges has been traversed outward. The search has not started in u and has not ended in u. Therefore, u has been entered $d(u)$ times, and each of its incident edges has been traversed inward. A contradiction, since u belongs in S. ■

Observe that the loop (Lines 2–10) is applied at most once for every passage, and the number of computational steps in each application of the loop is bounded by a constant. Thus, the time complexity is $O(|E|)$.

3.1.2 The Hopcroft-Tarjan Version of DFS

The Hopcroft and Tarjan version of DFS is essentially the same as Trémaux's, except that they number the vertices from 1 to $n(=|V|)$ in the order in which they are discovered. This is not necessary, as we have seen, for scanning the graph, but the numbering is useful in applying the algorithm for more advanced tasks. Let us denote the number assigned to vertex v by $k(v)$. Also, instead of marking passages, they mark edges as "used," and instead of using the F mark to indicate the edge through which the vertex was discovered and through which it is left for the last time, let us record for each vertex v other than s the vertex $f(v)$ from which v has been discovered. Then $f(v)$ is called the *father* of v; this name will be justified later. DFS is described in Algorithm 3.2.

Since this algorithm is just a simple variation of the previous one, our proof that the whole (connected) graph will be scanned, each edge once in each direction, still applies. Here, in Line 11, if $k(u) \neq 0$, then u is not a new vertex, and v, the center of activity, does not change. This is equivalent to the scanning of e from v to u and back to v, as was done in Trémaux's version. Also, moving our center of activity from v to $f(v)$ (Line 16) corresponds to traversing the edge v —— $f(v)$, in this direction. Thus, the whole algorithm is of time complexity $O(|E|)$, namely, linear in the size of the graph.

Now that we have applied DFS to a finite and connected $G(V, E)$, let us consider the set of edges E', consisting of all edges of the form $f(v)$ —— v through which new vertices have been discovered. Also, direct each such edge from $f(v)$ to v. The digraph (V, E') is called *the DFS tree*. This name is justified by the following Lemma:

Lemma 3.3 *The digraph* (V, E') *defined above is a directed tree with root* s.

```
Procedure DFS(G(V,E),s;k(·),f(·))

1   for every e ∈ E mark e "new"
2   for every u ∈ V do
3       k(u) ← 0
4       f(u) ← NIL
5   v ← s
6   k(s) ← 1
7   i ← 2
8   while v has a new incident edge or f(v) ≠ NIL do
9       if v has a new incident edge v —e— u then do
10          mark e "old"
11          if k(u) = 0 (u is a new vertex) then do
12              f(u) ← v
13              k(u) ← i
14              i ← i + 1
15              v ← u
16      else v ← f(v)
```

Algorithm 3.2: DFS.

Proof: Since $f(s) = \text{NIL}$, $d_{in}(s) = 0$. For every other vertex v, $d_{in}(v) = 1$. Now one can find a directed path

$$s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \cdots v_l = v$$

from s to any vertex v, where for every $1 \leqslant i \leqslant l$, e_i is the directed edge from $f(v_i)$ to v_i, by starting from v and tracing the path backwards. Since for every i, $f(v_i)$ was discovered before v_i, the directed path is simple. Also, every vertex, other than s, has a defined father, and therefore the only vertex in which this backwards search can end is s.

By Theorem 2.5, Part (c), (V, E') is a directed tree. ■

Clearly, if one ignores the edge directions, (V, E') is a spanning tree of G.

In a directed tree, vertex u is called an *ancestor* of v, and v is called a *descendant* of u if there is a directed path from u to v.

The following very useful lemma is due to Hopcroft and Tarjan [4, 6]:

Lemma 3.4 *Let* (V, E') *be the undirected version of a DFS tree,* T*, of* $G(V, E)$*. If an edge* $a \overset{e}{\text{---}} b$ *is in* $E \setminus E'$*, then either* a *is an ancestor of* b *or* a *is a descendant of* b *in* T*.*

Proof: Without loss of generality, assume that $k(a) < k(b)$. In the DFS algorithm, the center of activity (v in the algorithm) moves only along the edges of the tree (V, E'). If b is not a descendant of a, and since a is discovered before b, the center of activity must first move from a to some ancestor of a before it moves up to b. However, we backtrack from a (in Line 16 of Algorithm 3.2) only when all a's incident edges are "old." This means that e is "old." Thus, b is already discovered. A contradiction. ■

Let us call the edges of the DFS tree *tree edges*, and all other edges of the graph, *back edges*. The justification for this name is in Lemma 3.4; every nontree edge connects some vertex back to one of its ancestors.

Consider, as an example, the graph shown in Figure 3.2. We apply DFS to this graph, starting with $s = c$, and assume that we discover vertices d, e, f, g, b, a, in this order. The resulting vertex numbers, tree edges, and back edges are shown in Figure 3.3, where the tree edges are shown as solid lines and are directed from low to high, and the back edges are shown as dashed lines and are directed from high to low. In both cases the direction of the edge indicates the direction in which the edge was scanned first. For tree edges this is the defined direction, and for back edges we can prove it as follows: Assume $v \overset{e}{\text{---}} u$ is a back edge, and u is an ancestor of v. The edge e could not have been scanned first from u, for if v was new at that time, then e would have been a tree edge, and if v has already been discovered (after u), then the center of activity could have been in u only if we have backtracked from v, and this means that e has already been scanned from v.

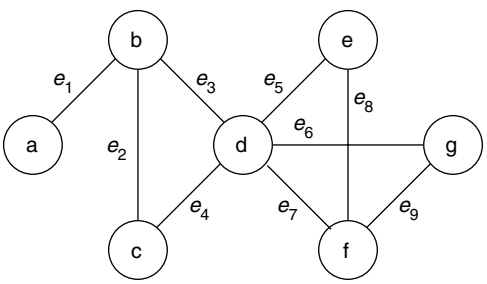


Figure 3.2: A graph to which DFS is applied.

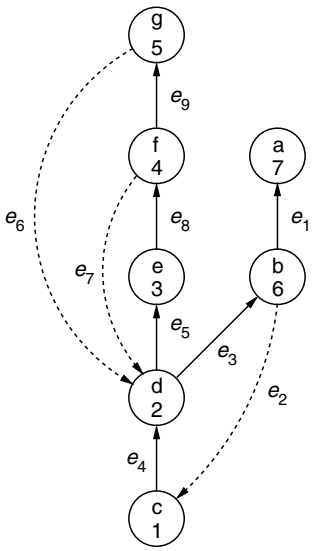


Figure 3.3: The DFS tree edges and back edges.

3.2 Algorithm for Nonseparable Components

A connected graph $G(V, E)$ is said to have a *separation vertex* v (sometimes also called an articulation point) if there exist vertices a and b, distinct from v, such that every path connecting a and b passes through v. In this case, we also say that v separates a from b. A graph that has a separation vertex is called *separable*, and one that has none is called *nonseparable*.

Let $V' \subset V$. The induced subgraph $G'(V', E')$ is called a *nonseparable component* if G' is nonseparable, and if for every $V' \subsetneq V'' \subset V$ the induced subgraph $G''(V'', E'')$ is separable. For example, in the graph shown in Figure 3.2, the subsets $\{a, b\}$, $\{b, c, d\}$ and $\{d, e, f, g\}$ induce the nonseparable components of the graph. If a graph $G(V, E)$ contains no separation vertex, then clearly the whole G is a nonseparable component. However, if v is a separation vertex, then $V \setminus \{v\}$ can be partitioned into $\{V_1, V_2, \ldots, V_k\}$ such that $V_1 \cup V_2 \cup \cdots \cup V_k = V \setminus \{v\}$, and if $i \neq j$, then $V_i \cap V_j = \emptyset$. Also, two vertices a and b are in the same V_i if and only if there is a path connecting them that does not include v. Thus, no nonseparable component can contain vertices from more than one V_i. We can next consider each of the subgraphs induced by $V_i \cup \{v\}$ and continue to partition them into smaller parts, if they are separable. Eventually, we end up with nonseparable parts. This shows that

no two nonseparable components can share more than one vertex because each such vertex is a separating vertex. Also, every simple circuit must lie entirely in one nonseparable component. Now, let us discuss how DFS can help to detect separating vertices.

Let the *lowpoint* of v, $L(v)$, be the least number, $k(u)$, of a vertex u that can be reached from v by a, possible empty, directed path consisting of tree edges, followed by at most one back edge. Clearly $L(v) \leqslant k(v)$, for we can use the empty path from v to itself. Also, if a nonempty path is used, then its last edge is a back edge, for a directed path of tree edges leads to vertices higher than v.

For example, in the graph of Figure 3.2, with the DFS as shown in Figure 3.3, the lowpoints are as follows: $L(a) = 7$, $L(b) = L(c) = L(d) = 1$ and $L(e) = L(f) = L(g) = 2$.

Lemma 3.5 *Let* G *be a graph whose vertices have been numbered by DFS. If* $u \longrightarrow v$ *is a tree edge,* $k(u) > 1$*, and* $L(v) \geqslant k(u)$*, then* u *is a separating vertex of* G.

Proof: Let S be the set of vertices on the path from the root r ($k(r) = 1$) to u, including r but not including u, and let T be the set of vertices on the subtree rooted at v, including v (i.e., all descendants of v, including v itself). By Lemma 3.4, there cannot be any edge connecting a vertex of T with any vertex of $V \setminus (S \cup \{u\} \cup T)$. Also, if there is any edge connecting a vertex $t \in T$ with a vertex $s \in S$, then the edge t —— s is a back edge and clearly $k(s) < k(u)$. Now, $L(v) \leqslant k(s)$, since one can take the tree edges from v to t, followed by t —— s. Thus, $L(v) < k(u)$, contradicting the hypothesis. Thus, u separates the S vertices from the T vertices and is, therefore, a separating vertex. ■

Lemma 3.6 *Let* $G(V, E)$ *be a graph whose vertices have been numbered by DFS. If* u *is a separating vertex, and* $k(u) > 1$*, then there exists a tree edge* $u \longrightarrow v$ *such that* $L(v) \geqslant k(u)$.

Proof: Since u is a separating vertex, there is a partition of $V \setminus \{u\}$ into $V_1, V_2, \ldots, V_m$ such that $m \geqslant 2$, and if $i \neq j$, then all paths from a vertex of V_i to a vertex of V_j, pass through u. Since $k(u) > 1$, the the search has not started in u. Let us assume that it starts in r, and w.l.o.g. $r \in V_1$. The center of activity of the DFS must pass through u. Let $u \longrightarrow v$ be the first tree edge for which $v \notin V_1$. W.l.o.g. assume $v \in V_2$. Since there are no edges connecting vertices of V_2 with vertices of $V \setminus (V_2 \cup \{u\})$, $L(v) \geqslant k(u)$. ■

Lemma 3.7 *Let* $G(V,E)$ *be a graph whose vertices have been numbered by DFS, starting with* r *(*$k(r) = 1$*). Vertex* r *is a separating vertex if and only if there are at least two tree edges out of* r*.*

Proof: Assume r is a separating vertex. Let $V_1, V_2, \ldots, V_m$ be a partition of $V \setminus \{r\}$ such that $m \geqslant 2$, and if $i \neq j$, then all paths from a vertex of V_i to a vertex of V_j, pass through u. Therefore, no path in the tree starting with $r \longrightarrow v$, $v \in V_i$, can lead to a vertex of V_j where $j \neq i$. Thus, there are at least two tree edges out of r.

Now, assume $r \longrightarrow v_1$ and $r \longrightarrow v_2$ are two tree edges out of r. Let T be the set of vertices in the subtree rooted at v_1. By Lemma 3.4, there are no edges connecting vertices of T with vertices of $V \setminus (T \cup \{r\})$. Thus, r separates T from the rest of the graph, which is not empty, since it includes at least the vertex v_2. ■

Let $C_1, C_2, \ldots, C_m$ be the nonseparable components of a connected graph $G(V,E)$, and let $s_1, s_2, \ldots, s_p$ be its separating vertices. Let us define $\tilde{G}(\tilde{V},\tilde{E})$, the *superstructure* of $G(V,E)$, as follows:

$$\tilde{V} = \{s_1, s_2, \ldots, s_p\} \cup \{C_1, C_2, \ldots, C_m\},$$

$$\tilde{E} = \{s_i \text{ —— } C_j \mid s_i \text{ is a vertex of } C_j \text{ in } G\}.$$

By the observations we made in the beginning of the section, $\tilde{G}$ is a tree. By Corollary 2.1, if $m > 1$, then there must be at least two leaves in $\tilde{G}$. However, the degree of a separating vertex s_i in $\tilde{G}$ is greater than 1. Thus, there are at least two *leaf components* in G, each containing only one separating vertex.

By Lemma 3.2, the whole graph will be explored by the DFS. Now, assume the search starts in a vertex r that is not a separating vertex. Even if it is in one of the leaf components, eventually, we will enter another leaf component C, say, via its separating vertex s and an edge $s \longrightarrow v$. By Lemma 3.6, $L(v) \geqslant k(s)$, and if $L(v)$ is known when we backtrack from v to s, then by using Lemma 3.5, we can detect that s is a separating vertex. Also, as far as the component C is concerned, from the time C is entered via $s \longrightarrow v$ until it is entirely explored, we can think of the algorithm as running on C alone, with s as the starting vertex. Thus, by Lemma 3.7, there is only one tree edge from s to other vertices of C, and all other vertices of C are descendants of v and are therefore explored after v is discovered and before the backtrack from v to s. This suggests the use of a stack (pushdown store) for producing the vertices of the component. We store the vertices in the stack in the order that they are discovered. If on backtracking from v to $f(v)$, we discover that $f(v)$ is a separating vertex, we

read off all vertices from the top of the stack down to and including v. All these vertices, plus $f(v)$, (which is not removed at this point from the stack even if it is the next on top) constitute a component. This, in effect, removes the leaf C from the tree $\tilde{G}$, and if its adjacent separating vertex s has now d(s) = 1, then we may assume that it is removed too. The new superstructure is again a tree, and the same process will repeat itself, detecting and trimming one leaf at a time until only one component is left when the DFS terminates.

If the search starts in a separating vertex r, then all but the components containing r are detected and produced as before. All components that contain r, except the last one, are detected by Lemma 3.7: Each time we backtrack into r, on $r \longrightarrow v$, if r still has additional unexplored incident edges, then we conclude that r is a separating vertex, and the vertices on the stack above, including v, plus r, constitute a component.

Finally, when the search is about to end, since we are going to backtrack to r, and r has no new incident edges, all vertices on the stack form the last component, although no separating vertex is discovered at this point.

The remaining problem is that of computing $L(v)$ in time; that is, its value should be known by the time we backtrack from v. If v is a leaf of the DFS tree, then $L(v)$ is the least element in the following set: $\{k(u) \mid u = v$ or v —— u is a back edge$\}$. Let us assign $L(v) = k(v)$ immediately when v is discovered, and as each back edge v —— u is explored, let us assign $L(v) = \min\{L(v), k(u)\}$. Clearly, by the time we backtrack from v, all the back edges have been explored, and $L(v)$ has the right value. If v is not a leaf of the DFS tree, then $L(v)$ is the least element in the following set: $\{k(u) \mid u = v$ or v —— u is a back edge$\} \cup \{L(u) \mid v \longrightarrow u$ is a tree edge$\}$. When we backtrack from v, we have already backtracked from all its sons earlier, and therefore already know their lowpoint. Thus, in addition to what we do for a tree leaf, it suffices to do the following: When we backtrack from u to $v = f(u)$, we assign $L(v) = \min\{L(v), L(u)\}$.

Putting together the ideas described above leads to the algorithm represented in Algorithm 3.3. In this representation, $L(\cdot)$ is the lowpoint function, and S is a stack of vertices. The remaining variables are as in DFS. For the given undirected, connected, and finite graph $G(V, E)$, $|V| > 1$, the algorithm produces the set of separating vertices and a list of its nonseparable components, both of which are assumed to be initially empty. Note that just before the last backtrack into s, Lines 26–27 produce the last nonseparable component, which may be the entire V if G has no separating vertices.

Although this algorithm is more complicated then the original DFS, its time complexity is still $O(|E|)$. This follows easily from the fact that each edge is still scanned exactly once in each direction. The number of operations per edge

```
Procedure NONSEPARABLE(G(V,E), s;set of separating vertices,
list of nonseparable components)

1   for every e ∈ E mark e "new"
2   for every u ∈ V do
3       k(u) ← 0
4       f(u) ← NIL
5   v ← s
6   k(s) ← 1
7   i ← 2
8   vacate S
9   push s into S
10  while v has a new incident edge or f(v) ≠ NIL do
11      if v has a new incident edge v —e— u then do
12          mark e "old"
13          if k(u) = 0 (u is a new vertex) then do
14              push u into S
15              f(u) ← v
16              k(u) ← i
17              L(u) ← i
18              i ← i + 1
19              v ← u
20          else (u is old) do
21              L(v) ← min{L(v), k(u)}
22      else (f(v) is defined)
23          if L(v) ⩾ k(f(v)) then do
24              if f(v) ≠ s or s has a new incident edge, then do
25                  add f(v) to the set of separating vertices
26              pop vertices from S down to and including v
27              the set of popped vertices, with f(v), is an element
                    of the set of nonseparable components
28          else (L(v) < k(f(v))) then do
29              L(f(v)) ← min{L(f(v)), L(v)}
30          v ← f(v)
```

Algorithm 3.3: Using DFS to find the separating vertices and nonseparable components.

is bounded by a constant, except when a nonseparable component is produced. Each vertex is pushed into S once, and popped once. Thus, the total time to produce the components is $O|V|$.

3.3 DFS on Directed Graphs

Running DFS on digraphs is similar to running it on undirected graphs. We start scanning from a new vertex and will scan all vertices (and edges) that can be reached from it via directed paths. A new scan is started from a new vertex, as long as the previous search has left unscanned vertices. Thus, the fact that the whole graph is scanned is trivial.

Assuming the given finite digraph is $G(V, E)$, where $|V| = n$. Again, we assign a number $k(u)$ to every vertex u, where $k : V \mapsto \{1, 2, \ldots, n\}$ is a bijection. If a vertex u is first discovered by scanning an edge $v \longrightarrow u$, then we assign $f(u) = v$. Here, too, v denotes the center of activity.

The algorithm is described in Algorithm 3.4. It is easy to see that the time complexity is $O(|V| + |E|)$.

Let us call a vertex v *ripe* if all its outgoing edges are old and the center of activity is at v. When v is ripe, and if $f(v) \neq \text{NIL}$, then we backtrack to $f(v)$. Otherwise, we return to Line 6 in Algorithm 3.4.

Let us denote by T_u the directed subtree whose root is u, and all vertices that are discovered from the time u is discovered to the time u is ripe are in it.[2] The edges of T_u are of the form $f(w) \xrightarrow{e} w$, where both $f(w)$ and w are in T_u, and w has been discovered via e.

Lemma 3.8 *If vertex w is new when u is discovered, and if at that time there is a directed path from u to w such that all its intermediate vertices (and edges) are new, then w is in T_u.*

Proof: By contradiction. Assume that w is as in the premise of the Lemma, but w is not in T_u. Let P be a directed path from u to w such that all its intermediate vertices are new when u is discovered, and let b be the first vertex on P that does not belong to T_u. Let $a \xrightarrow{e} b$ be the edge on P that enters b.

Since $a \in T_u$, and it eventually becomes ripe, e must have been investigated, as in Line 11 of Algorithm 3.4, and b has been discovered and belongs to T_u. A contradiction. ■

[2] Note that u is not necessarily a root of a search in the sense that it may not have been picked in Line 7, and may belong to some T_v for $v \neq u$.

```
Procedure Directed-DFS(G(V,E), s; k(·), f(·))

1   for every e ∈ E mark e "new"
2   for every u ∈ V do
3       k(u) ← 0
4       f(u) ← NIL
5   i ← 1
6   while there is a vertex u for which k(u) = 0 do
7       let v be such a vertex
8       k(v) ← i
9       i ← i + 1
10      while v has a new outgoing edge or f(v) ≠ NIL do
11          if v has a new outgoing edge v --e--> w, then do
12              mark e "old"
13              if k(w) = 0 (u is a new vertex) then do
14                  f(w) ← v
15                  k(w) ← i
16                  i ← i + 1
17                  v ← w
18          else v ← f(v)
```

Algorithm 3.4: DFS on a directed graph.

3.4 Strongly Connected Components of a Digraph

Let $G(V,E)$ be a finite digraph. Let us define the relation $\sim$, a subset of $V \times V$, in the following way: For $x, y \in V$, $x \sim y$ if there is a directed path from x to y and also a directed path from y to x.

The relation $\sim$ is easily seen to be reflexive, symmetric, and transitive. Thus, it is an equivalence relation. An equivalence class of this relation is called a *strongly connected component* or, in short, a *strong component*, and if there is only one equivalence class, then G is said to be *strongly connected*.

The *super-structure* of G, $\tilde{G}(\tilde{V},\tilde{E})$ is a digraph constructed as follows:

- $\tilde{V}$ is the set of strong components of V.
- $\tilde{E} = \{X \xrightarrow{e} Y \mid$ there exists an edge $x \xrightarrow{e} y$ in E such that $x \in X$ and $y \in Y\}$.

Observe that $\tilde{G}$ is a DAG (directed acyclic graph). A strong component C is called a *source* if there are no edges that enter C in $\tilde{G}$. A *sink* component is similarly defined.

Given a digraph $G(V, E)$, our purpose is to describe an efficient algorithm for finding its strong components. The first linear-time algorithm that achieved this goal was presented by Tarjan [6]. However, we present an algorithm attributed to Kosaraju and Sharir [1], which is simpler to explain.[3]
The algorithm consists of three phases:

Phase 1: Run a DFS-A on G, in which vertices are numbered $h : V \mapsto \{1, 2, \ldots, n\}$ in the order in which they become ripe. This is described in Algorithm 3.5.
Phase 2: Reverse the direction of all edges of $G(V, E)$ to obtain $G^R(V, E^R)$.
Phase 3: Run a DFS-B on G^R, where each time a new search begins, it is started in the new vertex v for which $h(v)$ is maximum; the set of vertices found in a search is declared to be a strong component of G. This is described in Algorithm 3.6.

Observe that the strong components of G^R are identical to those of G. Also, the algorithm consists of three phases, each of time complexity $O(|V| + |E|)$. Thus, the whole algorithm is of time complexity $O(|V| + |E|)$.

Lemma 3.9 *The function $h(\cdot)$, produced by DFS-A, satisfies the following condition: For every $u \in V$,*

$$h(u) = \max_{w \in T_u} \{h(w)\}.$$

Proof: Follows from the fact that u is the last vertex in T_u to become ripe. ■

Lemma 3.10 *Let C be a strong component of G, and let u be the vertex of C for which $h(u)$ is maximum. It follows that u is the first vertex of C to be discovered in DFS-A.*

Proof: If a is the first vertex of C to be discovered in DFS-A, then by Lemma 3.8, $u \in T_a$. By Lemma 3.9, $h(a) \geqslant h(u)$. Since h is a bijection, and $h(u)$ is maximum in C, it follows that $a = u$. ■

Corollary 3.1 *If the premise of Lemma 3.10 holds, then all vertices of C are in T_u.*

[3] The algorithm of Tarjan uses DFS once, while that of Kosaraju and Sharir uses two runs of DFS.

```
Procedure DFS-A(G(V,E);h(·))

1   for every e ∈ E mark e "new"
2   for every u ∈ V do
3       f(u) ← NIL
4       mark u "new"
5   i ← 1
6   while there are new vertices do
7       let v be such a vertex
8       mark v "old"
9       while v has new outgoing edges or f(v) ≠ NIL do
10          if v has new outgoing edges, then do
11              let v —e→ w be a new edge
12              mark e "old"
13              if w is new then do
14                  mark w "old"
15                  f(w) ← v
16                  v ← w
17          else (f(v) ≠ NIL) then do
18              h(v) ← i
19              i ← i + 1
20              v ← f(v)
21      h(v) ← i
22      i ← i + 1
```

Algorithm 3.5: DFS-A.

Lemma 3.11 *Assume DFS-A was applied to* G *and vertex* v *was assigned the highest value of* $h(v)$. *The strong component* C *to which* v *belongs is a source component.*

Proof: By contradiction. Assume there is an edge $a \xrightarrow{e} b$, such that $a \notin C$ and $b \in C$.

Let r be the root, chosen in Line 7, of the search in which a is discovered. Then $r \notin C$, for otherwise, there would be a directed path from C, which starts in r, to a, to b, and thus, a belongs to C, contradicting the assumption that $a \notin C$. We conclude that $r \neq v$.

```
Procedure DFS-B(G^R(V, E^R), h(·); strong components of G)

1   for every e ∈ E^R mark e "new"
2   for every u ∈ V do
3       f(u) ← NIL
4       mark u "new"
5   while there are new vertices do
6       let v be the new vertex for which h(v) is maximum
7       S ← {v}
8       mark v "old"
9       while v has new outgoing edges or f(v) ≠ NIL do
10          if v has new outgoing edges, then do
11              let v —e→ w be a new edge
12              mark e "old"
13              if w is new then do
14                  mark w "old"
15                  f(w) ← v
16                  S ← S ∪ {w}
17                  v ← w
18          else (f(v) ≠ NIL) then do
19              v ← f(v)
20      print: "The set S is a strong component"
```

Algorithm 3.6: DFS-B.

Observe that the last vertex to serve as a root of a search is the vertex for which $h(\cdot)$ is maximum. Thus, v is a root of a search. Also, T_r is constructed before v is discovered. But when r is chosen to be a root of a search, all vertices on the path from r to a, to b, to v, are new. Thus, by Lemma 3.8, $v \in T_r$, contradicting the fact that v is a root of a new search. ■

Since the component C, containing the vertex v for which $h(v)$ is maximum, is a source component of G, it is a sink component of G^R. It follows that in DFS-B, since the root of the first search is v, all vertices of C, and none else, are discovered in this search, and indeed, this set S is declared to be a strong component.

```
Procedure TARRY(G, s)

1   v ← s
2   while there is an unmarked passage in v or v has a passage marked F do
3       if there is an unmarked passage to edge v —e— u then do
4           mark the passage of e at v by E
5           if u has no marked passages then do
6               mark the passage of e at u by F
7           v ← u
8       else (there is a passage in v marked F) do
9           use the passage marked F to move to the neighboring vertex u
10          v ← u
```

Algorithm 3.7: The Tarry algorithm.

Now, assume we remove from G all vertices of the first declared component and their incident (outgoing) edges to form a directed subgraph G'. The remaining directed subforest, of the original DFS-A forest of G, and the values of $h(\cdot)$ for the remaining vertices are legitimate in the sense that one can run DFS-A on G' to yield exactly this forest and assignments of $h(\cdot)$. It follows that the next declared strong component is valid too. By induction, all declared strong components are valid as well.

3.5 Problems

Problem 3.1 Tarry's algorithm [7] is like Trémaux's, with the following change. Upon reaching an old vertex u, one moves the center of activity to that vertex anyway, instead of insisting on marking the passage through which one has just reached u by E and not moving the center of activity to u. Tarry's algorithm is described in Algorithm 3.7. Prove that Tarry's algorithm terminates after all edges of G have been traversed, once in each direction.

Problem 3.2 Consider the set of edges which upon termination of Tarry's algorithm (see Problem 3.1) have one endpoint marked E and the other marked F. Also, assume these edges are now directed from E to F.

(i) Prove that this set of edges form a directed spanning tree of G, with root s.
(ii) Does a statement like that of Lemma 3.4 hold in this case? Prove or disprove.

Problem 3.3 Fraenkel [2, 3] showed that the number of edge traversals can sometimes be reduced, in comparison to the Tarry algorithm (see Problem 3.1), if the use of a two-way counter is allowed. Each time a new vertex is entered, the counter is incremented. When it is realized that all incident edges of a vertex have been traversed at least in one direction, the counter is decremented. If the counter reaches the start value, the search is terminated. (One can return to s via the passages marked F.)

Write an algorithm that realizes this idea. (Hint: An additional mark that temporarily marks the passages used to reenter a vertex is used.) Prove the validity of your algorithm. Show that for some graphs the algorithm will traverse each edge exactly once for others; the savings depends on the choice of passages. Yet there are graphs for which the algorithm can save only one traversal, even if we do not insist on returning to s.

Problem 3.4 Assume that G is drawn in the plane in such a way that no two edges cross. Show how Trémaux's algorithm can be modified in such a way that the whole scanning path never crosses itself.

Problem 3.5 In an undirected graph G, a set of vertices K is called a *clique* if every two vertices of K are connected by an edge. Prove that, in the spanning (directed) tree resulting from running DFS on a finite and connected G, all vertices of a clique appear on one directed path. Do they necessarily appear consecutively on the path? Justify your answer.

Problem 3.6 Prove or disprove the following claim: If C is a simple circuit in an undirected, finite, and connected graph G to which DFS is applied, then all vertices of C appear on one directed path of the (directed) DFS tree. Does you answer change if C is an induced circuit? (A circuit C is an *induced circuit* in G if G does not contain an edge between two nonadjacent vertices in C.)

Problem 3.7 Prove that if C is a directed circuit of a finite digraph to which DFS is applied and v is a vertex on C for which $k(v)$ is minimum, then v is a root of a subtree of the resulting directed forest, and all vertices of C are in this subtree.

Problem 3.8 An edge e of a connected undirected graph G is called a *bridge* if the deletion of e destroys G's connectivity. Describe an algorithm to compute all bridges of a given finite G. (Hint: There are two ways to solve the problem. In the first solution, one modifies the graph and uses the algorithm for finding the separating vertices. In the second, one modifies the algorithm and applies the modified algorithm to G.)

Problem 3.9 (This problem was suggested by Silvio Micali.) Let G be a connected graph.

- Prove that a vertex $u \neq s$ is a separating vertex of G if and only if, upon termination of DFS on G, there is a tree edge $u \longrightarrow v$ for which there is no back edge $x \longrightarrow y$ such that x is a descendant of v and y is a proper ancestor of u.
- Describe an algorithm, of time complexity $O(|E|)$, which detects the separating vertices of G without numbering the vertices and, therefore, without the use of *lowpoint*. (Hint: For every back edge $x \longrightarrow y$, mark all tree edges on the path from y to x; proceed from x to y until an already marked edge is encountered or the edge tree outgoing from y is reached. The latter edge is not marked. The back edges are considered by rescanning G again, while remembering which edges are tree edges and which are back edges, only that in the second scan the back edges are processed first. When all this is over, $v \neq s$ is a separating vertex if and only if there is an unmarked tree edge out of v. Also note that a third scan can be used to produce the nonseparable components.)

Problem 3.10 (This problem was suggested by Alessandro Tescari.) Show that the algorithm for nonseparable components can be simplified by adding a new vertex r and a new edge r — s, and starting the search at r. Do we still need Lemma 3.7?

Bibliography

[1] J.E. Hopcroft, A.V. Aho, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] A.S. Fraenkel. "Economic traversal of labyrinths." *Math. Mag.*, 43(1):125–130, 1970.

[3] A.S. Fraenkel. "Economic traversal of labyrinths (correction)." *Math. Mag.*, 44, 1971.

[4] J. Hopcroft and R. Tarjan. "Algorithm 447: Efficient algorithms for graph manipulation." *Comm. of the ACM*, 16:372–378, 1973.

[5] E. Lucas. *Récreations Mathématiques*. Paris, 1882.

[6] R. Tarjan. "Depth-first search and linear graph algorithms." *SIAM J. on Computing*, 1:146–160, 1972.

[7] G. Tarry. "Le problème des labyrinthes." *Nouvelles Annales de Math.*, 14:187, 1895.

4

Ordered Trees

4.1 Uniquely Decipherable Codes

Let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. We call Σ an *alphabet* and its elements are called *letters*; the number of letters in Σ is σ. (Except for this numerical use of σ, the "numerical" value of the letters is ignored; they are just "meaningless" characters. We use the numerals just because they are convenient characters.) A finite sequence $a_1 a_2 \cdots a_l$, where a_i is a letter, is called a *word* whose *length* is l. We denote the length of a word w by $l(w)$. A set of (nonempty and distinct) words is called a *code*. For example, the code $\{102, 21, 00\}$ consists of three code-words: one code-word of length 3 and two code-words of length 2; the alphabet is $\{0, 1, 2\}$ and consists of three letters. Such an alphabet is called "ternary".

Let $c_1, c_2, \ldots, c_k$ be code-words. The *message* $c_1 c_2 \cdots c_k$ is the word resulting from the concatenation of the code-word c_1 with c_2, and so on. For example, if $c_1 = 00$, $c_2 = 21$, and $c_3 = 00$, then $c_1 c_2 c_3 = 002100$.

A code C over Σ (i.e., the code-words of C consist of letters in Σ) is said to be *uniquely decipherable* (UD) if every message constructed from code-words of C can be broken down into code-words of C in only one way. For example, the code $\{01, 0, 10\}$ is not UD because the message 010 can be parsed in two ways: $0, 10$ and $01, 0$.

Our first goal is to describe a test for deciding whether a given code C is UD. This test is an improvement on a test of Sardinas and Patterson [1] and can be found in Gallager's book [2].

If s, p, and w are words, and $ps = w$, then p is is called a *prefix* of w and s is called a *suffix* of w. We say that a word w is nonempty if $l(w) > 0$.

A nonempty word t is called a *tail* if there exist two messages $c_1 c_2 \cdots c_m$ and $c'_1 c'_2 \cdots c'_n$ with the following properties:

(1) $c_i, 1 \leqslant i \leqslant m$, and $c'_j, 1 \leqslant j \leqslant n$ are code-words, and $c_1 \neq c'_1$;
(2) t is a suffix of c'_n;
(3) $c_1c_2\cdots c_m t = c'_1c'_2\cdots c'_n$.

Lemma 4.1 *A code C is UD if and only if no tail is a code-word.*

Proof: If a code-word c is a tail then, by definition, there exist two messages $c_1c_2\cdots c_m$ and $c'_1c'_2\cdots c'_n$ that satisfy $c_1c_2\cdots c_m c = c'_1c'_2\cdots c'_n$, while $c_1 \neq c'_1$. Thus, there are two different ways to parse this message, and C is not UD.

If C is not UD, then there exist messages that can be parsed in more than one way. Let μ be such an ambiguous message, whose length is minimum: $\mu = c_1c_2\cdots c_k = c'_1c'_2\cdots c'_n$; i.e. all the c_i-s and c_j-s are code-words and $c_1 \neq c'_1$. Now, without loss of generality we can assume that c_k is a suffix of c'_n (or change sides). Thus, c_k is a tail. ■

The following algorithm generates all the tails. If a code is a tail, the algorithm terminates with a negative answer.

Algorithm for UD:

(1) For every two code-words, c_i and c_j $(i \neq j)$, do the following:
   a. If $c_i = c_j$, halt; C is not UD.
   b. If for some word s, either $c_i s = c_j$ or $c_i = c_j s$, put s in the set of tails.
(2) For every tail t and every code-word c do the following:
   a. If $t = c$, halt; C is not UD.
   b. If some word s, either $ts = c$ or $cs = t$, put s in the set of tails.
(3) Halt; C is UD.

Clearly, in Step (1), the words declared to be tails are indeed tails. In step (2), since t is already known to be a tail, there exist code-words $c_1, c_2, \ldots, c_m$ and $c'_1, c'_2, \ldots, c'_n$ such that $c_1c_2\cdots c_m t = c'_1c'_2\cdots c'_n$. Now, if $ts = c$, then $c_1c_2\cdots c_m c = c'_1c'_2\cdots c'_n s$, and therefore s is a tail; and if $cs = t$, then $c_1c_2\cdots c_m cs = c'_1c'_2\cdots c'_n$ and s is a tail.

Next, if the algorithm halts in (3), we want to show that all the tails have been produced. Once this is established, it is easy to see that the conclusion that C is UD follows; each tail has been checked, in Line (2a.), whether it is equal to a code-word, and no such equality has been found. By lemma 4.1, the code C is UD.

For every t let $m(t) = c_1c_2\cdots c_m$ be a shortest message such that $c_1c_2\cdots c_m t = c'_1c'_2\cdots c'_n$ and t is a suffix of c'_n. We prove by induction on

the length of $m(t)$ that t is produced. If $m(t) = 1$, then t is produced by ((1)b.), since $m = n = 1$.

Now assume that all tails p for which $m(p) < m(t)$ have been produced. Since t is a suffix of c'_n, let p denote the word such that $pt = c'_n$. Therefore, $c_1 c_2 \cdots c_m = c'_1 c'_2 \cdots c'_{n-1} p$, and p is a tail.

If $p = c_m$, then $c_m t = c'_n$, and t is produced in Step (1).

If p is a suffix of c_m, then p is a tail. Also, $m(p)$ is shorter then $m(t)$. By the inductive hypothesis, p has been produced. When Step (2b) is applied to $pt = c'_n$ (with p as a tail and c'_n as a code-word), the tail t is produced.

If c_m is a suffix of p, then $c_m t$ is a suffix of c'_n, and therefore, $c_m t$ is a tail. Also, $m(c_m t) = c_1 c_2 \cdots c_{m-1}$ and is shorter than $m(t)$. By the inductive hypothesis, $c_m t$ has been produced. When Step (2b) is applied to the tail $c_m t$ and code-word c_m, the tail t is produced.

This proves that the algorithm halts with the right answer.

Let the code consist of n words, and l be the maximum length of a code-word. Step (1) takes at most $O(n^2 \cdot l)$ elementary operations. The number of tails is at most $O(n \cdot l)$. Thus, Step (2) takes at most $O(n^2 l^2)$ elementary operations. Therefore, the whole algorithm is of time complexity $O(n^2 l^2)$. Other algorithms of the same complexity can be found in [3] and [4]; these tests are extendible to test for additional properties [5, 6, 7].

Theorem 4.1 *Let* $C = \{c_1, c_2, \ldots, c_n\}$ *be a UD code over an alphabet of* σ *letters. If* $l_i = l(c_i)$, $i = 1, 2, \ldots, n$, *then*

$$\sum_{i=1}^{n} \sigma^{-l_i} \leqslant 1. \tag{4.1}$$

The left-hand side of 4.1 is called the *characteristic sum* of C; clearly, it characterizes the vector $(l_1, l_2, \ldots, l_n)$, rather than C. The inequality 4.1 is called the *characteristic sum condition*. The theorem was first proved by McMillan [8]. The following proof is due to Karush [9].

Proof: Let e be a positive integer

$$\left(\sum_{i=1}^{n} \sigma^{-l_i} \right)^e = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_e=1}^{n} \sigma^{-(l_{i_1} + l_{i_2} + \cdots + l_{i_e})}.$$

There is a unique term, on the right-hand side, for each of the n^e messages of e code-words. Let us denote by $N(e, j)$ the number of messages of e code-words

whose length is j. It follows that

$$\sum_{i_1=1}^{n}\sum_{i_2=1}^{n}\cdots\sum_{i_e=1}^{n}\sigma^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_e})}=\sum_{j=e}^{e\hat{l}}N(e,j)\cdot\sigma^{-j},$$

where $\hat{l}$ is the maximum length of a code-word. Since C is UD, no two messages can be equal. Thus, $N(e,j)\leqslant\sigma^{j}$. We now have

$$\sum_{j=e}^{e\hat{l}}N(e,j)\cdot\sigma^{-j}\leqslant\sum_{j=e}^{e\hat{l}}\sigma^{j}\cdot\sigma^{-j}\leqslant e\cdot\hat{l}.$$

We conclude that for all $e\geqslant 1$,

$$\Big(\sum_{i=1}^{n}\sigma^{-l_i}\Big)^{e}\leqslant e\cdot\hat{l}.$$

This implies 4.1. ■

A code C is said to be *prefix* if no code-word is a prefix of another. For example, the code $\{00,10,11,100,110\}$ is not prefix, since 10 is a prefix of 100; the code $\{00,10,11,010,011\}$ is a prefix. A prefix code has no tails and is therefore UD. In fact, it is very easy to parse the messages: Reading the messages from left to right, as soon as we read a code-word, we know that it is the first code-word of the message, since it cannot be the beginning of another code-word. Therefore, in most applications, prefix codes are used. The following theorem, due to Kraft [10], in a sense shows us that we do not need nonprefix codes.

Theorem 4.2 *If the vector of integers,* $(l_1,l_2,\ldots,l_n)$*, satisfies*

$$\sum_{i=1}^{n}\sigma^{-l_i}\leqslant 1, \tag{4.2}$$

then there exists a prefix code $C=\{c_1,c_2,\ldots,c_n\}$ *over the alphabet of* σ *letters such that* $l_i=l(c_i)$.

Proof: Let $\lambda_1<\lambda_2<\cdots<\lambda_m$ be integers such that each l_i is equal to one of the λ_j-s and each λ_j is equal to at least one of the l_i-s. Let k_j be the number of

l_j-s that are equal to λ_j. We have to show that there exists a prefix code C such that the number of code-words of length λ_j is k_j.

Clearly, 4.2 implies that

$$\sum_{j=1}^{m} k_j \sigma^{-\lambda_j} \leqslant 1 \tag{4.3}$$

We prove by induction on r that for every $1 \leqslant r \leqslant m$ there exists a prefix code C_r such that, for every $1 \leqslant j \leqslant r$, the number of its code-words of length λ_j is k_j.

First assume that $r = 1$. Inequality 4.3 implies that $k_1 \sigma^{-\lambda_1} \leqslant 1$, or $k_1 \leqslant \sigma^{\lambda_1}$. Since there are σ^{λ_1} distinct words of length λ_1, we can assign any k_1 of them to constitute C_1.

Now, assume C_r exists. If $r < m$ then 4.3 implies that

$$\sum_{j=1}^{r+1} k_j \sigma^{-\lambda_j} \leqslant 1.$$

Multiplying both sides by $\sigma^{\lambda_{r+1}}$ yields

$$\sum_{j=1}^{r+1} k_j \sigma^{\lambda_{r+1}-\lambda_j} \leqslant \sigma^{\lambda_r+1},$$

which is equivalent to

$$k_{r+1} \leqslant \sigma^{\lambda_{r+1}} - \sum_{j=1}^{r} k_j \sigma^{\lambda_r+1-\lambda_j}. \tag{4.4}$$

The number of distinct words of length λ_{r+1}, a prefix of which of length λ_j is a code-word in C_r, equals $k_j \cdot \sigma^{\lambda_r+1-\lambda_j}$. Thus, 4.4 implies that among the $\sigma^{\lambda_{r+1}}$ words of length λ_{r+1}, there are at least k_{r+1} words, none of which has a prefix in C_r. The enlarged set of code-words is C_{r+1}. ■

This proof suggests an algorithm for the construction of a code with a given vector of code-word length. We return later to the question of prefix code construction, but first we introduce positional trees.

4.2 Positional Trees and Huffman's Optimization Problem

A *positional* σ*-tree* (or when σ is known, a positional tree) is a directed tree with the following property: Each edge out of a vertex v is associated with one

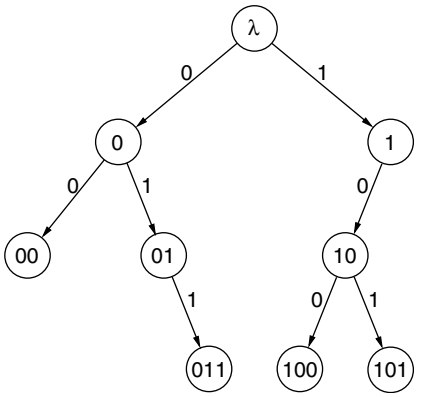


Figure 4.1: A positional 2-tree.

of the letters of the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$; different edges, out of v, are associated with different letters. It follows that the number of edges out of a vertex is at most σ, but may be less; in fact, a leaf has none.

We associate with each vertex v the word consisting of the sequence of letters associated with the edges on the path from the root r to v. For example, consider the binary tree (positional 2-tree) of Figure 4.1, where the associated word is written in each vertex. (λ denotes the empty word.)

Clearly, the set of words associated with the leaves of a positional tree is a prefix code. Also, every prefix code can be described by a positional tree in this way.

The *level* of a vertex v of a tree is the length of the directed path from the root to v; it is equal to the length of the word associated with v.

Our next goal is to describe a construction of an optimum code in a sense that we discuss shortly. It is described here as a communication problem, as it was viewed by Huffman [11], who solved it. In the next section, we shall describe one more application of this optimization technique.

Assume that words over a source alphabet of n letters have to be transmitted over a channel that can transfer one letter of the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ at a time, and $\sigma < n$. We want to construct a code over Σ with n code-words and associate a code-word with each source letter. A word over the source alphabet is translated into a message over the code, by concatenating the code-words that correspond to the source letters in the same order as they appear in the source word. This message can now be transmitted through the channel. Clearly, the code must be UD.

Assume further that the source letters have given probabilities $p_1, p_2, \ldots p_n$ of appearance and that the choice of the next letter in the source word is independent of its previous letters. If the vector of code-word lengths is $(l_1, l_2, \ldots, l_n)$, then the average code-word length, $\bar{l}$, is given by

$$\bar{l} = \sum_{i=1}^{n} p_i l_i \ . \tag{4.5}$$

We want to find a code for which $\bar{l}$ is minimum in order to minimize the expected length of the message.

Since the code must be UD, by Theorem 4.1, the vector of code-word lengths must satisfy the characteristic sum condition. This implies, by Theorem 4.2, that a prefix code with the same vector of code-word lengths exists. Therefore, in seeking an optimum code, for which $\bar{l}$ is minimum, we may restrict our search to prefix codes. In fact, all we have to do is find a vector of code-word lengths for which $\bar{l}$ is minimum, among the vectors that satisfy the characteristic sum condition.

First, let us assume that $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$. This is easily achieved by sorting the probabilities. We first demonstrate Huffman's construction for the binary case $(\sigma = 2)$. Assume the probabilities are 0.6, 0.2, 0.05, 0.05, 0.03, 0.03, 0.03, 0.01. We write this list as our top row (see Figure 4.2). We add the last (and therefore least) two numbers, and insert the sum in the proper place to maintain the nonincreasing order. We repeat this operation until we get a vector with only two probabilities. Now, we assign each probability a word-length 1, and start working our way back up by assigning each of the probabilities of the previous step its length in the present step, if it is not one of the last two, and each of the two last probabilities of the previous step is assigned a length that is larger by one than the length assigned to their sum in the present step.

Once the vector of code-word lengths is found, a prefix code can be assigned to it by the technique of the proof of Theorem 4.2. (An efficient implementation is discussed in Problem 4.6.) Alternatively, the back-up procedure can produce a prefix code directly. Instead of assigning lengths to the last two probabilities, we assign the two words of length one: 0 and 1. As we back up from a present step in which a word is already assigned to each probability, to the previous step, the rule is as follows: All but the last two probabilities of the previous step are assigned the same words as in the present step. The last two probabilities are assigned c0 and c1, where c is the word assigned to their sum in the present step.

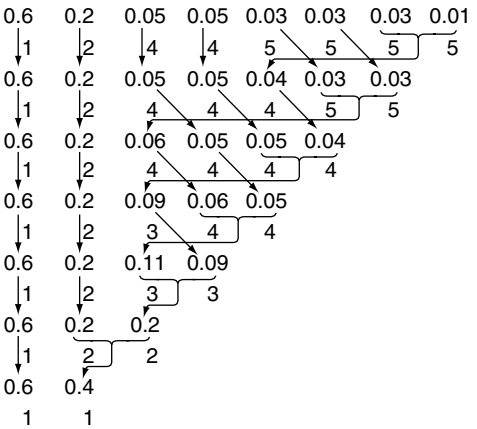


Figure 4.2: A demonstration of Huffman's construction for the binary case.

In the general case, when $\sigma \geqslant 2$, we add in each step the last d probabilities of the present vector of probabilities; if n is the number of probabilities of this vector then d is given by

$$1 < d \leqslant \sigma \quad \text{and} \quad n \equiv d \mod (\sigma - 1) \tag{4.6}$$

After the first step, the length of the vector, n', satisfies $n' \equiv 1 \mod (\sigma-1)$, and will be equal to one, mod $(\sigma - 1)$, from there on. The reason for this rule is that we should end up with exactly σ probabilities, each to be assigned length 1. Now, $\sigma \equiv 1 \mod (\sigma-1)$, and since in each ordinary step the number of probabilities is reduced by $\sigma - 1$, we want $n \equiv 1 \mod (\sigma-1)$. In case this condition is not satisfied by the given n, we correct it in the first step as is done by our rule. Our next goal is to prove that this indeed leads to an optimum assignment of a vector of code-word lengths.

Lemma 4.2 *If* $C = \{c_1, c_2, \ldots, c_n\}$ *is an optimum prefix code for the probabilities* $p_1, p_2, \ldots, p_n$, *then* $p_i > p_j$ *implies that* $l(c_i) \leqslant l(c_j)$.

Proof: Assume $l(c_i) > l(c_j)$. Make the following switch: Assign c_i to probability p_j, and c_j to p_i; all other assignments remain unchanged. Let $\tilde{l}$ denote the average code-word length of the new assignment, while $\bar{l}$ denotes the previous

one. By (4.5), we have

$$\begin{aligned}\bar{l} - l &= [p_i \cdot l(c_i) + p_j \cdot l(c_j)] - [p_i \cdot l(c_j) + p_j \cdot l(c_i)] \\ &= (p_i - p_j)(l(c_i) - l(c_j)) > 0\end{aligned}$$

contradicting the assumption that $\bar{l}$ is minimum. ■

Lemma 4.3 *There exists an optimum prefix code for the probabilities* $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$ *such that the positional tree that represents it has the following properties:*

(1) *All the internal vertices of the tree, except possibly one internal vertex* v*, have exactly* σ *sons.*
(2) *Vertex* v *has* $1 < \rho \leqslant \sigma$ *sons, where* $n \equiv \rho \mod (\sigma - 1)$.
(3) *Vertex* v *of ((1)) is on the lowest level that contains internal vertices, and its sons are assigned to* $p_{n-\rho+1}, p_{n-\rho+2}, \ldots, p_n$.

Proof: Let T be a positional tree representing an optimum prefix code. If there exists an internal vertex u that is not on the lowest level of T containing internal vertices and it has less than σ sons, then we can perform the following change in T: Remove one of the leaves of T from its lowest level and assign to the probability a new son of u. The resulting tree, and therefore its corresponding prefix code, has a smaller average code-word length. A contradiction. Thus, we conclude that no such internal vertex u exists.

If there are internal vertices on the lowest level of internal vertices that have less than σ sons, choose one of them, say v. Now eliminate sons from v and attach their probabilities to new sons of the others, so that their number of sons is σ. Clearly, such a change does not change the average length, and the tree remains optimum. If, before filling in all the missing sons, v has no more sons, we can use v as a leaf and assign to it one of the probabilities from the lowest level, thus creating a new tree that is better than T. A contradiction. Thus, we never run out of sons of v to be transferred to other lacking internal vertices on the same level. Also, when this process ends, v is the only lacking internal vertex (proving (1)), and its number of remaining sons must be greater than one, or its son can be removed and its probability attached to v. This proves that the number of sons of v, ρ, satisfies $1 < \rho \leqslant \sigma$.

If v's ρ sons are removed, the new tree has $n' = n - \rho + 1$ leaves and is *full* (i.e., every internal vertex has exactly σ sons). In such a tree, the number of leaves, n', satisfies $n' \equiv 1, \mod (\sigma - 1)$. This is easily proved by induction on

the number of internal vertices. Thus, $n-\rho+1\equiv 1 \mod(\sigma-1)$, and therefore $n\equiv\rho \mod(\sigma-1)$, proving (2).

We have already shown that v is on the lowest level of T that contains internal vertices, and the number of its sons is ρ. By Lemma 4.2, we know that the least ρ probabilities are assigned to leaves of the lowest level of T. If they are not sons of v, we can exchange sons of v with sons of other internal vertices on this level, to bring all the least probabilities to v without changing the average length. ■

For a given alphabet size σ and probabilities $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$, let $\theta_\sigma(p_1,p_2,\ldots,p_n)$ be the set of all σ-ary positional trees with n leaves, assigned with the probabilities $p_1,p_2,\ldots,p_n$ in such a way that $p_{n-d+1},p_{n-d+2},\ldots,p_n$ (see Equation 4.6) are assigned, in this order, to the first d sons of a vertex v, which has no other sons. By Lemma 4.3, $\theta_\sigma(p_1,p_2,\ldots,p_n)$ contains at least one optimum tree. Thus, we may restrict our search for an optimum tree to $\theta_\sigma(p_1,p_2,\ldots,p_n)$.

Lemma 4.4 *There is a one to one correspondence between* $\theta_\sigma(p_1,p_2,\ldots,p_n)$ *and the set of* σ*-ary positional trees, with* $n-d+1$ *leaves assigned with* $p_1,p_2,\ldots,p_{n-d},p'$, *where* $p'=\sum_{i=n-d+1}^{n} p_i$. *The average word-length* $\bar{l}$ *of the prefix code, represented by a tree* T *of* $\theta_\sigma(p_1,p_2,\ldots,p_n)$, *and the average code-word-length* $\bar{l}'$ *of the prefix code represented by the tree* T', *which corresponds to* T, *satisfy*

$$\bar{l}=\bar{l}'+p'. \tag{4.7}$$

Proof: The tree T' which corresponds to T is achieved as follows: Let v be the father of the leaves assigned $p_{n-d+1},p_{n-d+2},\ldots,p_n$. Remove all the sons of v and assign p' to it.

It is easy to see that two different trees T_1 and T_2 in $\theta_\sigma(p_1,p_2,\ldots,p_n)$ will yield two different trees T_1' and T_2', and that every σ-ary tree T' with $n-d+1$ leaves assigned $p_1,p_2,\ldots,p_{n-d},p'$, is the image of some T; establishing the correspondence.

Let l_i denote the level of the leaf assigned p_i in T. Clearly $l_{n-d+1}=l_{n-d+2}=\cdots=l_n$. Thus,

$$\begin{aligned}\bar{l}&=\sum_{i=1}^{n-d} p_i\cdot l_i+l_n\cdot\sum_{i=n-d+1}^{n} p_i=\sum_{i=1}^{n-d} p_i\cdot l_i+l_n\cdot p'\\&=\sum_{i=1}^{n-d} p_i\cdot l_i+(l_n-1)\cdot p'+p'=\bar{l}'+p'.\end{aligned}$$

■

Lemma 4.4 suggests a recursive approach to find an optimum T. For $\bar{l}$ to be minimum, $\bar{l}'$ must be minimum. Thus, let us first find an optimum T', and then find T by attaching d sons to the vertex of T' assigned p'; these d sons are assigned $p_{n-d+1}, p_{n-d+2}, \ldots, p_n$. This is exactly what is done in Huffman's procedure, thus proving its validity.

It is easy to implement Huffman's algorithm in time complexity $O(n^2)$. First, we sort the probabilities, and after each addition, the resulting probability is inserted in a proper place. Each such insertion takes at most $O(n)$ steps, and the number of insertions is $\lceil (n-\sigma)/(\sigma-1) \rceil$. Thus, the whole forward process is of time complexity $O(n^2)$. The back up process is $O(n)$ if pointers are left in the forward process to indicate the probabilities of which it is composed.

However, the time complexity can be reduced to $O(n \log n)$. One way of doing it is the following: First sort the probabilities. This can be done in $O(n \log n)$ steps [14]. The sorted probabilities are put on a queue S_1 in a non-increasing order from left to right. A second queue, S_2, initially empty, is used too. In the general step, we repeatedly take the least probability of the two (or one, if one of the queues is empty) appearing at the right hand side ends of the two queues, and add up d of them. The result, p', is inserted at the left hand side end of S_2. The process ends when after adding d probabilities both queues are empty. This adding process and the back up are $O(n)$. Thus, the whole algorithm is $O(n \log n)$.

The construction of an optimum prefix code, when the cost of the letters are not equal is discussed in Reference [12]; the case of alphabetic prefix codes, where the words must maintain lexicographically the order of the given probabilities, is discussed in Reference [13]. These references give additional references to previous work.

4.3 Application of the Huffman Tree to Sort-by-Merge Techniques

Assume that we have n items, and there is an order defined between them. For ease of presentation, let us assume that the items are the integers $1, 2, \ldots, n$, and the order is "less than." Assume that we want to organize the numbers in nondecreasing order, where initially they are put in L lists, $A_1, A_2, \ldots, A_L$. Each A_i is assumed to be ordered already. Our method of building larger lists from smaller ones is as follows: Let $B_1, B_2, \ldots, B_m$ be any m existing lists. We read the first, and therefore least, number in each of the lists, take the least number among them away from its list, and put it as the first number of the merged list. The list from which we took the first number is now shorter by one. We repeat this operation on the same m lists until they merge into one. Clearly, some of

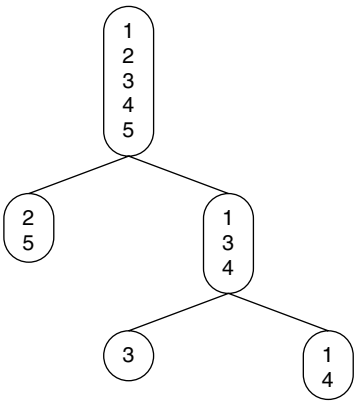


Figure 4.3: An example of a sort-by-merge described by a positional tree.

the lists become empty before others, but since this depends on the structure of the lists, we only know that the general step of finding the least number among m numbers (or less) and its transfer to a new list is repeated $b_1 + b_2 + \cdots + b_m$ times, where b_i is the number of numbers in B_i.

The number m is dictated by our equipment or decided upon in some other way. However, we shall assume that its value is fixed and predetermined. In fact, in most cases $m = 2$.

The whole procedure can be described by a positional tree. Consider the example shown in Figure 4.3, where $m = 2$. First, we merge the list $\langle 3 \rangle$ with $\langle 1,4 \rangle$. Next, we merge $\langle 2,5 \rangle$ with $\langle 1,3,4 \rangle$. The original lists, $A_1, A_2, \ldots, A_L$, correspond to the leaves of the tree. The number of transfers can be computed as follows: Let a_i be the number of numbers in A_i, and l_i be the level of the list A_i in the tree. The number of elementary merge operations is then

$$\sum_{i=1}^{L} a_i \cdot l_i. \tag{4.8}$$

Burge [15] observed that the attempt to find a positional m-ary that minimizes (4.8) is similar to that of the minimum average word-length problem solved by Huffman. The fact that the Huffman construction is in terms of probabilities does not matter, since the fact that $p_1 + p_2 + \cdots + p_L = 1$ is never used in the construction or its validity proof. Let us demonstrate the implied procedure by the following example:

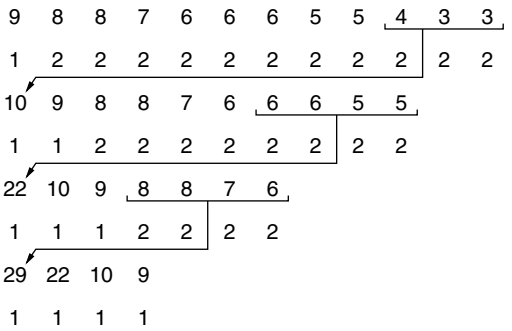


Figure 4.4: An example of sort-by-merge for $L = 12$ and $m = 4$.

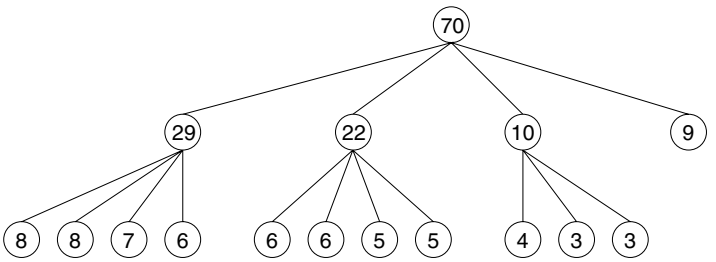


Figure 4.5: The positional tree describing the example in Figure 4.4.

Assume $L = 12$ and $m = 4$; the b_i's are given in nonincreasing order: $9,8,8,7,6,6,6,5,5,4,3,3$. Since $L \equiv 0 \pmod 3$, according to (4.6), $d = 3$.

Thus, in the first step we merge the last three lists to form a list of length 10, which is now put in the first place (see Figure 4.4). From there on, we merge each time the four lists of least length. The whole merge procedure is described in the tree shown in Figure 4.5.

4.4 Catalan Numbers

The set of *well-formed sequences of parentheses* is defined by the following recursive definition:

(1) The empty sequence is well formed.
(2) If A and B are well-formed sequences, so is AB (the concatenation of A and B).
(3) If A is well formed, so is (A).

(4) There are no other well-formed sequences.

For example, (()(())) is well formed; ()))(() is not.

Lemma 4.5 *A sequence of (left and right) parentheses is well formed if and only if it contains an even number of parentheses, half of which are left, and the other half, right. Also as we read the sequence from left to right, the number of right parentheses never exceeds the number of left parentheses.*

Proof: First let us prove the "only if" part. Since the construction of every well-formed sequence starts with no parentheses (the empty sequence), and each time we add on parentheses (Step 3) there is one left and one right, it is clear that there are n left parentheses and n right parentheses. Now, assume that for every well-formed sequence of m left and m right parentheses where $m < n$, it is true that as we read it from left to right the number of right parentheses never exceeds the number of left parentheses. If the last step in the construction of our sequence was (2), then since A is a well-formed sequence, as we read from left to right, as long as we still read A, the condition is satisfied. When we are between A and B, the count of left and right parentheses equalizes. From there on, the balance of left and right is safe, since B is well formed and contains less than n parentheses. If the last step in the construction of our sequence was (3), since A satisfies the condition, so does (A).

Now, we shall prove the "if" part, again by induction on the number of parentheses. (Here, as before, the basis of the induction is trivial.) Assume that the statement holds for all sequences of m left and m right parentheses, if $m < n$, and we are given a sequence of n left and n right parentheses that satisfies the condition. Clearly, if after reading $2m$ symbols of it from left to right, the number of left and right parentheses is equal, and if $m < n$, then this subsequence, A, by the inductive hypothesis, is well formed. Now, the remainder of our sequence, B, must satisfy the condition, too, and again by the inductive hypothesis is well formed. Thus, by Step (2), AB is well formed. If there is no such nonempty subsequence A, which leaves a nonempty B, then as we read from left to right, the number of right parentheses, after reading one symbol and before reading the whole sequence, is strictly less then the number of left parentheses. Thus, if we delete the first symbol, which is a "(", and the last, which is a ")", the remainder sequence, A, still satisfies the condition, and by the inductive hypothesis, is well formed. By Step (3) our sequence is well formed too. ■

We shall now show a one-to-one correspondence between the non-well-formed sequences of n left and n right parentheses, and all sequences of $n-1$ left parentheses and $n+1$ right parentheses.

Let $p_1 p_2 \cdots p_{2n}$ be a sequence of n left and n right parentheses that is not well formed. By Lemma 4.5, there is a prefix of it that contains more right parentheses than left parenthesis. Let j be the least integer such that the number of right parentheses exceeds the number of left parentheses in the subsequence $p_1 p_2 \cdots p_j$. Clearly, the number of right parentheses is then one larger than the number of left parentheses, or j is not the least index to satisfy the condition. Now, invert all p_i's for $i > j$ from left parentheses to right parentheses, and from right parentheses to left parentheses. Clearly, the number of left parentheses is now $n-1$, and the number of right parentheses is now $n+1$.

Conversely, given any sequence $p_1 p_2 \cdots p_{2n}$ of $n-1$ left parentheses and $n+1$ right parentheses, let j be the first index such that $p_1 p_2 \cdots p_j$ contains one right parenthesis more than left parentheses. If we now invert all the parentheses in the section $p_{j+1} p_{j+2} \cdots p_{2n}$ from left to right and from right to left, we get a sequence of n left and n right parentheses which is not well formed. This transformation is the inverse of the one in the previous paragraph. Thus, the one-to-one correspondence is established.

The number of sequences of $n-1$ left and $n+1$ right parentheses is

$$\binom{2n}{n-1},$$

for we can choose the places for the left parentheses, and the remaining places will have right parentheses. Thus, the number of well-formed sequences of length n is

$$\binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{1+n}\binom{2n}{n}. \tag{4.9}$$

These numbers are called *Catalan numbers*.

An *ordered tree* is a directed tree such that for each internal vertex there is a defined order of its sons. Clearly, every positional tree is ordered, but the converse does not hold: In the case of ordered trees there are no predetermined "potential" sons; only the order of the sons counts, not their position, and there is no limit on the number of sons.

An *ordered forest* is a sequence of ordered trees. We usually draw a forest with all the roots on one horizontal line. The sons of a vertex are drawn from left to right in their given order. For example, the forest shown in Figure 4.6 consists of three ordered trees whose roots are A, B, and C.

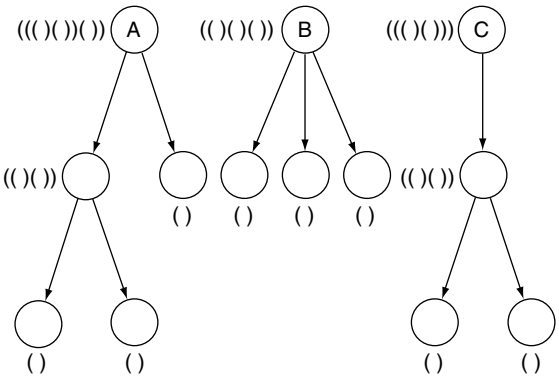


Figure 4.6: An example of three ordered trees.

There is a natural correspondence between well-formed sequences of n pairs of parentheses and ordered forests of n vertices. Let us label each leaf with the sequence (). Every vertex whose sons are labeled $w_1, w_2, \ldots, w_s$ is labeled with the concatenation $(w_1 w_2 \cdots w_s)$; clearly, the order of the labels is in the order of the sons. Finally, once the roots are labeled $x_1, x_2, \ldots, x_r$, the sequence corresponding to the forest is the concatenation $x_1 x_2 \cdots x_n$. For example, the sequence corresponding to the forest of Figure 4.6 is (((())())())(()())(((()))). The inverse transformation clearly exists, and thus the one-to-one correspondence is established. Therefore, the number of ordered forests of n vertices is given by (4.9).

We now describe a one-to-one correspondence between ordered forests and positional binary trees. The leftmost root of the forest is the root of the binary tree. The leftmost son of the vertex in the forest is the left son of the vertex in the binary tree. The next brother on the right, or, in the case of a root, the next root on the right is the right son in the binary tree. For example, see Figure 4.7, where an ordered forest and its corresponding binary tree are drawn. Again, it is clear that this is a one-to-one correspondence, and therefore the number of positional binary trees with n vertices is given by (4.9).

There is yet another combinatorial enumeration that is directly related to these.

A *stack* is a storage device that can be described as follows. Suppose that n cars travel on a narrow one-way street where no passing is possible. This leads into a narrow two-way street on which the cars can park or back up to enter another narrow one-way street (see Figure 4.8). Our problem is to find how may

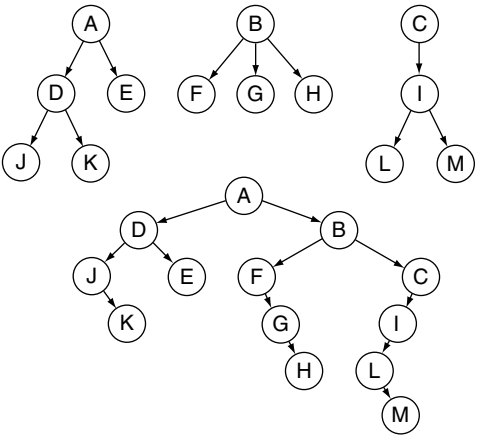


Figure 4.7: An ordered forest and its corresponding binary tree.

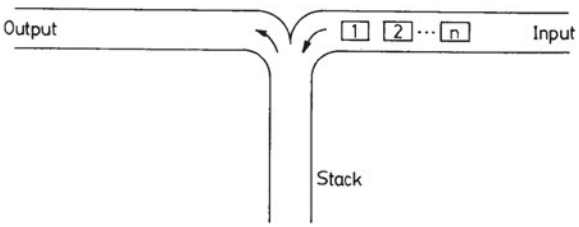


Figure 4.8: An example of a stack of cars.

permutations of the cars can be realized from input to output if we assume that the cars enter in the natural order.

The order of operations in the stack is fully described by the sequence of drive-in and drive-out operations. There is no need to specify which car drives in, for it must be the first one on the leading-in present queue; also, the only one that can drive out is the top one in the stack. If we denote a drive-in operation by "(", and a drive-out operation by ")", the whole procedure is described by a well-formed sequence of n pairs of parentheses.

The sequence must be well formed, by Lemma 4.5, since the number of drive-out operations can never exceed the number of drive-in operations. Also, every well-formed sequence of n pairs of parentheses defines a realizable sequence of operations, since, again by Lemma 4.5, a drive-out is never instructed when

the stack is empty. Also, different sequences yield different permutations. Thus, the number of permutations on n cars realizable by a stack is given by (4.9).

Let us now consider the problem of finding the number of full binary trees. Denote the number of leaves of a binary tree T by $L(T)$, and the number of internal vertices by $I(T)$. It is easy to prove, by induction on the number of leaves, that $L(T) = I(T) + 1$. Also, if all leaves of T are removed, the resulting tree of $I(T)$ vertices is a positional binary tree T'. Clearly, different T-s will yield different T'-s, and one can reconstruct T from T' by attaching two leaves to each leaf of T', and one leaf (son) to each vertex that in T' has only one son. Thus, the number of full binary trees of n vertices is equal to the number of positional binary trees of $(n-1)/2$ vertices. By (4.9) this number is

$$\frac{2}{n+1}\binom{n-1}{\frac{n-1}{2}}.$$

4.5 Problems

Problem 4.1 Prove the following theorem: A position between two letters in a message m over a UD code C is a separation between two code-words if and only if both the prefix and the suffix of m up to this position are messages over C.

Problem 4.2 Use the result of Problem 4.1 to construct an efficient algorithm for parsing a message m over a UD code C in order to find the words which compose m. (Hint: scan m from left to right, and mark all the positions such that the prefix m up to them is a message. Repeat from right to left to mark positions corresponding to suffixes that are messages.)

Problem 4.3 Test the following codes for UD:

(1) $\{00, 10, 11, 100, 110\}$,
(2) $\{1, 10, 001, 0010, 00000, 100001\}$,
(3) $\{1, 00, 101, 010\}$.

Problem 4.4 Construct a prefix binary code, with minimum average code-word length, which consists of ten words whose probabilities are $0.2, 0.18, 0.12, 0.1, 0.1, 0.08, 0.06, 0.06, 0.06, 0.04$. Repeat the construction for $\sigma = 3$ and 4.

Problem 4.5 Prove that, if a prefix code corresponds to a full positional tree, then its characteristic sum is equal to 1.

Problem 4.6 Prove that, if the word-length vector $(l_1, l_2, \ldots, l_n)$ satisfies the characteristic sum condition, and if $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n$, then there exists a positional tree with n leaves whose levels are the given l_i's, and the order of the leaves from left to right is as in the vector.

Problem 4.7 A code is called *exhaustive* if every word over the alphabet is the beginning of some message over the code. Prove the following:

(1) If a code is prefix, and its characteristic sum is 1, then the code is exhaustive.
(2) If a code is UD and exhaustive, then it is prefix and its characteristic sum is 1.

Problem 4.8 Construct the ordered forest, the positional binary tree and the permutation through a stack that corresponds to the following well-formed sequence of ten pairs of parentheses:

$$(()(()()))(((()())())).$$

Problem 4.9 A direct method for computing the number of positional binary trees of n vertices through the use of a generating function goes as follows: Let b_n be the number of trees of n vertices. Define $b_0 = 1$ and define the function

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots .$$

(1) Prove that $b_n = b_0 \cdot b_{n-1} + \cdots + b_{n-1} \cdot b_0$.
(2) Prove that $xB^2(x) - B(x) + 1 = 0$.
(3) Use the formula

$$(1+a)^{\frac{1}{2}} = 1 + \frac{\frac{1}{2}}{1!}a + \frac{\frac{1}{2}(\frac{1}{2}-1)}{2!}a^2 + \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)}{3!}a^3 + \cdots$$

to prove that

$$b_n = \frac{1}{n+1}\binom{2n}{n}.$$

Bibliography

[1] Sardinas, A. A., and Patterson, G. W., "A Necessary and Sufficient Condition for the Unique Decomposition of Coded Messages," IRE Convention Record, Part 8, 1953, pp. 104–108.
[2] Gallager, R. G., *Information Theory and Reliable Communication*, John Wiley, 1968. Problem 3.4, page 512.

[3] Levenshtein, V. I., "Certain Properties of Code Systems," Dokl. Akad. Nauk, SSSR, Vol. 140, No. 6, Oct. 1961, pp. 1274–1277. English translation: Soviet Physics, "Doklady," Vol. 6, April 1962, pp. 858–860.
[4] Even, S., "Test for Unique Decipherability," IEEE Trans. on Infor. Th., Vol. IT-9, No. 2, April 1963, pp. 109–112.
[5] Levenshtein, V. I., "Self-Adaptive Automata for Coding Messages," Dokl. Akad. Nauk, SSSR, Vol 140, Dec. 1961, pp. 1320–1323. English translation: Soviet Physics, "Doklady," Vol. 6, June 1962, pp. 1042–1045.
[6] Markov, Al. A., "On Alphabet Coding," Dokl. Akad. Nauk, SSSR, Vol. 139, July 1961, pp. 560–561. English translation: Soviet Pysics, "Doklady," Vol. 6, Jan. 1962, pp. 553–554.
[7] Even, S., "Test for Synchronizability of Finite Automata and Variable Length Codes," IEEE Trans. on Infor. Th., Vol. IT-10, No. 3, July 1964, pp. 185–189.
[8] McMillan, B., "Two Inequalities Implied by Unique Decipherability," IRE Tran. on Infor. Th., Vol. IT-2, 1956, pp. 115–116.
[9] Karush, J., "A Simple Proof of an Inequality of McMillan," IRE Tran. on Infor. Th., Vol. IT-7, 1961, pp. 118.
[10] Kraft, L. G., "A Device for Quantizing, Grouping and Coding Amplitude Modulated Pulses," M.S. Thesis, Dept. of E.E., M.I.T.
[11] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE, Vol. 40, No. 10, 1952, pp. 1098–1101.
[12] Perl, Y., Garey, M. R., and Even, S., "Efficient Generation of Optimal Prefix Code: Equiprobable Words Using Unequal Cost Letters," J.ACM, Vol. 22, No. 2, April 1975, pp. 202–214.
[13] Itai, A., "Optimal Alphabetic Trees," SIAM J. Comput., Vol. 5, No. 1, March 1976, pp. 9–18.
[14] Knuth, D. E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.
[15] Burge, W. H., "Sorting, Trees, and Measures of Order," *Infor. and Control*, Vol. 1, 1958, pp. 181–197.

5
Flow in Networks

5.1 Introduction

A *network* $N(G, s, t, c)$ consists of the following data:

- A finite digraph $G(V, E)$ with no self-loops and no parallel edges.[1]
- Two vertices s and t are specified; s is called the *source*; and t, the *sink*.[2]
- The *capacity* function, $c : E \to R^+$. The positive real number, $c(e)$, is called the *capacity* of edge e.

For every vertex $v \in V$, let $\alpha(v)$ denote the set of edges that enter v in G. Similarly, let $\beta(v)$ denote the set of edges that emanate from v.

A *flow function*, $f : E \to R$, is an assignment of a real number $f(e)$ to each edge e such that the following two conditions hold:

The Edge Rule. For every edge $e \in E$, $0 \leqslant f(e) \leqslant c(e)$.
The Vertex Rule. For every vertex $v \in V \setminus \{s, t\}$,

$$\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e).$$

The *total flow* F, of f in N, is defined by

$$F = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e)\,. \tag{5.1}$$

Namely, F is the net sum of flow into the sink.

[1] Self-loops are useless in this context, and parallel edges can be replaced with one edge whose capacity is the sum of the capacities of the parallel edges.
[2] The source is not necessarily a graphical source; i.e., it may have incoming edges. Similarly, the sink is not necessarily a graphical sink.

In the next two sections, we shall discuss methods for computing a flow function f for which F is maximum.

Given a set $S \subset V$, let $\bar{S} = V \setminus S$. In the following, we shall discuss sets S, such that $s \in S$ and $t \in \bar{S}$. Also, $(S;\bar{S})$ denotes the set of edges which are directed from a vertex in S to a vertex in $\bar{S}$; this set of edges is called a *forward cut*. The set $(\bar{S};S)$ is similarly defined, and is called a *backward cut*. The union of $(S;\bar{S})$ and $(\bar{S};S)$ is called the *cut* defined by S.

By definition, the total flow F is measured at the sink. Our purpose is to show that F can be measured at any cut.

Lemma 5.1 *For every* $\{s\} \subset S \subset (V \setminus \{t\})$ *and every flow function* f *of total flow* F*, the following holds:*

$$F = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e) \,. \tag{5.2}$$

Proof: By the vertex rule, for every $v \in (V \setminus \{s,t\})$,

$$0 = \sum_{e \in \alpha(v)} f(e) - \sum_{e \in \beta(v)} f(e) \,. \tag{5.3}$$

Also, consider again Equation 5.1:

$$F = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e) \,.$$

Now, add the equations, as in Equation 5.3, for every $v \in (\bar{S} \setminus \{t\})$, as well as Equation 5.1. The aggregate equation has F on the l.h.s.[3]

In order to see what happens on the r.h.s., consider an edge $x \xrightarrow{e} y$. If both x and y belong to S then $f(e)$ does not appear on the r.h.s. of the aggregate equation at all, in agreement with Equation 5.2. If both x and y belong to $\bar{S}$ then $f(e)$ appears twice on the r.h.s. of the aggregate equation; once positively, in the equation for y, and once negatively, in the equation for x. Thus, it is canceled out in the summation, again in agreement with Equation 5.2. If $x \in S$ and $y \in \bar{S}$ then $f(e)$ appears on the r.h.s. of the aggregate equation, as part of Equation 5.3 for y, positively, and in no equation for other vertices included in the summation. In this case $e \in (S;\bar{S})$, and again we have agreement with Equation 5.2. Finally, if $x \in \bar{S}$ and $y \in S$, $f(e)$ appears negatively on the r.h.s.

[3] Left-hand side.

of the aggregate equation, as part of Equation 5.3 for x, and again this agrees with Equation 5.2, since $e \in (\bar{S};S)$. ■

Let us denote by $c(S)$ the *capacity of the cut* determined by S, which is defined as follows:

$$c(S) = \sum_{e \in (S;\bar{S})} c(e) \; . \tag{5.4}$$

Lemma 5.2 *For every flow function* f*, with total flow* F*, and every* $\{s\} \subset S \subset (V \setminus \{t\})$*, the following inequality holds:*

$$F \leqslant c(S) \; . \tag{5.5}$$

Proof: By Lemma 5.1,

$$F = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e) \; .$$

By the edge rule, for every edge e, $0 \leqslant f(e) \leqslant c(e)$. Thus,

$$F \leqslant \sum_{e \in (S;\bar{S})} c(e) - 0 \; .$$

Finally, by Equation 5.4, $F \leqslant c(S)$. ■

The following is a very important corollary of Lemma 5.2. It allows us to detect that a given total flow F is maximum, and the capacity of a given cut, defined by S, is minimum.

Corollary 5.1 *If* F *and* S *satisfy Equation 5.5 by equality, then* F *is maximum and the cut defined by* S *is of minimum capacity.*

5.2 The Algorithm of Ford and Fulkerson

Ford and Fulkerson [7] suggested the use of *augmenting paths* to change a given flow function f in order to increase the total flow. A procedure for finding an augmenting path, if one exists, is used. If an augmenting path is found, then it is used to increase the total flow. If no augmenting path is found, then the present flow is declared maximum and the process terminates.

If the direction of the edges is ignored, an augmenting path P is a simple path from s to t. It is used to add a quantity $\Delta > 0$ to the total flow by pushing along

P Δ additional units of flow. To do this, if an edge e of P is directed in the direction from s to t, $c(e) \geqslant f(e) + \Delta$ must hold. But if e of P is directed in the opposite direction, we must be able to reduce its flow from $f(e)$ to $f(e) - \Delta$. Thus, $f(e) \geqslant \Delta$ must hold.

In attempt to find an augmenting path for a given flow, a labeling procedure is used. We first label s. Then, as long as there is an unlabeled vertex v for which an augmenting path from s to v is found, v is labeled. If t is labeled, then an augmenting path has been found and the labeling process is terminated.

Every vertex v is assigned a label $\lambda(v)$, which is one of the following:

- $\lambda(s) = \infty$; only vertex s gets this label, and it is the only label s gets.
- $\lambda(v) = \text{NIL}$; in this case, we say that v has no label.
- $\lambda(v) = (e, \sigma)$, where e is the edge through which v has been assigned its label, $\sigma = +$ if e has been used forwardly and $\sigma = -$ if e has been used backwardly.

An edge $u \xrightarrow{e} v$ is said to be *useful* from u to v if $\lambda(u) \neq \text{NIL}$, $\lambda(v) = \text{NIL}$ and one of the following conditions holds:

- $u \xrightarrow{e} v$ and $f(e) < c(e)$. In this case, we say that e is *forwardly useful*.
- $v \xrightarrow{e} u$ and $f(e) > 0$. In this case, we say that e is *backwardly useful*.

Algorithm 5.1 describes the labeling procedure, named LABEL. The input of LABEL consists of the network N and the present flow function f. $\lambda(\cdot)$ is defined for all vertices. The procedure may modify these labels. Thus, $\lambda(\cdot)$ is both an input and output. The output of LABEL includes a function $\Delta(\cdot)$, which is defined on a subset of E, and its value is a positive real number.

```
Procedure LABEL(N, f, λ, Δ)

1    while there is an edge u --e--> v which is useful from u to v and
     λ(t) = NIL do
2        if e is forwardly useful then do
3            λ(v) ← (e, +)
4            Δ(e) ← c(e) − f(e)
5        if e is backwardly useful then do
6            λ(v) ← (e, −)
7            Δ(e) ← f(e)
```

Algorithm 5.1: The labeling procedure.

```
Procedure AUGMENT(G, λ, Δ, f)

1   empty Q
2   Δ ← ∞
3   v ← t (v is called the center of activity)
4   while v ≠ s do
5       put λ(v) into Q
6       let λ(v) = (e, σ)
7       Δ ← min{Δ, Δ(e)}
8       let e be u —e→ v
9       v ← u
10  while Q is not empty do
11      remove the first label, (e, σ), from Q
12      if σ = + then do
13          f(e) ← f(e) + Δ
14      else (σ = −) do
15          f(e) ← f(e) − Δ
```

Algorithm 5.2: The augmenting procedure.

If $\lambda(t) \neq \mathrm{NIL}$ when LABEL is terminated, then an augmenting path exists. The augmenting path is found in the AUGMENT procedure and used to change f and thus, increment F. This is presented in Algorithm 5.2.

The input to AUGMENT consists of $G(V, E)$, $\lambda(\cdot)$ and $\Delta(\cdot)$. The flow function f is both an input and an output. A queue Q of vertex labels is used, as well as a real Δ – these variables are internal.

The Ford and Fulkerson procedure is described in Algorithm 5.3 and uses LABEL and AUGMENT as subroutines. Initially, f is any legitimate flow in the network N, and in the absence of better ideas, one can use $f(e) = 0$ for every edge e. Thus, the input is N and f is both an input and an output.

Note that, if for an edge $e \in \alpha(s)$, initially $f(e) = 0$, then $f(e)$ is never changed. The same is true for an edge $e \in \beta(t)$.

As an example, consider the networks shown in Figure 5.1. Next to each edge e we write $c(e), f(e)$, in this order. We assume a zero initial flow everywhere. A first wave of label propagation might be as follows: s is labeled, e_2 is used to label c, e_6 is used to label d, e_4 is used to label a, e_3 used to label b and finally,

Procedure FORD-FULKERSON(N, f)

```
1   for every v ∈ V do
2       λ(v) ← NIL
3   λ(s) ← ∞
4   call LABEL(N, f, λ, Δ)
5   while λ(t) ≠ NIL do
6       call AUGMENT(G, λ, Δ, f)
7       for every v ∈ V \ {s} do
8           λ(v) ← NIL
9       call LABEL(N, f, λ, Δ)
10  print "The present flow f is maximum"
```

Algorithm 5.3: The Ford-Fulkerson procedure.

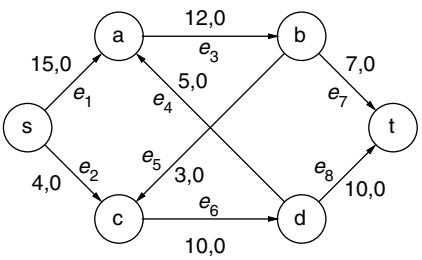


Figure 5.1: An example of a network.

e_7 is used to label t. The path is

$$s \xrightarrow{e_2} c \xrightarrow{e_6} d \xrightarrow{e_4} a \xrightarrow{e_3} b \xrightarrow{e_7} t\,,$$

$\Delta = 4$, and the new flow is shown in Figure 5.2. The second augmenting path may be

$$s \xrightarrow{e_1} a \xrightarrow{e_3} b \xrightarrow{e_5} c \xrightarrow{e_6} d \xrightarrow{e_8} t\,,$$

$\Delta = 3$ and the new flow is shown in Figure 5.3. The third augmenting path may be

$$s \xrightarrow{e_1} a \xrightarrow{e_3} b \xrightarrow{e_7} t\,,$$

$\Delta = 3$ and the new flow is shown in Figure 5.4. Up to now only forward labeling

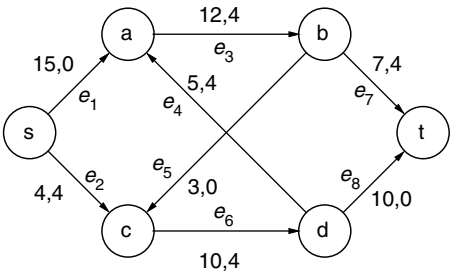


Figure 5.2: The example after the first augmenting path.

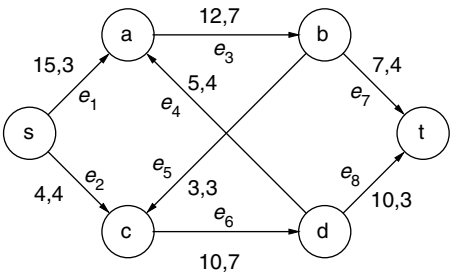


Figure 5.3: The example after the second augmenting path.

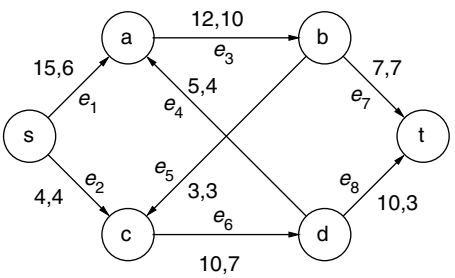


Figure 5.4: The example after the third augmenting path.

has been used. In the next application of LABEL, we can still label vertex a via e_1; and vertex b, via e_3, both of which are forward labeling, but no further forward labeling exists. However, e_4 is useful backwardly, and through it one

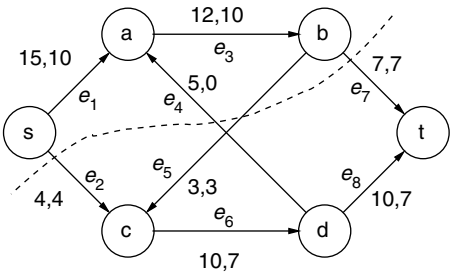


Figure 5.5: The example after the fourth augmenting path.

can label d. Now one can use e_8 to label t. The augmenting path is

$$s \xrightarrow{e_1} a \xleftarrow{e_4} d \xrightarrow{e_8} t\,,$$

$\Delta = 4$, and the new flow is shown in Figure 5.5. Now, the total flow, F, is equal 14. The next application of LABEL does not reach t. The set of labeled vertices S is $\{s, a, b\}$ and the forward cut, $(S;\bar{S})$ consists of the edges e_2, e_5, and e_7. All of these edges are saturated; that is, $f(e) = c(e)$. The backward cut, $(\bar{S};S)$, consists of one edge, e_4, and its flow is 0. Thus, $F = c(S)$, and by Corollary 5.1, F is maximum and $c(S)$ is minimum.

It is easy to see that the flow produced by the algorithm remains legitimate throughout. The definitions of $\Delta(\cdot)$ and Δ guaranty that forwardly used edges will not be overflowed, that is, $f(e) \leqslant c(e)$, and that backward edges will not be underflowed, that is, $f(e) \geqslant 0$. Also, since Δ units of flow are are pushed from s to t on a path, the incoming flow will remain equal to the outgoing flow in every vertex $v \in V \setminus \{s, t\}$.

Assuming the Ford and Fulkerson procedure halts, the last labeling process has not reached t. As above, let S be the set of vertices labeled in the last application of the labeling process. If $e \in (S;\bar{S})$, then $f(e) = c(e)$, since e is not forwardly useful. If $e \in (\bar{S}, S)$, then $f(e) = 0$, since e is not backwardly useful. By Lemma 5.1,

$$F = \sum_{e\in(S;\bar{S})} f(e) - \sum_{e\in(\bar{S};S)} f(e) = \sum_{e\in(S;\bar{S})} c(e) - \sum_{e\in(\bar{S};S)} 0 = c(S)\,.$$

By Corollary 5.1, F is maximum and $c(S)$ is minimum.

The question of whether the Ford and Fulkerson procedure will always halt remains to be discussed. Note first a very important property of the procedure:

If the initial flow is integral, for example, zero everywhere, and if all capacities are integers, then the algorithm never introduces fractions. The algorithm adds and subtracts, but it never divides. Also, if t is labeled, the resulting augmenting path is used to increase the total flow by at least one unit. Since there is an upper bound on the total flow (any cut), the process must terminate.

Ford and Fulkerson showed that their procedure may fail if the capacities are allowed to be irrational numbers. Their counterexample ([7, p. 21]) displays an infinite sequence of flow augmentations. The flow converges (in infinitely many steps) to a value which is one-fourth of the maximum total flow. We shall not present their example here; it is fairly complex, and as the reader will shortly discover, it is not as important any more.

One could have argued that, for all practical purposes, we may assume that the algorithm is sure to halt. This follows from the fact that our computations are usually through a fixed radix (decimal, binary, etc.) number representation with a bound on the number of digits used; in other words, all figures are multiples of a fixed quantum and the termination proof works here as it does for integers. However, a simple example shows the weakness of this argument. Consider the network shown in Figure 5.6.

Assume that M is a very large integer. If the algorithm starts with $f(e) = 0$ for every e, and alternately uses

$$s \longrightarrow a \longrightarrow b \longrightarrow t$$

and

$$s \longrightarrow b \longleftarrow a \longrightarrow t$$

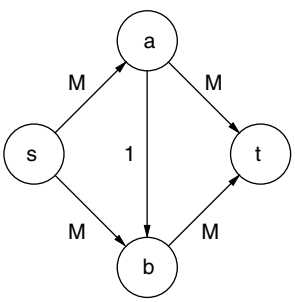


Figure 5.6: An example which demonstrating the possibility that the Ford and Fulkerson procedure may be inefficient.

as augmenting paths, it will use $2M$ augmentations before $F = 2M$ is achieved. This is exponential time in terms of the length of the input data, since it takes only $\lceil \log_2(M+1) \rceil$ bits to represent M.

Edmonds and Karp [4] were first to overcome this problem. They showed that if Breadth-First Search (BFS) is used in the labeling algorithm and thus, every augmenting path is a shortest one, the algorithm terminates in $O(|V| \cdot |E|^2)$ time, regardless of the capacities. (Here, of course, we assume that our computer can handle, in one step, any real number.) In the next section we shall present the more advanced work of Dinitz (see [2]); his algorithm has time complexity $O(|V|^2 \cdot |E|)$. A significantly more efficient algorithm was presented by Goldberg and Tarjan [9], but this algorithm is not described in this book.

The existence of the algorithm of Edmonds and Karp, or that of Dinitz, assures that if one proceeds according to a proper strategy in the labeling procedure, the algorithm is guaranteed to halt (in polynomial time). When it does, the total flow is maximum, and the cut indicated is minimum, thus providing the max flow min-cut theorem:

Theorem 5.1 *Every network has a maximum total flow which is equal to the capacity of a cut for which the capacity is minimum.*

5.3 The Dinitz Algorithm

As in the Ford and Fulkerson algorithm, the Dinitz algorithm[4] starts with some legitimate flow function f and improves it. When no improvement is possible the algorithm halts, and the total flow, F, is maximum.

Given a network $N(G(V,E), s, t, c)$ and a flow function f, let us define the *secondary* network $N'(G'(V,E'), s, t, c')$ as follows:

- For every $e \in E$ such that $f(e) < c(e)$, $e \in E'$. Also, $c'(e) = c(e) - f(e)$. (Such an edge is called *forwardly useful* and $c'(e)$ is called its *residual capacity*.)
- For every $e \in E$, $a \xrightarrow{e} b$ such that $f(e) > 0$, there is an edge $b \xrightarrow{e'} a$ in E'. Also, $c'(e') = f(e)$. (Such an edge is called *backwardly useful*.)

Note that if $0 < f(e) < c(e)$, then e gives rise to two antiparallel edges in G', since it is useful both in the forward and the backward directions. Thus, $|E| \leqslant |E'| \leqslant 2|E|$.

[4] In [3], Yefim Dinitz tells the story of his algorithm, and the differences between his version and the one presented here (G.E.).

```
Procedure LAYER(N′, V_i, vertex labels, T)

1   T ← ∅
2   while there is an edge u ⟶ v in N′, u ∈ V_i and v is not labeled, do
3       T ← T ∪ {v}
4       label v
```

Algorithm 5.4: Finding the next layer in the BFS of N'.

Before we go into a detailed description of the Dinitz algorithm, here is an abstract of its structure:

The Dinitz algorithm proceeds in phases. In each phase the current f is used to produce the corresponding secondary network N'. A *layered network*, N'', is then produced by a BFS on N', starting from the source s. If the sink t is not reached, the whole algorithm halts, and the current flow is maximum. If t is reached, a maximal (or blocking) flow, f'', is found in N''. The flow f is then augmented, by using f'', and a new phase is launched.

We now present a detailed description of the Dinitz algorithm and later discuss its validity and time complexity.

The construction of N'', from N', uses the BFS layers. The production of the layers is described in the subroutine LAYERS, which in turn uses a subroutine LAYER, described in Algorithm 5.4. The input to LAYER is the secondary network N', the previous layer (a set of vertices) V_i, and the vertex labels. The set of labeled verfices can change when LAYER is run (since additional vertices may get labeled), and is therefore also an output. The set of vertices that may become the next layer is provided by the output T.

The LAYERS subroutine is described in Algorithm 5.5. The input to LAYERS consist of N'. The output is l, and if t is reached, the layers are specified in $V_0, V_1, \ldots V_l$. Obviously, $l \leqslant n-1$.

If t has been reached, the *layered network* $N''(G''(V'', E''), s, t, c'')$ is constructed as follows:

- $V'' = \cup_{i=0}^{l} V_i$,
- $E_i'' = \{u \xrightarrow{e''} v \mid u \in V_i,\ v \in V_{i+1},\ e'' \in E'\}$,
- $E'' = \cup_{i=0}^{l-1} E_i''$,
- for every $e'' \in E''$ $\quad c''(e'') = c'(e'')$.

Procedure LAYERS(N';$V_0, V_1, \ldots V_{n-1}, l$)

Local variables: T a set of vertices, vertex labels.

```
1   label every v ∈ V "unlabeled"
2   label s
3   V_0 ← {s}
4   i ← 0
5   repeat
6       call LAYER(N', V_i, "vertex labels", T)
7       i ← i + 1
8       V_i ← T
9   until t ∈ T or T = ∅
10  if t ∈ T then do
11      l ← i
12      V_l ← {t}
13  else (T = ∅) declare: "f is a maximum flow."
```

Algorithm 5.5: Finding the BFS Layers of N'.

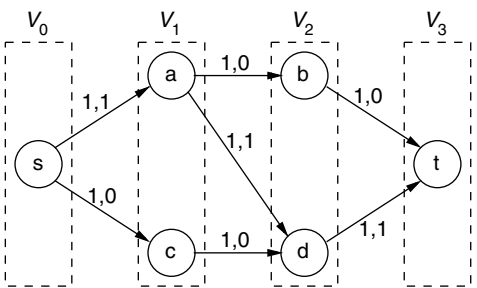


Figure 5.7: An example of a maximal flow in a layered network that is not maximum.

Next, we construct a *maximal*, or *blocking* flow in N''. A maximal flow f'' is a legitimate flow in N'' such that no augmenting path of length l exists in the presence of f''. A maximal flow may not be maximum, as demonstrated in Figure 5.7. The total flow is 1. This flow is not maximum, since there is a flow

```
Procedure FIND-PATH(N'', blocking-labels, S)

1   empty S
2   push (NIL, s) into S
3   while S is not empty and t is not the right element in the top pair of S do
4       let (x, v) be the top pair on S
5       if there is an unblocked edge v --e'--> v', then do
6           push (e', v') into S
7       else do
8           if x ≠ NIL then label x 'blocked'
9           pop (x, v) from S
```

Algorithm 5.6: Finding an augmenting path of length l in N''.

for which the total is 2. Yet it is maximal, since on every directed path of length 3 from s to t, there is at least one saturated edge.

It is easier to find maximal flow than maximum flow, since the flow in edges never has to decrease. In other words, no backward labeling of vertices is necessary.

The procedure MAXIMAL to find a maximal flow in a given layered network N'' starts with $f''(e'') = 0$ for every $e'' \in E''$. It uses two subroutines: FIND-PATH and INCREASE-FLOW. Initially, all edges are marked "unblocked". When an edge becomes saturated, its label changes to "blocked." Also, if we conclude that the flow in $u \xrightarrow{e''} v$ cannot increase since all paths from v to t are blocked, we change the label of e'' to "blocked."

FIND-PATH is described in Algorithm 5.6. Its input is N''. The blocking-labels are both an input and an output, and the stack S is an output. The procedure uses a DFS strategy. As long as there is an unblocked edge to continue on, we keep going, and store the sequence of the edges and their end-points in a stack S. Thus, S is a stack of pairs, such as (e, v), where v is the vertex that e enters. If t is reached, the sequence of edges stored in S is a directed augmenting path from s to t of length l. If there are no unblocked edges out of the vertex we have reached, we pop the top edge (and its endpoint) from S, change its label to 'blocked' and backtrack to the previous vertex on the path. If we are stuck at s, the search terminates with no edges in S.

The procedure INCREASE-FLOW is described in Algorithm 5.7. The procedure uses two local variables: S' is a stack of vertices, and Δ is a real number.

```
Procedure INCREASE-FLOW(N'', S, f'', blocking-labels)

1   empty S'
2   Δ ← ∞
3   while the pair on top of S is not (NIL, s) do
4       pop the top pair (e, v) from S
5       Δ ← min{Δ, c''(e) − f''(e)}
6       push e into S'
7   while S' is not empty do
8       pop the top edge e from S'
9       f''(e) ← f''(e) + Δ
10      if f''(e) = c''(e) then mark e 'blocked'
```

Algorithm 5.7: Increasing the flow in the edges of the augmenting path in N''.

N'' and S are the input, while f'' and the blocking-labels are both an input and an output. The augmenting path stored in S is scanned twice. During the first scan (Lines 3–6), the minimum residual capacity of the edges on the augmenting path is computed and recorded in Δ, and the augmenting path is restored S'. In the second scan (Lines 7–10), the flow in every edge of the augmenting path is increased by Δ, and if an edge becomes saturated, its label is changed from "unblocked" to "blocked". Clearly, at least one edge on the path becomes saturated.

The procedure MAXIMAL described in Algorithm 5.8 finds a maximal flow, f'', in a given layered network N''. The stack S and the blocking-labels are internal variables. Finally, we are ready to present the Dinitz algorithm, as described in Algorithm 5.9. Note that the input to DINITZ is the network N, and the output is a maximum flow f, in N. All other variables used in the procedure, such as N', N'' and all their components, including l, and f'', are internal variables.

Lemma 5.3 *If procedure DINITZ halts, the resulting* f *is a legitimate flow in* N.

Proof: First, observe that in each phase the flow f'' is legitimate in N''. The edge rule is observed since Δ is chosen in INCREASE-FLOW in a way that

```
Procedure MAXIMAL(N'', f'')

1   for every e ∈ E'' do
2       label e 'unblocked'
3       f''(e) ← 0
4   run FIND-PATH(N'', blocking-labels, S)
5   while S is not empty then do
6       run INCREASE-FLOW(N'', S, f'', blocking-labels)
7       run FIND-PATH(N'', blocking-labels, S)
```

Algorithm 5.8: Constructing a maximal flow in a layered network N''.

```
Procedure DINITZ(N;f)

1   for every e ∈ E do
2       f(e) = 0
3   N' ← N
4   run LAYERS(N';V_0, V_1, ..., V_{n-1}, l)
5   while f has not been declared to be maximum do
6       construct N''
7       run MAXIMAL(N'';f'')
8       for every e' ∈ E'' do
9           if e' corresponds to a forward edge e ∈ E then do
10              f(e) ← f(e) + f''(e')
11          else (u --e'--> v ∈ N'', but v --e--> u ∈ N) do
12              f(e) ← f(e) − f''(e')
13      construct N' from (N, f)
14      run LAYERS(N';V_0, V_1, ..., V_{n-1}, l)
```

Algorithm 5.9: The Dinitz algorithm.

ensures that no edge is overflowed. The vertex rule is observed, since the flow is built up by augmenting paths.

By the definition of c' and thus, c'', when f is changed by adding (subtracting) f'' to (from) the previous f, the edge rule is maintained in the edges of G. Also,

since f is the superposition of two flows, each of which observes the vertex rule, so does their sum. ■

Lemma 5.4 *If procedure DINITZ halts, the resulting* f *is a maximum flow in* N.

Proof: The proof here is very similar to the one in the Ford and Fulkerson algorithm. Consider the last phase, in which vertex t is not reached in procedure LAYERS (see Line 13). Let S be the union of $V_0, V_1, \ldots, V_i$, where V_i is the last (nonempty) layer before $T = \emptyset$ is encountered. Now consider the cuts $(S;\bar{S})$ and $(\bar{S};S)$ in G. Every edge $u \xrightarrow{e} v$ in $(S;\bar{S})$ is saturated, or else e is useful from $u \in V_i$ to v, and T is not empty. Also, for every edge $u \xrightarrow{e} v$ in $(\bar{S};S)$, $f(e) = 0$, or e is useful from $v \in V_i$ to u, and again, T is not empty. The remainder of the argument is identical to that of the Ford-Fulkerson case. ■

Lemma 5.5 *In each phase, except the last, the flow* f'' *is maximal in* N''.

Proof: Initially, all edges of N'' are unblocked (see Algorithm 5.8). An edge $u \xrightarrow{e} v$ is marked "blocked" in two cases:

- e is saturated. See Line 10 of Algorithm 5.7.
- There is no unblocked edge out of v. See Line 8 of Table 5.6.

In either case, when an edge is blocked, it is already known that no additional augmenting path through it is possible. Thus, when all edges out of s are blocked, the flow f'' is maximal. This event is detected by procedure FIND-PATH (see Algorithm 5.6) by the emptiness of S. ■

Lemma 5.6 *The running time of each phase is* $O(|V| \cdot |E|)$.

Proof: The time it takes to construct N'' is $O(|E|)$, since it is essentially a BFS on N'.

Note that in Algorithm 5.6 we traverse edges (see Lines 5–6). If we backtrack to a vertex other than s (Lines 7–9), an edge is marked blocked.

The number of consecutive edge traversals between two blockings of edges is bounded by l for the following reasons:

- If l traversals are performed consecutively without any backtracking, then t has been reached and an augmenting path is stored in S. Procedure INCREASE-FLOW is called and at least one edge becomes saturated and is marked blocked.

- It is possible that we have backtracked into a vertex other than s. In that case, in less than l consecutive traversals, either another backtrack occurs or t is reached, and as in the previous case, at least one edge is marked "blocked".

Since the number of edges in N'' is $O(|E|)$, and since every edge can be blocked at most once, the total number of edge traversals is $O(|V| \cdot |E|)$.

It is not hard to see that the time of all other operations in procedure MAXIMAL is bounded by a constant times the number of edge traversals. ■

Note that the length of a layered network N'' was denoted by l. Since we need to compare lengths of the layered networks of consecutive phases, let us denote by l_k the length of the layered network of the k-th phase, $k \geqslant 1$.

Lemma 5.7 *If the* $(k+1)$*-st phase is not the last, then* $l_{k+1} > l_k$.

Proof: There is a path P of length l_{k+1} in the layered network of the $(k+1)$-st phase, which starts with s and ends with t:

$$P:\ s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \cdots v_{l_{k+1}-1} \xrightarrow{e_{l_{k+1}}} v_{l_{k+1}} = t\ .$$

First, let us assume that all the vertices of P appear in the k-th layered network. Let V_j be the j-th layer of the k-th layered network. We claim that if $v_a \in V_b$, then $a \geqslant b$. This is proved by induction on a. For $a = 0$ ($v_0 = s$), the claim is obviously true. Now, assume $v_{a+1} \in V_c$.

By the inductive hypothesis, $(a \geqslant b)$, and if $b+1 \geqslant c$, then $a+1 \geqslant b+1 \geqslant c$, proving the inductive step. However, if $b+1 < c$, then the edge e_{a+1} has not been used in the k-th phase since it is not even in the k-th layered network, where all edges are between adjacent layers. If e_{a+1} has not been used in the k-th layered network and is useful from v_a to v_{a+1} in the beginning of phase $k+1$, then it has been useful from v_a to v_{a+1} in the beginning of phase k, as well. Thus, v_{a+1} cannot belong to V_c since by the algorithm it belongs to a previous layer of the k-th network.

Now, in particular, $t = v_{l_{k+1}} \in V_k$. Therefore, $l_{k+1} \geqslant l_k$. Also, equality cannot hold because in this case, the entire P is in the k-th layered network, and if all its edges are still useful in the beginning of phase $k+1$, then the final flow f'' of phase k was not maximal. This proves the claim of the lemma in the case that all vertices of P are in the k-th layered network.

If not all the vertices of P appear in the kth layered network, then let $v_a \xrightarrow{e_{a+1}} v_{a+1}$ be the first edge of P such that for some b, $v_a \in V_b$, but v_{a+1} is not in the

k-th layered network. Thus, e_{a+1} was not used in phase k. Since e_{a+1} is useful in the beginning of phase $k+1$, it was also useful in the beginning of phase k. The only possible reason for v_{a+1} not to belong to V_{b+1} is that $b+1 = l_k$ and $v_{a+1} \neq t$. Since $t = v_{l_{k+1}}$, it follows that $a+1 < l_{k+1}$. By the argument of the previous paragraph, $a \geqslant b$. Thus, $l_k = b+1 \leqslant a+1 < l_{k+1}$. ■

Corollary 5.2 *The number of phases is bounded by* $|V|$.

Proof: Clearly, $l_1 \geqslant 1$. By Lemma 5.7, for the k-th phase, if it is not the last, $l_k \geqslant k$. Since $k \leqslant l_k \leqslant |V|-1$, the number of phases, including the last, is bounded by $|V|$. ■

Theorem 5.2 *The Dinitz algorithm terminates in time* $O(|V|^2|E|)$ *and yields a maximum flow.*

Proof: By Corollary 5.2, the number of phases is bounded by $|V|$. By Lemma 5.6, each phase requires $O(|V| \cdot |E|)$ time. Thus, the whole algorithm takes $O(|V|^2|E|)$ time to terminate. By Lemma 5.4, the resulting flow is maximum in N. ■

Theorem 5.3 (The max-flow min-cut theorem) *Every finite network has a maximum flow and a minimum cut, and their values are equal.*

Proof: By Theorem 5.2 there is a max-flow, and by the proof of Lemma 5.4, there is a cut of the same value. By Corollary 5.1, this cut is of minimum value. ■

5.4 Networks with Upper and Lower Bounds

In the previous sections, we have assumed that the flow in each edge is bounded from above by the capacity of the edge, but that the lower bound on the flow in every edge is zero. The significance of this assumption is that the assignment of $f(e) = 0$, for every edge e, defines a legitimate flow, and the algorithm for improving the flow can be started with this zero flow.

In this section, in addition to the upper bound $c(e)$ on the flow in e, we assume that the flow is also bounded from below by $b(e)$. Thus, the *edge rule* is changed as follows: The flow $f(e)$, in every edge e, must satisfy

$$b(e) \leqslant f(e) \leqslant c(e). \tag{5.6}$$

The vertex rule remains unchanged.

Thus, the problem of finding a maximum flow in a given network $N(G(V,E), s, t, b, c)$ is divided into two subproblems. First, check whether N

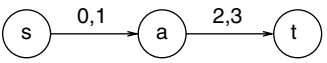


Figure 5.8: An example of a network which has no legitimate flow.

has legitimate flows, and if the answer is positive, find one. Second, increase this initial flow and find a maximum flow.

A simple example of a network that has no legitimate flow is shown in Figure 5.8. Here, next to each edge e we write $b(e), c(e)$.

The following method for testing whether a given network N has a legitimate flow function is due to Ford and Fulkerson [7]. It reduces the problem to one of determining a maximum flow in an *auxiliary network* $\tilde{N}(\tilde{G}(\tilde{V},\tilde{E}),\tilde{s},\tilde{t},\tilde{c})$, in which all lower bounds are zero. Here, $\tilde{s}$ is called the *auxiliary source*, and $\tilde{t}$ is called the *auxiliary sink*. In the auxiliary flow problem, s and t are not the source and the sink anymore and must satisfy the vertex rule. We shall show that the original N has legitimate flows if and only if the maximum flow in $\tilde{N}$ satisfies a condition to be specified shortly.

$\tilde{N}$ is defined as follows:

(i) $\tilde{V} = V \cup \{\tilde{s},\tilde{t}\}$, where $\tilde{s}$ and $\tilde{t}$ are two new vertices.

(ii) $\tilde{E} = E \cup \mathcal{S} \cup \mathcal{T} \cup \{e',e''\}$, where $\mathcal{S}$, $\mathcal{T}$ and $\{e',e''\}$ are sets of new edges defined as follows:

$$\mathcal{S} = \{\tilde{s} \longrightarrow v \mid v \in V\},$$

$$\mathcal{T} = \{v \longrightarrow \tilde{t} \mid v \in V\},$$

and $s \xrightarrow{e'} t$ and $t \xrightarrow{e''} s$.

(iii) $c : \tilde{E} \to R^{\geqslant 0} \cup \{\infty\}$ is defined separately for each of the four sets of edges:

- For $e \in E$,
$$\tilde{c}(e) = c(e) - b(e). \tag{5.7}$$
- For $\tilde{s} \xrightarrow{\sigma} v$, $\tilde{c}(\sigma) = \sum_{e \in \alpha(v)} b(e)$.
- For $v \xrightarrow{\tau} \tilde{t}$, $\tilde{c}(\tau) = \sum_{e \in \beta(v)} b(e)$.
- $\tilde{c}(e') = \infty$ and $\tilde{c}(e'') = \infty$.

Let us demonstrate this construction on the network shown in Figure 5.9. The auxiliary network is shown in Figure 5.10. The upper bounds are shown next to the edges to which they apply.

Now we can use the Ford and Fulkerson or the Dinitz algorithm to find a maximum flow in the auxiliary network. It is left to the reader to verify that the

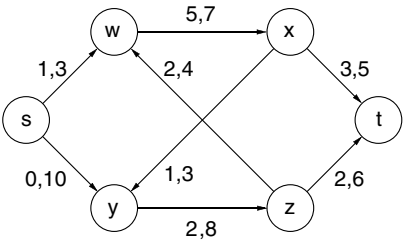


Figure 5.9: An example of a network for which we want to determine whether it has a legitimate flow.

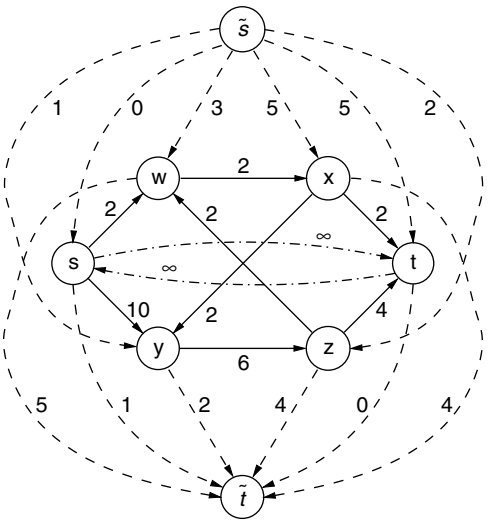


Figure 5.10: The auxiliary network of the example network.

maximum total flow in the auxiliary network of our example is 16. We return to this example shortly.

Theorem 5.4 *The original network* N *has a legitimate flow if and only if the maximum flow of the auxiliary network* $\tilde{\mathsf{N}}$ *saturates all the edges emanating from* $\tilde{s}$*; that is, all edges of* $\mathcal{S}$*.*

Clearly, if all edges of $\mathcal{S}$ are saturated, then so are all edges of $\mathcal{T}$. This follows from the fact that every edge $e \in \mathsf{E}$ contributes $b(e)$ to the capacity of one edge

in $\mathcal{S}$ and of one edge in $\mathcal{T}$. Thus, the sum of capacities of edges of $\mathcal{S}$ is equal to the sum of capacities of edges of $\mathcal{T}$.

Proof: First, we prove the sufficiency of the condition. Assume that a maximum flow function $\tilde{f}$ of $\tilde{N}$ saturates a11 edges of $\mathcal{S}$. Define the following flow function f, for N: For every $e \in E$,

$$f(e) = b(e) + \tilde{f}(e). \tag{5.8}$$

Since $\tilde{f}$ satisfies the edge rule in $\tilde{N}$, it follows that

$$0 \leqslant \tilde{f}(e) \leqslant \tilde{c}(e). \tag{5.9}$$

Thus,

$$b(e) \leqslant b(e) + \tilde{f}(e) \leqslant b(e) + \tilde{c}(e).$$

By Equations 5.8 and 5.7, one gets

$$b(e) \leqslant f(e) \leqslant c(e)\ , \tag{5.10}$$

and f satisfies the edge rule.

Now, let $v \in V \setminus \{s, t\}$. We remind the reader that $\alpha(v)$ is the set of edges that enter v in G, and $\beta(v)$ is the set of edges that emanate from v in G. Let $\tilde{s} \xrightarrow{\sigma} v$ be the new edge that enters v in $\tilde{G}$, and $v \xrightarrow{\tau} \tilde{t}$ be the new edge that emanates from v in $\tilde{G}$. Since $\tilde{f}$ satisfies the vertex rule in $\tilde{N}$, we have

$$\sum_{e \in \alpha(v)} \tilde{f}(e) + \tilde{f}(\sigma) = \sum_{e \in \beta(v)} \tilde{f}(e) + \tilde{f}(\tau)\ . \tag{5.11}$$

By the assumption, σ and τ are saturated. Thus,

$$\sum_{e \in \alpha(v)} \tilde{f}(e) + \tilde{c}(\sigma) = \sum_{e \in \beta(v)} \tilde{f}(e) + \tilde{c}(\tau)\ .$$

By the definitions of $\tilde{c}(\sigma)$ and $\tilde{c}(\tau)$, it follows that

$$\sum_{e \in \alpha(v)} \tilde{f}(e) + \sum_{e \in \alpha(v)} b(e) = \sum_{e \in \beta(v)} \tilde{f}(e) + \sum_{e \in \beta(v)} b(e)\ .$$

By Equation 5.8 and merging the sums on each side of the equation, one gets

$$\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e)\ , \tag{5.12}$$

and the vertex rule is proved. Thus, f is legitimate in N.

Next, we prove the necessity of the condition. Assume there is a legitimate flow f in N. Let us show that there is a maximum flow $\tilde{f}$ in $\tilde{N}$ for which all edges of $\mathcal{S}$ are saturated.

The steps of the previous proof are reversible, with minor modifications. For an edge $e \in E$, use Equation 5.8 to define $\tilde{f}(e)$. For an edge $\sigma \in \mathcal{S}$, define $\tilde{f}(\sigma) = \tilde{c}(\sigma)$, and for $\tau \in \mathcal{T}$ define $\tilde{f}(\tau) = \tilde{c}(\tau)$. Finally, if $F \geqslant 0$ is the total flow according to f, define $\tilde{f}(e'') = F$ and $\tilde{f}(e') = 0$, and if $F < 0$, define $\tilde{f}(e'') = 0$ and $\tilde{f}(e') = -F$.

To show that $\tilde{f}$ observes the edge rule, all we need to do is show it for $e \in E$; obviously, in the remaining edges, $\tilde{f}$ is defined in a way which satisfies the edge rule. Since f is legitimate in N, it satisfies Equation 5.10. By Equations 5.8 and 5.7, one concludes that Equation 5.9 holds.

To show that $\tilde{f}$ observes the vertex rule, first consider a vertex $v \in V \setminus \{s, t\}$. Since f observes the vertex rule in N, Equation 5.12 holds. By reversing the steps of the proof above, one gets Equation 5.11, which implies that v satisfies the vertex rule.

Finally, for vertex s, $\tilde{f}$ satisfies the vertex rule for the following reasons. In N, s may not be balanced. If $F > 0$, then the total outgoing flow is F units greater than the total incoming flow. The total outgoing flow in $\tilde{N}$ in edges of $\beta(s)$ and in τ is as it is in N, and the total incoming flow in $\tilde{N}$ in edges of $\alpha(s)$ and in σ is also as in N. However, $\tilde{f}(e'')$ causes the incoming and outgoing total flows to balance. Similar arguments show that the total flow in t is balanced as well, and that both s and t are balanced when $F < 0$. ■

Let us demonstrate the technique for establishing whether the network has a legitimate flow, and finding one in the case the answer is positive, on our example (Figure 5.9). First, we apply the Dinitz algorithm to the auxiliary network of Figure 5.10 and end up with the flow, as in Figure 5.11. The maximum flow saturates all edges that emanate from $\tilde{s}$, and we conclude that the original network has a legitimate flow. We use Equation 5.8 to define a legitimate flow in the original network; this is shown in Figure 5.12. (Next to each edge e we write $b(e), c(e),$ and $f(e)$, in this order.)

Once a legitimate flow f has been found in N, we turn to the question of optimizing it. First, let us consider the question of maximizing the total flow. One can use the Ford and Fulkerson algorithm, except that the backward labeling must be redefined as follows. An edge $v \xrightarrow{e} u$ is *backwardly useful* for labeling vertex v, if

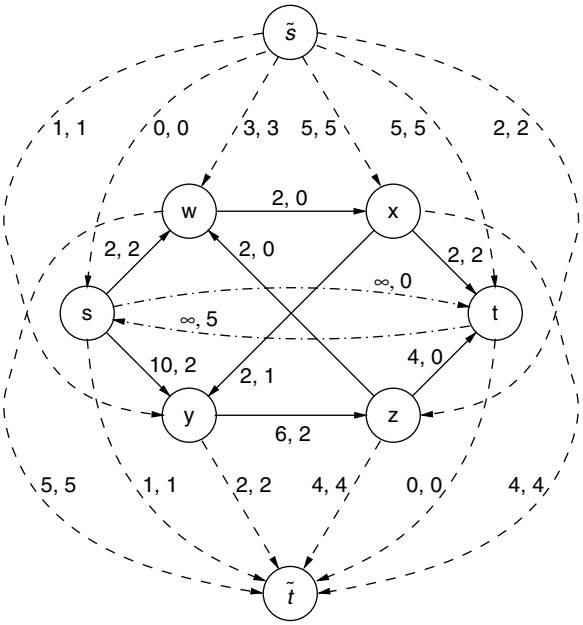


Figure 5.11: A maximum flow in the auxiliary network of the example network.

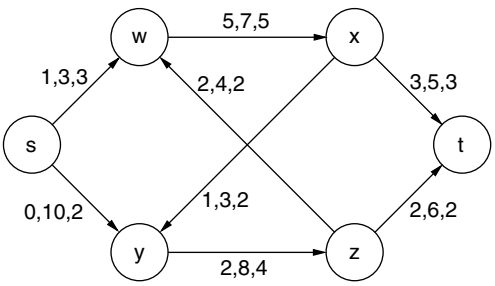


Figure 5.12: A legitimate flow in the original example network.

(i) u is labeled and v is not,
(ii) $f(e) > b(e)$.

The label that v gets is $(e, -)$. In this case, Line 7 of the procedure LABEL (see Algorithm 5.1) is changed to

$$\Delta(e) \leftarrow f(e) - b(e).$$

With this exception, the algorithm is exactly as described in Section 5.2. The proof that the flow is maximum when the algorithm terminates is similar, but we need to redefine the capacity of a cut determined by S (see Equation 5.4) as follows:

$$c(S) = \sum_{e \in (S;\bar{S})} c(e) - \sum_{e \in (\bar{S};S)} b(e) .$$

It is easy to prove that a statement just like Lemma 5.2 still holds; namely,

$$F \leqslant c(S) .$$

Now, the set of labeled vertices S, when the algorithm terminates, satisfies this inequality by equality. Thus, the flow is maximum, and the indicated cut is minimum.

Clearly, the Dinitz algorithm can also be used. It is left as an exercise for the reader to make the necessary changes.

In certain applications, what we want is a minimum flow, that is, a legitimate flow function f for which the total flow F is minimum. Consider a network $N(G(V,E),b,c)$, where function $b : E \to R$ specifies the lower bounds on the flow in the edges, and $c : E \to R$ specifies the upper bounds on the flow in the edges. Let $s,t \in V$ be any two vertices. Denote by $f : E \to R$ a legitimate flow function of the network $(G(V,E),s,t,b,c)$, where s plays the role of the source and t plays the role of the sink. Also, let $F_{s,t}$ be the corresponding total flow. Clearly, f is legitimate in $(G(V,E),t,s,b,c)$ as well, where the roles of the source and the sink are reversed. Also,

$$F_{s,t} = -F_{t,s} .$$

It follows that f is a minimum flow in $(G(V,E),s,t,b,c)$ if and only if f is a maximum flow in $(G(V,E),t,s,b,c)$. Thus, our techniques solve the problem of minimizing the total flow by simply exchanging the roles of s and t.

For a set $\{s\} \subset S \subset V \setminus \{t\}$, let us define $\underline{c}(S)$, (the value of the cut $(S;\bar{S})$ for minimum flow purposes) as follows:

$$\underline{c}(S) = \sum_{e \in (S;\bar{S})} b(e) - \sum_{e \in (\bar{S};S)} c(e) .$$

Clearly, $\underline{c}(S) = -c(\bar{S})$. Now, assume f is a maximum flow in $(G(V,E),t,s,b,c)$. By the max-flow min-cut theorem (Theorem 5.3), which also applies to networks with lower and upper bounds, provided they have legitimate flows, there

is a cut defined by some $\{s\} \subset S \subset V \setminus \{t\}$, such that

$$F_{t,s} = c(\bar{S}) \ .$$

Thus,

$$-F_{s,t} = -\underline{c}(S) \ ,$$

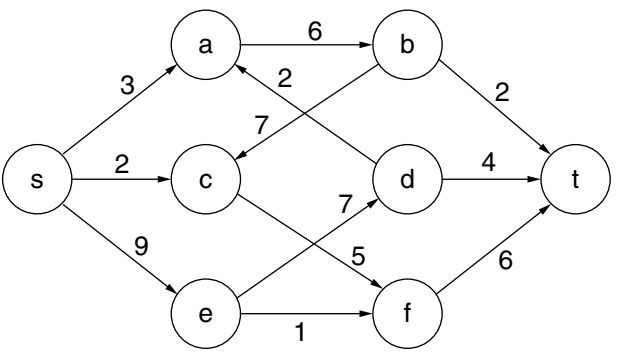
and

$$F_{s,t} = \underline{c}(S) \ .$$

One can prove and use a lemma similar to Lemma 5.2, with the inequality reversed. Thus, $\underline{c}(S)$ is maximum. It follows that for networks with lower and upper bounds, which have legitimate flows, a min-flow max-cut theorem holds.

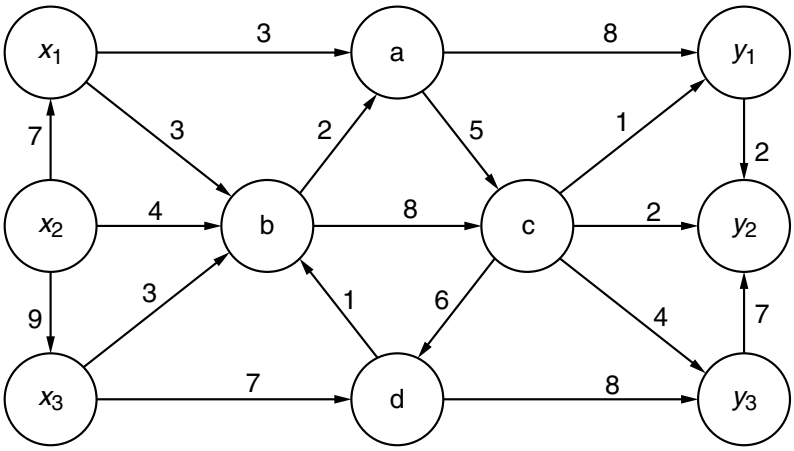
5.5 Problems

Problem 5.1 Find a maximum flow in the network shown below.



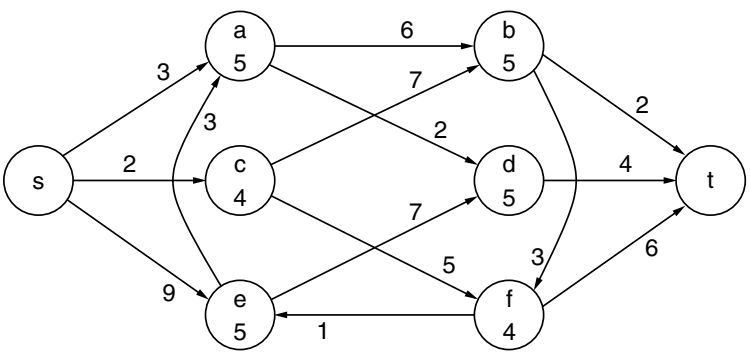
The number next to each edge is its capacity. Show a minimum cut. How many minimum cuts can you find?

Problem 5.2 In the following network x_1, x_2, x_3, are all sources (of the same commodity). The supply available at x_1 is 5, at x_2 is 10, and at x_3 is 5. The



vertices y_1, y_2, y_3, are all sinks. The demand required at y_1 is 5, at y_2 is 10, and at y_3 is 5. Find out whether all requirements can be met simultaneously. (Hint: One way of solving this type of problem is to reduce it to the familiar one-source one-sink format. Introduce auxiliary source s and sink t. Connect s to x_i through a directed edge of capacity equal to x_i's supply. Connect each y_i to t through a directed edge of capacity equal to y_i's demand. Find a maximum flow in the resulting network, and observe whether all demands are met.)

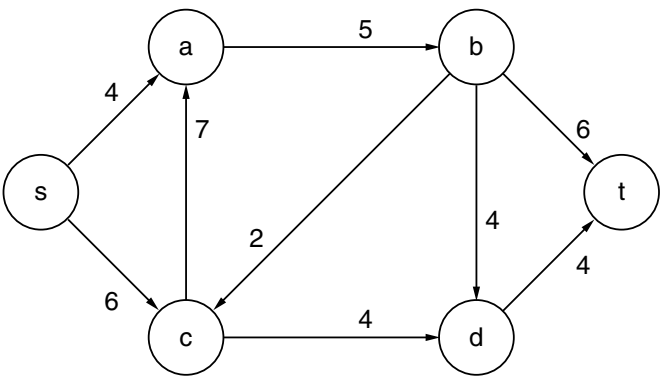
Problem 5.3 In the following network, in addition to the capacities of the edges, each vertex other than s and t has an upper bound on the flow that may



flow through it. These vertex capacities are written below the vertex labels. Find a maximum flow for this network. (Hint: One way of solving this type of problem is to replace each vertex v by two vertices v' and v'' with an edge $v' \xrightarrow{e} v''$, where $c(e)$ is the upper bound on the flow through v. All edges that previously entered v now enter v', and all edges that previously emanated from v now emanate from v''.)

Problem 5.4

(1) Describe an alternative labeling procedure, like that of Ford and Fulkerson, for maximizing the flow, except that the labeling starts at t, and if it reaches s, an augmenting path is found.
(2) Demonstrate your algorithm on the following network:



(3) Describe a method of locating an edge with the property that increasing its capacity increases the maximum flow in the network. (Hint: One way of doing this is to use both source-to-sink and sink-to-source labelings.) Demonstrate your method on the network above.
(4) Does an edge like this always exist? Prove your claim.

Problem 5.5 In a network $N(G(V,E),s,t,c)$, there are two sets, $\{s\} \subset S_1 \subset V \setminus \{t\}$ and $\{s\} \subset S_2 \subset V \setminus \{t\}$, and each of them defines a minimum cut. Prove that each of $S_1 \cup S_2$ and $S_1 \cap S_2$ defines a minimum cut as well.

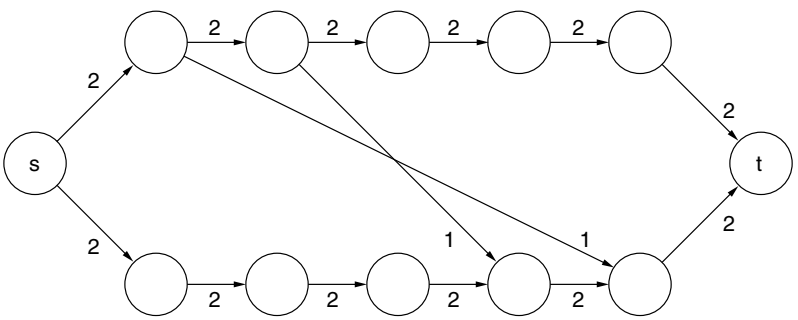
Problem 5.6 Let f be a maximum flow in $N(G(V,E),s,t,c)$, where all edge capacities are positive integers, and for every $e \in E$, $f(e)$ is a nonnegative integer.

The capacity of $u \xrightarrow{e} v$ is reduced by one unit. It is necessary to find a maximum flow in the new network.

Describe an algorithm to achieve this goal whose time complexity is $O(|E|)$. (Hint: Without loss of generality, assume $f(e) = c(e)$. If there is a directed circuit via e that carries flow, reduce the flow on every edge of the circuit by one unit. If not, first locate a directed path from s to t via e that carries flow, and reduce the flow in every edge of the path by one unit, and then look for an augmenting path from s to t in the new network.)

Problem 5.7 Let $G(V, E)$ be a finite directed graph without parallel edges. Describe an algorithm that is a modification of the algorithm of Ford and Fulkerson for finding a maximum flow in a given network $N(G, s, t, c)$, except that, in each stage, one looks for an augmenting path for which Δ, the value by which the flow is increased, is maximum. The time complexity of finding each such path should be $O(|V|^2)$. (Comment: This algorithm can be shown to be polynomial; see [4].)

Problem 5.8 In the following network, the capacities of the edges are written next to them. Use the Dinitz algorithm to find its maximum flow when the initial flow is zero everywhere. How many phases are there in which t is reached?



Problem 5.9 A flow in a network is said to have circuits if there is at least one directed circuit such that on all its edges the flow is positive. Such a circular flow is superfluous since it contributes nothing to the total flow.

- Show that if we start with zero flow everywhere and use the Dinitz algorithm to find a maximum flow, we may end up with a flow that has circuits. (Hint: Consider the graph depicted in Figure 5.13.)

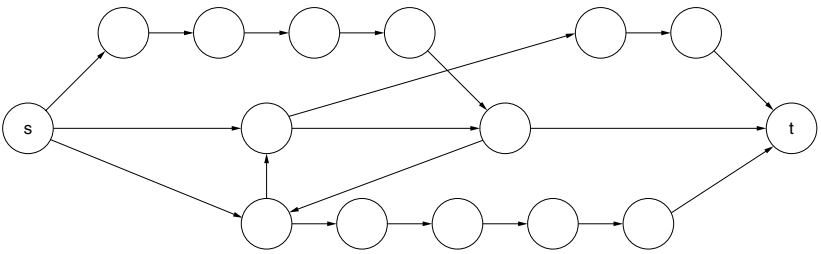


Figure 5.13: Hint for Problem 5.9.

- Describe an $O(|E|^2)$ algorithm to remove all circular flow from a given flow function.

Problem 5.10 Let $N(G(V,E),c)$ be a network where G is a finite directed graph, and c is a capacity function on the edges. For every $x,y \in V$, let F_{xy} denote the maximum flow in case x is the source and y is the sink.

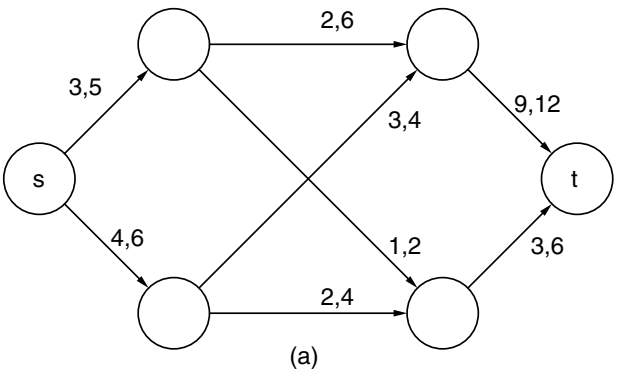
Prove that for every three vertices $u,v,w \in V$, $F_{uw} \geqslant \min\{F_{uv},F_{vw}\}$.

Problem 5.11 Prove that in a network with a lower bound $b(e) \geqslant 0$ for every edge e, but no upper bound ($c(e) = \infty$), there is a legitimate flow if and only if for every edge e, for which $b(e) > 0$, either e is in a directed circuit or e is in a directed path from s to t or from t to s. Show that in such a network, a legitimate flow can be found in time $O(|V| \cdot |E|)$.

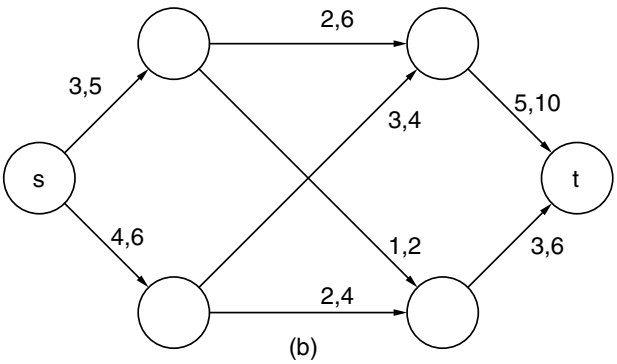
Problem 5.12 Find a minimum flow from s to t for the network of Problem 5.1, where the numbers next to the edges are now assumed to be lower bounds, and there are no upper bounds.

Problem 5.13 The two networks shown below have both lower and upper bounds on the flow through the edges. Which of the two networks has no legitimate flow? Find both a maximum flow and a minimum flow if a legitimate flow exists. If no legitimate flow exists, display a set of vertices that

includes neither the source nor the sink and is required to "produce" flow or to "absorb" it.[5]



(a)



(b)

Problem 5.14 Prove that a network with lower and upper bounds on the flow in the edges has no legitimate flow if and only if there exists a set of vertices which includes the neither source nor the sink and is required to "produce" flow or to "absorb" it.

[5] A set of vertices $A \subset V \setminus \{s,t\}$ is required to *absorb* flow if

$$\sum_{e \in (\bar{A};A)} b(e) > \sum_{e \in (A;\bar{A})} c(e) .$$

5.6 Notes by Andrew Goldberg

The augmenting path algorithm is due to Ford and Fulkerson [7, 6]. Dinitz [2] developed the *blocking flow* algorithm, which runs in polynomial time. The observation that augmenting along a shortest path leads to a polynomial-time algorithm has been made independently by Edmonds and Karp [4]. A more efficient, $O(|V|^3)$, variant of the blocking flow algorithm based on preflows is due to Karzanov [12]. Sleator and Tarjan [14] used the dynamic tree data structure to improve this bound to $O(|V||E|\log|V|)$. This has been further improved to $O(|V||E|\log(|V|^2/|E|))$ by Goldberg and Tarjan [10].

Goldberg and Tarjan [9] developed the push-relabel method that uses preflows and the *push* operation similar to Karzanov's algorithm. However, instead of building a layered network, the algorithm uses the *relabel* operation for fine-grains updates of vertex distances. The push-relabel method leads to the best currently known, strongly polynomial bound of King et al. [13]. This bounds comes very close, but does not quite achieve, $O(|V||E|)$. The push-relabel algorithm is also highly practical when used in combination with efficiency-enhancing heuristics [1].

Karzanov [11] and, independently, Even and Tarjan [5] have shown that on unit capacity networks, the blocking flow algorithm algorithm runs in $O(\min((|E|^{1/2}, |V|^{2/3})|E|)$ time. This leaves a polynomial gap between the general and the unit-capacity cases.

When talking about shortest augmenting paths, one has to assign lengths for residual edges. All polynomial-time algorithms mentioned above use the unit length function. Edmonds and Karp [4] observed that one can use other length functions. However, for a long time nobody was able to use this observation to improve the time bounds. Goldberg and Rao [8] use the binary length function (zero for high- and one for low-capacity edges) to get an $O(\min((|E|^{1/2}, |V|^{2/3})|E|\log\frac{|V|^2}{|E|}\log U)$ bound for networks with integral capacities in the range $[1, U]$. This closes the gap between the general and the unit-capacity case.

Bibliography

[1] B. V. Cherkassky and A. V. Goldberg. "On Implementing Push-Relabel Method for the Maximum Flow Problem." *Algorithmica*, 19:390–410, 1997.

[2] E. A. Dinic. "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation." *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[3] Yefim Dinitz. Dinitz' Algorithm: The original version and Even's version. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Essays in Memory of Shimon Even*, Vol. 3895 of *Lecture Notes in Computer Science*, pp. 218–240. Springer, 2006.

[4] J. Edmonds and R. M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems." *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[5] S. Even and R. E. Tarjan. "Network Flow and Testing Graph Connectivity." *SIAM J. Comput.*, 4:507–518, 1975.

[6] L. R. Ford, Jr. and D. R. Fulkerson. "Maximal Flow Through a Network." *Canadian Journal of Math.*, 8:399–404, 1956.

[7] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[8] A. V. Goldberg and S. Rao. "Beyond the Flow Decomposition Barrier." *J. Assoc. Comput. Mach.*, 45:753–782, 1998.

[9] A. V. Goldberg and R. E. Tarjan. "A New Approach to the Maximum Flow Problem." *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[10] A. V. Goldberg and R. E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation." *Math. of OR*, 15:430–466, 1990.

[11] A. V. Karzanov. "O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh." In *Matematicheskie Voprosy Upravleniya Proizvodstvom*, volume 5. Moscow State University Press, 1973. In Russian; title translation: "On Finding Maximum Flows in Networks with Special Structure and Some Applications."

[12] A. V. Karzanov. "Determining the Maximal Flow in a Network by the Method of Preflows." *Soviet Math. Dok.*, 15:434–437, 1974.

[13] V. King, S. Rao, and R. Tarjan. "A Faster Deterministic Maximum Flow Algorithm." *J. Algorithms*, 17:447–474, 1994.

[14] D. D. Sleator and R. E. Tarjan. "A Data Structure for Dynamic Trees." *J. Comput. System Sci.*, 26:362–391, 1983.

6

Applications of Network Flow Techniques

6.1 Zero-One Network Flow

Several combinatorial problems can be solved through network flow techniques. In the networks we get, the capacity of all the edges is one. To get better algorithms with lower time complexities, we need to study these network flow problems. We follow here the work of Even and Tarjan [1].

Consider a maximum flow problem where for every edge e of $G(V,E)$, $c(e)=1$.

The first observation is that in the Dinitz algorithm for maximal flow in a layered network, each time we find a path, all the edges on it become blocked; in case the last edge leads to a dead end, we backtrack on this edge, and it becomes blocked. Thus, the total number of edge traversals is bounded by $|E|$, and the whole phase is of time complexity $O(|E|)$. Since the number of phases is bounded by $|V|$, the Dinitz algorithm for maximum flow is of complexity $O(|V|\cdot|E|)$.

Our first goal is to prove a better bound yet: $O(|E|^{3/2})$. However, we need to prepare a few results beforehand.

Let $G(V,E)$ be a 0-1 network in which $c(e)=1$ for all $e\in E$ with some integral legal flow function f. Define $\tilde{G}(V,\tilde{E})$ as follows:

(i) If $u \xrightarrow{e} v$ in G, and $f(e)=0$, then $e\in\tilde{E}$.

(ii) If $u \xleftarrow{e} v$ in G, and $f(e)=1$, then $u \xrightarrow{e'} v$ is in $\tilde{G}$. Clearly, e' is a new edge that corresponds to e.

Thus, $|E|=|\tilde{E}|$. Clearly, the useful edges of the layered network that is constructed for G with present flow f, with their direction of usefulness, are all edges of $\tilde{G}$.

Let us denote by $(S\mathcal{I}\bar{S})_G$, where $s \in S$, $t \notin S$, and $\bar{S} = V - S$, the set of edges that emanate from a vertex of S and enter a vertex of $\bar{S}$, and let $c(S,G)$ be the capacity of the corresponding cut in G. Also, let M be the total maximum flow in G, while F is the total present flow.

Lemma 6.1 $\tilde{M} = M - F$.

Proof: Let S be a subset of V such that $s \in S$ and $t \notin S$. The definition of $\tilde{G}$ implies that

$$c(S,\tilde{G}) = |(S;\bar{S})_{\tilde{G}}| = \sum_{e \in (S;\bar{S})_G} (1 - f(e)) + \sum_{e \in (\bar{S};S)_G} f(e).$$

However,

$$F = \sum_{e \in (S;\bar{S})_G} f(e) - \sum_{e \in (\bar{S};S)_G} f(e).$$

Thus,

$$c(S,\tilde{G}) = |(S;\bar{S})_G| - F = c(S,G) - F.$$

This implies that the minimum cut of G corresponds to the minimum cut of $\tilde{G}$; that is, is defined by the same S. By the max-flow min-cut theorem (Theorem 5.1), the capacity of a minimum cut of $\tilde{G}$ is $\tilde{M}$ (the maximum total flow in $\tilde{G}$). Thus, the lemma follows. ■

Lemma 6.2 *The length of the layered network for the* 0-1 *network defined by* $G(V,E)$ *(with a given* s *and* t*) and zero flow everywhere is at most* $|E|/M$.

Proof: We remind the reader that V_i is the set of vertices of the i-th layer of the layered network, and E_i is the set of edges from V_{i-1} to V_i. Since $f(e) = 0$ for every $e \in E$, the useful directions are all forward. Thus, every E_i is equal to $(S;\bar{S})_G$, where $S = V_0 \cup V_1 \cup \ldots \cup V_{i-1}$. Thus, by Lemma 5.1,

$$M \leqslant |E_i|. \tag{6.1}$$

Summing up (6.1), for every $i = 1,2,\ldots,l$ where l is the length of the layered network, we get $l \cdot M \leqslant |E|$, or

$$l \leqslant |E|/M.$$

■

Theorem 6.1 *For* 0-1 *networks, Dinitz's algorithm is of time complexity* $O(|E|^{3/2})$.

Proof: If $M \leqslant |E|^{1/2}$, then the number of phases is bounded by $|E|^{1/2}$, and the result follows. Otherwise, consider the phase during which the total flow reaches $M - |E|^{1/2}$. The total flow F in $G(V,E)$ when the layered network for this phase is constructed satisfies

$$F < M - |E|^{1/2}.$$

This layered network is identical with the one constructed for $\tilde{G}$, with zero flow everywhere. Thus, by Lemma 6.1;

$$\tilde{M} = M - F > |E|^{1/2}.$$

By Lemma 6.2, the length l of this layered network satisfies

$$l \leqslant |E|/\tilde{M} < |E|/|E|^{1/2} = |E|^{1/2}.$$

Therefore, the number of phases up to this point is at most $|E|^{1/2} - 1$, and since the number of additional phases to completion is at most $|E|^{1/2}$, the total number of phases is at most $2|E|^{1/2}$. ■

A 0-1 network is of *type* 1 if it has no parallel edges. For such a network we can prove another upper bound on the time complexity. First, we prove a lemma similar to Lemma 6.2.

Lemma 6.3 *Let* $G(V,E)$ *define a* 0-1 *network of type 1, with maximum total flow* M *from* s *to* t. *The length* l *of the first layered network, when the flow is zero everywhere, is at most* $2|V|/M^{1/2}$.

Proof: Let V_i be the set of vertices of the i-th layer. Since there are no parallel edges, the set of edges, E_{i+1}, from V_i to V_{i+1} in the layered network satisfies $|E_{i+1}| \leqslant |V_i| \cdot |V_{i+1}|$ for every $i = 0, 1, \ldots, l-1$. Since each $|E_i|$ is the capacity of a cut, we get that

$$M \leqslant |V_i| \cdot |V_{i+1}|.$$

Thus, either $V_i \geqslant M^{1/2}$ or $|V_{i+1}| \geqslant M^{1/2}$. Clearly,

$$|V| \geqslant \sum_{i=0}^{\ell} |V_i| \geqslant \left\lfloor \frac{l+1}{2} \right\rfloor \cdot M^{1/2}.$$

Thus,

$$\frac{|V|}{M^{1/2}} \geqslant \left\lfloor \frac{l+1}{2} \right\rfloor \geqslant \frac{\ell}{2},$$

and

$$\frac{2|V|}{M^{1/2}} \geqslant \ell,$$

and the lemma follows. ■

Theorem 6.2 *For* 0-1 *networks of type* 1*, Dinitz's algorithm has time complexity* $O(|V|^{2/3} \cdot |E|)$.

Proof: If $M \leqslant |V|^{2/3}$, the result follows immediately. Let F be the total flow when the layered network, for the phase during which the total flow reaches the value $M - |V|^{2/3}$, is constructed. This layered network is identical with the first layered network for $\tilde{G}$ with zero flow everywhere. $\tilde{G}$ may not be of type 1 since it may have parallel edges, but it can have at most two parallel edges from one vertex to another; if e_1 and e_2 are antiparallel in G, $f(e_1) = 0$ and $f(e_2) = 1$, then in $\tilde{G}$ there are two parallel edges: e_1 and e_2. A result similar to Lemma 6.3 yields that

$$l < 2^{3/2}|V|/\tilde{M}^{1/2}.$$

Since $\tilde{M} = M - F > M - (M - |V|^{2/3}) = |V|^{2/3}$, we get

$$l < \frac{2^{3/2}|V|}{|V|^{1/3}} = 2^{3/2} \cdot |V|^{2/3}.$$

Thus, the number of phases up to this point is $O(|V|^{2/3})$. Since the number of phases from here to completion is at most $|V|^{2/3}$, the total number of phases is $O(|V|^{2/3})$. ■

In certain applications, the networks that arise satisfy the condition that for each vertex other than s or t, there is either only one edge emanating from it or only one edge entering it. Such 0-1 networks are called *type* 2.

Lemma 6.4 *Let the* 0-1 *network defined by* $G(V, E)$ *be of type* 2*, with maximum total flow* M *from* s *to* t*. The length* l *of the first layered network, when the flow is zero everywhere, is at most* $(|V| - 2)/M + 1$.

Proof: The structure of G implies that a max-flow in G can be decomposed into vertex-disjoint directed paths from s to t; that is, no two of these paths share any vertices, except their common start-vertex s and end-vertex t. (The flow may imply some directed circuits that are vertex-disjoint from each other and from the paths above, except possibly at s or t. These circuits are of no interest to us). The number of these paths is equal to M. Let λ be the length of a

shortest of these paths. Thus, each of the paths uses at least $\lambda - 1$ intermediate vertices. We have

$$M \cdot (\lambda - 1) \leqslant |V| - 2,$$

which implies $\lambda \leqslant (|V| - 2)/M + 1$. However, $l \leqslant \lambda$. Thus, the lemma follows. ■

Lemma 6.5 *If the* 0-1 *network defined by* G *is of type* 2*, and if the present flow function is* f*, then the corresponding* $\tilde{G}$ *also defines a type* 2*,* 0-1 *network.*

Proof: Clearly $\tilde{G}$ defines a 0-1 network. What remains to be shown is that in $\tilde{G}$, for every vertex v, there is either one emanating edge or only one entering edge. If there is no flow through v (per f), then, in $\tilde{G}$, v has exactly the same incident edges, and the condition continues to hold. If the flow going through v is 1, (clearly, it cannot be more) assume that it enters via e_1 and leaves via e_2. In $\tilde{G}$, neither of these two edges appears, but two edges e_1' and e_2' are added, which have directions opposite to e_1 and e_2, respectively. The other edges of G that are incident to v remain intact in $\tilde{G}$. Thus, the numbers of incoming edges and outgoing edges of v remain the same. Since G is of type 2, so is $\tilde{G}$. ■

Theorem 6.3 *For a* 0-1 *network of type* 2*, Dinitz's algorithm is of time complexity* $O(|V|^{1/2} \cdot |E|)$.

Proof: If $M \leqslant |V|^{1/2}$, then the number of phases is bounded by $|V|^{1/2}$, and the result follows. Otherwise, consider the phase during which the total flow reaches the value $M - |V|^{1/2}$. Therefore, the layered network for this phase is constructed when $F < M - |V|^{1/2}$. This layered network is identical with the first for $\tilde{G}$, with zero flow everywhere. Also, by Lemma 6.5, $\tilde{G}$ is of type 2. Thus, by Lemma 6.4, the length l of the layered network is at most $(|V| - 2)/\tilde{M} + 1$. Now, $\tilde{M} = M - F > M - (M - |V|^{1/2}) = |V|^{1/2}$. Thus,

$$l \leqslant \frac{|V| - 2}{|V|^{1/2}} + 1 = O(|V|^{1/2}).$$

Therefore, the number of phases up to this one is at most $O(|V|^{1/2})$. Since the number of phases to completion is at most $|V|^{1/2}$ more, the total number of phases is at most $O(|V|^{1/2})$. ■

6.2 Vertex Connectivity of Graphs

Intuitively, the connectivity of a graph is the minimum number of elements whose removal from the graph disconnect it. There are four cases. We may

discuss undirected graphs or digraphs; we may discuss the elimination of edges or vertices. We start with the problem of determining the vertex-connectivity of an undirected graph. The other cases, which are simpler, are discussed in the next section.

Let $G(V,E)$ be a finite undirected graph, with no self-loops and no parallel edges. A set of vertices, S, is called an (a,b) *vertex separator* if $\{a,b\} \subseteq V \setminus S$, and every path connecting a and b passes through at least one vertex of S. Clearly, if a and b are connected by an edge, no (a,b) vertex separator exists. Let $a \not- b$ mean that there is no such edge. In this case, let $N(a,b)$ be the least cardinality of an (a,b) vertex separator. Also, let $p(a,b)$ be the maximum number of pairwise vertex-disjoint paths connecting a and b in G; clearly, all these paths share the two end-vertices, but no other vertex appears on more than one of them.

Theorem 6.4 *If* $a \not- b$ *then* $N(a,b) = p(a,b)$.

This is one of the variations of Menger's theorem [2]. It is not only reminiscent of the max-flow min-cut theorem, but can be proved by it. Dantzig and Fulkerson [3] pointed out how this can be done, and we shall follow their approach.

Proof: Construct a digraph $\bar{G}(\bar{V},\bar{E})$ as follows. For every $v \in V$ put two vertices v' and v'' in $\bar{V}$ with an edge $v' \xrightarrow{e_v} v''$. For every edge $u \overset{e}{\text{---}} v$ in G, put two edges $u'' \xrightarrow{e'} v'$ and $v'' \xrightarrow{e''} u'$ in $\bar{G}$. Define now a network, with digraph $\bar{G}$, source a'', sink b', unit capacities for all the edges of the e_v type (let us call them *internal edges*), and infinite capacity for all the edges of the e' and e'' type (called *external edges*). For example, in Figure 6.1(b) the network for G, as shown in Figure 6.1(a), is demonstrated.

We now claim that $p(a,b)$ is equal to the total maximum flow F (from a'' to b') in the corresponding network. First, assume we have $p(a,b)$ vertex disjoint paths from a to b in G. Each such path, $a \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \overset{e_3}{\text{---}} \dots \overset{e_{l-1}}{\text{---}} v_{l-1} \overset{e_l}{\text{---}} b$, indicates a direct path in $\bar{G}$:

$$a'' \xrightarrow{e'_1} v'_1 \xrightarrow{e_{v_1}} v''_1 \xrightarrow{e'_2} v'_2 \xrightarrow{e_{v_2}} v''_2 \xrightarrow{e'_3} \dots \xrightarrow{e'_{l-1}} v'_{l-1} \xrightarrow{e_{v_{l-1}}} v''_{l-1} \xrightarrow{e'_l} b'$$

These directed paths are vertex-disjoint, and each can be used to flow one unit from a'' to b'. Thus,

$$F \geqslant p(a,b).$$

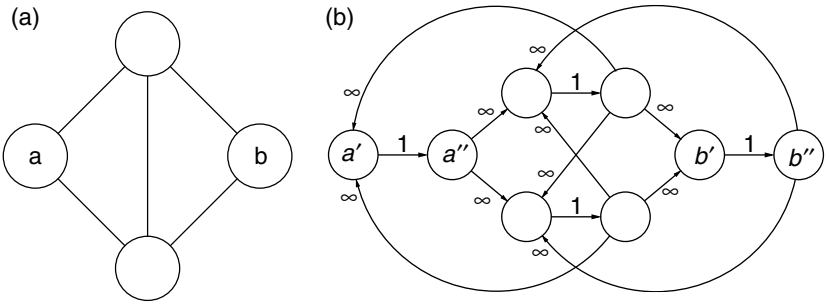


Figure 6.1: Construction in the proof of Theorem 6.4

Next, assume f is a flow function which achieves a maximum total flow F in the network. We may assume that $f(e)$ is either zero or one, for every $e \in \bar{E}$. This follows from the fact that one can use the Ford and Fulkerson algorithm, or the Dinitz algorithm, in which the flow is always integral. Also, the edges with infinite capacity enter a vertex with a single outgoing edge whose capacity is one and which must satisfy the conversation rule (C2), or they emanate from a vertex with only one incoming edge of unit capacity which is subject to C2; thus, the flow through them is actually bounded from above by one. (We have assigned them infinite capacity for convenience reasons, which will become clear shortly.) Therefore, the total flow F can be decomposed to paths, each describing the way that one unit reaches b' from a''. These paths are vertex-disjoint, since the flow through v' or v'', if $v \notin \{a, b\}$, is bounded by one. Each indicates a path in G. These F paths in G are vertex-disjoint too. Thus,

$$F \leqslant p(a, b).$$

We conclude that $F = p(a, b)$.

By the max-flow min-cut theorem, F is equal to the capacity $c(S)$ of some cut defined by some $S \subset \bar{V}$, such that $a'' \in S$ and $b' \notin S$. Since

$$c(S) = \sum_{e \in (S;\bar{S})} c(e),$$

the set $(S;\bar{S})$ consists of internal edges only. Now, every directed path from a'' to b' in $\bar{G}$ uses at least one edge of $(S;\bar{S})$. Thus, every path from a to b in G uses at least one vertex v such that $e_v \in (S;\bar{S})$. Therefore, the set $R = \{v \mid v \in V \text{ and } e'_v \in (S;\bar{S})\}$ is an (a, b) vertex separator. Clearly $|R| = c(S)$.

Thus, we have an (a,b) vertex separator whose cardinality is F. Proving that $N(a,b) \leqslant F \leqslant p(a,b)$.

Finally, it is easy to see that $N(a,b) \leqslant p(a,b)$, since every path from a to b uses at least one vertex of the separator, and no two paths can use the same one. ■

The algorithm suggested in the proof, for finding $N(a,b)$, when the Dinitz algorithm is used to solve the network problem, is of time complexity $O(|V|^{1/2}\cdot|E|)$. This results from the following considerations. The number of vertices in $\bar{G}$ is $2|V|$; the number of edges is $|V|+2|E|$. Assuming $|E| \geqslant |V|$, we have $|\bar{V}| = O(|V|)$ and $|\bar{E}| = O(|E|)$. Since we can assign unit capacity to all the edges without changing the maximum total flow, the network is of type 2. By Theorem 6.3, the algorithm is of time complexity $O(|V|^{1/2}\cdot|E|)$. We can even find a minimum (a,b) vertex separator as follows: Once the flow is maximum, change the capacity of the external edges back to ∞ and apply the construction of the layered network. The set of vertices which appear also in this layered network, S, defines a minimum cut which consists of internal edges only. Let R be the vertices of G which correspond to the internal edges in $(S;\bar{S})$. R is a minimum (a,b) vertex separator in G. This addition work is of time complexity $O(|E|)$.

The *vertex connectivity*, c, of an undirected graph $G(V,E)$ is defined as follows:

(i) If G is completely connected, (i.e., every two vertices are connected by an edge), then $c = |V| - 1$.
(ii) If G is not completely connected, then

$$c = \min_{a \neq b} N(a,b).$$

Lemma 6.6 *If* G *is not completely connected, then*

$$\min_{a \neq b} p(a,b) = \min_{a,b} p(a,b);$$

namely, the smallest value of $p(a,b)$ occurs also for some two vertices a and b that are not connected by an edge.

Proof: Let a,b be a pair of vertices such that $a \overset{e}{\text{—}} b$ and $p(a,b)$ is minimum over all pairs of vertices of the graph. Let G' be the graph obtained from G by dropping e. Clearly, the number of vertex disjoint paths connecting a and b in

G', $p'(a,b)$, satisfies

$$p'(a,b) = p(a,b) - 1.$$

Also, since $a \not\rightarrow b$ in G', then by Theorem 6.4, there is an (a,b) vertex separator R in G' such that $p'(a,b) = |R|$.

If $|R| = |V| - 2$, then $p(a,b) = |V| - 1$, and $p(a,b)$ cannot be the least of all $\{p(u,v) \mid u,v \in V\}$, since for any $u \not\rightarrow v$, $p(u,v) \leqslant |V| - 2$. Hence, $|R| < |V| - 2$. Therefore, there must be some vertex $v \in V - (R \cup \{a,b\})$. Now, without loss of generality, we may assume that R is also an (a,v) vertex separator (or exchange a and b). Thus, $a \not\rightarrow v$ in G and $R \cup \{b\}$ is an (a,v) vertex separator in G. We now have

$$p(a,v) \leqslant |R| + 1 = p(a,b),$$

and the lemma follows. ■

Theorem 6.5 $c = \min_{a,b} p(a,b)$.

Proof: If G is completely connected, then for every two vertices a and b, $p(a,b) = |V| - 1$, and the theorem holds. If G is not completely connected, then, by definition,

$$c = \min_{a \not\rightarrow b} N(a,b).$$

By Theorem 6.4, $\min_{a \not\rightarrow b} N(a,b) = \min_{a \not\rightarrow b} p(a,b)$. Now by Lemma 6.6, $\min_{a \not\rightarrow b} p(a,b) = \min_{a,b} p(a,b)$. ■

We can use the intermediate result,

$$c = \min_{a \not\rightarrow b} p(a,b),$$

to compute the vertex connectivity of G with time complexity $O(|V|^{5/2} \cdot |E|)$. However, a slightly better bound can be obtained.

Lemma 6.7 $c \leqslant 2|E|/|V|$.

Proof: The vertex (or edge) connectivity of a graph cannot exceed the degree of any vertex. Thus,

$$c \leqslant \min_{v} d(v).$$

Also,

$$\sum_{v} d(v) = 2 \cdot |E|.$$

Thus, $\min_v d(v) \leqslant 2 \cdot |E|/|V|$, and the lemma follows. ■

```
Procedure VERTEX-CONNECTIVITY(V, E)

1   Order the vertices v_1, v_2, ..., v_|V| in such a way that v_1 ≠ v for some v.
2   γ ← ∞
3   i ← 1
4   while i ≤ γ do
5       for every v such that v_i ≠ v do
6           γ ← min{γ, N(v_i, v)}
7   return γ.
```

Algorithm 6.1: Constructing the vertex connectivity of a graph $G = (V, E)$ that is not completely connected.

A procedure to find the vertex connectivity c of a graph G that is not completely connected is listed in Algorithm 6.1.

Theorem 6.6 *The procedure VERTEX-CONNECTIVITY terminates with* $\gamma = c$.

Proof: Clearly, after the first computation of $N(v_1, v)$ for some $v_1 \neq v$, γ satisfies

$$c \leqslant \gamma \leqslant |V| - 2. \tag{6.2}$$

From there on, γ can only decrease, but (6.2) still holds. Thus, for some $k \leqslant |V| - 1$, the procedure will terminate. When it does, $k \geqslant \gamma + 1 \geqslant c + 1$.

By definition, c equal to the cardinality of a minimum vertex separator R of G. Thus, at least one of the vertices $v_1, v_2, \ldots, v_k$ is not in R, say v_i. R separates the remaining vertices into at least two sets, such that each path from a vertex of one set to a vertex of another passes through at least on vertex of R. Thus, there exists a vertex v such that $N(v_i, v) \leqslant |R| = c$, and therefore $\gamma \leqslant c$. ■

Clearly, the time complexity of this procedure is $O(c \cdot |V|^{3/2} \cdot |E|)$. By Lemma 6.7, this is bounded by $O(|V^{1/2} \cdot |E|^2)$.

If $c = 0$, then G is not connected. We can use DFS (or BFS) to test whether this is the case in $O(|E|)$ time. If $c = 1$, then G is separable, and as we saw in Section 3.2, this can be tested also by DFS in $O(|E|)$ time. This algorithm determines also whether $c \geqslant 2$, that is, whether it is nonseparable. Before we discuss testing

for a given k, whether $c \geqslant k$, let us consider the following interesting theorem about equivalent conditions for G to be nonseparable.[1]

Theorem 6.7 *Let $G(V, E)$ be an undirected graph with $|V| > 2$ and no isolated vertices.* [2] *The following six statements are equivalent.*

1. G *is nonseparable.*
2. *For every two vertices* x *and* y *there exists a simple circuit which goes through both.*
3. *For every two edges* e_1 *and* e_2 *there exists a simple circuit which goes through both.*
4. *For every two vertices* x *and* y *and an edge* e *there exists a simple path from* x *to* y *which goes through* e.
5. *For every three vertices* x, y *and* z *there exists a simple path from* x *to* z *which goes through* y.
6. *For every three vertices* x, y *and* z *there exists a simple path from* x *to* z *which avoids*

Proof: First we prove that (1) is equivalent to (2).

(1) $\Rightarrow$ (2): Since G is nonseparable, $c \geqslant 2$. By Theorem 6.5, for every two vertices x and y $p(x, y) \geqslant 2$; thus, there is a simple circuit that goes through x and y.

(2) $\Rightarrow$ (1): There cannot exist a separation vertex in G, since every two vertices lie on some common simple circuit.

Next, let us show that (1) and (3) are equivalent.

(1) $\Rightarrow$ (3): From G construct G' as follows. Remove the edges $u_1 \xrightarrow{e_1} v_1$ and $u_2 \xrightarrow{e_2} v_2$ (without removing any vertices). Add two new vertices, x and y, and four new edges: $u_1 \text{---} x \text{---} v_1$, $u_2 \text{---} y \text{---} v_2$. Clearly, none of the old vertices become separation vertices by this change. Also, x cannot be a separation vertex, or either u_1 or v_1 are separation vertices in G. (Here, $|V| > 2$ is used.) Thus, G' is nonseparable. Hence, by the equivalence of (1) and (2), G' satisfies (2). Therefore, there exists a simple circuit in G' that goes through x and y. This circuit indicates a circuit through e_1 and e_2 in G.

(3) $\Rightarrow$ (1): Let x and y be any two vertices. Since G has no isolated vertices, there is an edge e_1 incident to x and an edge e_2 incident to y. (If $e_1 = e_2$, choose any other edge to replace e_2; the replacement need not even be incident to y; the replacement exists since there is at least one other vertex, and it is not isolated.)

[1] Many authors use the term "biconnected" to mean nonseparable. I prefer to call a graph biconnected if $c = 2$.

[2] Namely, for every $v \in V$, $d(v) > 0$. G has been assumed to have no self-loops.

By (3) there is a simple circuit through e_1 and e_2, and therefore a circuit through x and y. Thus, (2) holds, and (1) follows.

Now, let us prove that (3) $\Rightarrow$ (4) $\Rightarrow$ (5) $\Rightarrow$ (6) $\Rightarrow$ ((3)).

(3) $\Rightarrow$ (4): Since (3) holds, the graph G is nonseparable. Add a new edge $x \overset{e'}{\text{---}} y$, if such does no exist already in G. Clearly, the new graph G', is still nonseparable. By the equivalence of (1) and (3), G' satisfy statement (3). Thus, there is a simple circuit which goes through e and e' in G'. Therefore, there is a simple path in G from x to y through e.

(4) $\Rightarrow$ (5): Let e be an edge incident to vertex y; such an edge exists, since there are no isolated vertices in G. By (4), there is a simple path from x to z through e. Thus, this path goes through y.

(5) $\Rightarrow$ (6): Let p be a simple path which goes from x to y through z; such a path exists, since (5) holds for every three vertices. The first part of p, from x to z does not pass through y.

(6) $\Rightarrow$ (1): If (6) holds, then there cannot be any separation vertex in G. ■

Let us now return to the problem of testing the vertex connectivity of a given graph G; that is, testing whether c is greater than or equal to a given positive integer k. We have already seen that for $k = 1$ and 2, there is an $O(|E|)$ algorithm. Hopcroft and Tarjan [4] showed that $k = 3$ can also be tested in linear time, but their algorithm is quite complicated and does not seem to generalize for higher values of k. Let us present a method suggested by Kleitman [5] and improved by Even [6].

Let $L = \{v_1, v_2, \ldots, v_l\}$ be a subset of V, where $l \geqslant k$. Define $\tilde{G}$ as follows: $\tilde{G}$ includes all the vertices and edges of G. In addition, it includes a new vertex s connected by an edge to each of the vertices of L; $\tilde{G}$ is called the auxiliary graph.

Lemma 6.8 *Let* $u \in V - L$. *If* $p(v_i, u) \geqslant k$ *in* G, *for every* $v_i \in L$, *then, in* $\tilde{G}$, $p(s, u) \geqslant k$.

Proof: Assume not. Then $p(s, u) < k$. By Theorem 6.4, there exists a (s, u) vertex separator S in $\tilde{G}$ such that $|S| < k$. Let R be the set of vertices such that all paths in $\tilde{G}$ from s to $v \in R$ pass through at least one vertex of S. Clearly, $v_i \notin R$, since v_i is connected by an edge to s. However, since $l \geqslant k > |S|$, there exists some $1 \leqslant i \leqslant l$ such that $v_i \notin S$. All paths from v_i to u go through vertices of S. Thus, $p(v_i, u) \leqslant |S| < k$, contradicting the assumption. ■

Let $V = \{v_1, v_2, \ldots, v_n\}$. Let j be the least integer such that for some $i < j$, $p(v_i, v_j) < k$ in G.

Lemma 6.9 *Let* j *be as defined above and* $\tilde{G}$ *be the auxiliary graph for* $L = \{v_1, v_2, \ldots, v_{j-1}\}$. *In* $\tilde{G}$, $p(s, v_j) < k$

Proof: Consider a minimum (v_i, v_j) vertex separator S. By Theorem 6.4, $|S| < k$. Let R be the set of all vertices $v \in V$ such that all the paths from v_i to v in G pass through vertices of S. Clearly, $v_j \in R$. If for some $j' < j$, $v_{j'} \in R$, then $p(i, j') \leqslant |S| < k$, and the choice of j is wrong. Thus, v_j is the least vertex in R (i.e. the vertex for which the subscript is minimum). Hence, $L \cap R = \emptyset$. Thus, in $\tilde{G}$, S is an (s, v_j) vertex separator, and $p(s, v_j) < k$. ■

The following algorithm determines whether the vertex connectivity of a given undirected graph $G(V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, is at least k.

1. For every i and j such that $1 \leqslant i < j \leqslant k$, check whether $p(v_i, v_j) \geqslant k$. If for some i and j this test fails then halt; G's connectivity is less than k.
2. For every j such that $k+1 \leqslant j \leqslant n$, form $\tilde{G}$ (with $L = \{v_1, v_2, \ldots, v_{j-1}\}$), and check whether, in $\tilde{G}$, $p(s, v_j) \geqslant k$. If for some j this test fails, then halt; G's connectivity is less than k.
3. Halt; the connectivity of G is at least k.

The proof for the algorithm's validity is as follows: If G's connectivity is at least k, then clearly no failure will be detected in Step (6.2). Also, by Lemma 6.8, no failure will occur in Step (6.2), and the algorithm will halt in Step (6.2) with the right conclusion. If G's connectivity is less than k, and it is not detected directly in Step (6.2) then, by Lemma 6.9, it will be detected in Step (6.2).

Step (6.2) takes $O(k^3 \cdot |E|)$ steps, since we have to solve $k(k-1)/2$ flow problems. In each we have to find k augmenting paths, and each path takes $O(|E|)$ steps to find.

Step (6.2) takes $O(k \cdot |V| \cdot |E|)$ steps, since we have to solve $|V| - k$ flow problems, again for each up to total flow k.

Thus, if $k \leqslant |V|^{1/2}$ then the time complexity of the algorithm is $O(k \cdot |V| \cdot |E|)$.

The readers who are familiar with the interesting result of Gomory and Hu [7] for finding all $|V| \cdot (|V| - 1)$ total flows, for all source-sink pairs in an undirected network by solving only $|V| - 1$ network flow problems, should notice that this technique is of no help in our problem. The reason for that is that even if G is undirected, the network we get for vertex connectivity testing is directed.

6.3 Connectivity of Digraphs and Edge Connectivity

First, let us consider the problem of vertex-connectivity of a digraph $G(V, E)$. The definition of an (a, b) vertex separator is the same as in the undirected case,

except that now all we are looking at are directed paths from a to b; i.e., an (a,b) *vertex separator* is a set of vertices S such that $\{a,b\} \cap S = \emptyset$ and every directed path from a to b passes through at least one vertex of S. Accordingly, $N(a,b)$ and $p(a,b)$ are defined. The theorem analogous to Theorem 6.4, still holds, except that the algorithm is even simpler: For every edge $u \xrightarrow{e} v$ in G there is only one edge $u'' \xrightarrow{e'} v'$ in $\bar{G}$.

The *vertex connectivity*, c, of a digraph $G(V,E)$ is defined as follows:

(i) If G is completely connected, (i.e., for every two vertices a and b, there are edges $a \longrightarrow b$ and $b \longrightarrow a$), then $c = |V| - 1$.
(ii) If G is not completely connected, then

$$c = \min_{a \not\to b} N(a,b).$$

The lemma analogous to Lemma 6.6 still holds, and the proof goes along the same lines. Also, the theorem analogous to Theorem 6.5 holds, and the complexity it yields is the same. If G has no parallel edges, a statement like Lemma 6.7 holds, and the procedure and the proof of its validity (Theorem 6.6) extend to the directed case, except that for each v_i, we compute both $N(v_i, v)$ and $N(v, v_i)$.

The algorithm for testing k connectivity extends also to the directed case, and again all we need to change is that whenever $p(a,b)$ was computed, we now have to compute both $p(a,b)$ and $p(b,a)$.

Let us now consider the case of edge connectivity both in graphs and digraphs.

Let $G(V,E)$ be an undirected graph. A set of edges, T, is called an (a,b) *edge separator* if every path from a to b passes through at least one edge of T. Let $M(a,b)$ be the least cardinality of an (a,b) edge separator. Let $p(a,b)$ be now the maximum number of edge disjoint paths which connect a with b.

Theorem 6.8 $M(a,b) = p(a,b)$.

The proof is similar to that of Theorem 6.4, only simpler. There is no need to split vertices. Thus, in $\bar{G}$, $\bar{V} = V$. We still represent each edge $u \text{—} v$ of G by two edges $u \xrightarrow{e'} v$ and $v \xrightarrow{e''} u$ in $\bar{G}$. There is no loss of generality in assuming that the flow function in $\bar{G}$ satisfies the condition that either $f(e') = 0$ or $f(e'') = 0$; for if $f(e') = f(e'') = 1$ then replacing both by 0 does not change the total flow. The rest of the proof raises no difficulties.

The *edge connectivity*, c, of a graph G is defined by $c = \min_{a,b} M(a,b)$. By Theorem 6.8 and its proof, we can find c by the network flow technique. The

networks we get are of type 1. Both Theorem 6.1 and Theorem 6.2 apply. Thus, each network flow problem is solvable by Dinitz's algorithm with complexity $O(\min\{|E|^{3/2}, |V|^{2/3} \cdot |E|\})$.

Let T be a minimum edge separator in G; that is, $|T| = c$. Let v be any vertex of G. For every vertex v', on the other side of T, $M(v, v') = c$. Thus, in order to determine c, we can use

$$c = \min_{v' \in V - \{v\}} M(v, v').$$

We need to solve at most $|V| - 1$ network flow problems. Thus, the complexity of the algorithm is $O(|V| \cdot |E| \cdot \min\{|E|^{1/2}, |V|^{2/3}\})$.

In the case of edge connectivity of digraphs, we need to consider directed paths. The definition of an (a, b) *edge separator* is accordingly a set of edges, T, such that every directed path from a to b uses at least one edge of T. The definition of $p(a, b)$ again uses directed paths, and the proof of the statement analogous to Theorem 6.8 is the easiest of all, since $\bar{G}$ is now G with no changes.

In the definition of c, the edge connectivity, we need the following change:

$$c = \min\{M(a, b) \mid (a, b) \in V \times V\},$$

namely, we need to consider all ordered pairs of vertices.

The networks we get are still of type 1 and the complexity of each is still $O(|E| \cdot \min\{|E|^{1/2}, |V|^{2/3}\})$. The approach of testing for one vertex v, both $M(v, v')$ and $M(v', v)$ for all $v' \in V - \{v\}$ still works, to yield the same complexity: $O(|V| \cdot |E| \cdot \min\{|E|^{1/2}, |V|^{2/3}\})$. However, the same result, with an improvement only in the constant coefficient follows from the following interesting observation of Schnorr [8], which applies both to the directed and undirected edge connectivity problems.

Lemma 6.10 *Let $v_1, v_2, \ldots, v_n$ be a circular ordering (i.e. $v_{n+1} = v_1$) of the vertices of a digraph G. The edge connectivity, c, of G satisfies*

$$c = \min_{1 \leqslant i \leqslant n} M(v_i, v_{i+1}).$$

Proof: Let T be a minimum edge separator in G. That means that there are two vertices a and b such that T is an (a, b) edge separator. Define

$$L = \{v \mid \text{ there is a directed path from } a \text{ to } v \text{ which avoids } T\}.$$

$$R = \{v \mid \text{ there is no directed path from } a \text{ to } v \text{ which avoids } T\}.$$

Clearly, $L \cup R = V$ and $L \cap R = \emptyset$. Let $l \in L$ and $r \in R$. T is an (l, r) edge separator; for if it is not, then r belongs in L. Therefore, $M(l, r) \leqslant |T|$. Since T is a minimum edge separator, $M(l, r) = |T|$. Now neither L nor R are empty, since they contain a and b, respectively.

Consider now the circular ordering of V. There must be an i, $1 \leqslant i \leqslant n$, such that $v_i \in L$ and $v_{i+1} \in R$. Hence, the result. ■

In the case of graphs and digraphs we can test for k connectivity, easily, in time complexity $O(k \cdot |V| \cdot |E|)$. Instead of running each network flow problem to completion, we terminate it when the total flow reaches k. Each augmenting path takes $O(|E|)$ time and there are $|V|$ flow problems. As we can see, testing for k edge connectivity is much easier than for k vertex connectivity. The reason is that vertices cannot participate in the separating set which consists of edges.

We can also use this approach to determine the edge connectivity, c, in time $O(c \cdot |V| \cdot |E|)$. We run all the $|V|$ network flow problems in parallel, one augmenting path for each network in turn. When no augmenting path exists in any of the $|V|$ problems, we terminate. The cost increase is only in space, since we need to store all $|V|$ problems simultaneously. One can use binary search on c to avoid this increase in space requirements, but in this case the time complexity is $O(c \cdot |V| \cdot |E| \cdot \log c)$.

We conclude our discussion of edge connectivity with a very powerful theorem of Edmonds [9]. The proof presented here is due to Lovász [10].

Theorem 6.9 *Let* a *be a vertex of a digraph* $G(V, E)$ *and* $\min_{v \in V - \{a\}} M(a, v) = k$. *There are* k *edge-disjoint directed spanning trees of* G *rooted at* a.

Proof: The theorem trivially holds for $k = 1$. We prove the theorem by induction on k. Let us denote by $\delta_G(S)$ the number of edges in $(S; \bar{S})$ in G. If H is a subgraph of G then $G - H$ is the digraph resulting from the deletion of all the edges of H from G.

Clearly, the condition that $\min_{v \in V - \{a\}} M(v, a) \geqslant k$ is equivalent to the statement that, for every $S \subset V$, $S \neq V$ and $a \in S$, $\delta_G(S) \geqslant k$.

Let $F(V', E')$ be a subgraph of G such that

(i) F is a directed tree rooted at a (which is not necessarily spanning);
(ii) For every $S \subset V$, $S \neq V$ and $a \in S$, $\delta_{G-F}(S) \geqslant k - 1$.

If F is spanning directed tree then we get the result immediately; by the inductive hypothesis there are $k - 1$ edge-disjoint spanning trees rooted at a in $G - F$, and F is one more.

The crux of the proof is to show that if F is not spanning then an edge of the set $(V';\bar{V}')$ can be added to F, to increase its number of vertices by one and still satisfy both (i) and (ii).

Consider the following three conditions on a subset of vertices, S:

(1) $a \in S$,
(2) $S \cup V' \neq V$,
(3) $\delta_{G-F}(S) = k-1$.

Let us show that if no such S exists, then one can add any edge $e \in (V';\bar{V}')$ to F. Clearly, $F+e$ satisfies (i). Now, if (ii) does not hold, then there exists an S such that $S \neq V$, $a \in S$, and $\delta_{G-(F+e)}(S) < k-1$. It follows that $\delta_{G-F}(S)<k$. Now, by (ii), $\delta_{G-F}(S) \geqslant k-1$. Thus, $\delta_{G-F}(S) = k-1$, and S satisfies condition ((3)). Let u and v be vertices such that $u \xrightarrow{e} v$. Since $\delta_{G-(F+e)}(S) < k-1$ and $\delta_{G-F}(S) = k-1$, $v \notin S$. Also, $v \notin V'$. Thus, $S \cup V' \neq V$, satisfying condition ((2)). Therefore, S satisfies all three conditions; A contradiction.

Now, let A be a maximal [3] set of vertices that satisfies ((1)), ((2)), and ((3)). Since the edges of F all enter vertices of V',

$$\delta_{G-F}(A \cup V') = \delta_G(A \cup V') \geqslant k.$$

By condition ((3)),

$$\delta_{G-F}(A \cup V') > \delta_{G-F}(A).$$

The inequality implies that there exists an edge $x \xrightarrow{e} y$ that belongs to $(A \cup V';\overline{A \cup V'})$ and does not belong to $(A;\bar{A})$ in $G-F$. Hence, $x \in \bar{A} \cap V'$ and $y \in \bar{A} \cap \bar{V}'$. Clearly, $F+e$ satisfies (i). It remains to be shown that it satisfies (ii).

Let $S \subset V$, $S \neq V$ and $a \in S$. If $e \notin (S;\bar{S})$, then

$$\delta_{G-(F+e)}(S) = \delta_{G-F}(S) \geqslant k-1.$$

Assume $e \in (S;\bar{S})$. It is not hard to prove that for every two subsets of V, S, and A,

$$\delta_{G-F}(S \cup A) + \delta_{G-F}(S \cap A) \leqslant \delta_{G-F}(S) + \delta_{G-F}(A),$$

by considering the sets of edges connecting $S \cap A$, $S \cap \bar{A}$, $\bar{S} \cap A$, and $\bar{S} \cap \bar{A}$. Now, $\delta_{G-F}(A) = k-1$ and $\delta_{G-F}(S \cap A) \geqslant k-1$. Therefore,

$$\delta_{G-F}(S \cup A) \leqslant \delta_{G-F}(S).$$

[3] Namely, no larger set that contains A has all three properties.

Since $x \in S$ and $x \notin A$, $S \not\subseteq A$; namely, $S \cup A$ is larger than A. Also, $y \in \bar{S}$, $y \in \bar{A}$, and $y \in \bar{V}'$. Thus, $(S \cup A) \cup V' \neq V$. By the maximality of A, $\delta_{G-F}(S \cup A) \geqslant k$. This implies that $\delta_{G-F}(S) \geqslant k$; therefore $\delta_{G-(F+e)}(S) \geqslant k-1$, proving (ii). ■

The proof provides an algorithm for finding k edge-disjoint directed trees rooted at a. We look for a tree F such that $\min_{v \in V-\{a\}} M(a,v) \geqslant k-1$ in $G-F$, by adding to F one edge at a time. For each candidate edge e, we have to check whether $\min_{v \in V-\{a\}} M(a,v) \geqslant k-1$ in $G-(F+e)$. This can be done by solving $|V|-1$ network flow problems, each of complexity $O(k \cdot |E|)$. Thus, the test for each candidate edge is $O(k \cdot |V| \cdot |E|)$. No edge needs be considered more than once in the construction of F, yielding the time complexity $O(k \cdot |V| \cdot |E|^2)$. Since we repeat the construction k times, the whole algorithm is of time complexity $O(k^2 \cdot |V| \cdot |E|^2)$.

The following theorem was conjectured by Y. Shiloach and proved by Even, Garey, and Tarjan [11].

Theorem 6.10 *Let $G(V,E)$ be a digraph whose edge connectivity is at least k. For every two vertices, u and v, and every l, $0 \leqslant l \leqslant k$, there are l directed paths from u to v and $k-l$ directed paths from v to u, which are all edge disjoint.*

Proof: Construct an auxiliary graph $\bar{G}$ by adding to G a new vertex a, l parallel edges from a to u, and $k-l$ parallel edges from a to v. Let us first show that in $\bar{G}$ $\min_{w \in V} M(a,w) = k$.

If $\min_{w \in V} M(a,w) < k$, then there exists a set S such that $S \subset V \cup \{a\}$, $a \in S$, and $|(S;\bar{S})| < k$ in $\bar{G}$. Clearly, $S \neq \{a\}$ for $|(\{a\};V)| = k$. Let $x \in S - \{a\}$. Thus, $M(x,y) < k$ for every $y \in \bar{S}$, and the same also holds in G, since G is a subgraph of $\bar{G}$. This contradicts the assumption that in G, the edge connectivity is at least k.

Now, by Theorem 6.9, there are k edge-disjoint directed spanning trees, in $\bar{G}$, rooted at a. Exactly one edge out of a appears in each tree. Thus, each of the trees that uses an edge $a \longrightarrow u$ contains a directed path from u to v, and each of the trees that uses an edge $a \longrightarrow v$ contains a directed path from v to u. All these paths are, clearly, edge disjoint. ■

Corollary 6.1 *If the edge connectivity of a digraph is at least 2, then for every two vertices u and v there exists a directed circuit that goes through u and v in which no edge appears more than once.*

Proof: Use $k = 2$, $l = 1$ in Theorem 6.10. ■

It is interesting to note that no such easy result exists in the case of vertex connectivity and a simple directed circuit through given two vertices. In [11],

a digraph with vertex connectivity 5 is shown such that for every two of its vertices there is no simple directed circuit that passes through both. The author does not know whether any vertex connectivity will guarantee the existence of a simple directed circuit through any two vertices.

6.4 Maximum Matching in Bipartite Graphs

A set of edges, M, of a graph $G(V, E)$ with no self-loops, is called a *matching* if every vertex is incident to at most one edge of M. The problem of finding a maximum matching was first solved in polynomial time by Edmonds [12]. The best known result of Even and Kariv [13] is $O(|V|^{2.5})$. These algorithms are too complicated to be included here, and they do not use network flow techniques.

An easier problem is to find a maximum matching in a *bipartite* graph, that is, a graph in which $V = X \cup Y$, $X \cap Y = \emptyset$, and each edge has one end-vertex in X and one in Y. This problem is also known as the *marriage problem*. We present here its solution via network flow and show that its complexity is $O(|V|^{1/2} \cdot |E|)$. This result was first achieved by Hopcroft and Karp [14].

Let us construct a network $N(G)$. Its digraph $\bar{G}(\bar{V}, \bar{E})$ is defined as follows:

$$\bar{V} = \{s, t\} \cup V,$$

$$\bar{E} = \{s \longrightarrow x | x \in X\} \cup \{y \longrightarrow t | y \in Y\} \cup \{x \longrightarrow y | x \text{ —— } y \text{ in } G\}.$$

Let $c(s \longrightarrow x) = c(y \longrightarrow t) = 1$ for every $x \in X$ and $y \in Y$. For every edge $x \xrightarrow{e} y$ let $c(e) = \infty$. (This infinite capacity is defined in order to simplify our proof of Theorem 6.12. Actually, since there is only one edge entering x, with unit capacity, the flow in $x \longrightarrow y$ is bounded by 1.) The source is s and the sink is t. For example, consider the bipartite graph G shown in Figure 6.2(a). Its corresponding network $N(G)$ is shown in Figure 6.2(b).

Theorem 6.11 *The number of edges in a maximum matching of a bipartite graph* G *is equal to the maximum flow,* F*, in its corresponding network,* $N(G)$.

Proof: Let M be a maximum matching. For each edge $x \longrightarrow y$ of M, use the directed path $s \longrightarrow x \longrightarrow y \longrightarrow t$ to flow one unit from s to t. Clearly, all these paths are vertex-disjoint. Thus, $F \geqslant |M|$.

Let f be a flow function on $N(G)$, which is integral. (There is no loss of generality here. As we saw, in Chapter 5, every network with integral capacities has a maximum integral flow.) All the directed paths connecting s and t are of the form $s \longrightarrow x \longrightarrow y \longrightarrow t$. If such a path is used to flow (one unit) from s

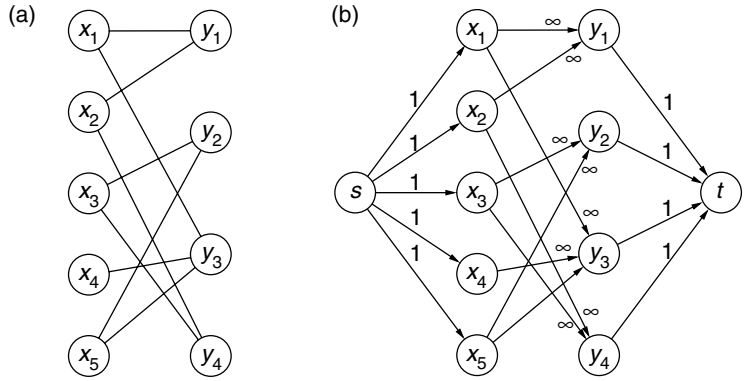


Figure 6.2: Construction of $N(G)$.

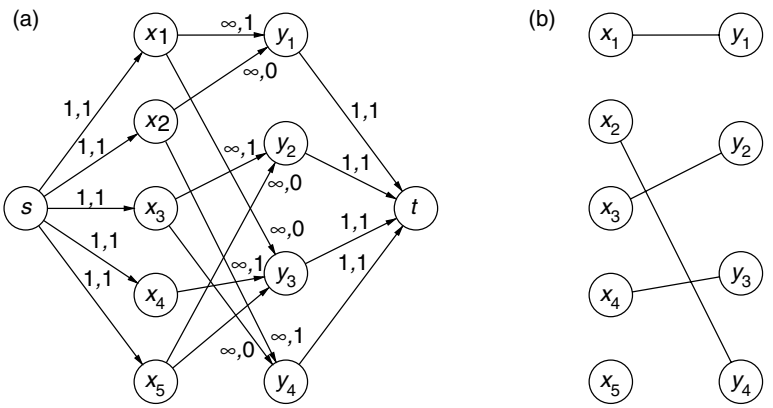


Figure 6.3: Maximum flow found by Dinitz's algorithm and its corresponding matching.

to t, then no other edge $x \longrightarrow y'$ or $x' \longrightarrow y$ can carry flow, since there is only one edge $s \longrightarrow x$ and its capacity is one, and the same is true for $y \longrightarrow t$. Thus, the set of edges $x \longrightarrow y$, for which $f(x \longrightarrow y) = 1$, constitutes a matching in G. Thus, $|M| \geqslant F$. ■

The proof indicates how the network flow solution can yield a maximum matching. for our example, a maximum flow, found by Dinitz's algorithm is shown in Figure 3(a) and its corresponding matching is shown in Figure 3(b). The algorithm described in the proof constructs a type 2 network; therefore, by Theorem 6.3, its running time is $O(|V|^{1/2} \cdot |E|)$.

Next, let us show that one can also use the max-flow min-cut theorem to prove a theorem of Hall's [15]. For every $A \subseteq X$, let $\Gamma(A)$ denote the set of vertices (all in Y) that are connected by an edge to a vertex of A. A matching M, is called *complete* if $|M| = |X|$.

Theorem 6.12 *A bipartite graph* G *has a complete matching* M *if and only if for every* $A \subset X$, $|\Gamma(A)| \geqslant |A|$.

Proof: Clearly, if G has a complete matching M, then each x has a unique "mate" in Y. Thus, for every $A \subset X$, $|\Gamma(A)| \geqslant |A|$.

Assume now that G does not have a complete matching. Let S be the set of labeled vertices (in the Ford and Fulkerson algorithm or the Dinitz algorithm) upon termination. Clearly, the maximum total flow is equal to $|M|$, but $|M| < |X|$. Let $A = X \cap S$. Since all the edges of the type $x \longrightarrow y$ are of infinite capacity, $\Gamma(A) \subset S$. Also, no vertex of $Y \setminus \Gamma(A)$ is labeled, since there is no edge from a labeled vertex to it. We have

$$(S;\bar{S}) = (\{s\};X - A) \cup (\Gamma(A);\{t\}).$$

Since $(S;\bar{S}) = |M| < |X|$, we get

$$|X - A| + |\Gamma(A)| < |X|,$$

which implies $|\Gamma(A)| < |A|$. ■

6.5 Two Problems on PERT Digraphs

The *Program Evaluation and Review Technique*, commonly abbreviated PERT, is a model for project management. A *PERT digraph* is a finite digraph $G(V, E)$ with the following properties:

(i) There is a vertex s, called the *start-vertex*, and a vertex $t(\neq s)$, called the *termination vertex*.
(ii) G has no directed circuits.
(iii) Every vertex $v \in V \setminus \{s, t\}$ is on some directed path from s to t.

A PERT digraph has the following interpretation: Every edge represents a process. Recall that $\alpha(v)$ denotes the set of edges that enter v; $\beta(v)$ denotes the set of edges that emanate from v. All the processes which are represented by edges of $\beta(s)$ can be started right away. For every vertex v, the process represented by edges of $\beta(v)$ can be started when all the processes represented by edges of $\alpha(v)$ are completed.

Our first problem deals with the question of how soon can the whole project can be completed; that is, what is the shortest time, from the moment the processes represented by $\beta(s)$ are started, until all the processes represented by $\alpha(t)$ are completed. We assume that the resources for running the processes are unlimited. For this problem to be well defined, let us assume that each $e \in E$ has an assigned *length* $l(e)$, which specifies the time it takes to execute the process represented by e. The minimum completion time can be found by the following algorithm:

1. Assign s the label 0 ($\lambda(s) \leftarrow 0$). All other vertices are "unlabeled."
2. Find a vertex v such that v is unlabeled and all edges of $\alpha(v)$ emanate from labeled vertices. Assign

$$\lambda(v) \leftarrow \max_{e \in \alpha(v)} \{\lambda(u) + l(e)\}.$$

3. If $v = t$, halt; $\lambda(t)$ is the minimum completion time. Otherwise, go to Step (2).

In Step (2), the existence of a vertex v such that all the edges of $\alpha(v)$ emanate from labeled vertices is guaranteed by conditions (i) and (iii): If no unlabeled vertex satisfies the condition, then for every unlabeled vertex v, there is an incoming edge which emanates from another unlabeled vertex. By repeatedly tracing back these edges, one finds a directed circuit. Thus, if no such vertex is found, then we conclude that either (i) or (ii) does not hold.

It is easy to prove by induction on the order of labeling, that $\lambda(v)$ is the minimum time in which all processes represented by the edges of $\alpha(v)$ can be completed.

The time complexity of the algorithm can be kept down to $O(|E|)$ as follows: For each vertex v we keep count of its incoming edges from unlabeled vertices; this count is initially set to $d_{in}(v)$; each time a vertex u gets labeled, we use the list $\beta(u)$ to decrease the count for all v such that $u \longrightarrow v$, accordingly; once the count of a vertex v reaches 0, it enters a queue of vertices to be labeled.

Once the algorithm terminates, by going back from t to s, via the edge that determined the label of the vertex, we can trace a longest path from s to t. Such a path is called *critical*.[4] Clearly, there may be more than one critical path. If one wants to shorten the completion time, $\lambda(t)$, then on each critical path at least one edge length must be shortened.

[4] The whole process is sometimes called the critical path method (CPM).

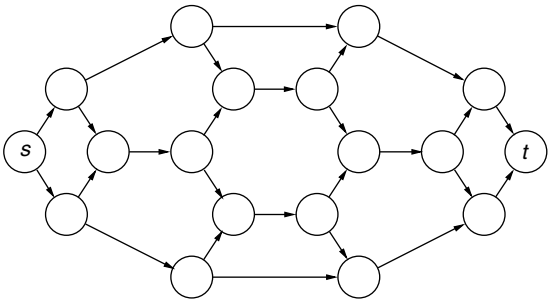


Figure 6.4: Example of a PERT digraph.

Next, we shall consider another problem concerning PERT digraphs, where there is no reference to edge lengths. Assume that each of the processes, represented by the edges, uses one processor for its execution. The question is: How many processors do we need to be sure that no execution will ever be delayed because of a shortage of processors? We want to avoid such a delay without relying on the values of $l(e)$'s either because they are unknown or because they vary from time to time.

Let us solve a minimum flow problem in the network whose digraph is G, source s, sink t, lower bound $b(e) = 1$ for all $e \in E$ and no upper bound (i.e., $c(e) = \infty$ for all $e \in E$). Condition (6.5) assures the existence of a legal flow (see Problem 6.5).

For example, consider the PERT digraph of Figure 6.4. The minimum flow (which in this case is unique) is shown in Figure 6.5(a), where a maximum cut is shown too.

A set of edges is called *concurrent* if for no two edges in the set there is a directed path that passes through both. Now, let T be the set of vertices that are labeled in the last attempt to find an augmenting path from t to s. Clearly, $t \in T$ and $s \notin T$. The set of edges $(\bar{T};T)$ is a maximum cut; there are no edges in $(T;\bar{T})$, for there is no upper bound on the flow in the edges, and any such edge would enable to continue the labeling of vertices. Thus, the set $(\bar{T};T)$ is concurrent.

If S is a set of concurrent edges, then the number of processors required is, at least $|S|$. This can be seen by assigning the edges of S a very large length; and all the others, a short length. Since no directed path leads from one edge of S to another, they all will be operative simultaneously. This implies that the number of processors required is at least $|(\bar{T};T)|$.

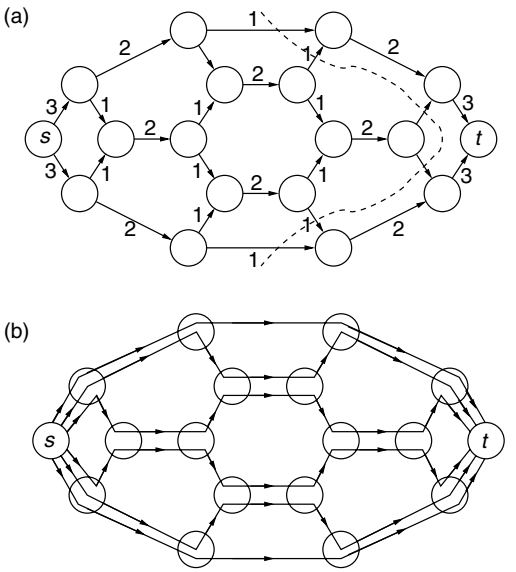


Figure 6.5: (a) The minimum flow in the PERT; (b) decompositions of the flow into the PERT into F directed paths.

However, the flow can be decomposed into F directed paths from s to t, where F is the minimum total flow, such that every edge is on at least one such (since $f(e) \geqslant 1$ for every $e \in E$). This is demonstrated for our example in Figure 6.5(b). We can now assign to each processor all the edges of one such path. Each such processor executes the processes represented by the edges of the path in the order in which they appear on the path. If one process is assigned to more than one processor, then one of them executes while the others are idle. It follows that whenever a process that corresponds to $u \longrightarrow v$ is executable (because all the processes which correspond to $\alpha(u)$ have been executed), the processor to which this process is assigned is available for its execution. Thus, F processors are sufficient for our purpose.

Since $F = |(\bar{T};T)|$, by the min-flow max-cut theorem, the number of processors thus assigned is minimum.

The complexity of this procedure is as follows: We can find a legal initial flow in time $O(|V| \cdot |E|)$, by tracing for each edge a directed path from s to t via this edge, and flow through it one unit. This path is found by starting from the edge, and going forward and backward from it until s and t are reached. Next, we solve

a maximum flow problem, from t to s. Thus, the complexity of the whole procedure is dominated by the complexity of solving one maximum flow problem.

6.6 Problems

Problem 6.1 Let $G(V, E)$ be an acyclic finite digraph.[5] We wish to find a minimum number of directed vertex-disjoint paths that cover all the vertices; that is, every vertex is on exactly one path. The paths may start anywhere and end anywhere, and their lengths are not restricted in any way. A path may be of zero length; that is, it may consist of one vertex.

1. Describe an algorithm for achieving this goal, and make it as efficient as possible. (Hint. Form a network as follows:

$$V' = \{s, t\} \cup \{x_1, x_2, \ldots, x_{|V|}\} \cup \{y_1, y_2, \ldots, y_{|V|}\}.$$

$$E' = \{s \longrightarrow x_i \mid 1 \leqslant i \leqslant |V|\} \cup \{y_i \longrightarrow t \mid 1 \leqslant i \leqslant |V|\}$$

$$\cup \{x_i \longrightarrow y_j \mid v_i \longrightarrow v_j \text{ in } G\}.$$

   The capacity of all edges is 1.
   Show that the minimum number of paths that cover V in G is equal to $|V| - F$, where F is the maximum total flow of the network.)
2. Is the condition that G is acyclic essential for the validity of your algorithm? Explain.
3. Give the best upper bound you can on the time complexity of your algorithm.

Problem 6.2 This problem is similar to Problem 6.1, except that the paths are not required to be vertex- (or edge-) disjoint.

1. Describe an algorithm for finding a minimum number of covering paths. (Hint. Form a network as follows:

$$V' = \{s, t\} \cup \{x_1, x_2, \ldots, x_{|V|}\} \cup \{y_1, y_2, \ldots, y_{|V|}\}.$$

$$E' = \{s \longrightarrow x_i \mid 1 \leqslant i \leqslant |V|\} \cup \{y_i \longrightarrow t \mid 1 \leqslant i \leqslant |V|\}$$

$$\cup \{x_i \longrightarrow y_i \mid 1 \leqslant i \leqslant |V|\} \cup \{y_i \longrightarrow x_j \mid v_i \longrightarrow v_j \text{ in } G\}.$$

   The lower bound of each $x_i \longrightarrow y_i$ edge is 1.
   The lower bound of all other edges is 0.
   The upper bound of all the edges is ∞. Find a minimum flow from s to t.)

[5] A digraph is called "acyclic" if it has no directed circuits.

2. Is the condition that G is acyclic essential for the validity of your algorithm? Explain.
3. Give the best upper bound you can on the time complexity of your algorithm. (Hint. $O(|V| \cdot |E|)$ is achievable.)
4. Two vertices u and v are called *concurrent* if no directed path exists from u to v or from v to u. A set of concurrent vertices is such that every two in the set are concurrent. Prove that the minimum number of paths that cover the vertices of G is equal to the maximum number of concurrent vertices. (This is Dilworth's Theorem [16].)

Problem 6.3 1. Let $G(X, Y, E)$ be a finite bipartite graph. Describe an efficient algorithm for finding a minimum set of edges such that each vertex is an end-vertex of at least one of the edges in the set.
2. Discuss the time complexity of your algorithm.
3. Prove that the size of a minimum set of edges which cover the vertices (as in (1)) is equal to the maximum size of an independent set of vertices of G.[6]

Problem 6.4 1. Prove that if $G(X, Y, E)$ is a complete bipartite graph (i.e., for every two vertices $x \in X$ and $y \in Y$, there is an edge x —— y) then the vertex connectivity of G is

$$c(G) = \min\{|X|, |Y|\}.$$

2. Prove that for every k, there exists a graph G such that $c(G) \geqslant k$ and G has no Hamilton path. (See Problem 1.4.)

Problem 6.5 Let M be a matching of a bipartite graph. Prove that there exists a maximum matching M' such that every vertex matched in M is matched also in M'.

Problem 6.6 Let $G(V, E)$ be a finite acyclic digraph with exactly one vertex s for which $d_{in}(s) = 0$ and exactly one vertex t for which $d_{out} = 0$. We say that the edge $a \longrightarrow b$ is greater than the edge $c \longrightarrow d$ if and only if there is a directed path in G from b to c. A set of edges is called a *slice* if no edge in it is greater than another and it is maximal; no other set of edges with this property contains it. Prove that the following three conditions on a set of edges, P, are equivalent:

[6] A set of vertices of a graph is called *independent* if there is no edge between two vertices of the set.

1. P is a slice.
2. P is an (s,t) edge separator in which no edge is greater than any other.
3. $P = (S;\bar{S})$ for some $\{s\} \subset S \subset V - \{t\}$ such that $(\bar{S};S) = \emptyset$.

Problem 6.7 (The problem of a system of distinct representatives [SDR]). Let $S_1, S_2, \ldots, S_m$ be finite sets. A set $\{e_1, e_2, \ldots, e_m\}$ is called an SDR if for every $1 \leqslant i \leqslant m$, $e_i \in S_i$.

1. Describe an efficient algorithm for finding an SDR, if one exists. (Hint. Define a bipartite graph and solve a matching problem.)
2. Prove that an SDR exists if and only if the union of any $1 \leqslant k \leqslant m$ of the sets contains at least k elements.

Problem 6.8 Let π_1 and π_2 be two partitions of a set of m elements, each containing exactly r disjoint subsets. We want to find a set of r elements such that each of the subsets of π_1 and π_2 is represented.

1. Describe an efficient algorithm to determine whether there is such a set of r representatives.
2. State a necessary and sufficient condition for the existence of such a set, similar to Theorem 6.12.

Problem 6.9 Let $G(V,E)$ be a completely connected digraph (see Problem 1.5); it is called *classifiable* if V can be partitioned into two nonempty classes, A and B, such that all the edges connecting between them are directed from A to B. Let $V = \{v_1, v_2, \ldots, v_n\}$ where the vertices satisfy

$$d_{out}(v_1) \leqslant d_{out}(v_2) \leqslant \ldots \leqslant d_{out}(v_n).$$

Prove that G is classifiable if and only if there exists a $k < n$ such that

$$\sum_{i=1}^{k} d_{out}(v_i) = \binom{k}{2}.$$

Problem 6.10 Let S be a set of people such that $|S| \geqslant 4$. We assume that acquaintance is a mutual relationship. Prove that if, in every subset of four people, there is one who knows all the others, then there is someone in S who knows everybody.

Problem 6.11 In the acyclic digraph shown in Figure 6.6, there are both AND vertices (designated by $\bigwedge$) and OR vertices (designated by $\bigvee$). As in a PERT network, the edges represent processes, and the edge length is the time the

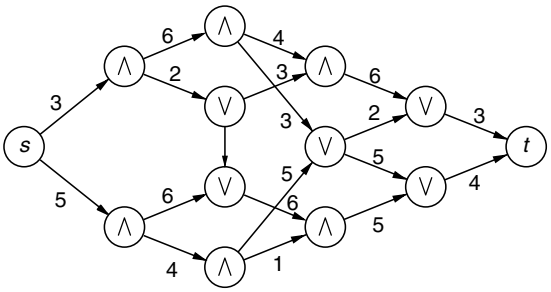


Figure 6.6: Graph for Problem 6.11.

process requires. The processes represented by the edges that emanate from an AND (OR) vertex can be started when all (at least one of) the incoming processes are (is) completed. Describe an algorithm for finding the minimum time from start (s) to reach termination (which depends on the type of t). Apply your algorithm on the given network. What is the complexity of your algorithm?

Problem 6.12 Consider Problem 6.11 on digraphs that are not necessarily acyclic. Show how to modify the algorithm to solve the problem or conclude that there is no solution.

Problem 6.13 In a school are n boys and n girls. Each boy knows exactly k girls $(1 \leqslant k \leqslant n)$, and each girl knows exactly k boys. Prove that if "knowing" is mutual, then all the boys and girls can participate in one dance, where every pair of dancers (a boy and a girl) know each other. Also show that it is always true that k consecutive dances can be organized so that everyone will dance once with everyone he or she knows.

Problem 6.14 Prove or disprove the following claim: If the vertex connectivity of a digraph is at least 2, then for every three vertices x, y, and z, there exists a simple directed path from x to z via y.

Problem 6.15 Let G be a finite undirected graph whose edge connectivity is at least 2. For each one of the following claims, determine whether it is true or false. Justify your answer.

1. For every three vertices x, y, and z there exists a path in which no edge appears more than once, from x to z via y.
2. For every three vertices x, y, and z there exists a path in which no edge appears more than once, from x to z, which avoids y.

Bibliography

[1] Even, S., Tarjan, R. E., "Network Flow and Testing Connectivity," *SIAM J. on comput*., Vol. 4, 1975, pp. 507–518.

[2] Menger, K., "Zur Aligemeinen Kurventheorie," *Fund. Math.,* Vol. 10, 1927, pp. 96–115.

[3] Dantzig, G. B., Fulkerson, D. R., "On the Max-Flow Min-Cut Theorem of Networks," Linear Inequalities and Related Systems, Annals of Math. Study 38, Princeton University Press, 1956, pp. 215–221.

[4] Hopcroft, J., and Tarjan, R. E., "Dividing a Graph into Triconnected Components", *SIAM J. on Comput.*, Vol. 2, 1973, pp. 135–158.

[5] Kleitman, D. J., "Methods for Investigating Connectivity of Large Graphs," *IEEE Trans. on Circuit Theory*, CT-16, 1969, pp. 232–233.

[6] Even, S., "Algorithm for Determining whether the Connectivity of a Graph is at Least k," *Siam J. on Comput.*, Vol. 4, 1977, pp. 393–396.

[7] Gomory, R. E., and Hu, T. C., "Multi-Terminal Network Flows," *J. of SIAM*, Vol. 9, 1961, pp. 551–570.

[8] Schnorr, C. P., "Multiterminal Network Flow and Connectivity in Unsymmetrical Networks," Department of Applied Math, University of Frankfurt, October 1977.

[9] Edmonds, J., "Edge-Disjoint Branchings," in *Combinatorial Algorithms, Courant Inst. Sci. Symp. 9*, R. Rustin, Ed., Algorithmics Press Inc., 1973, pp. 91–96.

[10] Lovász, L., "On Two Minimax Theorems in Graph Theory," *Journal of Combinatorial Theory, Series B*, Vol. 21:2, 1976, pp. 96–103.

[11] Even, S., Garey, M. R., and Tarjan, R. E., "A Note on Connectivity and Circuits in Directed Graphs," unpublished manuscript (1977).

[12] Edmonds, J., "Paths, Trees, and Flowers," *Canadian J. of Math.*, Vol. 17, 1965, pp. 449–467.

[13] Even, S., and Kariv, O., "An $O(n^{2.5})$ Algorithm for Maximum Matching in General Graphs," *16th Annual Symposium on Foundations of Computer Science*, IEEE, 1975, pp. 100–112.

[14] Hopcroft, J., and Karp, R. M., "An $O(n^{5/2})$ Algorithm for Maximum Matching in Bipartite Graphs," *SIAM J. on Comput.*, 1975, pp. 225–231.

[15] Hall, P., "On Representation of Subsets," *J. London Math. Soc.* Vol. 10, 1935, pp. 26–30.

[16] Dilworth, R. P., "A Decomposition Theorem for Partially Ordered Sets," *Ann. Math.*, Vol. 51, 1950, pp. 161–166.

7
Planar Graphs

7.1 Bridges and Kuratowski's Theorem

Consider a graph drawn in the plane in such a way that each vertex is represented by a point; each edge is represented by a continuous line connecting the two points that represent its end vertices, and no two lines, which represent edges, share any points, except in their ends. Such a drawing is called a *plane graph*. If a graph G has a representation in the plane that is a plane graph then it is said to be *planar*.

In this chapter, we shall discuss some of the classical work concerning planar graphs. The question of efficiently testing whether a given finite graph is planar is discussed in the next chapter.

Let S be a set of vertices of a nonseparable graph $G(V,E)$. Consider the partition of the set $V-S$ into classes, such that two vertices are in the same class if and only if there is a path connecting them that does not use any vertex of S. Each such class K defines a *component* as follows: The component is a subgraph $H(V',E')$, where $V' \supset K$. In addition, V' includes all the vertices of S that are connected by an edge to a vertex of K, in G. Also, E' contains all edges of G that have at least one end-vertex in K. An edge $u \overset{e}{\text{---}} v$, where both u and v are in S, defines a *singular component* $(\{u,v\},\{e\})$. Clearly, two components share no edges, and the only vertices they can share are elements of S. The vertices of a component that are elements of S are called its *attachments*.

In our study we usually use a set S, which is the set of vertices of a simple circuit C. In this case, we call the components *bridges*; the edges of C are not considered bridges.

For example, consider the plane graph over the vertices $a,\ldots,k$ shown in Figure 7.1. The edges of the plane graph are $1,\ldots,18$. Let C be the outside boundary:

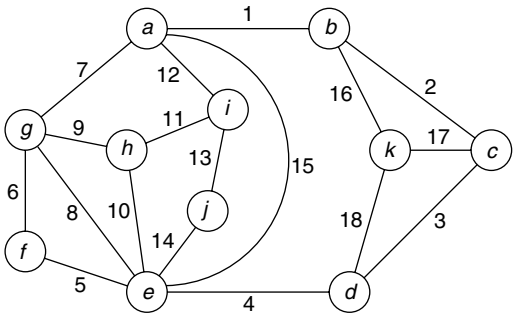


Figure 7.1: Example of a plane graph.

$$C = a \overset{1}{—} b \overset{2}{—} c \overset{3}{—} \ldots \overset{6}{—} g \overset{7}{—} a.$$

The circuit C partitions $V \setminus C$ into two classes: $\{i, j, h\}$ and $\{k\}$. The set of bridges is

$$(\{e, g\}, \{8\}),$$
$$(\{i, j, h, a, e, g\}, \{9, 10, 11, 12, 13, 14\}),$$
$$(\{a, e\}, \{15\}), \text{ and}$$
$$(\{k, b, c, d\}, \{16, 17, 18\}).$$

The first and third bridges are singular.

Lemma 7.1 *Let B be a bridge, and a_1, a_2, a_3, three of its attachments. There exists a vertex v, not an attachment, for which there are three vertex-disjoint paths in B: $P_1(v, a_1), P_2(v, a_2)$ and $P_3(v, a_3)$. ($P(a, b)$ denotes a path connecting a to b).*

Proof: Let $a_1 \overset{e_1}{—} v_1, a_2 \overset{e_2}{—} v_2$, and $a_3 \overset{e_3}{—} v_3$ be edges of B. If any of the v_i's $(i = 1, 2, 3)$ is an attachment, then the corresponding edge is a singular component and is not part of B. Thus, $v_i \in K$, where K is the class that defines B. Hence, there is a simple path $P'(v_1, v_2)$ that uses vertices of K only; if $v_1 = v_2$, this path is empty. Also, there is a simple path $P''(v_3, v_1)$ that uses vertices of K only. Let v be the first vertex of $P''(v_3, v_1)$ that is also on P'. Now, let $P_1(v, a_1)$ be the part of P' that leads from v to v_1, concatenated with $v_1 — a_1$; let $P_2(v, a_2)$ be the part of P' that leads from v to v_2, concatenated with $v_2 — a_2$; let $P_3(v, a_3)$ be the part of P'' that leads from v to v_3, concatenated with $v_3 — a_3$. It is easy to see that these paths are disjoint. ■

Let C be a simple circuit of a nonseparable graph G, and $B_1, B_2, \ldots, B_k$ be the bridges with respect to C. We say that B_i and B_j *interlace* if at least one of the following conditions holds:

1. There are two attachments of B_i, a and b, and two attachments of B_j, c and d, such that all four are distinct and appear on C in the order a, c, b, d.
2. There are three attachments common to B_i and B_j.

For each bridge B_i, consider the subgraph $C + B_i$. If any of these graphs is not planar, then clearly G is not planar. Now, assume all these subgraphs are planar. In every plane realization of G, C outlines a contour that divides the plane into two disjoint parts: its inside and outside. Each bridge must lie entirely in one of these parts. Clearly, if two bridges interlace they cannot be on the same side of C. Thus, in every plane realization of G, the set of bridges is partitioned into two sets: those drawn inside C and those drawn outside it. No two bridges in the same set interlace.

Lemma 7.2 *If* $B_1, B_2, \ldots, B_m$ *is set of bridges of a nonseparable graph* G *with respect to a simple circuit* C *and the following two conditions are satisfied:*

1. for every $1 \leqslant i \leqslant n$, $C + B_i$ *is planar, and*
2. no two bridges interlace,

then $C + B_1 + B_2 + \ldots + B_m$ *has a plane realization in which all these bridges are inside* C.

Proof: We shall only outline the proof. As we go clockwise around C there must be a bridge B_i such that we encounter all its attachments in some order: $a_1, a_2, \ldots, a_t$, and no attachment of any other bridge appears between a_1 and a_t on C. Such a bridge can be found by starting from any attachment a of B_1 and going around, say, clockwise. If before encountering all the attachments of B_1 we encounter an attachment of another bridge B_i, then all B_i's attachments are between consecutive attachments of B_1 that may also belong to B_i. We repeat the same process on B_i, and so on. Since the number of bridges is finite, and those discarded will not "interfere" with the new ones, the process will yield the desired bridge.

This observation allows a proof by induction. First B_i is drawn, and since no other bridge uses any of the vertices of C between a_1 and a_t, we can take C' to be the circuit that describes the part of C from a_t to a_1, clockwise, and the boundary of B_i from a_1 to a_t to form a simple circuit C', whose inside is so far empty. The remaining bridges are also bridges of C' and, clearly, satisfy (7.2) and (7.2) with respect to C'. ■

Corollary 7.1 *Let* G *be a nonseparable graph; and* C*, a simple circuit in* G*. In this case* G *is planar if and only if the bridges* $B_1, B_2, \ldots, B_k$ *of* G*, with respect to* C*, satisfy the following conditions:*

1. *For every* $1 \leqslant i \leqslant k$, $C + B_i$ *is planar.*
2. *The set of bridges can be partitioned into two subsets, such that no two bridges in the same subset interlace.*

Let us introduce the coloring of graphs. Here, we consider only 2-coloring of vertices. A graph $G(V, E)$ is said to be 2-*colorable* if V can be partitioned into V_1 and V_2 in such a way that there is no edge in G with two end vertices in $V_1(V_2)$. (Obviously, a 2-colorable graph is bipartite, and vice versa. It is customary to use the term "bipartite" if the partition is given, and "2-colorable" if one exists.) The following theorem is due to König [1].

Theorem 7.1 *A graph* G *is* 2*-colorable if and only if it has no odd length circuits.*

Proof: It is easy to see that if a graph has an odd length circuit, then it is not 2-colorable. In order to prove the converse, we may assume that G is connected, for if each component is 2-colorable, then the whole graph is 2-colorable.

Let v be any vertex. Let us perform BFS (see Section 1.5.1) starting from v. There cannot be an edge u —— w in G if u and w belong to the same layer, that is, are the same distance from v. For if such an edge exists then we can display an odd length circuit as follows: Let $P_1(v, u)$ be a shortest path from v to u. $P_2(v, w)$ is defined similarly and is of the same length. Let x be the last vertex that is common to P_1 and P_2. The part of P_1 from x to u, and the part of P_2 from x to w are of equal length, and together with u —— w, they form an odd length simple circuit.

Now, we can color all the vertices of even distance from v with one color and all the vertices of odd distance from v with a second color. ■

We can use the concept of 2-colorability to decide whether the bridges $B_1, B_2, \ldots, B_k$, with respect to a simple circuit C, can be partitioned into two pairwise noninterlacing subsets, as follows: Construct a graph G' whose vertices are the bridges $B_1, B_2, \ldots, B_k$, and two vertices are connected by an edge if and only if the corresponding bridges in G interlace. Now test whether the graph G' is 2-colorable, by giving one vertex color 1 and using some search technique, such as DFS or BFS to color alternate vertices with different colors

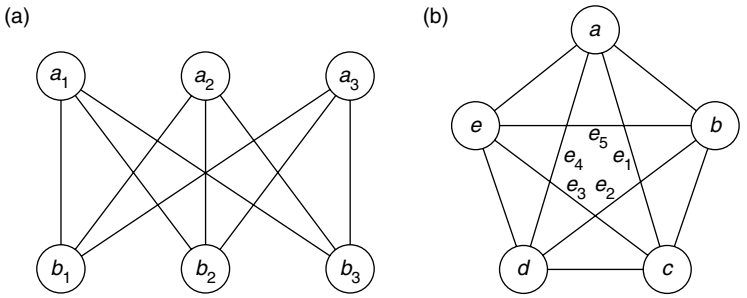


Figure 7.2: The graphs of Kuratowski: (a) $K_{3,3}$ and (b) K_5.

out of $\{1,2\}$. If no contradiction arises, a coloring is obtained and thus, a partition. If a contradiction occurs, there is no 2-coloring, and therefore no partition of the bridges; in this case, the graph is nonplanar, by Corollary 7.1.

Let us now introduce the graphs of Kuratowski [2]: $K_{3,3}$ and K_5. They are shown in Figure 7.2(a) and (b), respectively.

K_5 is completely connected graph of five vertices, or a clique of five vertices. $K_{3,3}$ is a completely connected bipartite graph with three vertices on each side.

Lemma 7.3 *Neither* $K_{3,3}$ *nor* K_5 *is planar.*

Proof: First, let us consider K_5, and its circuit C: a — b — c — d — e — a. Clearly, there are five bridges, all singular, corresponding to the edges e_1, e_2, e_3, e_4, and e_5. Let us construct G' as in the preceding discussion. It is shown in Figure 7.3. For example, the bridge e_1 interlaces with e_5 and e_2, and so on. Since G' contains an odd circuit, by Theorem 7.1, it is not 2-colorable, and the set of bridges of K_5 with respect to C is not partitionable. Thus, by Corollary 7.1, K_5 is not planar.

In the case of $K_{3,3}$, take C: a_1 — b_1 — a_2 — b_2 — a_3 — b_3 — a_1. The bridges a_1 — b_2, a_2 — b_3 and a_3 — b_1 form a triangle in the corresponding G'. ■

Before we take on Kuratowski's theorem, we need a few more definitions and a lemma.

Let $G(V,E)$ be a finite nonseparable plane graph with $|V| > 2$. A *face* of G is a maximal part of the plane such that, for every two points x and y in it, there is a continuous line from x and y that does not share any point with the realization of G. The contour of each face is a simple circuit of G; if any of these circuits is not simple, then G is separable. Each of these circuits is called a *window*.

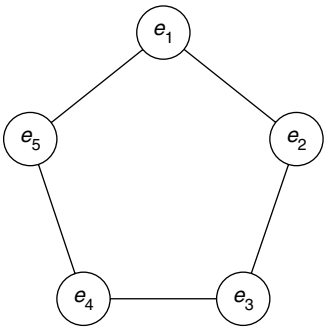


Figure 7.3: G' for the proof of Lemma 7.3.

One of the faces is of infinite area. It is called the *external* face, and its window is the external window. It is not hard to show that for every window W, there exists another plane realization, G', of the same graph, which has the same set of windows, but in G', the window W is external. First, draw the graph on a sphere, maintaining the windows; this can be achieved by projecting each point of the plane vertically up to the surface on a sphere whose center is in the plane and its intersecting circle with the plane encircles G. Next, place the sphere on a plane that is tangent to it, in such a way that a point in the face whose contour is W is the "north pole," that is, furthest from the plane. Project each point P of the sphere (other than the "north pole") to the plane by a straight line that starts from the "north pole" and goes through P. The graph is now drawn in the plane, and W is the external window.

Lemma 7.4 *Let* $G(V,E)$ *be a* 2*-vertex connected graph with a separating pair* a,b. *Let* $H_1,H_2,\ldots,H_m$ *be the components with respect to* $\{a,b\}$. G *is planar if and only if for every* $1 \leqslant i \leqslant m$, $H_i + (a \overset{e}{\text{---}} b)$ *is planar.*

By $H_i + (a \overset{e}{\text{---}} b)$ *we mean the subgraph obtained by adding the edge* $a \overset{e}{\text{---}} b$ *to* H_i.[1]

Proof: In each H_i, there is a path from a to b, or G is not 2-connected. Also $m > 1$. Thus, for each H_i we can find a path P from a to b in one of the other components. If G is planar, so is $H_i + P$, and therefore $H_i + (a \overset{e}{\text{---}} b)$ is planar. Now assume that each $H_i + (a \overset{e}{\text{---}} b)$ is planar. For each of these realization we can assume that e is on the external window. Thus, a planar realization of G exists, as demonstrated in Figure 7.4 in the case of $m = 3$. ■

[1] Since G is 2-vertex connected, it follows that $a,b \in H_i$. By definition, the component H_i does not contain e, even if $e \in E$. Thus, we add e to H_i regardless of whether $e \in E$.

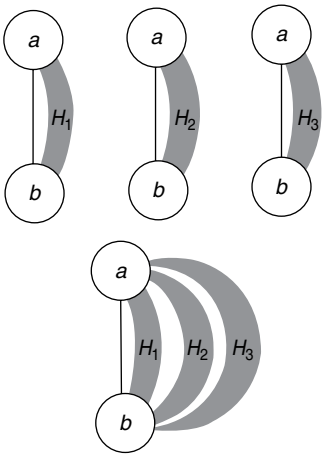


Figure 7.4: A demonstration of a planar realization of G, with $m = 3$, for the proof of Lemma 7.4.

Two graphs are said to be *homeomorphic* if both can be obtained from the same graph by subdividing edges, that is, an edge is replaced by a simple path whose intermediate vertices are all new.[2] Clearly, if two graphs are homeomorphic, then either both are planar or both are not. We are now ready to state Kuratowski's Theorem [2].

Theorem 7.2 *A graph* G *is non-planar if and only if there is a subgraph of* G *which is homeomorphic to either* $K_{3,3}$ *or* K_5.

Proof: If G has a subgraph H that is homeomorphic to either $K_{3,3}$ or K_5, then by Lemma 7.3, H is nonplanar, and therefore G is nonplanar. The converse is much harder to prove. We prove it by contradiction.

Let G be a graph that is nonplanar and which does not contain a subgraph that is homeomorphic to one of Kuratowski's graphs; in addition, let us assume that among such graphs, G has the minimum number of edges.

First, let us show that the vertex connectivity of G is at least 3. Clearly, G is connected and nonseparable, or the number of its edges is not minimum. Assume that it has a separating pair $\{a, b\}$, and let $H_1, H_2, \ldots, H_m$ be the components

[2] Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are said to be **isomorphic** if there are bijections $f : V_1 \to V_2$ and $g : E_1 \to E_2$ such that $f(u) \overset{g(e)}{\text{---}} f(v)$ in G_2 for every edge $u \overset{e}{\text{---}} v$ in G_1. Clearly, G_1 is planar if and only if G_2 is. Thus, we are not interested in the particular names of the vertices or edges, and we distinguish between graphs only up to isomorphism.

with respect to $\{a, b\}$, where $m > 1$. By Lemma 7.4, there exists an $1 \leqslant i \leqslant m$ for which $H_i + (a \text{—} b)$ is nonplanar. Clearly $H_i + (a \text{—} b)$ does not contain a subgraph that is homeomorphic to one of Kuratowski's graphs either. This contradicts the assumption that G has the minimum number of edges.

We now omit an edge $a_0 \overset{e_0}{\text{—}} b_0$ from G. The resulting graph, G_0, is planar. Since the connectivity of G is at least 3, G_0 is nonseparable. By Theorem 6.7, there is a simple circuit in G_0 that goes through a_0 and b_0. Let $\hat{G}_0$ be a plane realization of G_0 and C be a simple circuit that goes through a_0 and b_0, such that C encircles the maximum number of faces of all such circuits in all the plane realizations of G_0. Note that the selection of $\hat{G}_0$ and C is done simultaneously, and not successively. Assuming u and v are vertices on C, let $C[u, v]$ be the part of C going from u to v, clockwise. $C(u, v)$ is defined similarly, but the vertices u and v are excluded.

Consider now the external bridges of C in $\hat{G}_0$. If such a bridge, B, has two attachments either on $C[a_0, b_0]$ or $C[b_0, a_0]$, then C is not maximum. To see this, assume that B has two attachments, a and b, in $C[a_0, b_0]$. There is a simple path $P(a, b)$ connecting a to b via the edges and vertices of B, which is disjoint from C, and is therefore exterior to C. Form C' by adding to P the path $C[b, a]$. C' goes through a_0 and b_0, and the interior of C is either completely included in the interior of C' or in the exterior (see Figure 7.5). In the first case, C' has a larger interior than C, in $\hat{G}_0$. In the later case, we have to find another plane realization that is similar to G_0, but the exterior of C' is now the interior. In either case, the maximality of the choice of $\hat{G}_0$ and C is contradicted. Thus, each

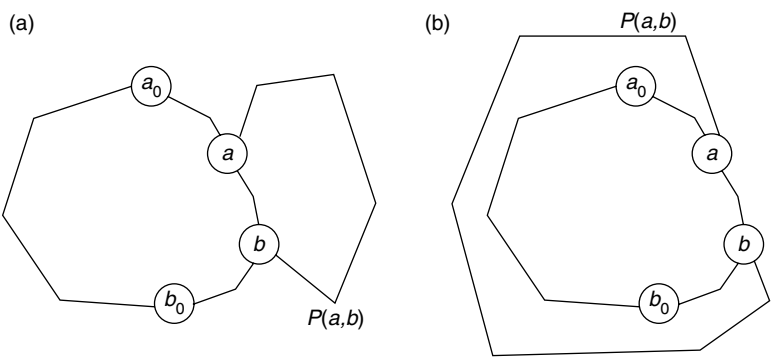


Figure 7.5: (a) The interior of the circuit C is contained in the interior of C'; (b) the interior of the circuit C is contained in the exterior of C'.

external bridge has at most one attachment in $C[a_0, b_0]$ and $C[b_0, a_0]$. Since an external bridge must have at least two attachments, it follows that neither a_0 nor b_0 can be an attachment of an external bridge.

It follows that each external bridge has at most two attachments. Since the vertex connectivity is at least 3, we conclude that all external bridges are singular (i.e., consist of a single edge).

Finally, there is at least one such external singular bridge; otherwise, one could draw the edge e_0 outside C, to yield a planar realization of G. Every external singular bridge interlaces with $a_0 \overset{e_0}{\text{---}} b_0$ since the attachments are on $C(a_0, b_0)$ and $C(b_0, a_0)$.

Similarly, there must be an internal bridge, B^*, which prevents the drawing of e_0 inside, and which cannot be transferred outside; that is, B^* interlaces with an external singular bridge, say, $a_1 \overset{e_1}{\text{---}} b_1$. The situation is schematically shown in Figure 7.6. We divide the argument to two cases according to whether B^* has any attachment other than a_0, b_0, a_1, b_1.

Case 1: B^* has an attachment a_2 other than a_0, b_0, a_1, b_1. Without loss of generality we may assume that a_2 is on $C(a_1, a_0)$. Since B^* prevents the drawing of e_0, it must have an attachment on $C(a_0, b_0)$. Since B^* interlaces e_1, it must have an attachment on $C(b_1, a_1)$.

Case 1.1: B^* has an attachment b_2 on $C(b_1, b_0)$. In B^*, there is a path P connecting a_2 with b_2. The situation is shown in Figure 7.7. The subgraph of

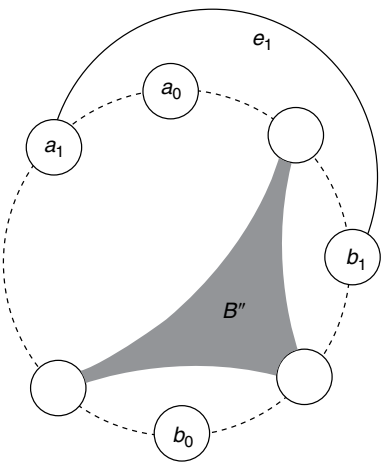


Figure 7.6: The bridge B^* in the proof of Theorem 7.2.

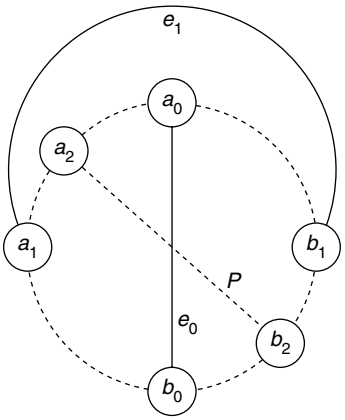


Figure 7.7: Case 1.1 in the proof of Theorem 7.2.

G shown in Figure 7.7 is homeomorphic to $K_{3,3}$, where a_1, a_0, and b_2 play the role of the upper vertices of Figure 7.2(a), and a_2, b_1 and b_0 play the role of the lower vertices.

Case 1.2: B^* has no attachments on $C(b_1, b_0)$. Thus, B^* has one attachment b_2' on $C(a_0, b_1)$; that is, it may be b_1 but not a_0. Also, B^* has an attachment b_2'' on $C[b_0, a_1)$. By Lemma 7.1, there exists a vertex v and three vertex-disjoint paths in B^*: $P_1(v, a_2), P_2(v, b_2'')$, and $P_3(v, b_2'')$. The situation is shown in Figure 7.8. If we erase from the subgraph of G, shown in Figure 7.8 the edges in the path $C[b_1, b_0]$ and all its intermediate vertices, the resulting subgraph is homeomorphic to $K_{3,3}$: Vertices a_2, b_2', and b_2'' play the role of the upper vertices; and a_0, a_1, and v, the lower vertices.

Case 2: B^* has no attachments other than a_0, b_0, a_1, b_1. In this case all four must be attachments; for, if a_0 or b_0 are not, then B^* and e_1 do not interlace; if a_1 or b_1 are not, then B^* does not prevent the drawing of e_0.

Case 2.1: There is a vertex v, in B^*, from which there are four disjoint paths in B^*: $P_1(v, a_0), P_2(v, b_0), P_3(v, a_1)$, and $P_4(v, b_1)$. This case is shown in Figure 7.9, and the shown subgraph is clearly homeomorphic to K_5.

Case 2.2: No vertex as in Case 2.1 exists. Let $P_0(a_0, b_0)$ and $P_1(a_1, b_1)$ be two simple paths in B^*. Let c_1 be the first vertex on P_1 that is common with P_0, and let c_2 be the last on P_1 that is common with P_0. We use only the first part, A,

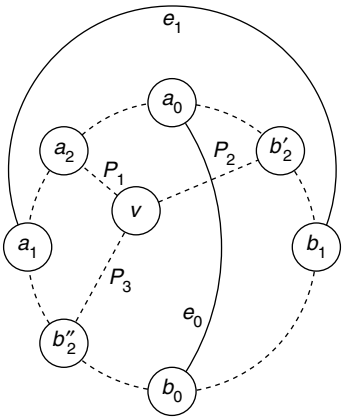


Figure 7.8: Case 1.2 in the proof of Theorem 7.2.

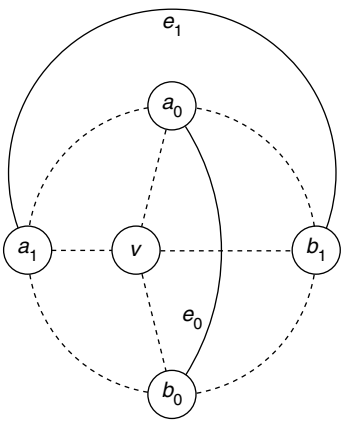


Figure 7.9: Case 2.1 in the proof of Theorem 7.2.

of P_1, connecting a_1 and c_1; and the last part, B, connecting c_2 with b_1. The pertaining subgraph of G is now shown in Figure 7.10, and is homeomorphic to $K_{3,3}$ after the edges in $C[a_0, b_1]$ and $C[b_0, a_1]$ and all their intermediate vertices are erased: Vertices a_0, b_1, and c_1 play the role of the upper vertices; and b_0, a_1, and c_2, the lower. (If c_1 is closer to a_0 than c_2, then we erase $C[a_1, a_0]$ and $C[b_1, b_0]$, instead, and the upper vertices are a_0, a_1 and c_2.) ■

Kuratowski's theorem provides a necessary and sufficient condition for a graph to be planar. However, it does not yield an efficient algorithm for planarity

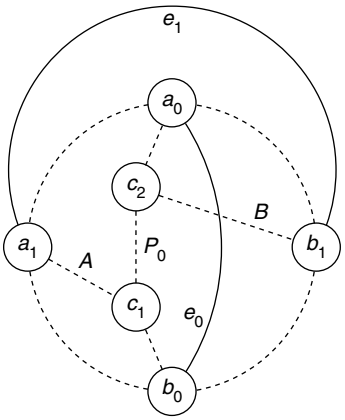


Figure 7.10: Case 2.2. in the proof of Theorem 7.2.

testing. The obvious procedure, that of trying for all subsets of five vertices to see whether there are ten vertex disjoint paths connecting all pairs, or for every pairs of three and three vertices whether there are nine paths, suffers from two shortcomings. First there are $\binom{|V|}{}$ choices of five-sets and $1/2 \cdot \binom{|V|}{3} \cdot \binom{|V|-3}{3}$ choices of three, and three vertices; this alone is $O(|V|^6)$. But what is worse, we have no efficient way to look for the disjoint paths; this problem may, in fact, be exponential.

Fortunately, there are $O(|E|)$ tests, as we shall see in the next chapter, for testing whether a given graph is planar.

7.2 Equivalence

Let $\hat{G}_1$ and $\hat{G}_2$ be two plane realizations of the graph G. We say that $\hat{G}_1$ and $\hat{G}_2$ are *equivalent* if every window of one of them is also a window in the other. G may be 2-connected and have nonequivalent plane realization; for example, see Figure 7.11

Let us restrict our discussion to planar finite graphs with no parallel edges and no self-loops. Our aim is to show that if the vertex connectivity of G, $c(G)$, is at least three then the plane realization of G is unique up to equivalence.

Lemma 7.5 *A planar nonseparable graph G is 2-connected if there is a plane realization of it, $\hat{G}$, and one of its windows has more than one bridge.*

Proof: If C is a window of $\hat{G}$ with more than one bridge, then all C's bridges are external. Therefore, no two bridges interlace. As in the first paragraph of the

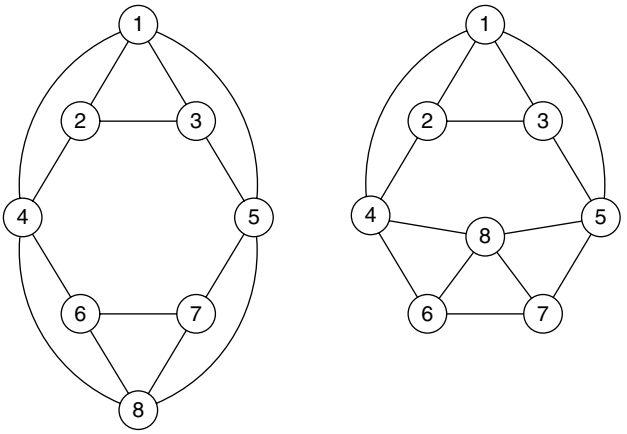


Figure 7.11: Two nonequivalent plane realizations of the same graph.

proof of Lemma 7.2, there exists a bridge B whose attachments can be ordered $a_1, a_2, \ldots a_t$, and no attachments of any other bridge appear on $C(a_1, a_t)$. It is easy to see that $\{a_1, a_t\}$ is a separating pair; it separates the vertices of B and $C(a_1, a_t)$ from the set of vertices of all other bridges and $C(a_t, a_1)$, where neither set can be empty since G has no parallel edges. ■

Theorem 7.3 *If G is a plane graph with no parallel edges and no self-loops and if its vertex connectivity, $c(G)$, is at least 3, then every two plane realizations of G are equivalent.*

Proof: Assume that G has two plane realizations $\hat{G}_1$ and $\hat{G}_2$ that are not equivalent. Without loss of generality we may assume that there is a window C in $\hat{G}_1$ that is not a window in $\hat{G}_2$. Therefore, C has at least two bridges; one interior and one exterior. By Lemma 7.5, and since C is a window in $\hat{G}_1$, G is 2-connected. A contradiction, since $c(G) \geqslant 3$. ■

7.3 Euler's Theorem

The following theorem is due to Euler.

Theorem 7.4 *Let $G(V, E)$ be a nonempty connected plane graph. The number of faces, f, satisfies*

$$|V| + f - |E| = 2. \tag{7.1}$$

Proof: By induction on $|E|$. If $|E| = 0$, then G consists of one vertex and there is one face, and Equation 7.1 holds. Assume that the theorem holds for all graphs with $m = |E|$. Let $G(V, E)$ be a connected plane graph with $m + 1$ edges. If G contains a circuit, then we can remove one of its edges. The resulting plane graph is connected and has m edges and, therefore, by the inductive hypothesis, satisfies Equation 7.1. Adding back the edge increases the number of faces by one and the number of edges by one, and thus Equation 7.1 is maintained. If G contains no circuits, then it is a tree. By Corollary 2.1, it has at least two leaves. Removing a leaf and its incident edge yields a connected graph with one less edge and one less vertex, which satisfies Equation 7.1. Therefore, G satisfies Equation 7.1 too. ■

The theorem implies that all connected plane graphs with $|V|$ vertices and $|E|$ edges have the same number of faces. One can draw many conclusions from the theorem. Some of them are the following:

Corollary 7.2 *If $G(V, E)$ is a connected plane graph with no parallel edges, no self-loops and $|V| > 2$, then*

$$|E| \leqslant 3|V| - 6. \tag{7.2}$$

Proof: Since there are no parallel edges, every window consists of at least three edges. Each edge appears on the windows of two faces, or twice on the window of one face. Thus, $3 \cdot f \leqslant 2 \cdot |E|$. By Equation 7.1, $|E| = |V| + f - 2$. Thus, $|E| \leqslant |V| + 2/3|E| - 2$, and (7.2) follows. ■

Corollary 7.3 *Every connected plane graph with no parallel edges and no self-loops has at least one vertex whose degree is 5 or less.*

Proof: Assume the contrary; that is, the degree of every vertex is at least 6. Thus, $2 \cdot |E| \geqslant 6 \cdot |V|$; note that each edge is counted in each of its two end-vertices. This contradicts (7.2). ■

7.4 Duality

Let $G(V, E)$ be a finite undirected and connected graph. A set $K \subseteq E$ is called a *cutset* if it is a minimal separating set of edges; that is, the removal of K from G interrupts its connectivity, but no proper subset of K does it. It is easy to see that a cutset separates G into two connected components: Consider first the removal of $K - \{e\}$, where $e \in K$. G remains connected. Now remove e. Clearly, G breaks into two components.

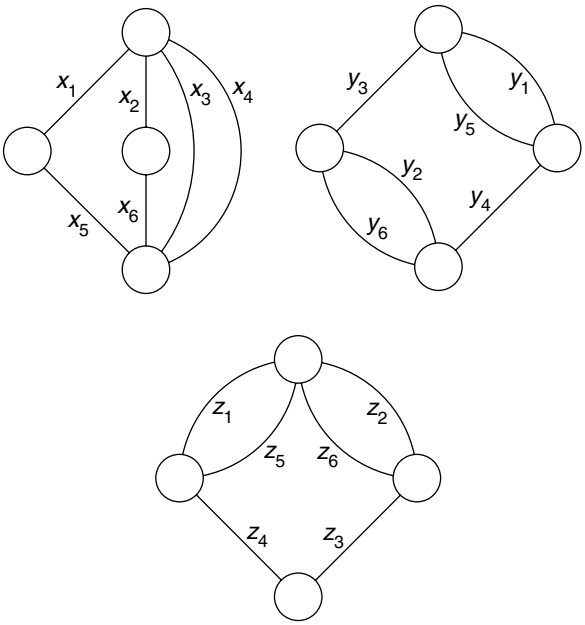


Figure 7.12: (a) A connected graph G_1; (b) G_2, a dual of G_1; (c) G_3, also a dual of G_1.

The graph $G_2(V_2, E_2)$ is said to be the *dual* of a connected graph $G_1(V_1, E_1)$ if there is a $1-1$ correspondence $f : E_1 \to E_2$, such that a set of edges S forms a simple circuit in G_1 if and only if $f(S)$ (the corresponding set of edges in G_2) forms a cutset in G_2. Consider the graph G_1 shown in Figure 7.12 (a). G_2 shown in Figure 7.12(b) is a dual of G_1, but so is G_3, shown in Figure 7.12(c), as the reader can verify by considering all (six) simple circuits of G_1 and all cutsets of G_2, or G_3.

A *contraction* of an edge $x \overset{e}{\text{—}} y$ of a graph $G(V, E)$ is the following operation: Delete the edge e, and merge x with y. The new contracted graph, G', has one less edge and one less vertex, if $x \neq y$. Clearly, if G is connected, so is G'. Also, graph G' is a contraction of G if by repeated contractions we can construct G' from G.

Lemma 7.6 *If a connected graph G_1 has a dual and G_1' is a connected subgraph of G_1, then G_1' has a dual.*

Proof: We can get G_1' from G_1 by a sequence of two kinds of deletions:

1. A deletion of an edge e of the present graph, which is not in G_1', and whose deletion does not interrupt the connectivity of the present graph.
2. A deletion of a leaf of the present graph, which is not a vertex of G_1', together with its incident edge.

We want to show that each of the resulting graphs, starting with G_1 and ending with G_1', has a dual.

Let G be one of these graphs, except the last, and its dual be G_d. First consider a deletion of type (7.4), of an edge e. Construct $f(e)$ in G_d, to get G_{dc}. If C is a simple circuit in $G - e$, then clearly it cannot use e, and therefore it is a circuit in G too. The set of edges of C is denoted by S. Thus, $f(S)$ is a cutset of G_d, and it does not include $f(e)$. Thus, the end vertices of $f(e)$ are in the same component of G_d with respect to $f(S)$. It follows that $f(S)$ is a cutset of G_{dc} too. If K is a cutset of G_{dc}, then it is a cutset of G_d too. Thus, $f^{-1}(K)$ form a simple circuit C' in G. However, $f(e)$ is not in K, and therefore e is not in C'. Hence, C' is a simple circuit of $G - e$.

Next, consider a deletion of type (7.4) of a leaf v and its incident edge e. Clearly, e, plays no role in a circuit. Thus, $f(e)$ cannot be a part of a cutset in G_d. Hence, $f(e)$ is a self-loop. The deletion of v and e form G, and the contraction of $f(e)$ in G_d (which effectively, only deletes $f(e)$ from G_d), does not change the sets of simple circuits in G and cutsets in G_d, and the correspondence is maintained. ■

Lemma 7.7 *Let G be a connected graph and e_1, e_2 be two of its edges, neither of which is a self loop. If for every cutset, either both edges are in it or both are not, then e_1 and e_2 are parallel edges.*

Proof: If e_1 and e_2 are not parallel, then there exists a spanning tree which includes both. (Such a tree can be found by contracting both edges and finding a spanning tree of the contracted graph.) The edge e_1 separates the tree into two connected components whose sets of vertices are S and $\bar{S}$. The set of edges between S and $\bar{S}$ in G is a cutset that includes e_1 and does not include e_2. A contradiction. ■

Lemma 7.8 *Let G be a connected graph with a dual G_d, and let f be the $1-1$ correspondence of their edges. If $u \xrightarrow{e_1} x_1 \xrightarrow{e_2} x_2 \xrightarrow{e_3} \ldots x_{l-1} \xrightarrow{e_l} v$ is a simple path or circuit in G such that $x_1, x_2, \ldots, x_{l-1}$ are all of degree two, then $f(e_1), f(e_2), \ldots, f(e_l)$ are parallel edges in G_d.*

Proof: Every circuit of G that contains one $e_i, 1 \leqslant i \leqslant l$, contains all the rest. Thus, in G_d, if one edge, $f(e_i)$, is in a cutset then all the rest are. By Lemma. 7.7, they are parallel edges. ■

Lemma 7.9 *If a connected graph G_1 has a dual, and G_2 is homeomorphic to G_1, then G_2 has a dual.*

Proof: If an edge $x \overset{e}{\text{---}} y$ of G_1 is replaced in G_2 by a path $x \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \text{ ---} \dots \overset{e_l}{\text{---}} y$, then in the dual $f(e)$ is replaced by parallel edges: $f(e_1), f(e_2), \dots, f(e_l)$. If a path of G_1 is replaced in G_2 by an edge e, then the edges of the dual that correspond to the edges of the path are all parallel (by Lemma 7.8) and can be replaced by a single edge $f(e)$. It is easy to see that every circuit which uses an edge e, when it is replaced by a path, will use all the edges of the path instead, while in the dual, every cutset that uses $f(e)$ will use all the parallel edges which replace it. Thus, the correspondence of circuits and cutsets is maintained. ■

Theorem 7.5 *A (connected) graph has a dual if and only if it is planar.*

Proof: Assume $G_1(V_1, E_1)$ is a planar graph and $\hat{G}_1$ is a plane realization of it. Choose a point p_i in each face F_i of $\hat{G}_1$. Let V_2 be the set of these points. Since G_1 is connected, the boundary of every face is a circuit (not necessarily simple) which we call its window. Let W_i be the window of F_i, and assume it consists of l edges. We can find l curved lines, all starting in p_i, but no two share any other point, such that each of the lines ends on one of the l edges of W_i, one line per edge. If a separating edge[3] e appears on W_i, then clearly it appears twice. In this case there will be two lines from p_i hitting e from both directions. These two lines can be made to end in the same point on e, thus creating a closed curve that goes through p_i and crosses e. If e is not a separating edge, then it appears on two windows, say, W_i and W_j. In this case, we can make the line from p_i to e meet e at the same point as does the line from p_j to e, to form a line connecting p_i with p_j, which crosses e. None of the set of lines thus formed crosses another, and we have one per edge of $\hat{G}_1$. Now define $\hat{G}_2(V_2, E_2)$ as follows: The set of lines connecting the p_i's is a representation of E_2. The $1-1$ correspondence $f : E_1 \to E_2$ is defined as follows: $f(e)$ is the edge of $\hat{G}_2$ which crosses e. Clearly, $\hat{G}_2$ is a plane graph that is a realization of a graph $G_2(V_2, E_2)$. It remains to show that there is a $1-1$ correspondence of the simple circuits of G_1 to the cutsets of G_2. The construction described

[3] An edge whose removal from G_1 interrupts its connectivity.

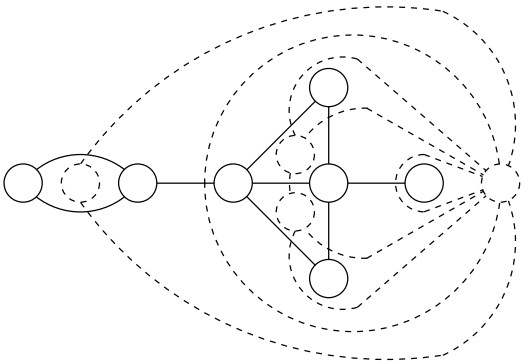


Figure 7.13: For the proof of Theorem 7.5.

above is demonstrated in Figure 7.13, where $\hat{G}_1$ is shown in solid lines, and $\hat{G}_2$ is shown in dashed lines.

Let C be a simple circuit of G_1. Clearly, in $\hat{G}_1$, C describes a simple closed curve in the plane. There must be at least one vertex of $\hat{G}_2$ inside this circuit, since at least one edge of $\hat{G}_2$ crosses the circuit, and it crosses exactly once. The same argument also applies to the outside. This implies that $f(S)$, where S is the set of edges of C, forms a separating set of G_2. Let us postpone the proof of the minimality of $f(S)$ for a little while.

Now let K be a cutset of G_2. Let T and $\bar{T}$ be the sets of vertices of the two components of G_2 formed by the deletion of K. The set of faces of $\hat{G}_1$ that correspond to the vertices of T, forms a continuous region, but not the whole plane. The minimality of K implies that the boundary of this region is a circuit in G_1, whose edges correspond to K. Thus, we have shown that every simple circuit of G_1 corresponds to a separating set of G_2, and every cutset of G_2 corresponds to a circuit of G_1.

Now let us handle the minimality. If S is the set of edges of a simple circuit C of G_1 and $f(S)$ is not a cutset of G_2, then there is a proper subset of $f(S)$ which is a cutset, say K. Therefore, $f^{-1}(K)$ is a circuit of G_1. However, $f^{-1}(K) \subsetneq S$. A contradiction to the assumption that C is simple. The proof that if K is a cutset then $f^{-1}(K)$ is a simple circuit is similar.

This completes the proof that the connected planar graph has a dual. We turn to the proof that a nonplanar graph has no dual. Here, we follow the proof of Parson [3].

First, let us show that neither $K_{3,3}$ nor K_5 have a dual. In $K_{3,3}$, the shortest circuit is of length four, and for every two edges there exists a simple circuit that one but does not use the other. Thus, in its dual no cutset consists of less

than four edges and there are no parallel edges. Therefore, the degree of each vertex is at least 4 and there must be at least five vertices. The number of edges is therefore at least $(5 \cdot 4/2) = 10$, while $K_{3,3}$ has 9 edges. Thus, $K_{3,3}$ has no dual. In the case of K_5, it has ten edges, ten simple circuits of length three and no shorter circuits, and for every two edges there is a simple circuit that uses one but not the other. Thus, the dual must have ten edges, ten cutsets of three edges, no cutset of lesser size and therefore the degree of every vertex is at least three. Also, there are no parallel edges in the dual. If the dual has five vertices, then it is K_5 itself (ten edges and no parallel edges), but K_5 has no cutsets of three edges. If the dual has seven vertices or more, then it has at least eleven edges ($\lceil 7 \cdot 3/2 \rceil$). Thus, the dual must have six vertices. Since it has ten clusters of three edges, there is one that separates S from $\bar{S}$, where neither consists of a single vertex. If $|S| = 2$, then it contains a vertex whose degree is 2. If $|S| = |\bar{S}| = 3$, then the maximum number of edges in the dual is nine. Thus, K_5 has no dual.

Now, if G is a nonplanar graph with a dual, then by Kuratowski's theorem (Theorem 7.2) it contains a subgraph G' that is homeomorphic to either $K_{3,3}$ or K_5. By Lemma 7.6, G' also has a dual. By Lemma 7.9, either $K_{3,3}$ or K_5 has a dual. A contradiction. Thus, no nonplanar graph has a dual. ■

Theorem 7.5 provides another necessary and sufficient condition for planarity, in addition to Kuratowski's theorem. However, neither has been shown to be useful for testing planarity.

There are many facts about duality that we have not discussed. Among them are the following:

1. If G_d is a dual of G, then G is a dual of G_d.
2. A 3-connected planar graph has a unique dual.

The interested reader can find additional information and references in the books of Harary [4], Ore [5], and Wilson [6].

7.5 Problems

Problem 7.1 The purpose of this problem is to prove a variation of Kuratowski's theorem.

1. Prove that if G is a connected graph and v_1, v_2, v_3 are three vertices, then there exists a vertex v and three (vertex) disjoint paths $P_1(v, v_1), P_2(v, v_2), P_3(v, v_3)$, one of which may be empty.
2. Prove that if G is a connected graph and S is a set of four vertices, then either there is a vertex v with four disjoint paths to the members of S or there

are two vertices u and v such that two members of S are connected to u by paths, two to v, and u is connected to v; all five paths are vertex-disjoint.

3. Show that if a graph is contractible to $K_{3,3}$, then it has a subgraph homeomorphic to $K_{3,3}$.
4. Show that if a graph is contractible to K_5, then either it has a subgraph that is homeomorphic to K_5 or it has a subgraph that is contractible to $K_{3,3}$.
5. Prove the theorem: A graph is nonplanar if and only if it contains a subgraph that is contractible to $K_{3,3}$ or K_5.

Problem 7.2 Show a graph that is nonplanar but not contractible to either $K_{3,3}$ or K_5. Does it contradict the result of Problem 7.1(7.1)?

Problem 7.3 Use Kuratowski's theorem to prove that the Petersen graph, shown in Figure 7.14, is nonplanar.

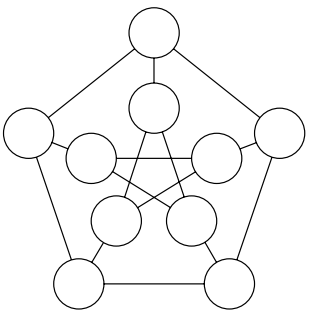


Figure 7.14: Peterson graph.

Problem 7.4 Is the graph depicted in Figure 7.15 planar? Justify your answer.

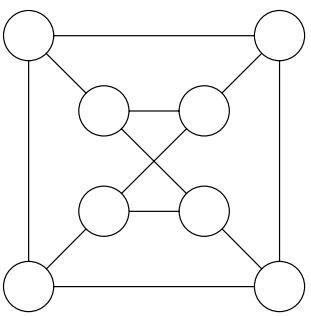


Figure 7.15: Is this graph planar?

Problem 7.5 A plane graph is called *triangular* if each of its windows is a triangle. Let N_i be the number of vertices whose degree is i.

1. Prove that every plane graph with three or more vertices that has no self-loops and no parallel edges can be made triangular by adding new edges without creating any self-loops or parallel edges. (This process is called *triangulation*.)
2. Let G be a triangular plane graph as in part 1, with $|V| > 3$. Prove that

$$N_1 = N_2 = 0.$$

3. Let G be a triangular plane graph as in part 1. Prove that

$$12 = 3 \cdot N_3 + 2 \cdot N_4 + N_5 - N_7 - 2 \cdot N_8 - 3 \cdot N_9 - 4 \cdot N_{10} \ldots$$

4. Prove that in a graph, in part 1, if there are no vertices of degree 3 or 4, then there are at least twelve vertices of degree 5.
5. Prove that if G is a planar graph with no self-loops or parallel edges, and $|V| > 3$, then it has at least 4 vertices with degrees less than 6, and if $N_3 = N_4 = 0$, then $N_5 \geqslant 12$.
6. Prove that if the vertex connectivity, $c(G)$, of a graph $G(V, E)$ is at least five, then $|V| \geqslant 12$.
7. Prove that if G is planar, then $c(G) < 6$.

Problem 7.6 Prove that if $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are homeomorphic, then

$$|E_1| - |V_1| = |E_2| - |V_2|.$$

Problem 7.7 Show a triangular plane graph without parallel edges that is not Hamiltonian.

Problem 7.8 Let G be a plane graph. The plane graph G^*, which is constructed by the procedure in the first part of the proof of Theorem 7.5, is called its *geometric dual*.

Prove that if G^* is the geometric dual of G, then G is the geometric dual of G^*.

Bibliography

[1] König, D., *Theorie der endlichen und unendlichen Graphen*. Leipzig, 1936. Reprinted Chelsea, 1950.
[2] Kuratowski, K., "Sur le problème des Courbes Gauches en Topologie," *Fund. Math.*, Vol. 15, 1930, pp. 217–283.
[3] Parson, T. D., "On Planar Graphs," *Am. Math.* Monthly, Vol. 78, No. 2, 1971, pp. 176–178.
[4] Harary, F., *Graph Theory*, Addison Wesley, 1969.
[5] Ore, O., *The Four-Color Problem*, Academic Press, 1976.
[6] Wilson, R. J., *Introduction to Graph Theory*, Longman, 1972.

8

Testing Graph Planarity

8.1 Introduction

There are two known planarity testing algorithms that have been shown to be realizable in a way that achieves linear time $(O(|V|))$. The idea in both is to follow the decisions to be made during the planar construction of the graph, piece by piece, as to the relative location of the various pieces. The construction is not carried out explicitly because there are difficulties, such as the crowding of elements into a relatively small portion of the area allocated, which, as yet, we do not know to avoid. Also, an explicit drawing of the graph is not necessary, as we shall see, to decide whether such a drawing is possible. We shall imagine that such a realization is being carried out, but will only decide where the various pieces are laid relative to each other, and not their exact shape. Such decisions may change later to make a place for later additions of pieces. In both cases, it was shown that the algorithm terminates within $(O(|V|))$ steps, and if it fails to find a "realization," then none exists.

The first algorithm starts by finding a simple circuit and adding to it one simple path at a time. Each such new path connects two old vertices via new edges and vertices. (Whole pieces are sometimes flipped over, around some line). Thus, we call it the *path addition* algorithm. The basic ideas were suggested by various authors, such as Auslander and Parter [1] and Goldstein [2], but the algorithm in its present form, both from the graph-theoretic point of view, and complexity point of view, is the contribution of Hopcroft and Tarjan [3]. They were first to show that planarity testing can be done in linear time.

The second algorithm adds one vertex in each step. Previously drawn edges incident to this vertex are connected to it, and new edges incident to it are drawn and their other endpoints are left unconnected. (Here, too, sometimes whole pieces have to be flipped around or permuted). The algorithm is due to Lempel, Even, and Cederbaum [4]. It consists of two parts. The first part was

shown to be linearly realizable by Even and Tarjan [5]; the second part was shown to be linearly realizable by Booth and Leuker [6]. We call this algorithm the *vertex addition* algorithm.

Each of these algorithms can be divided into its graph-theoretic part and its data structures and their manipulation. The algorithms are fairly complex and a complete description and proof would require a much more elaborate exposition. Thus, since this is a book on graphs, and not on programming, I have chosen to describe in full the graph-theoretic aspects of both algorithms and to only briefly describe the details of the data manipulation techniques. An attempt is made to convince the reader that the algorithms work, but in order to see the details which make it linear one will have to refer to the papers mentioned above.

Throughout this chapter, for reasons explained in Chapter 7, we shall assume that $G(V, E)$ is a finite undirected graph with no parallel edges and no self-loops. We shall also assume that G is nonseparable. The first thing that we can do is check whether $|E| \leqslant 3 \cdot |V| - 6$. By Corollary 7.1, if this condition does not hold, then G is nonplanar. Thus, we can restrict our algorithm to the cases where $|E| = O(|V|)$.

8.2 The Path Addition Algorithm of Hopcroft and Tarjan

The algorithm starts with a DFS of G. We assume that G is nonseparable. Thus, we drop from the DFS the steps for testing nonseparability. However, we still need the lowpoint function, to be denoted now $L1(v)$. In addition, we need the *second lowpoint* function, $L2(v)$, to be defined as follows: Let $S(v)$ be the set of values $k(u)$ of vertices u reachable from descendants of v by a single back edge. Clearly, $L1(v) = \min\{\{k(v)\} \cup S(v)\}$.

Define

$$L2(v) = \min\{\{k(v)\} \cup [S(v) - \{L1(v)\}]\}.$$

Let us now rewrite the DFS in order to compute these functions (we denote an assignment statement, "x gets the value of y" by $x := y$):

1. Mark all the edges "unused." For every $v \in V$, let $k(v) := 0$. Let $i := 0$ and $v := s$ (the vertex s is where we choose to start the DFS).
2. $i := i + 1, k(v) := i, L1(v) := i, L2(v) := i$.
3. If v has no unused incident edges, go to Step (8.2).
4. Choose an unused incident edge $v \overset{e}{\text{---}} u$. Mark e "used." If $k(u) \neq 0$ then do the following:
   If $k(u) < L1(v)$, then $L2(v) := L1(v), L1(v) := k(u)$.

If $k(u) > L1(v)$, then $L2(v) := \min\{L2(v), k(u)\}$.
Go to Step (8.2).
Otherwise $(k(u) = 0)$, let $f(u) := v$, $v := u$ and go to Step (8.2).

5. If $k(v) = 1$, halt.
6. $(k(v) > 1$; we backtrack).
   If $L1(v) < L1(f(v))$ then $L2(f(v)) := \min\{L2(v), L1(f(v))\}, L1(f(v)) := L1(v)$.
   If $L1(v) = L1(f(v))$, then $L2(f(v)) := \min\{L2(v), L2(f(v))\}$.
   Otherwise $(L1(v) > L1(f(v))), L2(f(v)) := \min\{L1(v), L2(f(v))\}$.
   $v := f(v)$.
   Go to Step (8.2).

From now on we refer to the vertices by their $k(v)$ number; that is, we change the name of v to $k(v)$.

Let $A(v)$ be the *adjacency list* of v; that is, the list of edges incident from v. We remind the reader that after the DFS each of the edges is directed; the tree edges are directed from low to high, and the back edges are directed from high to low. Thus, each edge appears once in the adjacency lists. Now, we want to reorder the adjacency lists, but first, we must define an order on the edges. Let the value $\phi(e)$ of an edge $u \xrightarrow{e} v$ be defined as follows:

$$\phi(e) = \begin{cases} 2 \cdot v & \text{if } u \xrightarrow{e} v \text{ is a back edge.} \\ 2 \cdot L1(v) & \text{if } u \xrightarrow{e} v \text{ is a tree edge and } L2(v) \geqslant u. \\ 2 \cdot L1(v) + 1 & \text{if } u \xrightarrow{e} v \text{ is a tree edge and } L2(v) < u. \end{cases}$$

Next, we order the edges in each adjacency list to be in nondecreasing order with respect to ϕ. [This can be done on $O(|V|)$ time as follows. First, compute for each edge it's ϕ value. Prepare $2 \cdot |V| + 1$ buckets, numbered $1, 2, \ldots, 2 \cdot |V| + 1$. Put each e into the $\phi(e)$ bucket. Now, empty the buckets in order, first bucket number 1, then 2 and so on, putting the edges taken out into the proper new adjacency lists, in the order that they are taken out of the buckets.]

The new adjacency lists are now used, in a second run of a DFS algorithm, to generate the paths, one by one. In this second DFS, vertices are not renumbered, and there is no need to recompute $f(v)$, $L1(v)$ or $L2(v)$. The tree remains the same, although the vertices may not be visited in the same order. The paths finding algorithm is as follows:

1. Mark all edges "unused" and let $v := 1$.
2. Start the circuit C; its first vertex is 1.

3. Let the first unused edge in $A(v)$ be $v \xrightarrow{e} u$.
   If $u \neq 1$, then mark e as used, add u to C, update $v := u$, and repeat Step (3). Otherwise, $(u = 1)$, C is closed. Output C.
4. If $v = 1$, halt.
5. If every edge in $A(v)$ is used, then update $v := f(v)$ and go to Step (4).
6. Start a path P with v as its first vertex.
7. Let $v \xrightarrow{e} u$ be the first unused edge in $A(v)$.
   If e is a back edge $(u < v)$, terminate P with u, output P, and go to (5).
8. Otherwise, (e is a tree edge). Add u to P, update $v := u$, and go to Step (7).

Lemma 8.1 *The paths finding algorithm finds first a circuit* C *that consists of a path from* 1 *(the root) to some vertex* v*, via tree edges, and a back edge from* v *to* 1.

Proof: Let $1 \longrightarrow u$ be the first edge of the tree to be traced (in the first application of Step (3)). We assume that G is nonseparable, and $|V| > 2$. Thus, by Lemma 3.7, this edge is the only tree edge out of 1, and $u = 2$. Also, 2 has some descendants, other than itself. Clearly, $2 \text{—} 3$ is a tree edge. By Lemma 3.5, $L1(3) < 2$, that is, $L1(3) = 1$. Thus, $L1(2) = 1$. The reordering of the adjacency lists assures that the first path to be chosen out of 1 will lead back to 1 as claimed. ■

Lemma 8.2 *Each generated path* P *is simple, and it contains exactly two vertices in common with previously generated paths; they are the first vertex,* f*, and the last,* l.

Proof: The edge scanning during the paths finding algorithm is in a DFS manner, in accord with the structure of the tree (but not necessarily in the same scanning order of vertices). Thus, a path starts from some (old) vertex f, goes along tree edges, via intermediate vertices which are all new, and ends with a back edge which leads to l. Since back edges always lead to ancestors, l is old. Also, by the reordering of the adjacency lists and the assumption that G is nonseparable, l must be lower than f. Thus, the path is simple. ■

Let S_v denote the set of descendants of v, including v itself.

Lemma 8.3 *Let* f *and* l *be the first and last vertices of a generated path* P *and* $f \longrightarrow v$ *be its first edge.*

1. *If* $v \neq l$ *then* $L1(v) = l$.
2. l *is the lowest vertex reachable from* S_f *via a back edge that has not been used in any path yet.*

Proof: Let us partition $S_f - \{f\}$ into two parts, α and β, as follows. A descendant u belongs to α if and only if when the construction of P begins, we have already backtracked from u. Clearly, $f \in \beta$ and all the back edges out of α have been scanned already. Let u be the lowest vertex reachable via an unused back edge from β. Clearly, the first remaining (unused) edge of $A(f)$ is the beginning of a directed path to u, which is either an unused back edge from f to u or a path of tree edges, via vertices of β, followed by a single back edge to u. Thus, $u = l$, and the lemma follows. ■

Lemma 8.4 *Let* P_1 *and* P_2 *be two generated paths whose first and last vertices are* f_1, l_1 *and* f_2, l_2*, respectively. If* P_1 *is generated before* P_2 *and* f_2 *is a descendant of* f_1 *then* $l_1 \leqslant l_2$.

Proof: The lemma follows immediately from Lemma 8.3. ■

So far, we have not made any use of $L2(v)$. However, Lemma 8.5 relies on it.

Lemma 8.5 *Let* P_1*:* $f \xrightarrow{e_1} v_1 \longrightarrow \ldots \longrightarrow l$ *and* P_2*:* $f \xrightarrow{e_2} v_2 \longrightarrow \ldots \longrightarrow l$ *be two generated paths, where* P_1 *is generated before* P_2*. If* $v_1 \neq l$ *and* $L2(v_1) < f$*, then* $v_2 \neq l$ *and* $L2(v2) < f$.

Proof: By the definition of $\phi(e)$, $\phi(e_1) = 2 \cdot l + 1$. If $v_2 = l$ or $L2(v2) \geqslant f$, then $\phi(e_2) = 2 \cdot l$, and e_2 should appear in $A(f)$ before e_1. A contradiction. ■

Let C be $1 \longrightarrow v_1 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow v_n \longrightarrow l$. Clearly $1 < v_1 < v_2 < \ldots < v_n$. Consider now the bridges of G with respect to C.

Lemma 8.6 *Let* B *be a nonsingular bridge of* G *with respect to* C*, whose highest attachment is* v_i*. There exists a tree edge* $v_i \xrightarrow{e} u$ *that belongs to* B*, and all other edges of* B *with endpoints on* C *are back edges.*

Proof: The lemma follows from the fact that the paths finding algorithm is a DFS. First C is found. We then backtrack from a vertex v_j only if all its descendants have been scanned. No internal part of B can be scanned before we backtrack into v_i. There must be a tree edge $v_i \longrightarrow u$, where u belongs to B, for the following reasons. If all the edges of B, incident to v_i are back edges, they all must come from descendants or go to ancestors of v_i (see Lemma 3.4). An edge from v_i to one of its ancestors (which must be on C) is a singular bridge and is not part of B. An edge from a descendant w of v_i implies that w cannot be in B, for it has been scanned already, and we have observed that no internal part of B can be scanned before we backtrack into v_i. If any other edge $v_k \text{---} x$ of B is also a tree edge, then, by definition of a bridge, there is a path

connecting u and x that is vertex-disjoint from C. Along this path there is at least one edge that contradicts Lemma 3.4. ■

Corollary 8.1 *Once a bridge* B *is entered, it is completely traced before it is left.*

Proof: By Lemma 8.6, there is only one edge through which B is entered. Since eventually the whole graph is scanned, and no edge is scanned twice in the same direction, the corollary follows. ■

Assuming that C and the bridges explored from vertices higher than v_i have already been explored and drawn in the plane. Lemma 8.7 provides a test for whether the next generated path could be drawn inside; the answer is negative, even if the path itself can be placed inside, but it is already clear that the whole bridge to which it belongs cannot be placed there.

Lemma 8.7 *Let* $v_i \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \ldots \longrightarrow u_l (= v_j)$ *be the first path,* P*, of the bridge* B *to be generated by the paths finding algorithm.* P *can be drawn inside (outside)* C *if there is no back edge* $w \longrightarrow v_k$ *drawn inside (outside) for which* $j < k < i$*. If there is such an edge, then* B *cannot be drawn inside (outside).*

Proof: The sufficiency is immediate. If there is no back edge drawn inside C, that enters the path, $v_j \longrightarrow v_{j+1} \longrightarrow \cdots \longrightarrow v_i$ in one of its internal vertices, then there cannot be any inside edge incident to these vertices, since bridges to be explored from vertices lower than v_i have not been scanned yet. Thus, P can be drawn inside if it is placed sufficiently close to C.

Now, assume there is a back edge $w \longrightarrow v_k$, drawn inside, for which $j < k < i$. Let P' be the directed path from v_p to v_k whose last edge is the back edge $w \longrightarrow v_k$. Clearly, v_p is on C and $p \geqslant i$; P' is not necessarily generated in one piece by the path-finding algorithm, if it is not the first path to be generated in the bridge B' to which it belongs.

Case 1: $p > i$. The bridges B and B' interlace by part (i) of the definition of interlacement. Thus, B cannot be drawn inside.

Case 2: $p = i$. Let P'' be the first path of B' to be generated, $P'' : v_i \longrightarrow x_1 \longrightarrow x_2 \longrightarrow \ldots \longrightarrow v_q$. By Lemma 8.4, $q \leqslant j$, since B' is explored before B.

Case 2.1: $q < j$. Since v_i and v_j are attachments of B, v_k and v_q are attachments of B' and $q < j < k < i$, the two bridges interlace. Thus, B cannot be drawn inside.

Case 2.2: $q = j$. P'' cannot consist of a single edge, for in this case, it is a singular bridge and v_k is not one of its attachments. Also, $L2(x_1) < v_k$. Thus, $L2(x_1) < v_i$. By Lemma 8.5, $u_1 \neq v_j$ and $L_2(u_1) < v_i$. This implies that B and B' interlace by either part (i) or part (ii) of the definition of interlacement, and B cannot be drawn inside.

■

The algorithm assumes that the first path of the new bridge B is drawn inside C. Now, we use the results of Corollary 7.1, Theorem 7.1 and the discussion that follows it, to decide whether the part of the graph explored so far is planar, assuming that $C + B$ is planar. By Lemma 8.7, we find which previous bridges interlace with B. The part of the graph explored so far is planar if and only if the set of its bridges can be partitioned into two sets such that no two bridges in the same set interlace. If the answer is negative, the algorithm halts, declaring that graph nonplanar. If the answer is positive, we still have to check whether $C + B$ is planar.

Let the first path of B be P: $v_i \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \ldots \longrightarrow v_j$. We now have a circuit C' consisting of $C[v_j, v_i]$ and P. The rest of C is an outside path P', with respect to C', and it consists of $C[v_i, 1]$ and $C[1, v_j]$. The graph $B + C[v_j, v_i]$ may have bridges with respect to C', but none of them has all its attachments on $C[v_j, v_i]$, for such a bridge is also a bridge of G with respect to C, and is not a part of B.

Thus, no bridge of C', with attachments of $C(v_j, v_i)$ may be drawn outside C', since it interlaces with P'. We conclude that $C + B$ is planar if and only if $B + C[v_j, v_i]$ is planar and its bridges can be partitioned into an inside set and an outside set so that the outside set contains no bridge with attachments in $C(v_j, v_i)$. The planarity of $B + C[v_j, v_i]$ is tested by applying the algorithm recursively, using the established vertex numbering L_1, L_2, f functions and ordering of the adjacency lists. If $B + C[v_j, v_i]$ is found to be nonplanar, clearly G is nonplanar. If it is found to be planar, we check whether all its bridges with attachments in $C(v_j, v_i)$ can be placed inside. If so, $C + B$ is planar; if not, G is nonplanar.

Hopcroft and Tarjan devised a simple tool for deciding whether the set of bridges can be partitioned properly. It consists of three stacks Π_1, Π_2, and Π_3. Of these Π_1 contains in nondecreasing order (the greatest entry on top) the vertices of $C(1, v_i)$ (where v_i is the last vertex of C into which we have backtraced), which are attachments of bridges placed inside C in the present partition of the bridges. A vertex may appear on Π_1 several times if there are several back edges into it from inside bridges. Π_2 is similarly defined for outside bridges. Both Π_1

and Π_2 are doubly linked lists, in order to enable switching over of complete sections from one of them to the other, investing, per section, time bounded by some constant. Π_3 consists of pairs of pointers to entries in Π_1 and Π_2. Its task will be described shortly.

Let S be a maximal set of bridges, explored up to now, such that a decision for any one of these bridges as to the side it is in, implies a decision for all the rest. (S corresponds to a connected component of the graph of bridges, as in the discussion following Theorem 7.1.) Let the set of entries in Π_1 and Π_2, which correspond to back edges from the bridges of S, be called a *block*.

Lemma 8.8 *Let K be a block, whose highest element is v_h and lowest element is v_l. If v_p is an entry in Π_1 or Π_2, then v_p belongs to K if $v_l < v_p < v_h$.*

Proof: We prove the lemma by induction on the order in which the bridges are explored. At first, both Π_1 and Π_2 are empty and the lemma is vacuously true. The lemma trivially holds after one bridge is explored.

Assume the lemma is true up to the exploration of the first path P of the new bridge B, where $P : v_i \longrightarrow \ldots \longrightarrow v_j$. If there is no vertex v_k on Π_1 or Π_2 such that $v_j < v_k < v_i$ then clearly the attachments of B (in $C(1, v_i)$) form a mew block (assuming $C + B$ is planar), and the lemma holds. However, if there are vertices of Π_1 or Π_2 in between v_j and v_i, then by Lemma 8.7, the bridges they are attachments of all interlace with B. Thus, the old blocks which these attachments belong to, must now be merged into one new block with the attachments of B (in $C(1, v_i)$). Now, let v_l be the lowest vertex of the new block and v_h be the highest. Clearly, v_l was the lowest vertex of some old block whose highest vertex was v'_h, and $v'_h > v_j$. Thus, by the inductive hypothesis, no attachment of another block could be between v_l and v'_h and therefore cannot be in this region after the merger. Also, all the attachments in between v_j and v_h are in the new block since they are attachments of bridges which interlace with B. Thus, all the entries of Π_1 or Π_2 which are in between v_l and v_h belong to the new block. ■

Corollary 8.2 *The entries of one block appear consecutively on $\Pi_1(\Pi_2)$.*

Thus, when we consider the first path $P : v_i \longrightarrow \ldots \longrightarrow v_j$ of the new bridge B in order to decide whether it can be drawn inside, we check the top entries t_1 and t_2 of Π_1 and Π_2, respectively.

If $v_j \geqslant t_1$ and $v_j \geqslant t_2$, then no merger is necessary; the attachments of B (in $C(1, v_i)$) are entered as a block (if $C + B$ is found to be planar) on top of Π_1.

If $v_j < t_1$ and $v_j \geqslant t_2$, then we first join all the blocks for which their highest entry is higher than v_j. To this end there is no need to check Π_2, since $v_j \geqslant t_2$;

however, several blocks still may merge. Next, switch the sections of the new block, that is, the section on Π_1 exchanges places with the section on Π_2. Finally, place the attachments of B in nondecreasing order on top of Π_1; these entries join the new block.

If $v_j \geqslant t_1$ and $v_j < t_2$, then again we join all blocks whose highest element is greater than v_j; only the sections on Π_2 need to be checked. The attachments of B join the same block are placed on top of Π_1.

If $v_j < t_1$ and $v_j < t_2$, then all blocks whose highest element is greater than v_j are joined into one new block. As we join the blocks, we examine them one by one. If the highest entry in the section on Π_1 is higher than v_j, then we switch the sections. If it is still higher, then we halt and declare the graph nonplanar. If all these switches succeed, then the merger is completed by adding the attachments of B on top of Π_1.

In order to handle the sections switching, the examination of their tops and mergers efficiently, we use a third stack, Π_3. It consists of pairs of pointers, one pair (x, y) per block; x points to the lowest entry of the section in Π_1; and y, to the lowest entry of the section in Π_2. (If the section is empty, then the pointer's value is 0). Two adjacent blocks are joined by simply discarding the pair of the top one. When several blocks have to be joined together upon the drawing of the first path of a new bridge, only the pair of the lowest of these blocks need remain, except when one of its entries is 0. In this case, the lowest nonzero entry of the pairs above it on the same side, if any such entry exists, takes its place.

When we enter a recursive step, a special "end of stack" marker E is placed on top of Π_2, and the three stacks are used as in the main algorithm. If the recursive step ends successfully, we first attempt to switch sections for each of the blocks with a nonempty section on Π_2, above the top most E. If we fail to expose E, then $C + B$ is nonplanar and we halt. Otherwise, all the blocks created during the recursion are joined to the one which includes v_j (the end vertex of the first path of B). The exposed E, on top of Π_2, is removed, and we continue with the previous level of recursion. When we backtrack into a vertex v_i, all occurrences of v_i are removed from the top of Π_1 and Π_2, together with pairs of pointers of Π_3 which point to removed entries on both Π_1 and Π_2. (Technically, instead of pointing to an occurrence of v_i, we point to 0 and pairs $(0,0)$ are removed).

Theorem 8.1 *The complexity of the path addition algorithm is* $O(|V|)$.

Proof: As in the closing remarks of Section 8.1, we can assume $|E| = O(|V|)$. The DFS and the reordering of the adjacency lists have been shown to be $O(|V|)$. Each edge in the paths finding algorithm is used again, once in each direction. The total number of entries in the stacks Π_1 and Π_2 is bounded by the number of

back edges ($|E|-|V|+1$), and is therefore $O(|V|)$. After each section switching the number of blocks is reduced by one; thus the total work invested in section switchings is $O(|V|)$. ■

8.3 Computing an st-Numbering

In this section, we define an st-numbering and describe a linear time algorithm to compute it. Thus numbering is necessary for the vertex addition algorithm, for testing planarity, of Lempel, Even, and Cederbaum.

Given any edge s —— t of a nonseparable graph $G(V,E)$, a 1-1 function $g: V \to \{1,2,\ldots,|V|\}$ is called an st-*numbering* if the following conditions are satisfied:

1. $g(s) = 1$.
2. $g(t) = |V|$ $(= n)$.
3. For every $v \in V - \{s,t\}$ there are adjacent vertices u and w such that $g(u) < g(v) < g(w)$.

Lempel, Even, and Cederbaum showed that, for every nonseparable graph and every edge s —— t, there exists an st-numbering. The algorithm described here, following the work of Even and Tarjan [5], achieves this goal in linear time.

The algorithm starts with a DFS whose first vertex is t and first edge is t —— s. (i.e., $k(t) = 1$ and $k(s) = 2$). This DFS computes for each vertex v, its DFS number, $k(v)$, its father, $f(v)$, and its lowpoint $L(v)$ and distinguishes tree edges from back edges. This information is used in the paths-finding algorithm to be described next, which is different from the one used in the path addition algorithm.

Initially, s, t, and the edge connecting them are marked "old," and all the other edges and vertices are marked "new." The path-finding algorithm starts from a given vertex v and finds a path from it. This path may be directed from v or into v.

1. If there is a "new" back edge $v \xrightarrow{e} w$ (in this case $k(w) < k(v)$), then do the following:
   Mark e "old."
   The path is $v \xrightarrow{e} w$.
   Halt.
2. If there is a "new" tree edge $v \xrightarrow{e} w$ (in this case $k(w) > k(v)$), then do the following:

Trace a path whose first edge is e, and from there it follows a path that defined $L(w)$, that is, it goes up the tree and ends with a back edge into a vertex u such that $k(u) = L(w)$. All vertices and edges on the path are marked "old." Halt.

3. If there is a "new" back edge $w \xrightarrow{e} v$ (in this case $k(w) > k(v)$), then do the following:
   Start the path with e (going backwards on it) and continue backwards via tree edges until you encounter an "old" vertex. All vertices and edges on the path are marked "old." Halt.
4. (All edges incident to v are "old"). The path produced is empty. Halt.

Lemma 8.9 *If the path finding algorithm is always applied from an "old" vertex $v \neq t$, then all the ancestors of an "old" vertex are "old" too.*

Proof: By induction on the number of applications of the path finding algorithm. Clearly, before the first application, the only ancestor of s is t, and it is old. Assuming the statement is true up to the present application, it is easy to see that if any of the four steps is applicable, the statement continues to hold after its application. ■

Corollary 8.3 *If G is nonseparable and under the condition of Lemma 8.9, each application of the path-finding algorithm from an "old" vertex v produces a path, through "new" vertices and edges, to another "old" vertex, or in case all edges incident to v are "old", it returns the empty path.*

Proof: The only case which requires a discussion is when case (2) of the path finding algorithm is applied. Since G is nonseparable, by Lemma 3.5, $L(w) < k(v)$. Thus, the path ends "bellow" v, in one of its ancestors. By Lemma 8.9, this ancestor is "old." ■

We are now ready to present the algorithm which produces an st-numbering. It uses a stack S that initially contains only t and s, s on top of t.

1. $i \leftarrow 1$.
2. Let v be the top vertex on S. Remove v from S. If $v = t$ then $g(t) \leftarrow i$ and halt.
3. ($v \neq t$) Apply the path finding algorithm to v. If the path is empty, then $g(v) \leftarrow i$, $i \leftarrow i + 1$ and go to Step (2).
4. (The path is not empty) Let the path be v — u_1 — u_2 — … — u_l — w. Put $u_l, u_{l-1}, \ldots, u_2, u_1, v$ on S in this order (v comes out on top), and go to Step (2).

Theorem 8.2 *The algorithm above computes an* st*-numbering for every nonseparable graph* $G(V,E)$.

Proof: First, we make a few observations about the algorithm:

1. No vertex ever appears in two or more places on S at the same time.
2. Once a vertex v is placed on S, nothing under v receives a number until v does.
3. A vertex is permanently removed from S only after all its incident edges become "old."

Next, we want to show that each vertex v is placed on S before t is removed. Since t and s are placed on S initially, the statement needs to be proved for $v \neq s, t$ only. Since G is nonseparable, there exists a simple path from s to v which does not pass through t (see Theorem 6.7 part (**??**)). Let this path be $s = u_1 \text{—} u_2 \text{—} \cdots \text{—} u_l = v$. Let m be the first index such that u_m is not placed on S. Since u_{m-1} is placed on S, t can be removed only after u_{m-1} (fact (ii)), and u_{m-1} is removed only after all its incident edges are "old" (fact (ii)). Thus, u_m must be placed on S before t is removed.

It remains to be shown that the algorithm computes an st-numbering.

Since each vertex is placed on S and eventually is removed, each vertex v gets a number $g(v)$. Clearly, $g(s) = 1$, for it is the first to be removed. After each assignment i is incremented. Thus, $g(t) = |V|$. Every other vertex v is placed on S, for the first time, as an intermediate vertex on a path. Thus, there is an adjacent vertex stored below it, and an adjacent vertex stored above it. The one above it (by fact (ii)) gets a lower number and the one below it, a higher number. ■

It is easy to see that the whole algorithm is of time complexity $O(|E|)$: First, the DFS is $O(|E|)$. The total time spent on path finding is also $O(|E|)$ since no edge is used more than once. The total number of operations in the main algorithm is bounded also by $O(|E|)$ because the number of stack insertions is exactly $|E|+1$.

8.4 The Vertex Addition Algorithm of Lempel, Even, and Cederbaum

In this section, we assume that $G(V,E)$ is a nonseparable graph whose vertices are st-numbered. From now on, we shall refer to the vertices by their st-number. Thus, $V = \{1,2,\ldots,n\}$. Also, the edges are now directed from low to high.

A (graphical) *source* of a digraph is a vertex v such that $d_{in}(v) = 0$; a (graphical) *sink* is a vertex v such that $d_{out}(v) = 0$. (Do not confuse with the source

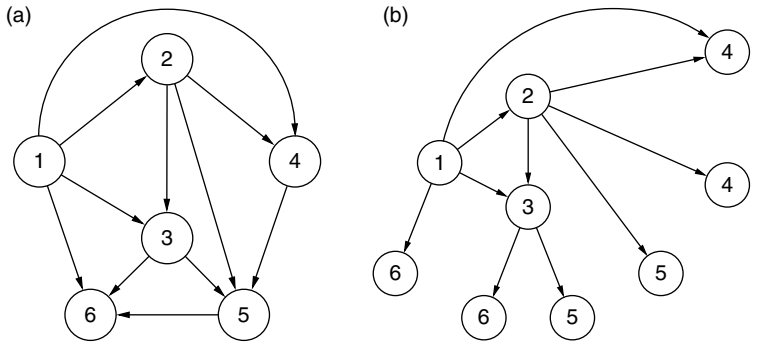


Figure 8.1: (a) An example of a digraph; (b) B_3 of the digraph from (a).

and sink of a network. The source of a network is not necessarily a (graphical) source, etc.) Clearly, vertex 1 is a source of G and vertex n is a sink. Furthermore, due to the st-numbering, no other vertex is either a source or a sink. Let $V_k = \{1, 2, \ldots, k\}$. Then, $G_k(V_k, E_k)$ is the digraph induced by V_k, that is, E_k consists of all the edges of G whose endpoints are both in V_k.

If G is planar, let $\hat{G}$ be a plane realization of G. It contains a plane realization of G_k. The following simple lemma reveals the reason for the st-numbering.

Lemma 8.10 *If $\hat{G}_k$ is a plane realization of G_k contained in a plane digraph $\hat{G}$, then all the edges and vertices of $\hat{G} - \hat{G}_k$ are drawn in one face of $\hat{G}_k$.*

Proof: Assume that a face F of $\hat{G}_k$ contains vertices of $V - V_k$. Since all the vertices on F's window are lower than the vertices in F, the highest vertex in F must be a sink. Since G has only one sink, only one such face is possible. ■

Let B_k be the following digraph. G_k is a subgraph of B_k. In addition, B_k contains all the edges of G which emanate from vertices of V_k and enter in G, vertices of $V - V_k$. These edges are called *virtual edges*, and the leaves they enter in B_k are called *virtual vertices*. These vertices are labeled as their counterparts in G, but they are kept separate; that is, there may be several virtual vertices with the same label, each with exactly one entering edge. For example, consider the digraph shown in Figure 8.1(a). B_3 of this digraph is shown in Figure 8.1(b).

By Lemma 8.10, we can assume that if G is planar, then there exists a plane realization of B_k in which all the virtual edges are drawn in the outside face. Furthermore, since $1 \xrightarrow{e} n$ is always an edge in G, if $k < n$ then vertex 1 is on the outside window and one of the virtual edges is e. In this case, we can

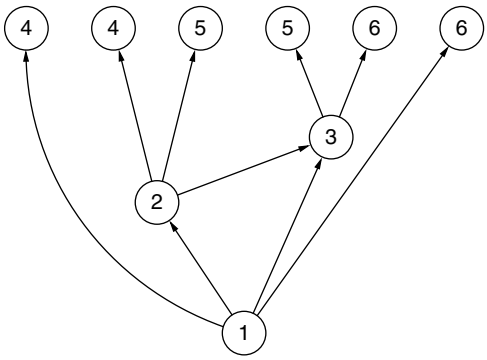


Figure 8.2: A bush form of B_3, of our example.

draw B_k in the following form: Vertex 1 is drawn at the bottom level. All the virtual vertices appear on one horizontal line. The remaining vertices of G_k are drawn in such a way that vertices with higher names are drawn higher. Such a realization is called a *bush form*. A bush form of B_3, of our example, is shown in Figure 8.2.

In fact, Lemma 8.10 implies that, if G is planar, then there exists a bush form of B_k such that all the virtual vertices with labels $k+1$ appear next to each other on the horizontal line.

The algorithm proceeds by successively "drawing" $B_1, B_2, \ldots, B_{n-1}$ and G. If in the realization of B_k all the virtual vertices labeled $k+1$ are next to each other, then it is easy to draw B_{k+1}: One joins all the virtual vertices labeled $k+1$ into one vertex and "pulls" it down from the horizontal line. Now all the edges of G that emanate from $k+1$ are added, and their other endpoints are labeled properly and placed in an arbitrary order on the horizontal line, in the space evacuated by the former virtual vertices labeled $k+1$.

However, a difficulty arises. Indeed, the discussion up to now guarantees that if G is planar, then there exists a sequence of bush forms such that each one is "grown" from the previous one. But since we do not have a plane realization of G, we may put the virtual vertices, out of $k+1$, in a "wrong" order. It is necessary to show that this does not matter; namely, by simple transformations it will be possible later to correct the "mistake."

Lemma 8.11 *Assume v is a separation vertex of B_k. If $v > 1$, then exactly one component of B_k, with respect to v, contains vertices lower than v.*

Note that here we ignore the direction of the edges, and the lemma is actually concerned with the undirected underlying graph of B_k.

Proof: The st-numbering implies that for every vertex u there exists a path from 1 to u such that all the vertices on the path are less than u. Thus, if $u < v$ then there is a path from 1 to u that does not pass through v. Therefore, 1 and u are in the same component. ■

Lemma 8.11 implies that a separation vertex v of B_k is the lowest vertex in each of the components, except the one which contains 1 (in case $v > 1$). Each of these components is a sub-bush; that is, it has the same structure as a bush form, except that its lowest vertex is v rather than 1. These subbushes can be permuted around v in any of the $p!$ permutations, in case the number of these subbushes is p. In addition, each of the subbushes can be flipped over. These transformations maintain the bush form. It is our purpose to show that if $\hat{B}_k^1$ and $\hat{B}_k^2$ are bush forms of a planar B_k, then through a sequence of permutations and flippings, one can change $\hat{B}_k^1$ into a $\hat{B}_k^3$ such that the virtual vertices of B_k appear in $\hat{B}_k^2$ and $\hat{B}_k^3$ in the same order.

For efficiency reasons, to be become clear to those readers who will study the implementation through P Q-trees, we assume that when a subbush is flipped, smaller subbushes of other separation vertices of the component are not flipped by this action. For example, consider the bush form shown in Figure 8.3(a). The bush form of Figure 8.3(b) is achieved by permuting about 1, Figure 8.3(c) by flipping about 1 and Figure 8.3(d) by flipping about 2.

Lemma 8.12 *Let* H *be a maximal nonseparable component of* B_k *and* $y_1, y_2, \ldots, y_m$ *be the vertices of* H *that are also endpoints of edges of* $B_k - H$. *In every bush form* $\hat{B}_k$, *all the* y*'s are on the outside window of* H *and in the same order, except that the orientation may be reversed.*

Proof: Since $\hat{B}_k$ is a bush form, all the y's are on the outside face of H. Assume there are two bush forms $\hat{B}_k^1$ and $\hat{B}_k^2$ in which the realizations of H are $\hat{H}^1$ and $\hat{H}^2$, respectively. If the y's do not appear in the same order on the outside windows of $\hat{H}^1$ and $\hat{H}^2$, then there are two y's, y_i and y_j which are next to each other in $\hat{H}^1$ but not in $\hat{H}^2$ (see Fig. 8.4). Therefore, in $\hat{H}^2$, there are two other y's, y_k and y_l which interpose between y_i and y_j on the two paths between them on the outside window of $\hat{H}^2$. However, from $\hat{H}^1$ we see that there are two paths, $P_1[y_i, y_j]$ and $P_2[y_k, y_l]$, which are completely disjoint. These two paths cannot exist simultaneously in $\hat{H}^2$. A contradiction. ■

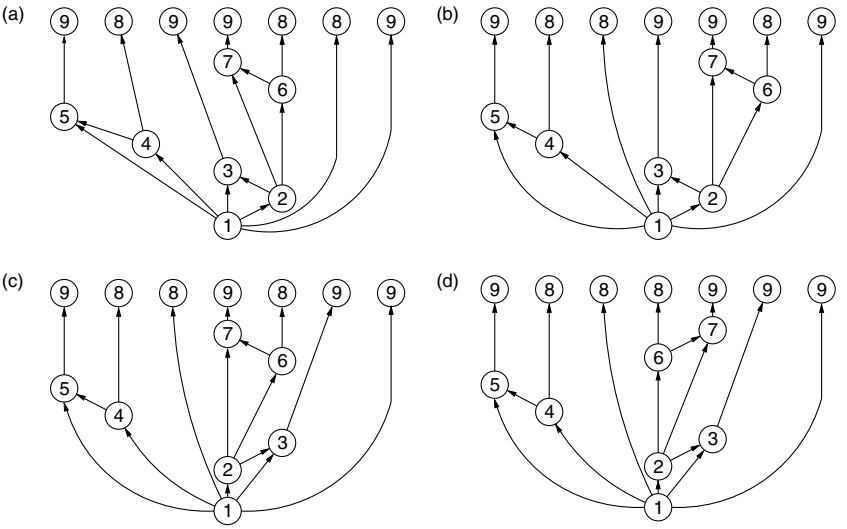


Figure 8.3: (a) A bush form; (b) a bush form obtained by permuting (a) about vertex 1; (c) a bush form obtained by flipping (a) about vertex 1; (d) a bush form obtained by flipping (c) about vertex 2.

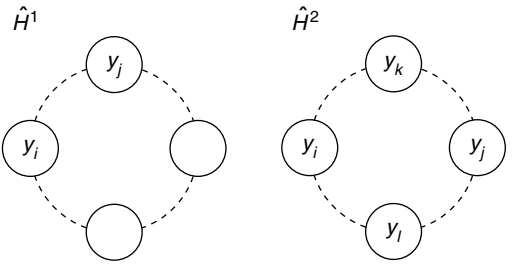


Figure 8.4: Proof of Lemma 8.12.

Theorem 8.3 *If* $\hat{B}^1_k$ *and* $\hat{B}^2_k$ *are bush forms of the same* B_k*, then there exists a sequence of permutations and flippings that transforms* $\hat{B}^1_k$ *into* $\hat{B}^3_k$*, such that in* $\hat{B}^2_k$ *and* $\hat{B}^3_k$*, the virtual vertices appear in the same order.*

Proof: By induction on the size of bush or subbush forms.[1] Clearly, if each of the two (sub-)bushes consists of only one vertex and one virtual vertex, then the statement is trivial. Let v be the lowest vertex in the (sub-)bushes

[1] Size the number of vertices.

$\hat{B}^1$ and $\hat{B}^2$ of the same B. If v is a separation vertex, then the components of B appear as subbushes in $\hat{B}^1$ and $\hat{B}^2$. If they are not in the same order, it is possible, by permuting them in $\hat{B}^1$, to put them in the same order as in $\hat{B}^2$. By the inductive hypothesis, there is a sequence of permutations and flippings that will change each subbush of $\hat{B}^1$ to have the same order of its virtual vertice as in its counterpart in $\hat{B}^2$, and therefore the theorem follows.

If v is not a separating vertex then let H be the maximal nonseparable component of B which contains v. In $\hat{B}^1(\hat{B}^2)$ there is a planar realization $\hat{H}^1(\hat{H}^2)$ of H. The vertices $y_1, y_2, \ldots, y_m$ of H, which are also endpoints of edges of $B - H$, by Lemma 8.12, must appear on the outside window of H in the same order, up to orientation. If the orientation of the y's in $\hat{H}^1$ is opposite to that of $\hat{H}^2$, flip the (sub-)bush $\hat{B}^1$ about v. Now, each of the y's is the lowest vertex of some subbush of B, and these subbushes appear in (the new) $\hat{B}^1$ and $\hat{B}^2$ in the same order. By the inductive hypothesis, each of these subbushes can be transformed by a sequence of permutations and flippings to have its virtual vertices in the same order as its counterpart in $\hat{B}^2$. ■

Corollary 8.4 *If* G *is planar and* $\hat{B}_k$ *is a bush form of* B_k*, then there exists a sequence of permutations and flippings which transforms* $\hat{B}_k$ *into a* $\hat{B}'_k$ *in which all the virtual vertices labeled* $k+1$ *appear together on the horizontal line.*

It remains to be shown how one decides which permutation or flipping to apply, how to represent the necessary information, without actually drawing bush forms, and how to do it all efficiently. Lempel, Even, and Cederbaum described a method that uses a representation of the pertinent information by proper expressions. However, a better representation was suggested by Booth and Leuker. They invented a data structure, called PQ-trees, through which the algorithm can be run in linear time. PQ-trees were used to solve other problems of interest; see [6].

I shall not describe PQ-trees in detail. The description in [6] is long (30 pages) although not hard to follow. It involves ideas of data structure manipulation, but almost no graph theory. The following is a brief and incomplete description.

A PQ-tree is a directed ordered tree, with three types of vertices: P-vertices, Q-vertices and leaves. Each P-vertex or Q-vertex, has at least one son. The sons of a P-vertex, which in our application represents a separating vertex v of B_k, may be permuted into any new order. Each sons and its subtree, represents a subbush. A Q-vertex represents a maximal nonseparable component, and its sons, which represent the y's, may not be permuted, but their order can be reversed. The leaves represent the virtual vertices.

The attempt to gather all the leaves labeled $k+1$ into an unbroken run, is done from sons to fathers, starting with the leaves labeled $k+1$. Through a technique of template matching, vertices are modified while the $k+1$ labeled leaves are bunched together. Only the smallest subtree containing all the $k+1-$ labeled leaves is scanned. All these leaves, if successfully gathered, are merged into one P-vertex, and its sons represent the virtual edges out of $k+1$. This procedure is repeated until $k+1=n$.

8.5 Problems

Problem 8.1 Demonstrate the path-addition algorithm on the Peterson graph (see Problem 7.3). Show the data for all the steps: the DFS for numbering the vertices, defining the tree and computing L1 and L2. The Φ function on the edges. The sorting of the adjacency lists. Use the path-finding algorithm in the new DFS to decompose the graph into C and a sequence of paths. Use Π_1, Π_2, Π_3 and end-of-stack markers to carry out all the recursive steps up to planarity decision.

Problem 8.2 Repeat the path-addition planarity test, as in Problem 8.1, for the graph depicted in Figure 8.5.

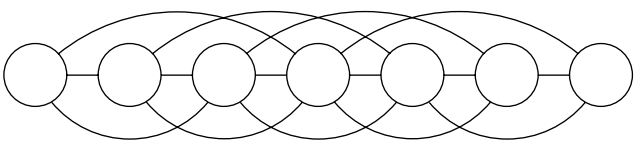


Figure 8.5: A graph for Problem 8.2.

Problem 8.3 Demonstrate the vertex-addition planarity test on the Peterson graph. Show the steps for the DFS, the st-numbering, and the sequence of bush forms.

Problem 8.4 Repeat the vertex-addition planarity test for the graph of Problem 8.2.

Problem 8.5 Show that if a graph is nonplanar, then a subgraph homeomorphic to one of the Kuratowski's graphs can be found in $O(|V|^2)$. (Hints: Only $O(|V|)$ edges need to be considered. Delete edges if their deletion does not make the graph planar. What is left?)

Bibliography

[1] Auslander, L., and Parter, S. V., "On Embedding Graphs in the Plane," *J. Math. and Mech.,* Vol. 10, No. 3, May 1961, pp. 517–523.

[2] Goldstein, A. J., "An Efficient and Constructive Algorithm for Testing Whether a Graph Can Be Embedded in a Plane," Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Math., Princeton Univ., May 16–18, 1963, 2 pp.

[3] Hopcroft, J., and Tarjan, R., "Efficient Planarity Testing," *JACM,* Vol. 21, No. 4, Oct. 1974, pp. 549–568.

[4] Lempel, A., Even, S., and Cederbaum, I., "An Algorithm for Planarity Testing of Graphs," *Theory of Graphs, International Symposium, Rome*, July, 1966. P. Rosenstiehl, Ed., Gordon and Breach, N.Y. 1967, pp. 215–232.

[5] Even, S., and Tarjan, R. E., "Computing and st-numbering," *Th. Comp. Sci.*, Vol. 2, 1976, pp. 339–344.

[6] Booth, K. S., and Lueker, G. S., "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms," *J. of Comp. and Sys. Sciences*, Vol. 13, 1976, pp. 335–379.

Index