EATCS
Monographs on
Theoretical Computer Science
W. Brauer G. Rozenberg A. Salomaa (Ed.)

KURT MEHLHORN

Data Structures and Algorithms 2:

Graph Algorithms and NP-Completeness

Springer-Verlag
Berlin Heidelberg New York Tokyo

Kurt Mehlhorn

Data Structures and Algorithms 2:

Graph Algorithms and NP-Completeness

With 54 Figures

Springer-Verlag
Berlin Heidelberg New York Tokyo 1984

Editors
Prof. Dr. Wilfried Brauer
FB Informatik der Universität
Rothenbaum-Chausee 67–69, 2000 Hamburg 13, Germany

Prof. Dr. Grzegorz Rozenberg
Institut of Applied Mathematics and Computer Science
University of Leiden, Wassenaarseweg 80, P.O. Box 9512
2300 RA Leiden, The Netherlands

Prof. Dr. Arto Salomaa
Department of Mathematics, University of Turku
20500 Turku 50, Finland

Author
Prof. Dr. Kurt Mehlhorn
FB 10, Angewandte Mathematik und Informatik
Universität des Saarlandes, 6600 Saarbrücken, Germany

With the permission of B. G. Teubner publishers, Stuttgart, arranged, solely authorized and revised English translation of the original German edition: Effiziente Allgorithmen (1977)

ISBN 3-540-13641-X Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-13641-X Springer-Verlag New York Heidelberg Berlin Tokyo

Offsetprinting: Beltz Offsetdruck, Hemsbach/Bergstrasse
Binding: J. Schäffer OHG, Grünstadt
2145/3140-543210

For Ena, Uli, Steffi, and Tim

Preface to Volume 2

The design and analysis of data structures and computer algorithms has gained considerable importance in recent years: The concept of "algorithm" is central in computer science and "efficiency" is central in the world of money.

This book treats graph algorithms and the theory of NP-completeness and comprises chapters IV to VI of the three volume series "Data Structures and Efficient Algorithms". The material covered in this book derives its importance from the universal role played by graphs in many areas of computer science. The other two volumes treat sorting and searching (chapters I to III) and multi-dimensional searching and computational geometry (chapters VII to VIII). All three volumes are organized according to problem areas. In addition, we have included a chapter (chapter IX) in all three volumes which gives a paradigm oriented view of the entire series and orders the material according to algorithmic methods.

In chapter IV we deal with algorithms on graphs. We start out with a discussion of various methods for representing a graph in a computer, and of simple algorithms for topological sorting and the transitive closure problem. The concept of a random graph is also introduced in these sections. We then turn to methods for graph exploration which we later refine to depth first and breadth first search. Depth first search is the basis for connectivity, biconnectivity and planarity algorithms for undirected graphs, and for an algorithm for strong connectivity of directed graphs. In the section on planar graphs we also present the planar separator theorem as well as a shortest path algorithm for planar graphs. Breadth first search is the basis for efficient least cost path algorithms and for network flow algorithms. Several algorithms for unweighted and weighted network flow and their application to matching and connectivity problems are discussed in detail. Finally, there is a section on minimum spanning trees.

Chapter V explores the algebraic interpretation of path problems on graphs. The concept of a path problem over a closed semi-ring is defined, a general solution is presented, and the connection with matrix multiplication is established. Then fast algorithms for matrix multiplication over rings are discussed, transformed to boolean matrix multiplication, and their implication for special path problems is investigated. Finally, a lower bound on the monotone complexity of boolean matrix multiplication is derived.

Chapter VI covers the theory of NP-completeness. Many efficient algorithms have been found in the past; nevertheless, a large number of problems have not yielded to the attack of algorithm designers. A particularly important class of such problems is the class of NP-complete problems. In the first half of the chapter this class is defined, and many well-known problems are shown to belong to the class. In the second half of the chapter we discuss methods for solving NP-complete problems. We first treat branch-and-bound and dynamic programming and then turn to approximation algorithms. The chapter closes with a short discussion of other complexity classes.

The book covers advanced material and leads the reader to very recent results and current research. It is intended for a reader who has some knowledge in algorithm design and analysis. The reader must be familiar with the fundamental data structures such as queues, stacks, and linked list structures. This material is covered in chapter I of volume 1 and also in many other books on computer science. Knowledge of the material allows the reader to appreciate most of the book. For some sections more advanced knowledge is required. Priority queues and balanced trees are used in sections IV.7, IV.8 and IV.9.1, algorithms for the union – find problem are used in IV.8, and bucket sort is employed at several places. Information about these problems can be found in volume 1 but also in many other books about algorithms and data structures.

The organization of the book is quite simple. There are three chapters which are numbered using roman numerals. Sections and subsections of chapters are numbered using arabic numerals. Within each section, theorems and lemmas are numbered consecutively. Cross references are made by giving the identifier of the section (or subsection) and the number of the theorem. The common prefix of the identifiers of origin and destination of a cross reference may be suppressed, i. e., a cross reference to section VII.1.2 in section VII.2 can be made by either referring to section VII.1.2 or to section 1.2.

Each chapter has an extensive list of exercises and a section on bibliographic remarks. The exercises are of varying degrees of difficulty. In many cases hints are given, or a reference is provided in the section on bibliographic remarks.

Most parts of this book were used as course notes either by myself or by my colleagues N. Blum, Th. Lengauer, and A. Tsakalidis. Their comments were a big help. I also want to thank H. Alt, O. Fries, St. Hertel, B. Schmidt, and K. Simon who collaborated with me on several sections and I want to thank the many students who helped to improve the presentation by their criticism. Discussions with many colleagues helped to shape my ideas: B. Becker, J. Berstel, B. Commentz-Walter, H. Edelsbrunner, B. Eisenbarth, Ph. Flajolet, M. Fontet, G. Gonnet, R. Güttler, G. Hotz, S. Huddleston, I. Munro, J. Nievergelt, Th. Ottmann, M. Overmars, M. Paterson, F. Preparata, A. Rozenberg, M. Stadel, R. E. Tarjan, J. van Leeuwen, D. Wood, and N. Ziviani.

The drawings and the proof reading were done by my student Hans Rohnert. He did a fantastic job. Of course, all remaining errors are my sole responsibility. Thanks to him, there should not be too many left. The typescript was prepared by Christel Korten-Michels, Martina Horn, Marianne Weis and Doris Schindler under sometimes hectic conditions. I thank them all.

Saarbrücken, April 1984 Kurt Mehlhorn

Contents Vol. 2: Graph Algorithms and NP-Completeness

IV. Algorithms on Graphs

In this chapter we treat efficient algorithms for many basic problems on graphs: topological sorting and transitive closure, connectivity and biconnectivity, least cost paths, least cost spanning trees, network flow problems and matching problems and planarity testing. Most of these algorithms require methods for the systematic exploration of a graph. We will introduce such a method in section 4 and then specialize it to breadth first and depth first search.

IV. 1. Graphs and their Representation in a Computer

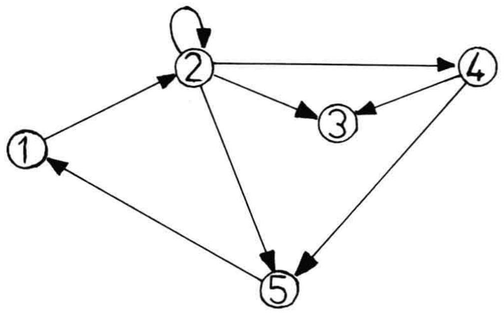
A directed graph $G = (V,E)$ consists of a set $V = \{1,2,\ldots,|V|\}$ of nodes and a set $E \subseteq V \times V$ of edges. A pair $(v,w) \in E$ is called an edge from v to w. Throughout this chapter we set $n = |V|$ and $e = |E|$.
Two methods for storing a graph are customary.

a) Adjacency matrix: A graph $G = (V,E)$ is represented by a $|V| \times |V|$ boolean matrix $A_G = (a_{ij})_{1\le i,j\le n}$ with

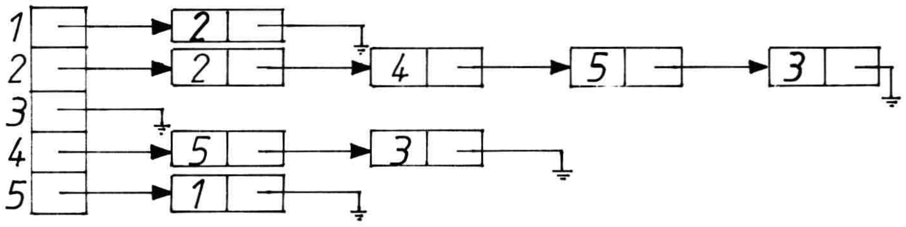
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

The storage requirement of this representation is clearly $\Theta(n^2)$.

b) Adjacency lists: A graph $G = (V,E)$ is represented by n linear lists. The i-th list contains all nodes j with $(i,j) \in E$. The headers of the n lists are stored in an array. The storage requirement of this representation is $O(n + e)$. The lists are not necessarily in sorted order.



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$



The above example shows a graph, and its representation by adjacency matrix and adjacency lists.

Since $0 \le e \le n^2$ we conclude that the adjacency list represention is often much smaller than the adjacency matrix representation and never much larger. Since most graphs which come up in applications are sparse, i.e. $e \ll n^2$, this is an important point to remember. Even more important is the fact that the choice of the representation can have a drastic influence on the time complexity of graph algorithms. We will see in this chapter that many graph problems can be solved in linear time $O(n + e)$ if the adjacency list representation is used. However, any algorithm using the matrix representation must have running time $\Omega(n^2)$. For this reason we will always use the adjacency lists except when explicitely stated otherwise (chapter V).

In more detail, the adjacency list representation is based on the following declarations:

```
    type node = record name : [1 ..n];
                            :
                            next : ↑ node
                end
```

and

```
    ADLHEAD :   array [1 ..n] of ↑ node
```

Array ADLHEAD contains the heads of the adjacency lists. The elements of the adjacency lists are of type node, each element representing an edge. In some cases these elements will contain additional information, e.g. the length of an edge, a pointer to the reverse edge in an undirected graph, ...

We need some more definitions. Let $G = (V,E)$ be a digraph. A path from v to w, $v, w \in V$, is a sequence $v_0, v_1, \ldots, v_k$ of nodes such that $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$ for $0 \le i < k$; k is the length of the path. Note that there is always the path of length zero from v to v. A path is simple if $v_i \neq v_j$ for $0 \le i < j < k$. A cycle is a path from v to v. If, in addition, the path is simple then the cycle is simple. A graph is acyclic if it contains no non-trivial cycle. Let $T \subseteq E$. We write $v \underset{T}{\rightarrow^*} w$ iff there is a path from v to w using only edges in T.

The indegree of a node v is the number od edges ending in v, $\text{indeg}_G(v) = |\{w; (w,v) \in E\}|$. Similarly, the outdegree of v is the number of edges starting in v, $\text{outdeg}_G(v) = |\{w; (v,w) \in E\}|$.

A digraph $G' = (V',E')$ is a subgraph of $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$. If $G = (V,E)$ is a digraph and $V' \subseteq V$ then the subgraph induced by V' is $(V', E \cap (V' x V'))$. $G - V'$ denotes the subgraph induced by $V - V'$. If $V' = \{v\}$ is a singleton then we write $G - v$ instead of $G - \{v\}$.

A digraph $A = (V,T)$ is a directed forest if A is acyclic and $\text{indeg}_A(v) \le 1$ for all $v \in V$. A node v with $\text{indeg}_A(v) = 0$ is called a root of the forest. Note that a directed forest has at least one root. If $A = (V,T)$ is a directed tree then $|T| = |V| - 1$. Also, there is a unique path from the root r to any node v of a directed tree. Finally, if v is any node of a directed tree then the subtee A_v rooted at v is the subgraph induced by the descendants of v, i.e. A_v is the subgraph induced by $\{w; v \underset{T}{\rightarrow^*} w\}$.

Let $G = (V,E)$ be a digraph. A directed forest $A = (V,T)$ with $T \subseteq E$ is called a spanning forest of G. If A is a tree then it is called a spanning tree of G.

An undirected graph (or simply graph) is a digraph $G = (V,E)$ with a symmetric relation E, i.e. $(v,w) \in E$ iff $(w,v) \in E$. In a graph the indegree of a node is always equal to its outdegree and is simply called the degree of the node. An undirected graph is called acyclic if it contains no simple cycles of length at least three (Note that an "undirected" edge between v and w always gives rise to a simple cycle, namely v,w,v). An acyclic undirected graph is called an undirected forest.

IV. 2. Topological Sorting and the Representation Problem

A topological sort of a digraph $G = (V,E)$ is a mapping $\text{ord}: V \to \{1,\ldots,n\}$ such that for all edges $(v,w) \in E$ we have $\text{ord}(v) < \text{ord}(w)$. Clearly, if a graph G has a topological sort then G is acyclic. The converse is also true and is easily proved by induction on the number of nodes. So suppose, $G = (V,E)$ is acyclic. If $n = |V| = 1$ then G has a topological sort. If $n > 1$ then G must have a node v with indegree 0. (Such a node can be found by starting at an arbitrary node w and walking back edges. Since the graph is acyclic no node is entered twice in this process, and hence the process terminates. It terminates in a node with indegree 0). Deleting v leaves us with an acyclic graph G' with one less node. G' has a topological sort and so does G.

Actually, the argument given above, describes an algorithm for computing the mapping ord.

```
(1)  $G_{current} \leftarrow G$; COUNT $\leftarrow$ 0;
(2)  while $G_{current}$ has at least one node with no predecessor
(3)  do   let v be a node with no predecessor;
(4)       COUNT     $\leftarrow$ COUNT + 1;
(5)       ORD[v]    $\leftarrow$ COUNT;
(6)       $G_{current} \leftarrow G_{current} - v$
(7)  od;
(8)  if   $G_{current}$ is nonempty
(9)  then cyclic else acyclic fi
```

The correctness of this algorithm is immediate from the preceding discussion. With respect to complexity the crucial lines are lines (3) and (6). How do we find a node with indegree 0 efficiently in line (3)? A brute force approach would be a complete search of graph $G_{current}$. Since such a search would take time at least $\Omega(n)$ the entire algorithm would be $\Omega(n^2)$ at best.

A better approach is to look at the interdependence of lines (3) and (6). In line (6) node v and all edges leaving v are deleted. Exactly the indegrees of the other endpoints are changed. This suggests to use an array INDEG[1 .. n] to store the current indegree of all nodes. Array INDEG is updated in line (6). In line (3) we need to know one node with indegree O; the indegree of a node can only become zero in line (6) and it is easy to detect that fact there. It is therefore wise to keep all nodes with indegree O in $G_{current}$ in a set ZEROINDEG.

The following refinement of our algorithm makes use of the variables INDEG: <u>array</u> [1 .. n] <u>of</u> <u>integer</u> and ZEROINDEG: subset of V. The graph $G_{current}$ is not stored explicitely. Rather it is the subgraph of G induced by the nodes which have not received a number ord yet. ZEROINDEG contains the points of zero indegree in $G_{current}$ and INDEG contains the indegree of all nodes in $G_{current}$. Initially $G_{current}$ = G and so INDEG should be initialized to the indegrees in G. This can be done efficiently by traversing all adjacency lists.

Algorithm: Topological sort

```
(1.1)  COUNT ← O;
(1.2)  ZEROINDEG ← ∅; for all i ∈ V do INDEG[i] ← O od;
(1.3)  for all i ∈ V
(1.4)  do for all j ∈ V with (i,j) ∈ E
(1.5)     do INDEG[j] ← INDEG[j] + 1
(1.6)     od
(1.7)  od;
(1.8)  for all i ∈ V
(1.9)  do if INDEG[i] = O then add i to ZEROINDEG fi
(1.1o) od;
(2)    while ZEROINDEG ≠ ∅
(3.1)  do let v be any node in ZEROINDEG;
(3.2)     delete v from ZEROINDEG;
(4)       COUNT ← COUNT + 1;
(5)       ORD[v] ← COUNT;
(6.1)     for all w ∈ V with (v,w) ∈ E
(6.2)     do INDEG[w] ← INDEG[w] - 1;
(6.3)        if INDEG[w] = O
```

```
(6.4)       then add w to ZEROINDEG fi;
(6.5)    od
(7)   od ;
(8)   if COUNT < n
(9)   then Halt ("graph is cyclic") else Halt ("graph is acyclic") fi
```

It remains to specify an implementation for set ZEROINDEG. On this set the following operations are performed: Insertion, deletion of an unspecified element, and test for emptiness. In chapter I we saw that implementing ZEROINDEG by a stack or by a queue will allow us to execute each one of these operations in time O(1). We prefer the stack for its simplicity and higher efficiency, so ZEROINDEG is a stack of elements of V (stack of [1.. n]).

Finally, we have to explain lines (1.4) and (6.1) in more detail. They are realized by stepping through the adjacency list corresponding to nodes i and w respectively and take time proportional to the outdegree of those nodes. A detailed program for lines (1.4) and (1.5) is given by (p is of type ↑node):

```
p ← ADJHEAD[i];
while p ≠ nil
do j ← p↑.name;
   INDEG[j] ← INDEG[j] + 1 ;
   p ← p↑.next
od
```

We are now in a position to determine the performance of our algorithm for topological sorting. Line (1) takes time O(1), lines (1.2) and lines (1.8) - (1.10) take O(n). Execution of (1.4) and (1.5) for a fixed i takes time $O(\text{outdeg}_G(i))$ and hence lines (1.3) - (1.7) take time O(n + e). Altogether, initialization takes time O(n + e).

The main loop is executed O(n) times and hence the total time spent in lines (3.1), (3.2), (4) and (5) is O(n). For a fixed v, lines (6.1) - (6.4) take time $O(\text{outdeg}_G(v))$. Since every node v is deleted from ZEROINDEG at most once total running time of that loop is O(n + e). This shows that the running time of the entire algorithm is O(n + e).

Theorem 1: A topological sort of digraph $G = (V,E)$ can be computed in linear time $O(n + e)$.

Proof: By the discussion above. □

Next we will show that getting the graph as a matrix will doom any algorithm to inefficiency.

Theorem 2: Any algorithm for topological sorting which gets the digraph as an adjacency matrix has running time $\Omega(n^2)$.

Proof: Consider the behaviour of any such algorithm on the empty graph, i.e. on the all zero matrix. Suppose there is a pair i,j of nodes, $i \neq j$, such that the algorithm neither inspects a_{ij} nor a_{ji}. Then we could change both entries to one and the algorithm would still return with a topological sort. However, the graph is cyclic after adding edges (i,j) and (j,i). This shows that the algorithm has to inspect at least half of the entries of the matrix and hence has running time $\Omega(n^2)$. □

We saw that a topological sort of an acyclic graph can be computed in linear time. Given the mapping $\text{ord}: V \to \{1,\ldots,|V|\}$ it is then easy to reorder the adjacency lists in increasing order as follows: Generate all pairs $\{(\text{ord}(v), \text{ord}(w));\ (v,w) \in E\}$ and sort them by bucket sort according to the second component and then according to the first component. This takes time $O(n + e)$ and generates the adjacency lists in sorted order.

IV. 3. Transitive Closure of Acyclic Digraphs

Let $G = (V,E)$ be a digraph. Digraph $G^* = (V,E^*)$ is the reflexive, transitive closure of G if $(v,w) \in E^*$ if and only if there is a path from v to w in G. In this section we present an algorithm for computing the transitive closure of an acyclic digraph; the algorithm is extended to general digraphs in section 6. We will assume that the acyclic digraph is topologically sorted, i.e. $(i,j) \in E$ implies $i < j$ and that the adjacency lists are sorted in increasing order. We saw in the previous section that this can be achieved in linear time $O(n + e)$.

The idea for the algorithm is very simple. We step through the nodes of

G in decreasing order. Suppose that we consider node i. Then for every $j > i$ we have already computed the set of nodes reachable from j, $REACH[j] = \{k;\ j \to^* k\}$. Then

$$REACH[i] = \{i\} \cup (\cup\{REACH[j]\ ;\ (i,j) \in E\})$$

This suggests to step through the nodes j with $(i,j) \in E$ and to compute the union of the sets REACH[j]. We will see that this is a costly process. It can be improved somewhat as follows. We step through nodes j with $(i,j) \in E$ in <u>increasing</u> order. When edge (i,j) is encountered, we will first test whether $j \in$ REACH[i] already. If this is the case then there must be a node $h \neq j$ with $i \to h \to^* j$ and hence $REACH[h] \supset REACH[j]$ and thus we do not have to add REACH[j] to REACH[i]. This observation will lead to considerable savings in many cases. Here is the complete algorithm.

```
(1)  BREACH ← ∅;                      -- BREACH is a bitvektor
(2)  for i from n downto 1
(3)  do REACH[i] ← BREACH ← {i}       -- REACH[i] is a linear list
(4)     for all j with (i,j) ∈ E      -- in increasing order!!
(5)     do if j ∉ BREACH
(6)        then for all k ∈ REACH[j]
(7)                do if k ∉ BREACH
(8)                   then add k to BREACH and REACH[i]
(9)                   fi
(1o)               od
(11)       fi
(12)    od;
(13)    for all k ∈ REACH[i]
(14)    do delete k from BREACH
(15)    od
(16) od
```

There is one subtle point about this algorithm, the two faces of set REACH[i]. REACH[i] is kept as a linear list and as a bitvektor BREACH. We initialize both of them to {i} in time O(1) in line (3). Note that BREACH is empty prior to the first execution of the loop and that this is ensured for later executions by lines (13) to (15). In lines (4)-(12) we step through the direct descendants of i in increasing order. Remember that the adjacency lists are sorted. The tests in line (5) and (7) take time O(1) since BREACH