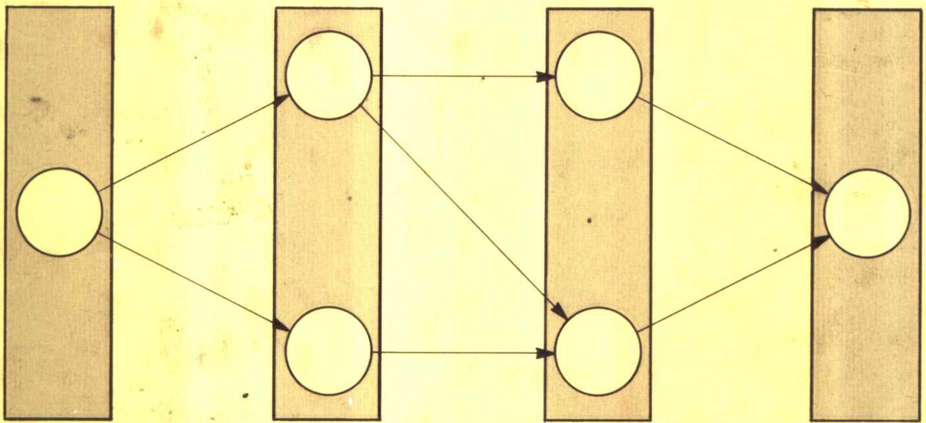
GRAPH ALGORITHMS

SHIMON EVEN



COMPUTER SCIENCE PRESS

GRAPH ALGORITHMS

SHIMON EVEN
Technion Institute

Computer Science Press

Copyright © 1979 Computer Science Press, Inc.

Printed in the United States of America.

All rights reserved. No part of this work may be reproduced, transmitted, or stored in any form or by any means, without the prior written consent of the Publisher.

Computer Science Press, Inc.
11 Taft Ct.
Rockville, Maryland 20850

4 5 6 85 84 83

Library of Congress Cataloging in Publication Data

Even, Shimon.
Graph algorithms.

(Computer software engineering series)
Includes bibliographies and index.
1. Graph theory. 2. Algorithms. I. Title.
II. Series.
QA166.E93 511'.5 79-17150
ISBN 0-914894-21-8
UK 0-273-08467-4

PREFACE

Graph theory has long become recognized as one of the more useful mathematical subjects for the computer science student to master. The approach which is natural in computer science is the algorithmic one; our interest is not so much in existence proofs or enumeration techniques, as it is in finding efficient algorithms for solving relevant problems, or alternatively 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 of Chapters 3, 5, 6, 8, 9 and 10 is less than ten years old.

This book is meant to be a textbook of an upper level undergraduate, or 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, University of California at Berkeley and the Technion. There is more than enough material for a one semester course, and I am sure that most teachers will have to omit parts of the book from their course. 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, 3.4. 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 preparatory reading assignment. The mathematical background needed for understanding Chapter 1 to 8 is some knowledge of set theory, combinatorics and algebra, which the computer science student usually masters during his freshman year through a course on discrete mathematics and a course on linear algebra. However, the student will also need to know a little about data structures and programming techniques, or he may not appreciate the algorithmic side or miss the complexity considerations. It is my experience that after two courses in programming the students have the necessary knowledge. However, in order to follow Chapters 9 and 10, 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 reliance on previous knowledge is made beyond the general background discussed above. No comments such as "the rest of the proof is left to the reader" or "this is beyond the scope of this book" is ever made. Some unproved results are mentioned, with a reference, but not used later in the book.

At the end of each chapter there are a few problems which the teacher 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 joint work and the influence they had on my work, and therefore on this book: I. Cederbaum, M. R. Garey, J. E. Hopcroft, R. M. Karp, A. Lempel, A. Pnueli, A. Shamir and R. E. Tarjan. Also, I would like to thank some of my former Ph.D. students for all 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.

Technion, Haifa, Israel

试读结束：需要全本请在线购买： www.ertongbook.com

CONTENTS

PREFACE v

1. PATHS IN GRAPHS

1.1 Introduction to graph theory 1
1.2 Computer representation of graphs 3
1.3 Euler graphs 5
1.4 De Bruijn sequences 8
1.5 Shortest-path algorithms 11
Problems 18
References 20

2. TREES

2.1 Tree definitions 22
2.2 Minimum spanning tree 24
2.3 Cayley's theorem 26
2.4 Directed tree definitions 30
2.5 The infinity lemma 32
2.6 The number of spanning trees 34
2.7 Optimum branchings and directed spanning trees 40
2.8 Directed trees and Euler circuits 46
Problems 49
References 51

3. DEPTH-FIRST SEARCH

3.1 DFS of undirected graphs 53
3.2 Algorithm for nonseparable components 57
3.3 DFS on digraphs 63
3.4 Algorithm for strongly-connected components 64
Problems 66
References 68

4. ORDERED TREES

4.1 Uniquely decipherable codes 69
4.2 Positional trees and Huffman's optimization problem 74
4.3 Application of the Huffman tree to sort-by-merge techniques........ 80
4.4 Catalan numbers 82
Problems........ 87
References 88

5. MAXIMUM FLOW IN A NETWORK

5.1 The Ford and Fulkerson algorithm........ 90
5.2 The Dinic algorithm 97
5.3 Networks with upper and lower bounds 104
Problems........ 112
References 115

6. APPLICATIONS OF NETWORK FLOW TECHNIQUES

6.1 Zero-one network flow........ 116
6.2 Vertex connectivity of graphs........ 121
6.3 Connectivity of digraphs and edge connectivity........ 130
6.4 Maximum matching in bipartite graphs........ 135
6.5 Two problems on PERT digraphs........ 138
Problems........ 142
References 146

7. PLANAR GRAPHS

7.1 Bridges and Kuratowski's theorem 148
7.2 Equivalence 160
7.3 Euler's theorem 161
7.4 Duality 162
Problems........ 168
References 170

8. TESTING GRAPH PLANARITY

8.1 Introduction........ 171
8.2 The path addition algorithm of Hopcroft and Tarjan...... 172
8.3 Computing an st-numbering 180

8.4 The vertex addition algorithm of Lempel, Even and Cederbaum 183
Problems 190
References 191

9. THE THEORY OF NP-COMPLETENESS

9.1 Introduction 192
9.2 The NP class of decision problems 195
9.3 NP-complete problems and Cook's theorem 198
9.4 Three combinatorial problems which are NPC 204
Problems 209
References 211

10. NP-COMPLETE GRAPH PROBLEMS

10.1 Clique, independent set and vertex cover 212
10.2 Hamilton paths and circuits 214
10.3 Coloring of graphs 217
10.4 Feedback sets in digraphs 223
10.5 Steiner tree 225
10.6 Maximum cut 226
10.7 Linear arrangement 230
10.8 Multicommodity integral flow 233
Problems 240
References 243

INDEX 245

Chapter 1

PATHS IN GRAPHS

1.1 INTRODUCTION TO GRAPH THEORY

A *graph* $G(V, E)$ is a structure which consists 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 is *incident* to the elements of an unordered pair of vertices $\{u, v\}$ which 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 represented 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 edge e_2 is incident to v_1 and v_2, which are called its *endpoints*. The edges e_4 and e_5 have the same endpoints and therefore are called *parallel* edges. Both endpoints of the edge e_1 are the same; 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 of the edges. Clearly, a self-loop uses its endpoint twice. Thus, in our example $d(v_4) = 1$, $d(v_2) = 3$ and $d(v_1) = 4$. Also, a vertex v whose degree is zero is called *isolated*; in our example v_3 is isolated since $d(v_3) = 0$.

Lemma 1.1: The number of vertices of odd degree in a finite graph is even.

Proof: Let $|V|$ and $|E|$ be the number of vertices and edges, respectively. Then,

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

since each edge contributes two to the left hand side; one to the degree of each of its two endpoints, if they are different, and two to the degree of its endpoint if it is a self-loop. It follows that the number of odd degrees must be even.

Q.E.D.

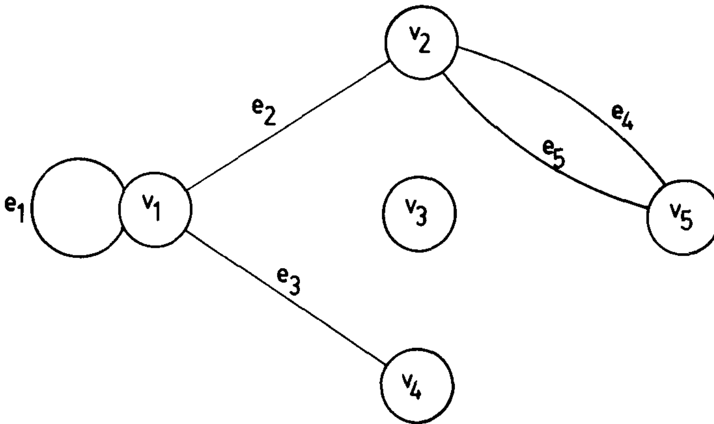


Figure 1.1

The notation $u \overset{e}{-} v$ means that the edge e has u and v as endpoints. In this case we also say that e *connects* vertices u and v, and that u and v are *adjacent*.

A *path* is a sequence of edges $e_1, e_2, \ldots$ such that:

(1) e_i and e_{i+1} have a common endpoint;
(2) if e_i is not a self-loop and is not the first or last edge then it shares one of its endpoints with e_{i-1} and the other with e_{i+1}.

The exception specified in (2) is necessary to avoid the following situation: Consider the graph represented in Figure 1.2.

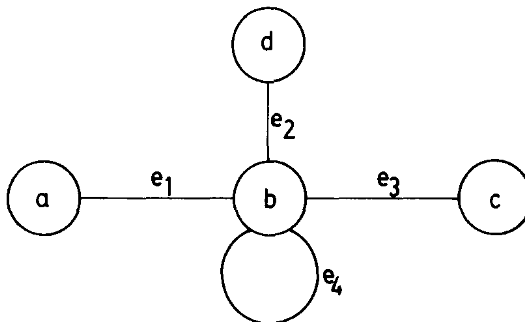


Figure 1.2

We do not like to call the sequence e_1, e_2, e_3 a path, and it is not, since the only vertex, b, which is shared by e_1 and e_2 is also the only vertex shared by e_2 and e_3. But we have no objection to calling e_1, e_4, e_3 a path. Also, the sequence e_1, e_2, e_2, e_3 is a path since e_1 and e_2 share b, e_2 and e_2 share d, e_2 and e_3 share b. It is convenient to describe a path as follows: $v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} v_2 \cdots v_{l-1} \overset{e_l}{—} v_l$. Here the path is $e_1, e_2, \ldots, e_l$ and the endpoints shared are transparent; v_0 is called the *start* and v_l is called the *end* vertex. The *length* of the path is l.

A *circuit* is a path whose start and end vertices are the same.

A path is called *simple* if no vertex appears on it more than once. A circuit is called *simple* if no vertex, other than the start-end vertex, appears more than once, and the start-end vertex does not appear elsewhere in the circuit; however, $u \overset{e}{—} v \overset{e}{—} u$ is not considered a simple circuit.

If for every two vertices u and v there exists a path whose start vertex is u and whose end vertex is v then the graph is called *connected*.

A *digraph* (or *directed graph*) is defined similarly to a graph except that the pair of endpoints of an edge is now ordered; the first endpoint is called the *start-vertex* of the edge and the second (which may be the same) is called its *end-vertex*. The edge $(u \overset{e}{\rightarrow} v)$ e is said to be *directed* from u to v. Edges with the same start vertex and the same end vertex are called *parallel*, and if $u \neq v$, $u \overset{e_1}{\rightarrow} v$ and $v \overset{e_2}{\rightarrow} u$ then e_1 and e_2 are *antiparallel*. An edge $u \rightarrow u$ is called a *self-loop*.

The *outdegree*, $d_{out}(v)$, of a vertex v is the number of edges which have v as their start-vertex; *indegree*, $d_{in}(v)$, is defined similarly. Clearly, for every graph

$$\sum_{i=1}^{|V|} d_{in}(v_i) = \sum_{i=1}^{|V|} d_{out}(v_i).$$

A *directed path* is a sequence of edges $e_1, e_2, \ldots$ such that the end vertex of e_{i-1} is the start vertex of e_i. A directed path is a *directed circuit* if the start vertex of the path is the same as its end vertex. The notion of a directed path or circuit being *simple* is defined similarly to that in the undirected case. A digraph is said to be *strongly connected* if for every vertex u and every vertex v there is a directed path from u to v; namely, its *start-vertex* is u and its *end-vertex* is v.

1.2 COMPUTER REPRESENTATION OF GRAPHS

In order to understand the time and space complexities of graph algorithms one needs to know how graphs are represented in the computer

试读结束：需要全本请在线购买： www.ertongbook.com

memory. In this section two of the most common methods of graph representation are briefly described.

Graphs and digraphs which have no parallel edges are called *simple*. In cases of simple graphs, the specification of the two endpoints is sufficient to specify the edge; in cases of digraph the specification of the start-vertex and end-vertex is sufficient. Thus, we can represent a graph or digraph of n vertices by an $n \times n$ matrix C, where $C_{ij} = 1$ if there is an edge connecting vertex v_i to v_j and $C_{ij} = 0$, if not. Clearly, in the case of graphs $C_{ij} = 1$ implies $C_{ji} = 1$; or in other words, C 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 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 contributes two to the count. For a digraph, the number of ones in the i row is equal to $d_{\text{out}}(v_i)$ and the number of ones in the i column is equal to $d_{\text{in}}(v_i)$.

The adjacency matrix is not an efficient representation of the graph in case the graph is *sparse*; namely, the number of edges is significantly smaller than n^2. In these cases the following representation, which also allows parallel edges, is preferred.

For each of the vertices, the edges incident to it are listed. This *incidence list* may simply be an array or may be a linked list. We may need a table which tells us the location of the list for each vertex and a table which tells us for each edge its two endpoints (or start-vertex and end-vertex, in case of a digraph).

We can now trace a path starting from a vertex, by taking the first edge on its incidence list, look up its other endpoint in the edge table, finding the incidence list of this new vertex etc. This saves the time of scanning the row of the matrix, looking for a one. However, the saving is real only if n is large and the graph is sparse, for instead of using one bit per edge, we now use edge names and auxiliary pointers necessary in our data structure. Clearly, the space required is $O(|E| + |V|)$, i.e., bounded by a constant times $|E| + |V|$. Here we assume that the basic word length of our computer is large enough to encode all edges and vertices. If this assumption is false then the space required is $O((|E| + |V|) \log (|E| + |V|))$*.

In practice, most graphs are sparse. Namely, the ratio $(|E| + |V|)/|V|^2$ tends to zero as the size of the graphs increases. Therefore, we shall prefer the use of incidence lists to that of adjacency matrices.

---

*The base of the log is unimportant (clearly greater than one), since this estimate is only up to a constant multiplier.

The reader can find more about data structures and their uses in graph theoretic algorithms in references 1 and 2.

1.3 EULER GRAPHS

An *Euler path* of a finite undirected graph $G(V, E)$ is a path $e_1, e_2, \ldots, e_l$ such that every edge appears on it exactly once; thus, $l = |E|$. An undirected graph which has an Euler path is called an *Euler graph.*

Theorem 1.1: A finite (undirected) connected finite 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 1.3 is an Euler graph, because both have four vertices of odd degree. The graph shown in Figure 1.3(a) is the famous *Königsberg bridge problem* solved by Euler in 1736. The graph shown in Figure 1.3(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 which 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 a Euler circuit, then all vertices are of even degree.

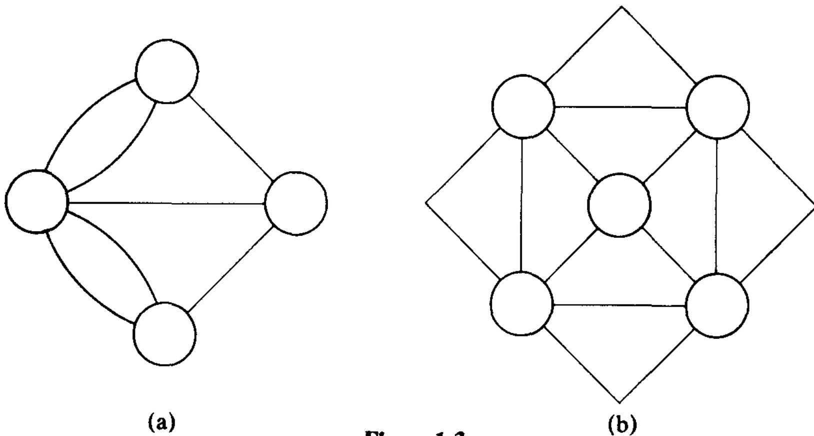


Figure 1.3

Assume now that G is a finite graph with exactly two vertices of odd degree, a and b. We shall described now an algorithm for finding a Euler path from a to b. Starting from a we choose any edge adjacent to it (an edge of which a is an endpoint) and trace it (go to its other endpoint). Upon entering a vertex we search for an unused incident edge. If the vertex is neither a nor b, each time we pass through it we use up two of its incident edges. The degree of the vertex is even. Thus, the number of unused incident edges after leaving it is even. (Here again, a self-loop is counted twice.) Therefore, upon entering it there is at least one unused incident edge to leave by. Also, by a similar argument, whenever we reenter a we have an unused edge to leave by. It follows that the only place this process can stop is in b. So far we have found a path which starts in a, ends in b, and the number of unused edges incident to any vertex is even. Since the graph is connected, there must be at least one unused edge which is incident to one of the vertices on the existing path from a to b. Starting a trail from this vertex on unused edges, the only vertex in which this process can end (because no continuation can be found) is the vertex in which it started. Thus, we have found a circuit of edges which were not used before, and in which each edge is used at most once: it starts and ends in a vertex visited in the previous path. It is easy to change our path from a to b to include this detour. We continue to add such detours to our path as long as not all edges are in it.

The case of all vertices of even degrees is similar. The only difference is that we start the initial tour at any vertex, and this tour must stop at the same vertex. This initial circuit is amended as before, until all edges are included.

Q.E.D.

In the case of digraphs, a *directed Euler path* is a directed path in which every edge appears exactly once. A *directed Euler circuit* is defined similarly. Also a *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 shown in Figure 1.4(a) is shown in Figure 1.4(b).

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

1. 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 all other vertices v, $d_{out}(v) = d_{in}(v)$.
2. For all vertices v, $d_{out}(v) = d_{in}(v)$.

If 1 holds then every directed Euler path starts in a and ends in b. If 2 holds then every directed Euler path is a directed Euler circuit.

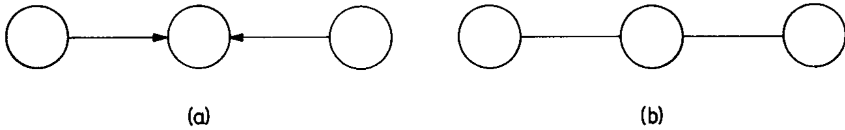


Figure 1.4

The proof of the theorem is along the same lines as the proof of Theorem 1.1, and will not be repeated here.

Let us make now a few comments about the complexity of the algorithm for finding an Euler path, as described in the proof of Theorem 1.1. Our purposed is to show that the time complexity of the algorithm is $O(|E|)$; namely, there exists a constant K such that the time it takes to find an Euler path is bounded by $K \cdot |E|$.

In the implementation, we use the following data structures:

1. Incidence lists which describe the graph.
2. A doubly-linked list P describing the path. Initially this list is empty.
3. A vertex table, specifying for each vertex v the following data:
   (a) A mark which tells whether v appears already on the path. Initially all vertices are marked "unvisited".
   (b) A pointer $N(v)$, to the next edge on the incidence list, which is the first not to have been traced from v before. Initially $N(v)$ points to the first edge on v's incidence list.
   (c) A pointer $E(v)$ to an edge on the path which has been traced from v. Initially $E(v)$ is "undefined".
4. An edge table which specified for each edge its two endpoints and whether it has been used. Initially, all edges are marked "unused".
5. A list L of vertices all of which have been visited. Each vertex enters this list at most once.

First let us describe a subroutine TRACE(d, P), where d is a vertex and P is a doubly linked list, initially empty, for storage of a traced path. The tracing starts in d and ends when the path, stored in P, cannot be extended.

TRACE(d, P):

(1) $v \leftarrow d$
(2) If v is "unvisited", put it in L and mark it "visited".
(3) If $N(v)$ is "used" but is not last on v's incidence list then have $N(v)$ point to the next edge and repeat (3).
(4) If $N(v)$ is "used" and it is the last edge on v's incidence list then stop.

(5) $e \leftarrow N(v)$
(6) Add e to the end of P.
(7) If $E(v)$ is "undefined" then $E(v)$ is made to point to the occurrence of e in P.
(8) Mark e "used".
(9) Use the edge table to find the other endpoint u of e.
(10) $v \leftarrow u$ and go to (2).

The algorithm is now as follows:

(1) $d \leftarrow a$
(2) TRACE(d, P). [Comment: The subroutine finds a path from a to b.]
(3) If L is empty, stop.
(4) Let u be in L. Remove u from L.
(5) Start a new doubly linked list of edges, P', which is initially empty. [Comment: P' is to contain the detour from u.]
(6) TRACE(u, P')
(7) Incorporate P' into P at $E(u)$. [Comment: This joins the path and the detour into one, possibly longer path. (The detour may be empty.) Since the edge $E(u)$ starts from u, the detour is incorporated in a correct place.]
(8) Go to (3).

It is not hard to see that both the time and space complexity of this algorithm is $O(|E|)$.

1.4 DE BRUIJN SEQUENCES

Let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ be an alphabet of σ letters. Clearly there are σ^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 i such that

$$a_i a_{i+1} \cdots a_{i+n-1} = w,$$

where the computation of the indices is modulo L. Clearly if the sequence satisfies this condition, the $L = \sigma^n$. The most important case is that of $\sigma = 2$.

*Sometimes they are called *maximum-length shift register sequences.*

Binary de Bruijn sequences are of great importance in coding theory and are implemented by shift registers. (See Golomb's book [3] on the subject.) The interested reader can find more information on de Bruijn sequences in references 4 and 5. The only problem we shall discuss here is the existence of de Bruijn sequences for every $\sigma \geq 2$ and every n.

Let us describe a digraph $G_{\sigma,n}(V, E)$ which has the following structure:

1. V is the set of all σ^{n-1} words of length $n - 1$ over Σ.
2. E is the set of all σ^n words of length n over Σ.
3. The edge $b_1b_2 \cdots b_n$ starts at vertex $b_1b_2 \cdots b_{n-1}$ and ends at vertex $b_2b_3 \cdots b_n$.

The graphs $G_{2,3}$, $G_{2,4}$, and $G_{3,2}$ are shown in Figures 1.5, 1.6 and 1.7 respectively.

These graphs are sometimes called de Bruijn diagrams, or Good's diagrams, or shift register state diagrams. The structure of the graphs is such that the word w_2 can follow the word w_1 in a de Bruijn sequence only if the edge w_2 starts at the vertex in which w_1 ends. Also it is clear that if we find a directed Euler circuit (a directed circuit which uses each of the graph's edges exactly once) of $G_{\sigma,n}$, then we also have a de Bruijn sequence. For example, consider the directed Euler circuit of $G_{2,3}$ (Figure 1.5) consisting of the following sequence of edges:

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

The implied de Bruijn sequence, 00011101, follows by reading the first letter of each word in the circuit. Thus, the question of existence of de Bruijn sequences is equivalent to that of the existence of direct Euler circuits in the corresponding de Bruijn diagram.

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

Proof: We wish to use Theorem 1.2 to prove our theorem. First we have to show that the underlying undirected graph is connected. In fact, we shall show that $G_{\sigma,n}$ is strongly connected. Let $b_1b_2 \cdots b_{n-1}$ and $c_1c_2 \cdots c_{n-1}$ be any two vertices; the directed path $b_1b_2 \cdots b_{n-1}c_1$, $b_2b_3 \cdots b_{n-1}c_1c_2$, ..., $b_{n-1}c_1c_2 \cdots c_{n-1}$ leads from the first to the second. Next, we have to show that $d_0(v) = d_1(v)$ for each vertex v. The vertex $b_1b_2 \cdots b_{n-1}$ is entered by

试读结束：需要全本请在线购买： www.ertongbook.com