ON OPTIMAL AND NEAR-OPTIMAL ALGORITHMS FOR SOME COMPUTATIONAL GRAPH PROBLEMS

A Thesis
submitted to the
University of Newcastle upon Tyne
for the degree of
Doctor of Philosophy

Jayme Luiz Szwarcfiter

June 1975

E R R A T A

page 3	line -5:	for "appears" read "appear"
page 4	line -13:	for "paths" read "path"
page 4	line -12:	for "not" read "no"
page 54	line 23:	for "stroing" read "storing"
page 58	line 9:	for the last letter read "B_i"
page 68	line 11:	"later" should be read after "stack"
page 76	line 1:	for "which" read "that"
page 76	line -7:	delete the commas between "when reached"
page 78	line 2:	delete "the"
page 78	line 3:	for "actual Johnson's" read "Johnson's actual"
page 79	line 11:	for "chosen" read "chosen,"
page 79	line 12:	for "module" read "modulo"
page 82	line 7:	for "store," read "store"
page 106	line 3:	for "is" read "in"
page 191	line 35-37:	this reference should read:

"[SzLa74] J.L. Szwarcfiter and Peter E. Lauer, Finding the Elementary Cycles of a Directed Graph in O(N+M) per Cycle, Tech. Rep. 60, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, 1974."

<u>ABSTRACT</u>

Some computational graph problems are considered in this thesis and algorithms for solving these problems are described in detail. The problems can be divided into three main classes, namely, problems involving partially ordered sets, finding cycles in graphs, and shortest path problems. Most of the algorithms are based on recursive procedures using depth-first search. The efficiency of each algorithm is derived and it can be concluded that the majority of the proposed algorithms are either optimal and near-optimal within a constant factor. The efficiency of the algorithms is measured by the time and space requirements for their implementation.

ACKNOWLEDGEMENTS

I am grateful to Mr. L.B. Wilson for his helpful supervision, encouragement and for his careful reading and valuable comments on the manuscripts of this thesis. To Dr. C.R. Snow for his valuable comments on the manuscripts of this thesis and related problems and for his helpful supervision during part of my research. To Professors E.S. Page and B. Randell for their encouragement and general guidance.

I wish to thank Mrs. I. Compson and Miss J. Lennox for the typing of this thesis.

I gratefully acknowledge the Conselho Nacional de Pesquisas, Brazil and Universidade Federal do Rio de Janeiro, Brazil for the financial support.

To

Regina, Claudio and Lila

CONTENTS

INTRODUCTION AND BASIC DEFINITIONS

Introduction

Computational graph theory is an area of research which is related to both computing science and mathematical graph theory. We can roughly characterize it as being the branch of computing science concerned with solving graph theoretic problems. This characterization explicitly has considered computational graph theory as part of computing science. However, it could be argued that computational graph theory should be considered as an area of graph theory concerned with finding algorithmic solutions for graph theoretic problems. Examining these two characterizations, we observe that a basic difference exists between them, namely the former mentions implicitly the use of the computer for solving the graph problems, whereas the latter is more concerned with problems of existence. This difference is fundamental. The time and space constraints imposed by the use of the computer dictate the general strategy to be adopted for the derivation of the solutions to the graph problems. A "pure graph theoretician", for instance, could perhaps demonstrate little interest in the depth-first search of a graph [Ta72]. On the other hand, some fundamental graph theoretic theorems as Kuratowski's planarity criterion [Ku30], have so far been of no relevant interest for solving graph problems with a computer. The implications of the use of the computer are reflected not only in the type of approach to be adopted for solving the problems, but even in the selection of the actual problems of study. A pure graph theoretician for instance is not likely to be attracted by problems as "find the elementary cycles of a graph".

We mention the following quotation by Corneil [Co74] in the section "History of the analysis of graph theoretical algorithms".

"In the late 1800's and early 1900's the interest in graph theory was blossoming. The criterion for evaluating the algorithms designed in this period was whether or not they worked. Since the algorithms were designed for use 'by hand' very little consideration was given to the timing requirements; naturally, no consideration was given to storage requirements. With the advent of electronic computers, programmers were forced to look for effective algorithms to solve their graph theoretical problems. Due to the cost and lack of computer storage on early computers the main evaluation criterion was the storage requirement of the algorithms. As storage became cheaper and mor readily available, the timing of an algorithm became increasingly important."

In this thesis we are concerned with the computational solutions of certain graph problems. In most cases, our primary preoccupation is the efficiency of the solutions, i.e. the time and space required for a computer implementation of the proposed methods. We employ backtracking, or depth-first search, as a basic tool for solving most of the considered problems. Backtracking has been commonly used and described as an important strategy for solving some computational problems (Knuth [Kn75], Golomb and Baumert [GoBa65]). It is perhaps an older idea, but its full importance in solving computational graph theoretical problems was not completely realized until the beginning of this decade, when Hopcroft and Tarjan principally, initiated its extensive use in many different graph algorithms ([Ta72], [Ta73], [Ta74], [Ta74a], [Ta74b], [HoTa73] , [HoTa74], [HoTa73a], among others). Clearly, the use of backtracking for solving graph problems started before the 70's (Roberts and Flores [RoFl66], for instance). However its widespread use for computational graph problems has been during the last three or four years.

The following are the problems that have been considered in this thesis:

In chapter 1 we have examined the relationship between a certain class of ternary trees (ternary search trees), and topological sorting, given a partially ordered set.
Ternary search trees and ternary sequence search trees have been defined as natural extensions of the binary case. Topological sorting by ternary tree insertion has also been considered as a natural generalization of binary tree (complete) sorting. On the other hand, although the problem of quasi-topological sorting has been regarded as a generalization of the topological sorting problem, it has been shown to be soluble using ternary trees. A possible meaning of topological searching is also presented.

Chapter 2 considers the problem of generating the complete set of solutions of the topological sorting problem, given a partially ordered set or given an acyclic digraph. A backtracking algorithm has been presented which enumerates the topological sorting arrangements, in a time at most propqrtional to the size of the digraph per arrangement. Most of the contents of this chapter describe the results which appears in [KnSz74].

Some cycle problems are considered in chapter 3. We first approach the problem of finding the elementary cycles of a directed graph. The most successful extant algorithms enumerating the elementary cycles of a directed graph are known to be based on a backtracking strategy. Such existing algorithms are discussed and a backtracking algorithm is proposed, whose time is at most proportional to the size of the digraph, per cycle enumerated. This part of section 3 appears in [SzLa75]. Next we examine the problem of generating a fundamental set of cycles of an undirected graph. This problem has been shown to be simpler than the former and in fact a backtracking algorithm has been presented, which generates such a set in a time linear in the size of the graph. However, the explicit output of the generated cycles requires a time at most proportional to the product of the number of vertices and edges of the graph. The problem of enumerating the elementary cycles of an undirected graph is considered next. The strategy adopted was to modify the algorithm for obtaining the elementary cycles of a directed graph, so as to operate for undirected graphs. The modifications which were introduced did not alter the overall time bound.

Chapter 4 examines some different shortest paths problems in acyclic digraphs. If a digraph has not cycles then it is possible in some cases, to take advantage of this fact and present particular algorithms that are more efficient than corresponding strategies supposed to operate in digraphs with cycles. In some cases the difference in efficiency is substantial. For example, an algorithm has been presented in this chapter, for finding the shortest path between two given vertices of an acyclic digraph visiting a given subset of vertices, which requires a time linear in the size of the graph. Known algorithms for solving the same problem for general (not necessarily acyclic) digraphs have a time bound exponential in the number of vertices of the graph. The following algorithms for acyclic digraphs, have been presented in this chapter: finding the shortest path between two given vertices; from all vertices to a fixed vertex; from a fixed vertex to all others; between all pairs of vertices; finding the shortest path between two given vertices, visiting a specified subset of vertices; finding the k-shortest paths between two given vertices; from all vertices to a fixed vertex; from a fixed vertex to all others; between all pairs of vertices; finding the longest path of the digraph; finding the k-longest paths of the digraph. The majority of these algorithms are based on backtracking procedures.

The extension of the k-shortest paths algorithms, for handling digraphs in which cycles are allowed, is the subject of chapter 5. Unlike some other shortest paths problems, we found the k-shortest paths algorithms for general digraphs to be nearly as efficient as the k-shortest paths algorithms for acyclic digraphs. This result however applies only for the part of the algorithm for finding the second, third, ..., k-th shortest path. Methods for finding the <u>shortest</u> path in general digraphs are known to be less efficient than corresponding algorithms for acyclic digraphs.

A solution to a problem related to Dilworth's decomposition theorem for partially ordered sets is presented in the appendix. The problem consists of given a partially ordered set, obtain a minimal covering by disjoint chains, from maximal antichains. The appendix contains an example of a derivation of a recursive algorithm from a mathematical proof by induction, in opposition to derivations of proofs of correctness by induction from recursive algorithms, which appear in some other parts of the thesis.

The algorithms presented in this thesis have been described in a structured go-to-less ALGOL-like formulation, following Dijkstra in [DaDiHo72]. Nearly all of them are based on recursive procedures.

Practical computer experiments have been carried out using ALGOLW ([Si71], [** 72]) and MTS, in operation at the Computing Laboratory, University of Newcastle upon Tyne.

Basic Definitions

Next, we present the basic definitions which are relevant to the contents of this thesis. The graph nomenclature that we have used was mostly taken from Harary [Ha69] and Harary, Norman and Cartwright [HaNoCa65].

A graph (V,E) is a finite non-empty set V, together with a set E of pairs of distinct elements of V. The elements of V and E are the vertices and edges, respectively of the graph. We denote by N and M respectively the number of vertices and edges of a graph. A directed graph (digraph) D(V, E) is a graph in which the edges are ordered pairs. An undirected graph G(V, E) is a graph in which the edges are unordered pairs. We denote an edge e by the pair of vertices (v, w) that forms it. At most one edge (v, w) may exist, for two given vertices v and w. Given an edge e = (v, w), v and w are adjacent and e is incident to both v and w. The degree of a vertex v is the number or vertices which are adjacent to v.

In a digraph, an edge (v, w) is said to be from v to w; the indegree and outdegree of a vertex v are the number of edges to and from v respectively. We denote them by indegree(v) and outdegree(v) respectively. A source vertex is a vertex v with indegree(v) = 0, while a sink vertex v has outdegree(v) = 0.

A sequence of vertices $v_1, v_2, \ldots, v_k$ such that for every i $1 \leq i < k$ we have $(v_i, v_{i+1}) \in E$, is called a path from v_1 to v_k.

The length of the path $v_1, v_2, \ldots, v_k$ is defined as $k - 1$. Vertex v_1 is said to reach v_k. A trivial path is composed by a sole vertex. A path is elementary if it contains no vertex twice.

A weighted graph is a graph in which there is associated a finite weight d_{vw} to each of its edges (v, w). Given a path $v_1, v_2, \ldots, v_k$ in a weighted graph, we define its weighted path length as the sum of the weights of the edges which form the path. By convention, the weighted path length of a trivial path is zero and if there is no path from vertex v to w, we say that the weighted path length from v to w is equal to infinity. When dealing with weighted graphs, we may use the terminology "path length", as referring to "weighted path length".

A cycle is a path $v_1, v_2, \ldots, v_k$ with $v_k = v_1$ and containing at least two different edges. A cycle $v_1, v_2, \ldots, v_{k-1}, v_k$ is elementary if $v_1, v_2, \ldots, v_{k-1}$ is an elementary path. If a graph has no cycles it is called acyclic. Two elementary cycles involving exactly the same edges are considered to be identical.

A graph (V', E') is a partial subgraph of a given graph (V, E) if $V' \subseteq V$ and $E' \subseteq E$. If additionally for any $v, w \in V'$, $(v, w) \in E$ implies $(v, w) \in E'$, the graph (V', E') is called a subgraph of (V, E). A partial subgraph (V', E') of a graph (V, E) is a spanning partial subgraph of (V, E), if $V = V'$.

An undirected graph is connected if there is a path between every two vertices of the graph; otherwise it is disconnected. A graph with no edges is totally disconnected. The maximal connected subgraphs of an undirected graph are its connected components. A digraph is strongly connected if for every (v,w) $v,w \in V$, there is a path from v to w. The maximal strongly connected partial subdigraphs of a digraph are called its _strongly connected components_.

A _complete graph_ is a graph which has a maximum number of edges, for the given set of vertices. A _complete acyclic digraph_ is an acyclic digraph in which the addition of any new edge, between two of its vertices, creates a cycle.

A _tree_ is a connected undirected graph with no cycles. A set of disjoint trees is called a _forest_. A _directed rooted tree_ or simply a _rooted tree_ is an acyclic digraph in which exactly one vertex, the _root_, has indegree zero, whilst every other vertex has indegree one. If there is a path from vertex v to w, in a directed rooted tree, then v is an _ancestor_ of w, and w is a _descendant_ of v. A _subtree_ (_rooted subtree_) of a tree (rooted tree) T is a partial subgraph of T, which is itself a tree (rooted tree).

A _spanning tree_ of an undirected graph G is a spanning partial subgraph of G which is a tree. It follows that an undirected graph G has a spanning tree if and only if it is connected, otherwise the set of spanning trees of its connected components defines a _spanning forest_ of G.

Given an elementary cycle c, we can represent it as a vector $(e_1, e_2, \ldots, e_M)$, with $e_i = 1$ if edge i belongs to the cycle and $e_i = 0$, otherwise. The cycles of an undirected graph generate a vector space called _cycle vector space_, with addition of cycles c_1 and c_2 defined as the ring sum (or boolean addition) of c_1 and c_2, under the representation above. The ring sum of c_1 and c_2 may produce either another cycle or an edge disjoint union of cycles. A _fundamental set of cycles_, corresponding to a spanning forest F of an undirected graph G, is a maximal set of elementary cycles such that each cycle of the set contains exactly one edge of G, which does not belong to F. If the graph has K connected components, then a fundamental set of cycles has precisely M-N+K cycles. This number of cycles is called the cycle rank of the graph. This set of cycles is a basis for the cycle vector space of the graph.

The following are some matrices related to a graph (V, E). The adjacency matrix is a N x N matrix, such that each element a_{ij} is defined as $a_{ij} = 1$ if $(v_i, v_j) \in E$ and $a_{ij} = 0$, otherwise. The reachability matrix is a N x N matrix, where each element a_{ij} is such that $a_{ij} = 1$ if vertex v_i reaches vertex v_j and $a_{ij} = 0$ otherwise. For an undirected graph, the incidence matrix is a N x M matrix, with each element a_{ij} defined as $a_{ij} = 1$ if edge e_j is incident to vertex v_i and $a_{ij} = 0$, otherwise.

A binary (ternary) tree T is recursively defined as a finite set of elements called vertices, that is either empty or consists of a single vertex called the root, together with two (three) disjoint binary (ternary) trees, called left and right (left, central and right) subtrees of the root respectively (see [Kn68] for the definition of t-ary trees). A vertex is a terminal vertex if all its subtrees are empty. We denote by root (T) the vertex which is the root of T and by L(x), C(x) and R(x) respectively the left, central and right subtrees of vertex x of a ternary tree. A path from a vertex x_1 to a vertex x_k, is a sequence of distinct vertices $x_1, x_2, \ldots, x_k$ such that either x_{i+1} is the root of a subtree of x_i, or x_i is the root of a subtree of x_{i+1}, $1 \leq i < k$. A path with k vertices is said to be of length k - 1. The level of a vertex x is the length of the path from x to the root. A binary (ternary) tree is balanced when (i) No non-terminal vertex has any empty subtree and (ii) all terminal vertices have the same level. A binary (ternary) tree is complete when the deletion of the vertices with maximal level produces a balanced binary (ternary) tree. The binary (ternary) tree consisting of a single vertex is also complete, by convention. The (internal) path length of T is the sum of the levels of all vertices of T.

A partially ordered set (poset) $(S, \preccurlyeq)$ is a set S together with a binary relation $\preccurlyeq$ on S, which satisfies the following properties for any elements $x, y, z \in S$: (i) reflexivity: $x \preccurlyeq x$; (ii) anti-symmetry: $x \preccurlyeq y$ and $y \preccurlyeq x$ implies $x = y$; and (iii) transitivity: $x \preccurlyeq y$ and $y \preccurlyeq z$ implies $x \preccurlyeq z$. S is said to be partially ordered by $\preccurlyeq$ and the relation itself is called a partial ordering on S. The relation $\prec$ defined by $x \prec y$ iff $x \preccurlyeq y$ and $x \neq y$ for every $x, y \in S$, satisfies the following properties for $x, y, z \in S$: (i) irreflexivity: $x \not\prec x$; (ii) asymmetry: $x \prec y$ implies $y \not\prec x$; and (iii) transitivity: $x \prec y$ and $y \prec z$ implies $x \prec z$. The relation $\prec$ defines similarly a poset $(S, \prec)$, which can also be characterized by the relation $\succ$ defined by $x \succ y$ iff $y \prec x$, for every $x, y \in S$. We use the following notation [Kn74a]: $x \| y$ when $x \not\prec y \not\prec x$, where x and y are distinct elements of S. The elements x and y are called independent when $x \| y$. If S is finite and non-empty - and we always assume so - then a poset $(S, \prec)$ can be represented by an inclusion diagram, in which there is a directed line from x to y iff $x \prec y$ and there exists no z, such that $x \prec z \prec y$, for all $x, y, z \in S$ [MaBi67]. It follows from this definition that an inclusion diagram is an acyclic digraph. A source is an element $x \in S$ such that there is no $y \in S$, with $y \prec x$. A sink is an element $x \in S$ such that there is no $y \in S$, with $x \prec y$. A chain is a subset of S, in which any two elements are related by $\prec$. An antichain is a subset of S, in which any two elements are independent.

The problem of topological sorting is to obtain a permutation $x_1 x_2 \dots x_N$ of S, such that for every x_i, x_j if $x_i < x_j$ then $i < j$. Such a permutation is called a topological sorting arrangement of $(S, <)$. Clearly, the solution of the topological sorting problem is not unique. In fact, for a given poset $(S, <)$ there is at least one topological sorting arrangement and at most N!, depending on the relation $<$ being maximal or minimal, with regard to its number of elements, respectively.

Let D(S, E) be an acyclic digraph. Define the digraph $D_t(S,<)$ by: $(x, y) \in <$ iff y is reachable from x in D and $x \neq y$, for all $x, y \in S$. D_t is called the transitive closure digraph of D. It follows that D_t is a poset since $<$ satisfies the required conditions. Also any spanning partial subdigraph D' of D_t such that the reachability of D_t is preserved in D', can "represent" the poset $(S, <)$. In particular the inclusion diagram of the poset $(S, <)$ corresponds to the minimal subdigraph of D_t, which can represent the poset. Observe also that in terms of graphs, the topological sorting problem is equivalent to the problem of finding an appropriate ordering of the vertices of an acyclic digraph, such that all the edges are oriented in the same way, from left to right, for instance, when drawing the digraph with the vertices represented by points and the edges by directed lines.

There are many different ways of representing a graph in a computer. For example, either the adjacency or incidence matrices can be used for storing a graph defined as a matrix. Another usual and in general convenient form of representing a graph inside a computer, consists of storing it as a set of adjacency lists A, with one list A(v) per vertex v of the graph. The members of A(v) are the vertices w such that $(v, w) \in E$. If there is no w such that

$(v, w) \in E$, then list $A(v)$ is empty. For some other graph representations see [Be73], [We71] for instance.

The performances of the algorithms proposed in this thesis have been evaluated in terms of expressions in O-notation for the time and space requirements of the algorithms. Assume that f is a function defined for the discrete variables $n_1, n_2, \ldots, n_p$. The notation $O(f)$ means that there exists a positive constant C, such that the number m represented by $O(f)$, satisfies $|m| \leq C|f(n_1, \ldots, n_p)|$ (see [Kn68]).

Finally, we mention that we have assumed, throughout the thesis, that the set V of vertices of a graph is $V = \{1, 2, \ldots, N\}$, unless otherwise stated. The symbols $\cup$, $\cap$, $\subseteq$ and $\setminus$ have their usual meaning of set union, intersection, inclusion and difference respectively.

CHAPTER 1

TERNARY TREES AND TOPOLOGICAL SORTING

1.1 Introduction

In this chapter we describe certain properties of ternary trees, related to partially ordered and quasi-ordered sets. Ternary search trees are defined and topological sorting is considered to be an extension of the usual sorting, similarly ternary search trees are extensions of the binary case. A particular case of searching - topological searching - is also presented, as an example. These operations are performed on (finite) partially ordered sets, or acyclic directed graphs, and we will use either structure, whichever is more convenient to our particular purpose.

Section 1.2 defines ternary search trees and shows how they may be related to partially ordered sets. Ternary sequence search trees are also defined and considered to be natural extensions of binary sequence search trees. These trees suggest, naturally, the idea of topological sorting and searching. However, one of our conclusions is that practical implementations using these sorting and searching methods should be restricted to a particular class of problems (those whose structure provides an easy and quick way for finding the type of relationship between any two elements). For such problems, this method may be efficient although our aim is not to present a method of sorting, but to point out some properties of ternary trees, when related to partially ordered or quasi-ordered sets. Section 1.3 presents this topological sorting with ternary trees. One interesting aspect of this method is that it works in a similar way, with respect to input and output, to the usual sorting, i.e. the "sort mechanism" converts an input permutation into a sorted output. The meaning of topological searching is described in 1.4. Finally, section 1.5 presents the case of quasi-topological sorting and some further general remarks are found in section 1.6.

1.2 Ternary search trees

Let $(S, <)$ be a poset. A ternary search tree associated with $(S, <)$ or simply a ternary search tree is a ternary tree T whose vertices are the elements of S, and such that:

$y \in L(x)$ implies $y \prec x$,

$y \in C(x)$ implies $y \| x$,

$y \in R(x)$ implies $y \succ x$,

for all $x, y \in S$, where $L(x)$, $C(x)$ and $R(x)$ denote, respectively, the left, central and right subtrees of x.

As an example, consider the poset of figure 1.1. It is represented by an adjacency matrix (m_{ij}), with $m_{ij} = 1$ if $x_i < x_j$ and otherwise 0, for all $x_i, x_j \in S$. The ternary tree, shown in figure 1.4, is a ternary search tree associated with this poset. Figures 1.2 and 1.3 illustrate two different digraphs that represent the poset (the digraph of figure 1.3 is the minimal digraph that represents it). Therefore, the ternary search tree of figure 1.4 is associated with any of the structures of figures 1.1, 1.2 or 1.3.

If follows from the definition above, that a given poset does not uniquely determine a ternary search tree associated with it. Nor does a given ternary search tree uniquely determine a poset with which the ternary tree is associated. The ternary tree of figure 1.5, distinct from that of figure 1.4, is another ternary search tree associated with the poset of figure 1.1. On the other hand, the poset

	A	B	C	D	E	F	G
A	0	0	1	1	0	1	1
B	1	0	1	1	0	1	1
C	0	0	0	1	0	0	0
D	0	0	0	0	0	1	0
E	0	0	0	1	0	1	0
F	0	0	0	0	0	0	0
G	0	0	0	0	0	1	0

Figure 1.1

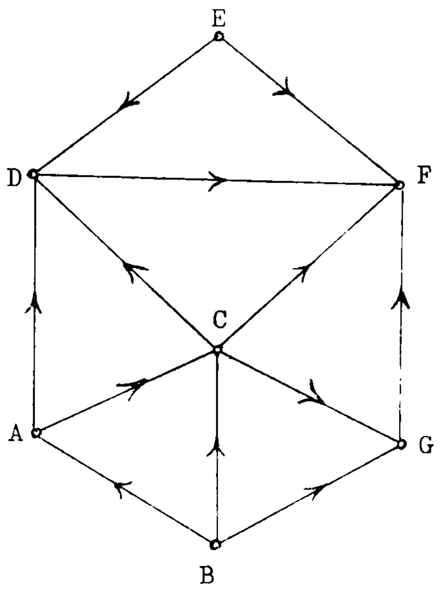


Figure 1.2

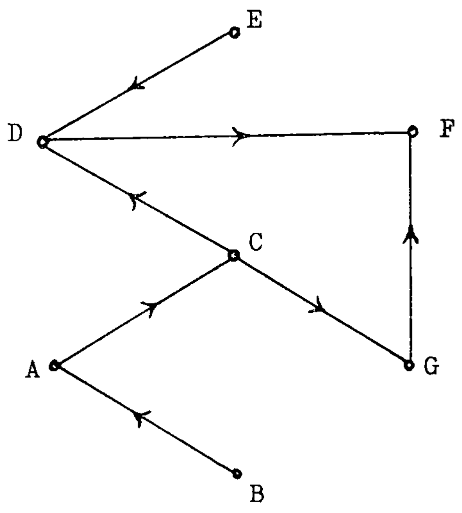


Figure 1.3

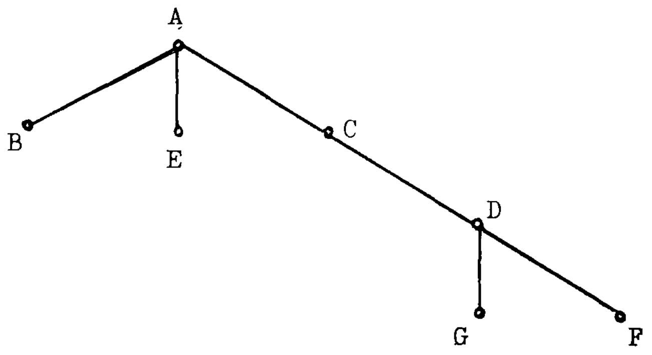
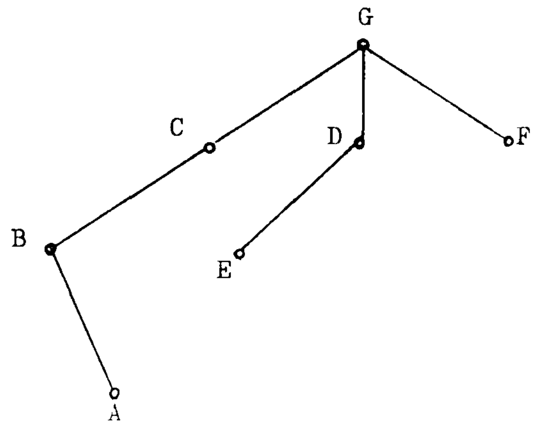


Figure 1.4

Figure 1.5



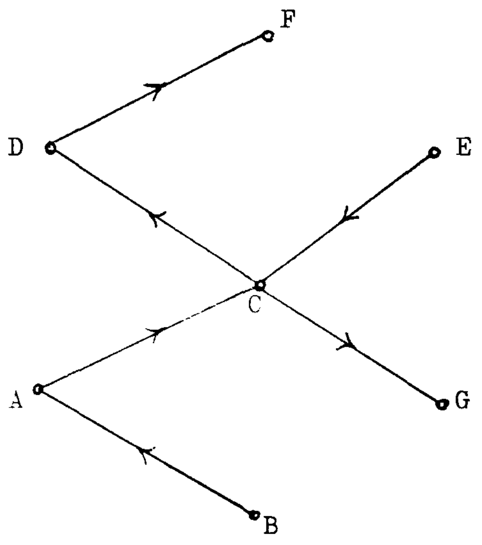


Figure 1.6

represented by the digraph of figure 1.6 is distinct from that of figure 1.1, and the ternary tree of figure 1.4 is also associated with it. However, the ternary search tree of figure 1.5 is <u>not</u> associated with the poset given by the digraph of figure 1.6, since in the ternary tree, vertex $E \in C(G)$ and in the poset $E \prec G$.

Given a ternary search tree T associated with a poset $(S, \prec_1)$, an element $y \notin S$ and a relation $\prec_2$ between $\{y\}$ and S, such that $(S \cup \{y\}, \prec)$, with $\prec = \prec_1 \cup \prec_2$, is still a poset, we can construct a ternary search tree T', associated with the latter poset, simply by properly inserting a new vertex in the ternary tree T. In fact, given T, y and $\prec_2$, the ternary search tree T' is uniquely determined. The basic idea for obtaining it is similar to that used for inserting a new vertex y in a binary search tree [Hi62]. We find a path from the root of T, to a vertex z of the ternary tree, such that if x is a vertex of this path, then the vertex following x in the path is at the left, centre or right of x, in T according to whether $y \prec x$, $y \| x$ or $y \succ x$, respectively. If the vertex so defined - which should follow x in the path - does not exist because the corresponding subtree is empty, then $x = z$ and y is inserted in T, in the place of that empty subtree.

Algorithm 1.1 follows the above strategy. It uses a recursive procedure INSERT which finds that appropriate path. It is assumed that the vertices of the ternary tree T are stored in a list, with one 4-field node in the list for each vertex of T. If p is the address in the list, of a vertex x of T, then <u>info</u>(p) = x and <u>left</u>(p), <u>central</u>(p), and <u>right</u>(p) are the addresses of the root of the left, central and right subtrees of x, respectively. If a certain subtree of x is empty, then its address is <u>null</u>. It is also assumed that a list pool contains the available memory space, from which the space for a newly created vertex is to be taken.

The following is a formulation of this algorithm. The symbols $\prec$, $||$ and $\succ$, correspond to the relation between $\{y\}$ and the elements of the poset with which the ternary search tree T is associated. Clearly, this relation is assumed to be known.

ALGORITHM 1.1:

```
begin comment an algorithm for inserting a new vertex in a ternary
              search tree;
   procedure  INSERT (pointer p, integer y);
   begin      if p = null then
              begin p := address of an available space memory, from pool;
                    left(p) := central(p) := right(p) := null;
                    info(p) := y
              end
              else if $y < info(p)$ then INSERT (left(p), y)
                   else if y||info(p) then INSERT(central(p), y)
                        else INSERT (right(p), y)
                             comment since $y \neq info(p)$, this last condition
                                     corresponds to $y > info(p)$;
   end INSERT;
   pointer q;
   integer y;
   read the ternary search tree T;
   read the element y to be inserted in T;
   q := address of the root of T;
   INSERT (q, y)
end
```

As a further remark about algorithm 1.1, observe that the computation of each invoked call of the procedure INSERT can be performed in, at most, a constant number of steps (excluding the computations of the recursive calls that may occur in it), if the poset is given in a suitable representation - its adjacency matrix, for instance. Also observe that if z is the vertex of T having that empty subtree in which y was inserted, the total number of calls of procedure INSERT equals one plus the number of vertices of T in the path from the root to vertex z.

Now, consider a poset (S, $<$) and a permutation $x_1 x_2 \ldots x_N$ of S. Construct a ternary search tree associated with S, by first choosing x_1 as its root and afterwards, iteratively inserting x_2, x_3, $\ldots$, x_N in the ternary search tree obtained in the previous iteration. Since the insertion of a new vertex, in a ternary search tree, is an operation that produces an unique new ternary search tree, we conclude that the final such tree - obtained after the insertion of x_N - is uniquely characterized by the poset and the permutation. We call it the ternary sequence search tree associated with the poset (S, $<$) and the permutation $x_1 x_2 \ldots x_N$ of S - briefly TSST. Clearly this idea constitutes an extension of Hibbard's concept of binary sequence search tree [Hi62].

The construction of a ternary sequence search tree is implemented by algorithm 1.2, which iteratively invokes the procedure INSERT defined in algorithm 1.1.

ALGORITHM 1.2

```
begin comment an algorithm for constructing a ternary sequence search tree:
    procedure INSERT (pointer p;  integer y);
    begin  ...
           ...     as in algorithm 1.1
           ....
    end INSERT;
    pointer q;
    integer y;
    read the poset (S, $\prec$);
    read the permutation $x_1 x_2 \ldots x_N$;
    INSERT (null, $x_1$);
    q := address of vertex $x_1$ in the ternary search tree;
    for j := 2 step 1 until N do INSERT (q, $x_j$)
end
```

As an example, the ternary search tree of figure 1.4 is the ternary sequence search tree associated with the poset of figure 1.1 and the permutation ABECDGF.

As has been mentioned, a poset (S, $\prec$) and a permutation p of S uniquely determine the ternary search tree T associated with them. However, the same poset (S, $\prec$) and another permutation p' $\neq$ p of S may have as their associated TSST a ternary tree T' such that T' = T. For instance, the poset of figure 1.1 and the permutation ACDGFEB are also associated with the TSST of figure 1.4. The problem that we pose now is to calculate the total number of such permutations that correspond to the same TSST. In fact, the following problems are equivalent:

i) Find the number α of distinct permutations p of S, $|S| = N$, which together with a poset (S, $\prec$) determine the same ternary sequence search tree.

ii) Find the number α of distinct permutations p, of a set of N numbers, which correspond to the same binary sequence search tree T, according to the usual construction of T, starting from p. Such a construction has been given by Hibbard [Hi62], Knuth [Kn73], Page and Wilson [PaWi73], Harrison [Ha73], among others.

iii) Find the number α of ways to label the N vertices of a binary tree T, with the labels of $\{1, \ldots, N\}$, such that the label of each vertex is less than that belonging to any subtree of this vertex.

For any of these 3 above problems, the value of α can be calculated by the following expression, which appears in [Kn73], as the answer to the problem (iii):

$$\alpha = \frac{N!}{\pi\ |T(x)|} ,$$

where T(x) denotes the ternary (binary) subtree whose root is x, and $|T(x)|$ is its number of vertices. The basic reason why this formula solves those problems is that : if x is a vertex in the ternary (binary) tree T, and y is another vertex belonging to one of the subtrees of x, then x must necessarily precede y in any of the permutations p. Hence, the total number of permutations p such that, in these permutations x precedes every vertex belonging to any of its subtrees, is $N!/|T(x)|$. By considering all vertices of the ternary (binary) tree, we obtain the above formula for α.

There are, for example $7!/7.4.3.1.1.1.1 = 60$ permutations of $\{A,B,C,D,E,F,G\}$ which together with the poset of figure 1.1 determine the ternary search tree of figure 1.4. There are $6!/6.3.2.1.1.1 = 20$ permutations of $\{1,2,\ldots,6\}$ which correspond to the binary search tree of figure 1.7. Similarly, there are 20 ways to label the binary tree of figure 1.7, with the labels $\{1, \ldots, 6\}$, such that the label of any vertex is less than that belonging to any of its subtrees.

Once the shape of the ternary search tree is established by the poset (S, $\prec$) and a permutation p of S, the value α obtained by the above formula is calculated disregarding the poset with which the search tree is associated. This suggests that an algorithm for finding the complete set of permutations p of S, which - together with a given digraph - are associated to the same TSST, does not need to manipulate the digraph at all (see section 1.4, for a further comment on this property).

Even when a digraph is disconnected, each associated TSST is still well defined. In the extreme case, when the digraph is totally disconnected, any TSST has L(x) = R(x) = empty, for all vertices of x. For example, the ternary search tree of figure 1.9 is the TSST

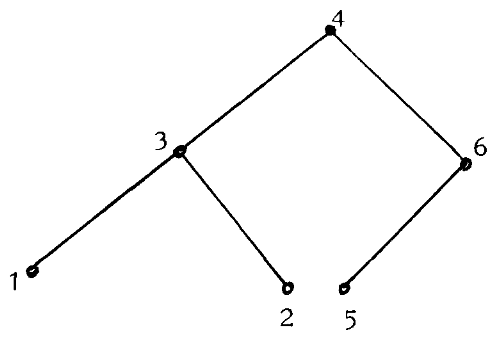


Figure 1.7

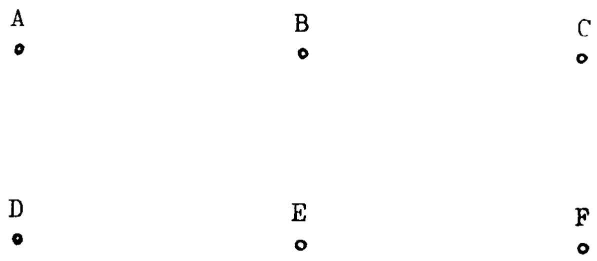


Figure 1.8

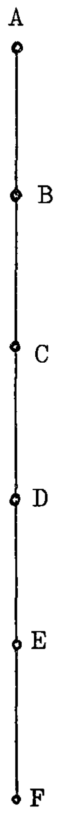


Figure 1.9

associated with the digraph of figure 1.8 and with the permutation ABCDEF.

1.3 Topological searching

The ternary search tree suggests a method for searching for items in a data base whose structure could be represented by an acyclic digraph. Consider such a case, and let T be a ternary search tree associated with that acyclic digraph (S, E), which represents the poset $(S, \prec)$. To search for a node x, for instance, we "compare" initially, x with root (T). If $x \neq \text{root}(T)$ then according to which of $x \prec \text{root}(T)$, $x \| \text{root}(T)$ or $x \succ \text{root}(T)$ is satisfied, the way L(root(T)), C(root(T)) or R(root(T)) respectively is chosen. Afterwards x is compared with the root of the chosen subtree, and so on. Observe that the term "compare", in this context, could mean the computation of a function like $f\colon S \times S \to \{0, 1, 2, 3\}$, with

$$
\begin{aligned}
x_1 &= x_2 \text{ implies } f(x_1, x_2) = 0\\
x_1 &\prec x_2 \text{ implies } f(x_1, x_2) = 1\\
x_1 &\| x_2 \text{ implies } f(x_1, x_2) = 2\\
x_1 &\succ x_2 \text{ implies } f(x_1, x_2) = 3, \quad \text{for all } x_1,\ x_2 \in S.
\end{aligned}
$$

Clearly, this method of searching has practical interest only if f could be easily and efficiently computed. By analogy with the usual terminology, we call it <u>topological searching</u>.

Now suppose we have an acyclic digraph D(S, E), $S = \{x_1, \ldots, x_n\}$, in which we want to search for an element of S and let us examine some basic differences, between a binary and topological searching. The binary case, is well known: obtain a one-to-one mapping $g\colon S \to R$, with R a subset of the reals, and construct an optimal binary search tree, with vertices $g(x_1), \ldots, g(x_n)$. The average number of operations to perform this search is $O(\log_2 n)$. For the topological searching, we would consider, instead a ternary search tree associated with D, with path length minimized. Unlike the binary case, a ternary search tree which minimizes path length is not necessarily complete (the elements of S, being considered equiprobably, with respect to searching), but is the "nearest" possible to a complete one, which still maintains its association with the digraph. For this reason, the average number of operations to perform a topological searching depends also upon the structure of D. Let M be the number of elements of the partial ordering represented by the digraph D. If $M = \min = 0$ then the topological searching is equivalent to a linear search, and we have $O(n)$ average operations. If $M = \max = n(n-1)/2$, then the topological searching is equivalent to the binary search. However, there is an optimal interval for M, in which the average number of operations is minimum, being $O(\log_3 n)$. As an example, consider the case where the elements to be searched for constitute the set S, of the poset $(S, \prec)$, where:

S = set of positive integers, which divide 120

$x \prec y$ if and only if x divides y for all distinct $x, y \in S$.

Figure 1.10 illustrates a digraph that represents this poset. Figure 1.11 presents a classical binary search tree, with minimal path length, in which a binary search would be performed. The vertices of this binary tree are the elements of S, and therefore, the function g, in this case, is the identity function. Figure 1.12 pictures a ternary search tree, associated with the digraph of figure 1.10 with minimal path length, for accomplishing a topological searching. Observe that, in this example, the topological searching would provide a search tree with less path length than the binary searching.

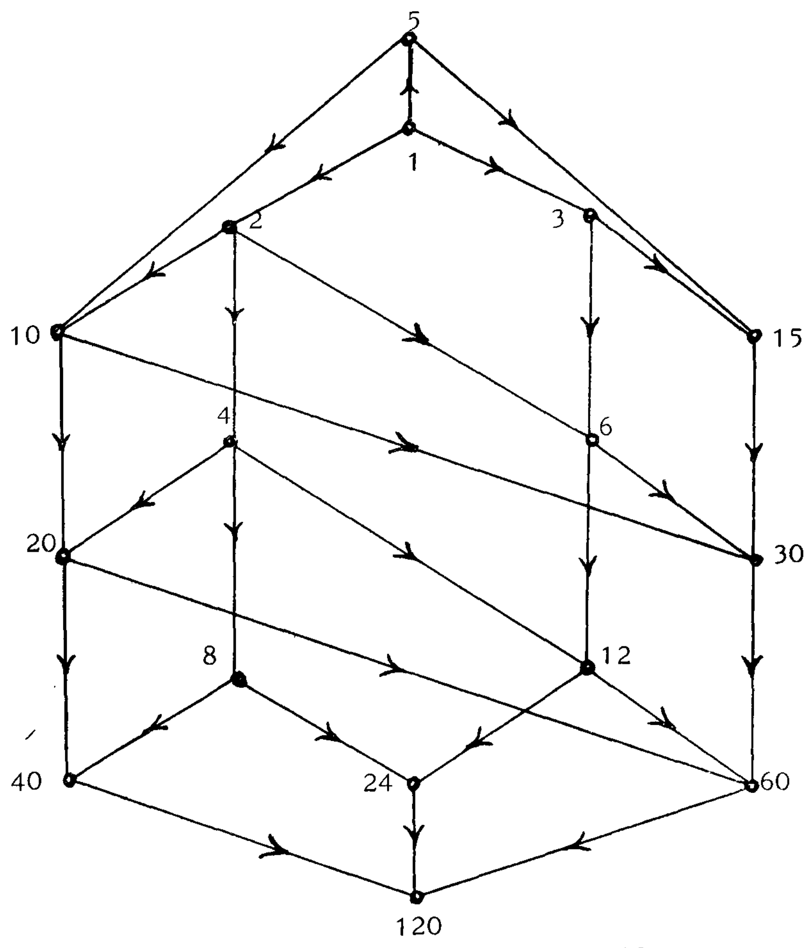


Figure 1.10

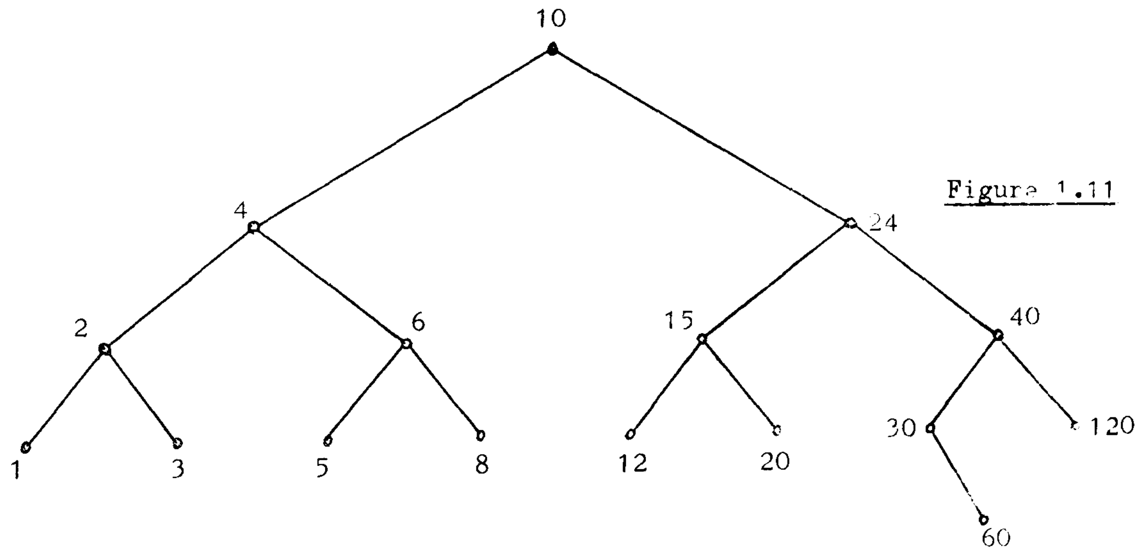


Figure 1.11

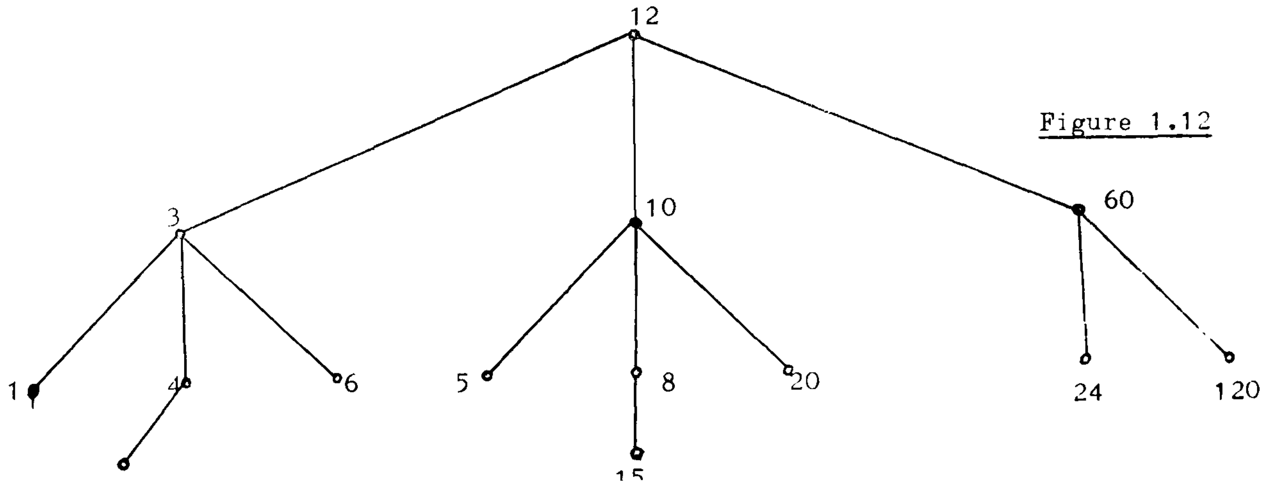


Figure 1.12

1.4 Topological Sorting

Given a ternary tree, we define its A-order traversal. recursively, by: do nothing if the ternary tree is empty; otherwise:

i) Traverse the left subtree in A-order

ii) Visit the root

iii) Traverse the central subtree in A-order

iv) Traverse the right subtree in A-order

Actually this definition constitutes a slight extension of Knuth's definition of symmetric traversal of binary trees [Kn68]. We assert that if T is a ternary search tree, associated with a poset (S, $\prec$), then the A-order traversal of T produces a topological sorting arrangement of (S, $\prec$). A proof of this fact is presented in the next section. A topological sorting arrangement of the poset represented in figure 1.1 obtained by the A-order traversal of the ternary tree of 1.4 is BAECDGF. Observe that items ii) and iii) of the definition, can be swapped, and the new permutation produced by such a traversal is still a topological sorting arrangement. The particular solution of the topological sorting problem obtained with this method, depends on the particular TSST which was used, i.e. depends on the permutation used for building that ternary tree.

If we consider a TSST T corresponding to a poset (S, $\prec$), and we define a relation Δ on S, by x Δ y if and only if y $\in$ T(x), for all x, y $\in$ S, (which is clearly a partial ordering) then the value α, given in section 1.2 corresponds to the total number of distinct topological sorting arrangements of the poset (S, Δ). Also a permutation pof S is a topological sorting arrangement of (S, Δ) if and only if T is the TSST associated with (S, $\prec$) and p. This means that the problem of generating all permutations p of S, which together with the poset $(S, \prec)$ are associated with the same TSST T, is equivalent to the problem of generating all topological sortings of the poset (S, Δ). An algorithm for solving this problem is presented in Chapter 2.

The average number of comparisons required to construct a TSST associated with a given poset $(S, \prec)$ and a random permutation p of S depends on $(S, \prec)$ (note that the average number of comparisons to build a binary sequence search tree for $\{1, 2, \ldots, N\}$ and a random permutation of this set, depends only on N). The worst case for the TSST corresponds, clearly, to a ternary tree having N - 1 vertices with exactly 2 empty subtrees each and 1 terminal vertex (a "zig-zag" ternary tree). In this case the number of comparisons is approximately proportional to N^2. The best case corresponds to a complete ternary tree in which approximately $N\log_3 N$ comparisons are required in average. Since the A-order traversal of a ternary tree can be performed in O(N) steps, these two figures, $O(N^2)$ and $O(N\log_3 N)$ respectively, give us an idea of upper and lower bounds for the timing of a topological sorting algorithm, using this method, if we consider that the computation of a function like f (of section 1.3) can be performed in a constant number of steps (this is true if f were already available as the adjacency matrix of the poset for instance).

1.5 Quasi-topological sorting

Given a set S and a binary relation Q on S which satisfies only reflexivity and transitivity, the pair (S, Q) is called a quasi ordered set. Again we can relate a finite quasi-ordered set (S, Q) to a digraph by a similar construction to that used for posets: the quasi-ordered set (S, Q) can be represented by a digraph D(S, E).

possibly with cycles, where D is a partial subdigraph of the digraph D'(S, Q), such that reachability is preserved. We can also think about a minimal digraph representing the quasi-ordered set, although, unlike the poset case, this minimal digraph is no longer unique. We adopt a notation which is similar to that used for posets:

$x \prec y$ when xQy and $y\not{Q}x$

$x \succ y$ when $x\not{Q}y$ and yQx

$x \| y$ when $x\not{Q}y$ and $y\not{Q}x$

$x \prec\succ y$ when xQy and yQx,

and therefore for any $x, y \in S$ there are exactly five possibilities: $x \prec y$, $x \succ y$, $x\|y$, $x \prec\succ y$ or $x = y$. Observe, however that they are not exclusive, since $x = y$ implies $x \prec\succ y$. Given a quasi-ordered set (S, Q) a permutation $x_1 x_2 \ldots x_N$ of S, is called a <u>quasi-topological sorting</u> arrangement when

$x_i \prec x_j$ implies $i < j$,

for all i, j = 1, ..., N. We define a <u>ternary search tree associated with the quasi-ordered</u> (S, Q) as a ternary tree T such that:

if $y \in L(x)$ implies $y \prec x$

$y \in C(x)$ implies $y\|x$ or $y \prec\succ x$

$y \in R(x)$ implies $y \succ x$,

for all $x, y \in T$, and S being the set of vertices of T. As an example, the ternary search tree of figure 1.14 is a ternary search tree associated with the quasi-ordered set, given by the digraph of figure of 1.13.

The notion of a quasi-ordered set suggests a partition in the set $\mathcal{D}$ of all directed graphs. Consider the digraphs $D_1(V_1, E_1)$, $D_2(V_2, E_2) \in \mathcal{D}$ and define the binary relation $\sim_Q$ by:

$D_1 \sim_Q D_2$ iff $V_1 = V_2$ and

vertex v_1 reaches v_2 in D_1 iff

vertex v_1 reaches vertex v_2 in D_2

for all $v_1, v_2 \in V_1 = V_2$.

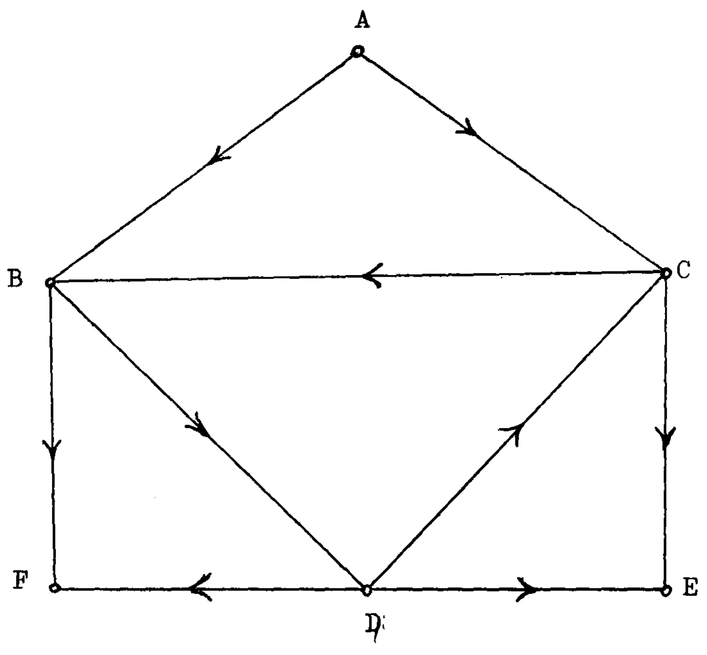


Figure 1.13

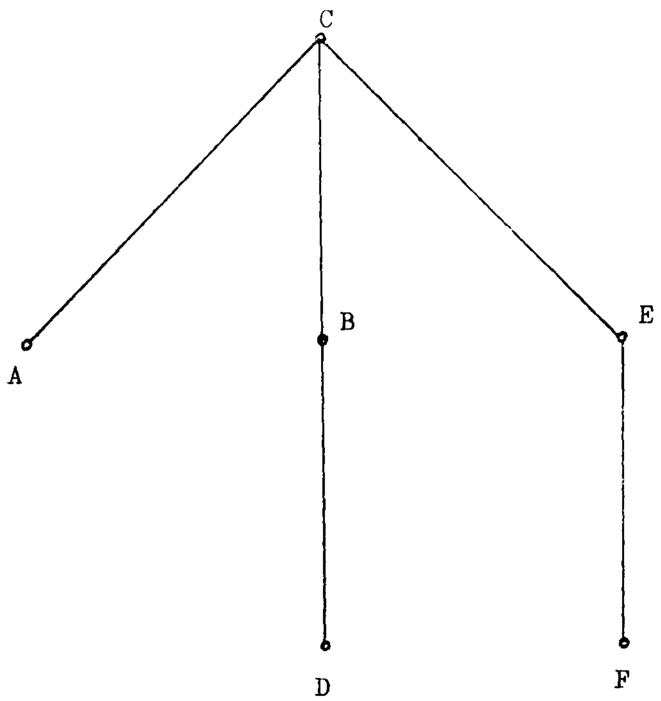


Figure 1.14

It follows that the relation $\sim_Q$ is an equivalence relation and the quotient set $\mathcal{D}/\sim_Q$ is isomorphic to the set of all finite quasi-ordered sets. The digraph with maximal number of edges in each class is a quasi-ordered set and the class itself corresponds precisely to all digraphs that represent this quasi-ordered set. Any minimal digraph in a class is clearly a minimal digraph representation of the corresponding quasi-ordered set. A similar construction can also be defined for posets considering (obviously) the set of all acyclic digraphs, instead of the set $\mathcal{D}$ of all digraphs.

The following theorem generalizes the topological sorting property of ternary trees, presented in the last section.

Theorem 1.1:

If T is a ternary search tree associated with a quasi-ordered set (S, Q) then the A-order traversal of T produces a quasi-topological sorting arrangement of (S, Q).

Proof:

The proof is by induction on the traversal of the subtrees of T. Consider a subtree T' of T, with x = root(T'). If T' is empty then there is no questions about whether or not their vertices are in an appropriate ordering. Otherwise, assume that

$$y_1 \dots y_r, \qquad z_1 \dots z_s \text{ and } w_1 \dots w_n$$

are the A-order traversals of L(x), C(x) and R(x), respectively, any of these possibly empty. By the induction hypothesis, each of these sequences satisfies the definition of a quasi-topological sorting, i.e., in the first sequence

$$\text{if } y_i \prec y_j \text{ then } i < j,$$

for all i, j = 1, ..., r. Similarly for the other two sequences.

The A-order traversal of T' would produce a sequence of the form

$$y_1 \ldots y_r x z_1 \ldots z_s w_1 \ldots w_n$$

and the proof consists of showing that any two vertices of it satisfy the definition of quasi-topological sorting. From the definition of a ternary search tree, we conclude that pairs of the form (y_i, x), (x, z_j) or (x, w_k) are in an appropriate relative ordering in the sequence. Now, we should examine the relative ordering between vertices belonging to different subtrees of T':

i) y_i and z_j:

The definition of quasi-topological sorting is not satisfied only if $y_i \succ z_j$. Assume then that $y_i \succ z_j$. Since $y_i \prec x$ it follows that

$x \succ y_i$ and $y_i \succ z_j$ implies $x \succ z_j$,

which contradicts $z_j \in C(x)$.

ii) y_i and w_k:

The definition is not satisfied only when $y_i \succ w_k$. Assume then that $y_i \succ w_k$. Since $w_k \succ x$, it follows

$y_i \succ w_k$ and $w_k \succ x$ implies $y_i \succ x$,

which contradicts $y_i \in L(x)$.

iii) z_j and w_k:

The definition is not satisfied only when $z_j \succ w_k$. But since $w_k \succ x$ it follows that

$z_j \succ w_k$ and $w_k \succ x$ implies $z_j \succ x$,

which contradicts $z_j \in C(x)$.

Observe that the transitivity of $\succ$ follows from the transitivity of Q. Hence the theorem is true for any subtree T' and therefore is true for T = T'.

Clearly, a poset is a particular case of a quasi-ordered set, in which anti-symmetry holds. Thus, we have also shown the topological sorting property of the ternary search tree. A complete (or linear) ordered set is a particular case of a poset in which trichotomy (i.e. any two elements are related) holds. So this also extends ~~the known~~ usual sorting property of binary search trees, with respect to symmetric order traversal. Finally, we mention that the concept of quasi-topological searching could be introduced, as a natural extension of topological searching.

1.6 Conclusions

We have pointed out some properties of ternary trees, relating these structures to quasi-ordered or partially ordered sets, and quasi-topological or topological sorting. Topological sorting is an important operation that we may wish to perform in acyclic digraphs - or partially ordered sets. For example, in some cases a topological sorting is performed as an initial step in an algorithm, for solving a more complex problem. Some algorithms have been devised for obtaining one solution for the topological sorting problem: [Kn68], [Ka63] [La61], [Ka62], among others. However, the premises for applying the topological sorting method, presented in this chapter are slightly different from those other algorithms, since as in usual (complete) sorting the algorithm is given an input sequence and must produce a sorted output sequence from it. Practical applications of the presented methods - for both topological and quasi-topological sorting - should be restricted to those cases for which there exists an easy and efficient way of determining the type of relationship between two given elements of the set.

One aspect that we would emphasize is the close relation which exists between this method for performing topological or quasi-topological sorting and tree insertion (complete) sorting. The latter method can be considered as a particular case of the former - whilst the problems of topological sorting and (complete) sorting are generally considered separately. In fact, the ordering of a set is complete if and only if an associated ternary search tree has all its central subtrees empty. In that case, the ternary search tree is equivalent to the usual binary search tree, the topological sorting becomes the usual sorting, the solution of the sorting problem is unique, the presented method of topological sorting becomes the usual tree insertion sorting, and - what is relevant - the structure of the input/output data and the algorithm work exactly in the same way.

CHAPTER 2

GENERATION OF ALL TOPOLOGICAL SORTINGS

2.1 Introduction

As mentioned in chapter 1, some algorithms are known that obtain one solution for the topological sorting problem. In particular, the algorithm in [Kn68] requires O(N+M) time for producing one topological sorting arrangement, of an acyclic digraph with N vertices and M edges.

The present chapter describes an algorithm which extends that of Knuth, and finds all solutions of the topological sorting problem, for a given acyclic digraph. Most of the ideas explained in this chapter are the result of a joint work with D. E. Knuth, reported in [KnSz74]. In that paper, the algorithm for obtaining all topological sorting arrangements was used as an example of structured programming and a discussion of some techniques for changing recursion into iteration. A more detailed appreciation of the recursion-iteration translation can be found in [Kn74]. Section 2.2 of this chapter presents the algorithm for all topological sorting arrangements, and a description of the method on which it was based. Its correctness and performance constitute sections 2.3 and 2.4, respectively. Further remarks concerning the proposed method are found in section 2.5, and some conclusions form the last section.

2.2 The algorithm

The algorithm in [Kn68], for one topological sorting arrangement, assumes the acyclic digraph D(V, E) to be represented by a set of adjacency lists A. The additional data structures used are a vector count and a queue qlink. For any vertex v $\in$ V, count(v) is initialised with the number of vertices w, such that $(w, v) \in E$. At any stage of the process, count(v) contains the number of above vertices w which were not scheduled for output yet. A vertex v is considered to be scheduled for output, when all vertices w, such that $(w, v) \in E$, have also been scheduled, some time before. The vertices that are at a given moment, awaiting output are kept in the queue qlink. This queue is therefore initialised as containing the vertices with zero indegree in the digraph. Consider, now the exploration of the first vertex v, in qlink. Since there is no vertex w, such that $(w, v) \in E$, v can be output, according to the rules of topological sorting. The algorithm proceeds by erasing all edges from v. This is accomplished simply by decreasing by one, each count(w), for each vertex $w \in A(v)$. Now, if the newly decremented count(w) dropped to zero, this means that all vertices v', such that $(v', w) \in E$, have already been scheduled for output (i.e., all edges to w have already been explored.) Therefore, these vertices w can be transferred to qlink. The vertex that now stands at the front of the queue qlink is afterwards output and explored, and so on, until all vertices have been explored and output, which is given by the condition that qlink becomes empty. The implementation has assigned to the count and qlink structures the same space in memory. This can be done because of the fact that the vertices are not inserted in qlink, whilst $count(v) \neq 0$.

The algorithm [KnSz74] finds all solutions of the topological sorting problem by extending this scheme. A recursive backtracking procedure ALLTOPSORTS(k) is used. This procedure will output all topological sorting arrangements, which begin with a sequence of vertices $v_1 v_2 \ldots v_k$, that has already been output. The count vector is retained from the original algorithm, but the queue qlink is replaced by an output-restricted deque U, where all deletions from U occur at its right, and insertions may occur at either of its ends. At the entry of the computation of ALLTOPSORTS(k), deque U contains precisely those vertices v, for which count(v) = 0. The computation of this procedure may change the contents of deque U, or those of count fields, however both are restored to their entry values, upon exit. The vertices for output are taken from the right of U and assigned to a variable q. The output of the k-th vertex, of a topological sorting sequence is performed by procedure ALLTOPSORTS(k-1), which has depth k. A variable base is used for storing the value of the rightmost vertex of U (when U is non-empty), at the start of each computation of the recursive procedure.

Assume that $v_1v_2 \ldots v_k$ is the current topological sorting subsequence that has already been output, and let us examine the computation of ALLTOPSORTS(k).

(i) If U is empty then there are no more vertices to be output in the present sequence. The depth of this call is therefore equal to N+1. Exit from the procedure occurs.

(ii) If U is not empty and contains $y_1 \ldots y_r$ on entry to ALLTOPSORTS(k), the procedure will set base := y_r and q := y_r. Then it will erase all edges of the form (q, j) by decreasing each count(j) by one for each vertex j $\in A(q)$; if $z_1, \ldots, z_s$ are the values of j whose count drop to zero at this time, U will be changed to $y_1 \ldots y_{r-1} z_1 \ldots z_s$. After outputting y_r and performing ALLTOPSORTS, beginning with subsequence $v_1 \ldots v_k y_r$, the strategy consists of retrieving all edges of the form (q, j), with j $\in$ A(q).

by adding one to each such count (j), and U is changed to $y_1 \ldots y_{r-1}$. The same process occurs again, with $q = y_{r-1}, y_{r-2}, \ldots, y_1$ until, finally all topological sorting arrangements beginning with $v_1 v_2 \ldots v_k$ will have been produced, and U is again $y_1 \ldots y_r$. Exit from ALLTOPSORTS(k) therefore, occurs.

The following is an ALGOL-like formulation of this algorithm. The erasure and retrieval of the edges, which appear in the formulation of the algorithm refer to those operations which were described in the text of this section.

ALGORITHM 2.1 [KnSz74]

```
begin comment an algorithm for all topological sorting arrangements;
      procedure ALLTOPSORTS (integer value k);
      comment this procedure will output all topological sorting arrange-
              ments which begin with a sequence $v_1 \dots v_k$ , that has
              already been output.  Let $R = \{1, \dots, N\} \setminus \{v_1, \dots, v_k\}$
              be the set of all vertices not yet output.  The procedure
              assumes that, for all $y \in R$, the current value of global
              variable count(y) is the number of edges (z, y) for
              $z \in R$, and that there is a linear list U containing
              precisely those elements $y \in R$ such that count(y) = 0;
      begin integer q, base;
            if U not empty then
            begin base := rightmost vertex of U;
                  repeat set q to rightmost vertex of U and delete it
                                                 from U;
                         erase all edges of the form (q, j);
                         output q in column k + 1;
                         if k = N - 1 then start a new output line;
                         ALLTOPSORTS(k+1);
                         retrieve all edges of the form (q, j);
                         insert q at the left of U;
                  until  rightmost vertex of U = base;
            end
      end ALLTOPSORTS;
      read the digraph and construct its adjacency lists A;
      initialise count(v) values;
      comment count(v) = indegree(v), for all vertices v;
      for v := 1 step 1 until N do
            if count(v) = 0 then insert v at the right of U;
      ALLTOPSORTS(0);
end
```

A previous version of the algorithm took base and q from the left of U, while all insertions in U were performed on the right only. This made U an input-restricted deque, so that a two-way linking for the implementation of U was originally needed. Thus, a slight modification of the strategy turned the deque to an output-restricted one with only one-way linking required for its implementation.

2.3 Correctness

The correctness of the above strategy can be shown by the following lemmas:

Consider an acyclic directed graph D(V, E), with N vertices, input to algorithm 2.1:

Lemma 2.1:

If $v_1 \ldots v_k$ is the topological sorting subsequence, already output at the start of ALLTOPSORTS(k), then deque U contains precisely those vertices v, $v \neq v_1, v_2, \ldots, v_k$ such that count(v) = 0, at that moment.

Proof:

Induction on k. If k = 0 the lemma holds trivially, since the initialisation of the process ensures that the vertices v with count(v) = 0 are precisely the vertices that constitute deque U. By the induction hypothesis, if $v_1 \ldots v_{k-1}$ is the topological sorting subsequence already output at the entry of ALLTOPSORTS(k-1), then U contains those vertices v, $v \neq v_1, \ldots, v_{k-1}$ such that count(v) = 0. Suppose that the content of U, at that moment, is $y_1 \ldots y_r$, and assume without loss of generality that $y_r = v_k$. The computation of ALLTOPSORTS(k-1) then sets base := v_k, q := v_k and deletes v_k from U. Afterwards, all edges of the form (v_k, j), for $j \in A(v_k)$, are erased and if $z_1, \ldots, z_s$ are such vertices j whose count dropped to zero, then U is changed to $y_1 \ldots y_{r-1} z_1 \ldots z_s$. In what follows, v_k is output and the call ALLTOPSORTS(k) occurs. At that moment, deque U contains precisely those vertices v, such that $v \neq v_1, \ldots, v_k$ and count(v) = 0.

Lemma 2.2:

If $v_1 \ldots v_k$ is the topological sorting subsequence, already output at the entry of ALLTOPSORTS(k), then at the exit of this computation, all topological sorting arrangements that begin with $v_1 \ldots v_k$ have been output.

Proof:

Induction on decreasing k. If k = N then by lemma 2.1 deque U is necessarily empty, which will produce an immediate exit from ALLTOPSORTS(k). Consequently, the lemma holds trivially, in this case. By the induction hypothesis, if $v_1 \ldots v_k\ v_{k+1}$ is a topological sorting subsequence that has already been output at the entry of ALLTOPSORTS(k + 1), then all topological sorting arrangements starting with $v_1 \ldots v_k\ v_{k+1}$ have been output, at the exit of this computation. Now assume the computation of ALLTOPSORTS(k), with deque U containing $y_1 \ldots y_r$ and $v_1 \ldots v_k$ being the subsequence already output, at the entry of this procedure: the computation sets base := y_r and q := y_r. The erasure of all edges (y_r, j) occurs for all $j \in A(y_r)$. Assume $z_1, \ldots, z_s$ to be those vertices j whose count drop to zero during this erasure. Therefore U is changed to $y_1 \ldots y_{r-1}\ z_1 \ldots z_s$ and y_r is output. The call ALLTOPSORTS(k + 1) occurs and by the induction hypothesis, all topological sorting arrangements, starting with $v_1 \ldots v_k y_r$ have been output, upon exit of this call. Next, the retrieval of the edges

(y_r, j) occurs for all $j \in A(y_r)$, which means that U is changed to $y_1 y_2 \ldots y_{r-1}$. Afterwards, y_r is inserted in the left of U, which becomes $y_r y_1 \ldots y_{r-1}$. The same process occurs again with $q = y_{r-1}$, $y_{r-2}, \ldots, y_1$ which cause calls ALLTOPSORTS(k+1) whose computations output all topological sorting sequences starting with $v_1 \ldots v_k y_{r-1}$, $v_1 \ldots v_k y_{r-2}, \ldots, v_1 \ldots v_k y_1$, respectively. By return of the last of such calls and after retrieving the edges of the form (y_1, j) for all $j \in A(y_1)$, and inserting y_1 at the left of U, the contents of the deque is $y_1 \ldots y_r$, which ensures exit from ALLTOPSORTS(k), since base $= y_r$ is the rightmost vertex of U. Since, as a consequence of lemma 2.1, the vertices of deque U, at the entry of ALLTOPSORTS(k) are precisely those vertices which - by the definition of topological sorting arrangement - can follow vertex v_k in such an arrangement, we conclude that all topological sorting arrangements starting with $v_1 \ldots v_k$ have been output at the exit of this procedure.

2.4 Performance

The performance of the method presented can be evaluated by the following theorem:

Theorem 2.1:

Let D(V, E) be an acyclic digraph of N vertices and M edges, input to algorithm 2.1. Let T by the total number of distinct topological sorting arrangements, of the vertices of D. Then the algorithm requires O(N+M) space and O((N+M)T) time, for the output of all such T arrangements.

Proof:

The space bound follows from the fact that the representation of the digraph by a set of adjacency lists requires O(N+M) space, and the remaining data structures require O(N) space. For the time bound, we observe that the cost of the output of one vertex v, in any topological sorting arrangement corresponds, at most, to the cost of the execution of one iteration of the repeat block - without considering the recursive call of ALLTOPSORTS inside this block, whose cost is charged to the corresponding vertices that are output in its computation. The cost of each such iteration is O((outdegree(v)) and corresponds to the erasure and retrieval operations, since all other computations, inside this block, can be executed in a constant number of steps. Since in each topological sorting arrangement, at most N vertices are output - and they are all distinct - we conclude that O(N+M) time is required, at most, per arrangement. On the other hand, precisely one call of the type ALLTOPSORTS(N) is invoked for each arrangement; these calls find deque U empty and therefore require O(T) time, for the whole process. Since O(N+M) time is spent by the algorithm, outside the recursive procedure, we conclude that the total time bound, for the entire computation, is O((N+M)T).

2.5 Further remarks

As mentioned in section 2.2, the algorithm supposes that the input digraph is represented as a set of adjacency lists. Clearly, two acyclic digraphs that correspond to the same partially ordered set will cause the algorithm to produce identical sets of topological sorting arrangements. However from theorem 2.1 we conclude that the fewer the number of edges of the input digraph, the faster the process is likely to be. Therefore, for a given partially ordered set, the best results are obtained when the input corresponds to the minimal digraph representing the poset.

The ordering in which the topological sorting arrangements are output depends on the ordering of the vertices in the adjacency lists. For instance, if $z_1, \ldots, z_s$ are the vertices whose count dropped to zero during the erasure of the edges of the form (y, j), for $j \in A(y)$ - which occured when the subsequence already output was $v_1 \ldots v_k$ - and assuming that if z_i precedes z_j in the adjacency list A(y) then $i < j$, thus we can conclude that the ordering in which those vertices z_i are inserted in deque U is precisely $z_1, z_2, \ldots z_s$. However, as they are inserted at the right end of U, and also taken for output at its right we conclude that the first topological sorting arrangement to be output is of the form

$$v_1 \ldots v_k y \ldots z_s z_{s-1} \ldots z_1 \ldots$$

The digraph of figure 2.1, if input as the following sequence of edges

(1, 3), (2, 1), (2, 4), (4, 3) and (4, 5)

is represented by the set of adjacency lists shown in figure 2.2 and causes algorithm 2.1 to print the five topological sorting arrangements as displayed in figure 2.3. Observe that redundant printing has been suppressed. Note also that the amount of work required to output a certain arrangement is proportional to the number of vertices actually printed in this arrangement plus the sum of their outdegrees. For instance, in the output

1 3 5

of figure 2.3 - which corresponds to the topological sorting 24135 - neither the vertices 2 and 4 nor the edges from them are manipulated.

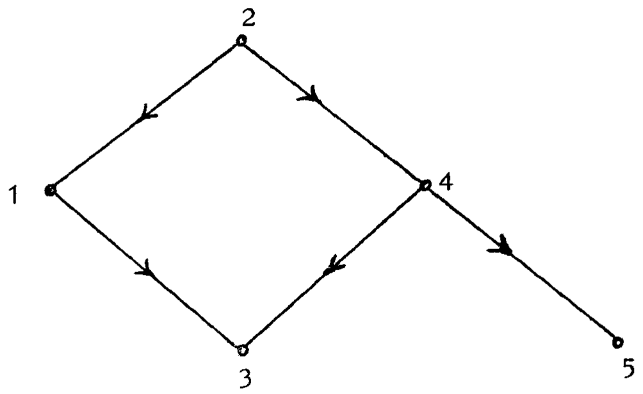


Figure 2.1

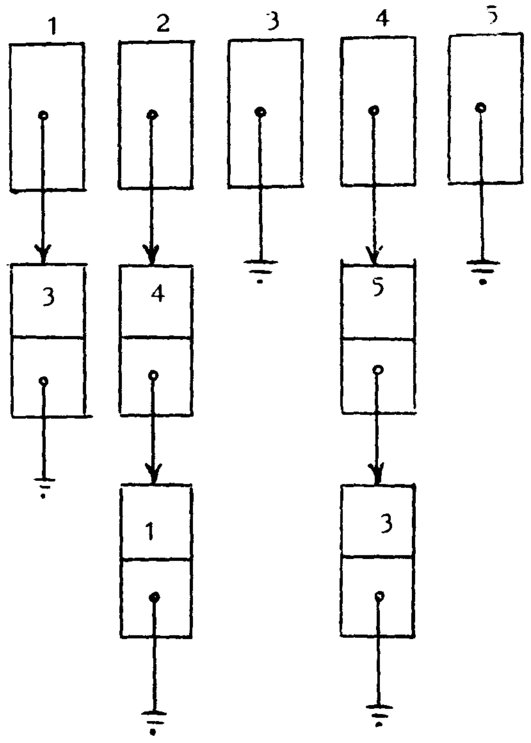


Figure 2.2

```
2   1   4   3   5
            5   3
    4   5   1   3
        1   3   5
            5   3
```

Figure 2.3

```
1:   2   1   4   3   5
2:               5   3
3:       4   5   1   3
4:           1   ...  see line 1
```

Figure 2.4

In fact, the number of times in the entire process that the upper time bound O(N+M) per output arrangement is attained equals the number of source vertices in the digraph. This value also equals the number of arrangements which are "fully" printed, i.e. in which all N vertices are explicitly printed. For the output of all other arrangements the bound is not attained.

If the set of edges of the digraph is empty, i.e. M = 0, any permutation of N is a topological sorting arrangement. Therefore, the algorithm operates as a permutation generator and outputs all T = N! permutations of N. Observe that, in this case, the total number of elements (vertices) that are actually printed is

$$N + N(N-1) + N(N-1)(N-2) + \ldots + N! = \lfloor N!e \rfloor - 1.$$

This follows from the fact that there are exactly N permutations in which the first element is printed, exactly N(N-1) permutations in which the second element is printed, and so on. From this result, we conclude that the average number of elements that are printed per permutation, is about <u>e</u>, and perhaps surprisingly, it is independent of N. For instance, when N = 4 a total of 64 elements are printed for the 24 permutations, and the average number of printings per permutation is 64/24, about 2.66. When N = 5, a total of 325 elements are printed, and the average is 325/120, about 2.70. The total time bound O((N+M)T), for the output of all T=N! permutations, becomes simply O(N!). This means that, within a constant factor, algorithm 2.1 is also efficient, as a permutation generator.

If N is large, the volume of output can be very large, as it may be concluded from the above case. For this sort of digraphs an interesting way of reducing the output has been suggested by Knuth in [KnSz74]. The idea is to allocate $O(2^N)$ more memory cells and to modify the recursive procedure so that it "remembers" similar past situations. A new global variable, which corresponds to the current value of the set $\{v_1, \ldots, v_k\}$ has to be added and the procedure ALLTOPSORTS ought to remember which sets it has seen before and where it occurred in the output. Whenever a set is repeated, the output can now be replaced by a simple cross-reference to the appropriate line. Figure 2.4 illustrates this new scheme of output, corresponding to the input digraph of figure 2.1.

The feature of "remembering" past situations can be applied to the problem of generating permutations with some additional simplifications, suggesting therefore a scheme for obtaining all permutations of a given set with reduced output. In fact, suppose we want the permutations of $\{1, 2, \ldots, N\}$ using a strategy similar to that of algorithm 2.1, i.e. at every stage we are looking for permutations starting with the sequence $x_1 x_2 \ldots x_k$ that has already been output. If $x_1 < x_2 < \ldots < x_k$ then this is the first appearance of this pattern. Otherwise, consider the smallest $j, 1<j<N$, such that $x_{j-1} > x_j$. Thus, the permutation that the first output of the type $x_1 \ldots x_j \ldots$ should remember is precisely the first appearance of the pattern $p_1 \ldots p_j$, where $p_1 \ldots p_j$ and $x_1 \ldots x_j$ are identical <u>combinations</u>, and $p_1<p_2<\ldots<p_j$. This can be easily detected in the algorithm if we do not allow $x_1 \ldots x_k$ to contain any $x_{i-1} < x_i$, $i \neq k$. Therefore, the algorithm can remember if a similar pattern occurred before simply by applying this test. The remaining problem is to find <u>where</u> that similar pattern occurred before, in the output. This is also computable and therefore no patterns need to be stored. To every permutation printed, a numeric label s is attached, calculated by:

(i) $s = 1$, for the first permutation printed

(ii) If s is the label of the current permutation $x_1 \ldots x_k \ldots$,
then the next printed permutation will be labelled

$s + (N - j)!$,

where j is the smallest index such that $x_{j-1} > x_j$.
If no such j exists, then s = 1, and define j to be equal to N.

Observe that if <u>all</u> permutations had been printed applying a strategy similar to algorithm 2.1, then s would have been the sequential line number of the corresponding permutation. Now the permutation $p_1 \ldots p_j$ which ought to be remembered and referenced from $x_1 \ldots x_j$, has as label s, the value computed by:

$$s(p_1 \ldots p_j) = 1 + \sum_{i=1}^{j} (N-i)!(p_i - p_{i-1} - 1), \text{ with } p_0 = 0.$$

This scheme, therefore, can be implemented using just O(N) space.

As an example, with N = 5, instead of printing the permutation, say (5)2341,-digits within parenthesis represent the redundant printing mentioned earlier -the following would occur: since 5 > 2, we have $j = 2$, $x_1 x_2 = 52$ and $p_1 p_2 = 25$. We know that the patterns corresponding to the set of permutations starting with 52 have already been printed before, and there are a total of $(N-j)! = 6$ such patterns, starting with the label s, computed by:

$$s(25) = 1 + 4!(2-0-1) + 3!(5-2-1) = 37.$$

Therefore, the printing of the permutations

```
(5)  2    3    4   1
(5) (2)  (3)   1   4
(5) (2)   4    1   3
(5) (2)  (4)   3   1
(5) (2)   1    3   4
(5) (2)  (1)   4   3
```

which would have occurred in an algorithm like 2.1, would be replaced b
the single line

(5) 2 ... see 6 permutations from label 37

and label 37 would be:

37: (2) 5 1 ... see 2 permutations from label 5

39: (2) (5) 3 ... see 2 permutations from label 27

41: (2) (5) 4 ... see 2 permutations from label 31

label 5 would be:

5: (1) (2) 5 3 4

6: (1) (2) (5) 4 3

label 27 would be:

27: (2) (3) 5 1 4

28: (2) (3) (5) 4 1

and label 31 would be:

31: (2) 4 5 1 3

32: (2) (4) (5) 3 1.

2.6 Conclusions

An algorithm for obtaining all topological sorting arrangement for a given acyclic digraph has been presented. The algorithm utilises a recursive backtracking procedure which outputs each arrangement in at most O(N+M) time. An iterative machine-oriented translation of the algorithm appears in [KnSz74]. Both versions of it - recursive and iterative - have been implemented and as expected, the iterative version produced better running times than the recursive one when both were applied to identical inputs.

A simple test can be added to the strategy to enable it to detect the presence of cycles in the input digraph which is supposed t be acyclic. If deque U is empty in any computation of ALLTOPSORTS(k) except ALLTOPSORTS(N), this means that the subdigraph formed by the subset of vertices not yet output is such that no vertex has outdegree zero in it. Hence a cycle must exist. If a digraph D containing cycles is input to algorithm 2.1 as it stands, then the output obtained is the set of all topological sorting arrangements of a subdigraph D' of D such that D' is the maximal subdigraph of D which does not contain any cycle nor any vertex which is reachable from a vertex belonging to a cycle of D.

CHAPTER 3

SOME GRAPH CYCLE ALGORITHMS

3.1 Introduction

Some problems, such as determining whether a graph has certain properties, or constructing a set of objects related to the graph admit of algorithmic solutions which have a time bound linear in the size of the graph. These include the problems of finding: the strongly connected components of a directed graph [Ta72], the biconnected components of a graph [Ta72], the graph from its given line graph [Ro73], [Le74], partitions of a graph into simple paths [HoTa73]. Testing planarity of a graph can be performed in a time just proportional to the number of its vertices [HoTa74]. Clearly, any algorithm for obtaining and explicitly listing, a set of objects related to the graph <u>must</u> be at least proportional to the total number of such objects. If this number grows exponentially with the size of the graph, the algorithm has an exponential running time. For such problems, a given algorithm may be additionally characterized by introducing a time bound per object obtained, and two algorithms can be compared according to their bounds per object. The problem of finding all elementary cycles of a directed graph falls into this category. Among the great number of cycle algorithms surveyed by Prabhaker and Deo [PrDe74] the algorithm by Johnson [Jo73a] presents the best time bound, namely a linear bound in the size of the graph, per cycle. This algorithm was devised by imposing further constrain on the backtracking performed by an already constrained backtracking algorithm [Ta73].

The present thesis proposes a cycle finding algorithm that has a similar (worst case) time bound as [Jo73a]. However, while maintaining all the constraints of [Jo73a], we are proposing new strategies that represent further restrictions to the backtracking.

The elementary cycles algorithm for digraphs has been also adapted for handling undirected graphs, resulting in an algorithm for solving the problem of finding all elementary cycles for undirected graphs. So far this method has been shown to be more efficient than algorithms specially devised for undirected graphs.

Another cycle problem considered in this chapter consists of a method for finding a fundamental set of cycles for an undirected graph. This problem is less complex than finding all the elementary cycles, as it can be verified from the discussion of the problem, later in this chapter. Our proposed solution may be regarded as a variation of an algorithm by Paton [Pa69].

Section 3.2 to 3.6 of this chapter, handles the problem of finding the elementary cycles of a digraph: a discussion of some existing methods is the subject of section 3.2; section 3.3 presents our proposed algorithm, which is shown to be correct in 3.4; the evaluation of its performance appears in section 3.5; a more detailed appreciation of Johnson's algorithm [Jo73a] and its comparison with our proposed method constitute section 3.6. Finding a fundamental set of cycles of an undirected graph is the subject of sections 3.7 to 3.10: an overview of some existing methods appears in section 3.7; our proposed strategy is described in 3.8; and its correctness and performance are discussed in sections 3.9 and 3.10, respectively. The problem of finding the elementary cycles in an undirected graph is handled in sections 3.11 to 3.14: a general appreciation of the problem is presented in section 3.11; our proposed method is described in section 3.12; remarks about its correctness and an evaluation of its performance constitute sections 3.

and 3.14 respęctively. Finally, some further comments about the results which appear in this chapter form the content of section 3.15. As already mentioned, the contents of 3.2 to 3.6 was reported in [SzLa75].

3.2 Elementary cycles in directed graphs

Tiernan [Ti70] finds all elementary paths $v_1, \ldots, v_k$, $v_1 < v_i$, $1 < i \leq k$ and $1 \leq k \leq N$. If $(v_k, v_1) \in E$ then the cycle $v_1, \ldots, v_k, v_1$ is enumerated. This strategy corresponds to an essentially unconstrained backtracking and was also presented by Roberts and Flores [RoFl66] and Berztiss [Be71]. Floyd [Fl67] has described a non deterministic version of this algorithm. Weinblatt [We72] also searches for elementary paths, but proposes to improve execution time by storing cycles already found and constructing new ones from these. In [Ta72], Tarjan gives examples illustrating that the algorithms [Ti70] and [We72] may take exponential time in the number of cycles enumerated. Lauer [La73] discusses the generalisation of Tiernan's algorithm to different representations of digraphs, improves storage requirements and proposes alternate proofs. Another backtracking algorithm presented by Berztiss [Be73] has been shown by Prabhaker and Deo [PrDe74] also to have a time bound exponential in the number of cycles. The algorithm by Syslo [Sy73, Sy75] is also based on a backtracking strategy and constitutes a variation of Tiernan's method.

Tarjan's algorithm [Ta73] is based on Tiernan's depth-first method. It makes use of two stacks, the <u>point stack</u> for stroing the path currently being examined and a <u>mark stack</u>, as well as a boolean vector called <u>mark vector</u>. The mark stack is used as a set of pointers to the mark vector. Whenever a new cycle is found, all vertices in the current point stack will eventually be unmarked when popped from this stack.

If no cycles is found involving a vertex, it will be deleted from the point stack, but continue to be marked. Some of the unnecessary work done by Tiernan is avoided by the condition that if a vertex is reached but is found marked, then it is not re-explored at this stage. However, we mention two points where this algorithm still does unnecessary work. First, whenever a vertex v is going to be unmarked because a cycle involving it was found, <u>all</u> the vertices that are above v in the mark stack will also be unmarked, even if some of them are involved in no cycle. Second, Tarjan follows Tiernan's principle of only searching for elementary cycles $v_j, \ldots, v_k$ with $v_j < v_i$, $1 < i \leq k$, where v_j is the vertex at the bottom of the stack, called <u>start vertex.</u> The inefficiency involved in this is discussed in section 3.6. The algorithm [Ta73] is bounded by O(N.M(C+1)) time and O(N+M) space.

Another method was developed by Ehrenfeucht, Fosdick and Osterweil [EhFoOs73] which includes both breadth-first and depth-first search, and makes use of an additional phase for collecting information about the digraph. This pre-processing requires $O(N^3)$ time and the actual process enumeration of the elementary cycles is bounded by O(N. M) time per cycle.

The algorithm by Read and Tarjan [ReTa73] first determines the set of all start vertices, to be used later during the search. Each strongly connected component is processed separately, and a vertex s will be used as a start vertex if there exists an edge (r,s) where r is a descendant of s in a directed rooted tree, generated by a depth-first search. For each start vertex s the algorithm invokes a recursive backtracking procedure BACKTRACK(s), which initiates the construction of an elementary path from s. If a recursive call BACKTRACK(v) occurred then v had been added to the current path before, and is the end of this path. Assume the computation of BACKTRACK(v_k) and v_1 , v_2, ..., v_k the current path. Initially, the set of connectable vertices is determined. A vertex w is <u>connectable</u> if there exists a path from w to s which does not involve any vertex of the current path. The path may only be extended with a connectable vertex w such that $(v_k , w) \in E$ and therefore any addition to the path is sure to lead to a new elementary cycle. If w=s then such a cycle has been found. Otherwise a recursive call BACKTRACK(w) occurs, unless there is exactly one edge (w,x) from w, such that x is connectable. In this last alternative, x is assigned to w and is added directly to the path, with no call BACKTRACK(x). This process is iterated until either w=s or there is more than one connectable vertex x, with $(w,x) \in E$.

The above strategy is therefore simple and elegant. However, it has an important drawback, which is the cost of the determination of the connectable vertices. Consider the digraph of figure 3.1, with $N=K^2-K+1$ vertices, $M=K^2+K-2$ edges and $C=2K-2$ elementary cycles, and a numbering of the vertices as obtained by a depth-first search [Ta72]. The algorithm would find $\{1,2,\ldots,K\}$ to be the set of start vertices. Let us examine the computation with start vertex 1. When a vertex $v \in \{2, \ldots,K\}$ is added to the current path, at any stage of the computation, there exist exactly two vertices $w, (v,w) \in E$, which are connectable, namely, a vertex from the subdigraph B_v, and vertex v+1 if $v \neq K$ or vertex 1 otherwise. Consequently, each time a vertex $v \neq 1$ is added to the current path, a call BACKTRACK(v) occurs. Therefore, for the enumeration of the K-2 elementary cycles (1,2,3,4,..., K,1; 1,3,4,...,K,1; ...;1,K-1,K,1) with start vertex 1, a total of $(K^2-K)/2$ calls of BACKTRACK occur. Since for the computation of the connectable vertices, in each of these calls, every one of the K^2-K+1 vertices is explored (i.e. marked unconnectable), the algorithm requires

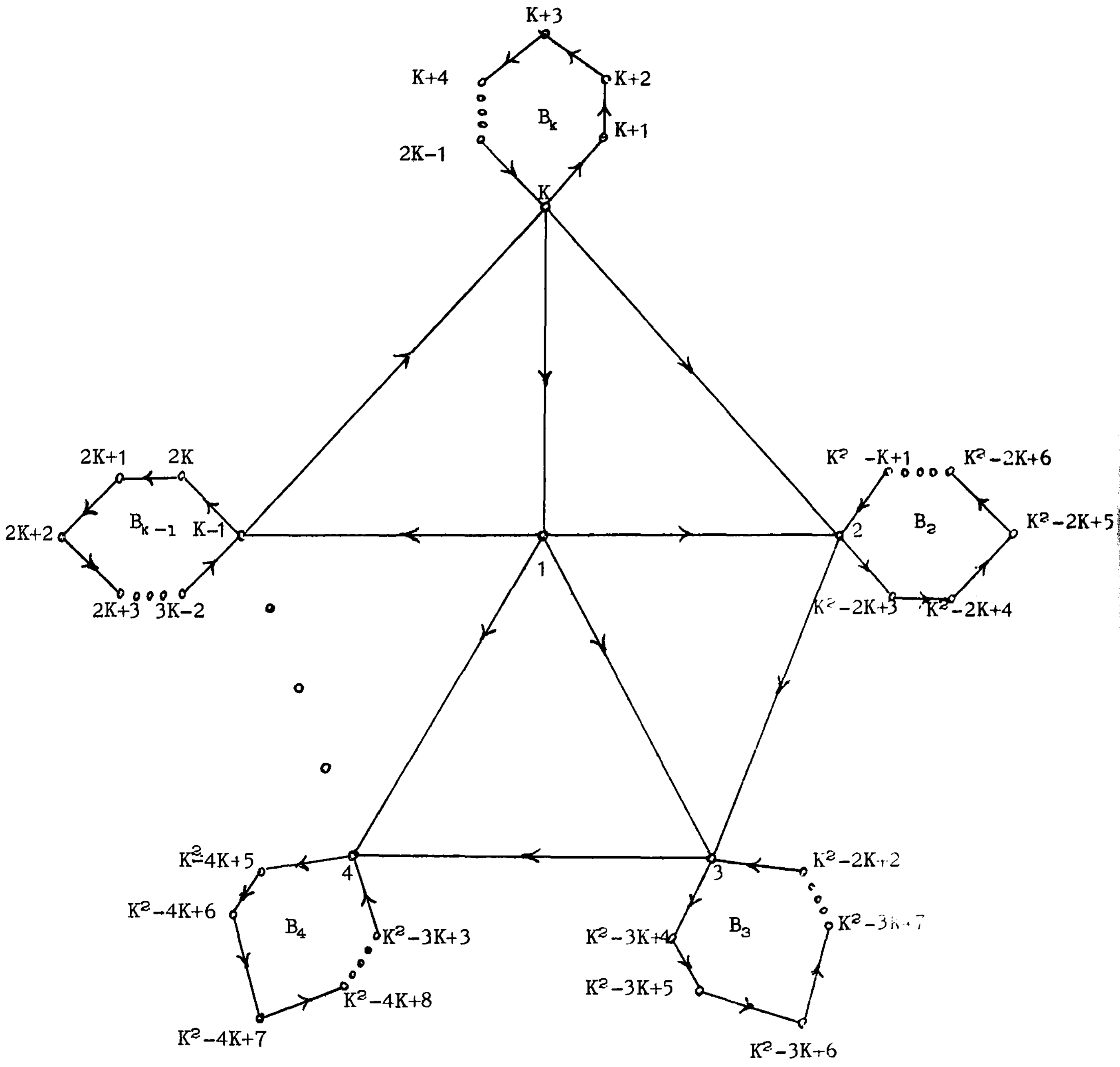

K+3
K+4
K+2
B_k
K+1
2K-1
K
2K+1
2K
2K+2
B_{k-1}
K-1
2K+3
3K-2
1
K^2 -K+1
K^2-2K+6
2
B_2
K^2-2K+5
K^2-2K+3
K^2-2K+4
K^2-4K+5
4
K^2-4K+6
B_4
K^2-3K+3
K^2-4K+8
K^2-4K+7
3
K^2-2K+2
K^2-3K+7
K^2-3K+4
B_3
K^2-3K+5
K^2-3K+6

Figure 3.1

at least $O(K^4)$ time. This contradicts [ReTa73] which mentions the time bound $O(N+(C+1)M)$.

However, the algorithm [ReTa73] could be implemented in such a way that by introducing a convenient scheme of lists, vertices that were already unconnectable at the beginning of the computation for the current connectable vertices, would not need to be marked unconnectable again. For evaluating the performance of the algorithm in this case, consider the digraph of figure 3.2. It was obtained from figure 3.1 by appending K subdigraphs to the subdigraph B_k, as indicated. Any subdigraph B_i of figure 3.2 is isomorphic to any subdigraph B_j, of figure 3.1, hence we still have, for the digraph of figure 3.2, $N=O(K^2)$, $M=O(K^2)$ and $C = O(K)$. If vertex 1 is the start vertex, again a call BACKTRACK(v), $v \neq 1$, follows the addition of vertex v to the current path and therefore $(K^2-K)/2$ calls of the procedure are invoked. Upon exit of any of the $O(K^2)$ calls of BACKTRACK(v), with $v \neq K$ and start vertex 1, all $O(K^2)$ vertices of B_k, B_{k+1}, ..., B_{2k} are connectable and therefore they ought to be explored in every one of those $O(K^2)$ computations. Therefore $O(K^4)$ time is required for this digraph and we conclude that the time bound is not $O(N+(C+1)M)$ even if this more efficient implementation is realized. A time bound for this algorithm is $O(N+M+NMC)$.

The algorithm by Johnson [Jo73a] also employs the technique of constructing elementary paths from a start vertex, in a stack. For each strongly connected component, the start vertex is chosen so as to be the least vertex of this component. Subsequently, a new maximal strongly connected partial subdigraph is obtained, which does not contain that vertex. The new start vertex is chosen to be the least in this partial subdigraph and so on. For each start vertex s, a recursive backtracking procedure is invoked and its computation is similar to that of Tarjan's

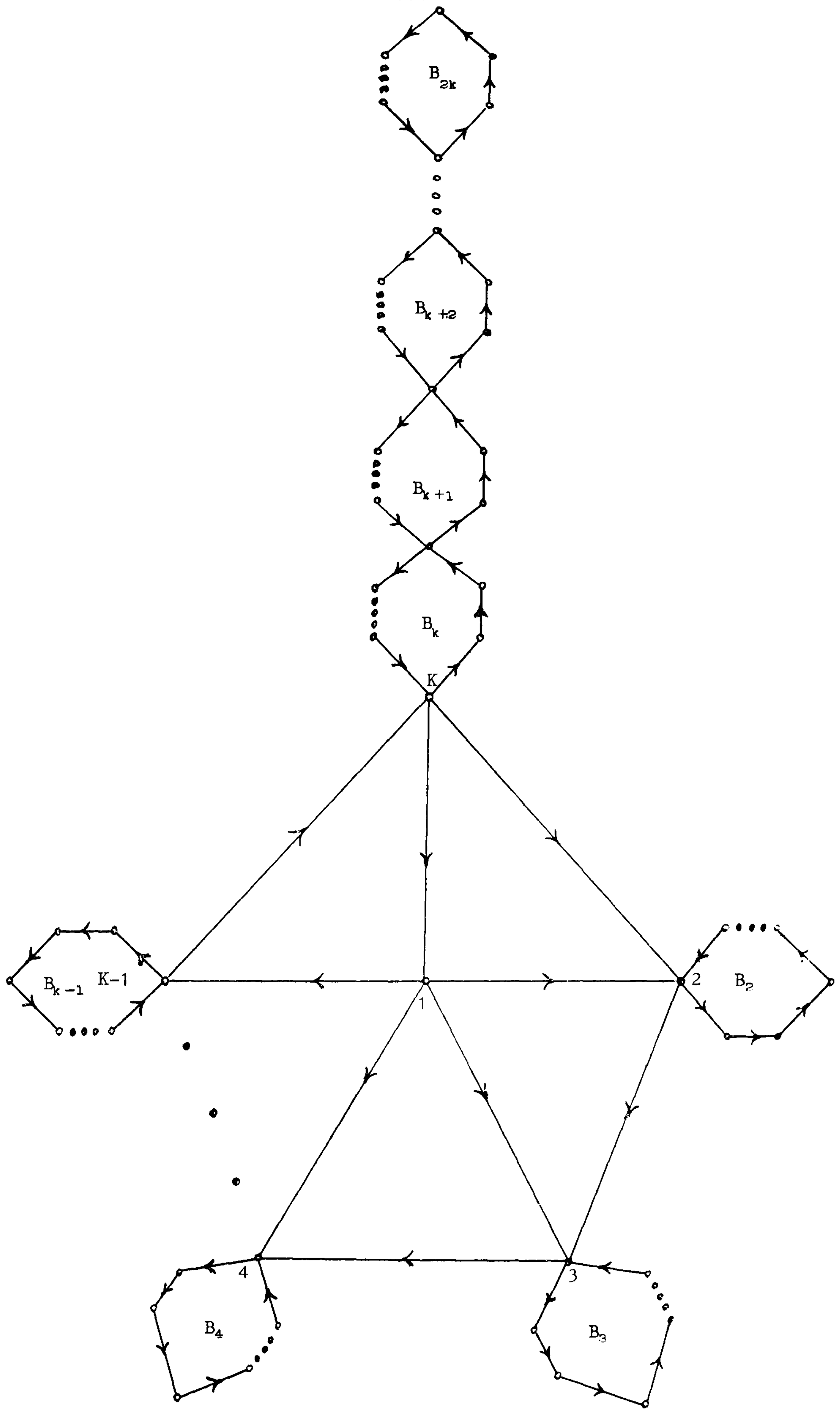


Figure 3.2

algorithm, except for the marking system, which was considerably enhanced. A vertex v is marked each time it enters the stack. Upon leaving the stack, if an elementary cycle was found involving v and the start vertex s, then v is unmarked. Otherwise, it remains marked until another vertex u is popped from the stack and such that an elementary cycle existed involving u and s, and there exists a path from v to u consisting of vertices that are marked and not in the stack. Johnson implements this strategy efficiently, using a scheme of lists B, one list B(v) per vertex v. At any given moment, B(v) contains those vertices u such that (u,v) $\in$ E and u is marked and not in the stack. The actual unmarking is performed by a procedure UNBLOCK(v) which will recursively call UNBLOCK(u), if u$\in$B(v). This algorithm is bounded by O(N+(C+1)M) time and O(N+M) space. Further remarks concerning this method and comparisons with the proposed algorithm can be found in section 3.6.

3.3 The Proposed Algorithm

Our algorithm also uses a recursive backtracking procedure but a more efficient system for detecting elementary cycles. This detection occurs as soon as the elementary cycle is generated anywhere in the current path under examination. This path is kept in a stack (Tarjan's point stack). The boolean vector is retained but not the mark stack. Instead, we have utilised and slightly modified Johnson's marking system using one list B(v), per vertex v. A vertex u is inserted in list B(v) if (u,v)$\in$ E and the exploration of edge (u,v) has not lead to a new elementary cycle. In addition to these structures, we use a position vector and a boolean reach vector. If a vertex v is the j-th vertex from the bottom of the stack, then position (v)=j; when v is deleted from the stack then position (v)=N+1. If a vertex v has not yet left the stack for the first time, then reach(v) = false, otherwise reach(v)

= true. A vertex v is marked when it enters the stack, and the mark is kept at least as long as this vertex remains in the stack. Upon leaving the stack, v is unmarked only if a new elementary cycle was found with v but not necessarily with the vertex at the bottom of the stack (start vertex). If v leaves the stack with the mark on, then it will be unmarked when a vertex z_1 is popped from the stack in such a way that a new elementary cycle was found with z_1, and there exists a path $z_k, z_{k-1}, \ldots, z_1, (z_k = v)$ such that $z_{i+1} \in B(z_i)$, $k < i \leq 1$, at that time.

The digraph is represented by a set of adjacency lists with one list A(v) per vertex v. A pre-processing is performed to find the strongly connected components of the digraph, using the method described in [Ta72]. For each strongly connected component a start vertex is chosen to be the vertex with maximal indegree in this component. The present method ensures that, when this start vertex is deleted from the stack, all the elementary cycles of this component have been enumerated. Therefore only one start vertex per component is required. As it can be observed from the proposed strategy, if a start vertex would have been chosen to be an arbitrary vertex of the digraph - instead of a vertex with maximal indegree in a strongly connected component - the algorithm could be easily modified so as to avoid finding the strongly connected components. The modified algorithm would have the same time bound as the one currently described.

The basic idea of the algorithm is similar to all previously described methods, namely to try to extend the current elementary path under examination. Consider the case where the content of the stack is $v_1 v_2 \ldots v_{k-1}$ and edge (v_{k-1}, v_k) is reached:

(i) If v_k is not marked then necessarily v_k is not in the stack, the elementary path will be extended with v_k, and an edge from v_k will be examined.

(ii) If v_k is marked and not in the stack then necessarily, there can be no new elementary cycles generated from the path v_1, v_2, ..., v_{k-1}, v_k and therefore v_k is not re-explored, at this stage. Vertex v_{k-1} is inserted in list $B(v_k)$ and v_k is deleted from $A(v_{k-1})$.

(iii) If v_k is marked and lies in the stack then an elementary cycle was found, and it can be recorded at once. The algorithms [We72] and [Be73] also consider this cycle at that stage. However, some efficient algorithms as [Ta73], [EhFoOs73], [Jo73a] and [ReTa73] disregard it, if v_k is not the start vertex. The problem that arises when considering such a cycle with $v_k \neq v_1$, is that a mechanism for detecting duplicate cycles must be set up. The nature of this mechanism follows from the observation that a cycle is a <u>new</u> cycle, if and only if at least one of its vertices had never been deleted from the stack. The fact that it has not been deleted before is indicated by setting a variable q, local to the recursive procedure. For a given computation of this procedure q indicates the top most vertex of the stack that has never been deleted from it. Therefore, if position $(v_k) \leq q$ a new elementary cycle is found. Otherwise, this is a duplicate cycle: v_{k-1} is inserted in $B(v_k)$ and v_k is deleted from $A(v_{k-1})$.

In cases (ii) and (iii), when v_k is marked the elementary path is not extended. If a certain elementary path cannot be extended any more, the algorithm backtracks to the previous vertex in the stack, and so on. When the start vertex is deleted from the stack, a new strongly connected component is considered, and so on, until all such components have been processed.

Below is an ALGOL-like formulation of the proposed algorithm. The combined action of variables f and g ensures the correct propagation of the information that a new elementary cycle was found with a certain vertex v at the top of the stack, for all vertices that are below v in the stack. The following procedure CYCLE processes only non-trivial strongly connected components (those which have more than one vertex). If this condition is relaxed then the algorithm would still be correct, but corollary 3.1 of section 3.5 would have to be reformulated.

ALGORITHM 3.1

```
begin comment algorithm for finding the elementary cycles of a digraph;
    procedure CYCLE (integer value v,q;  logical result f);
    begin procedure NOCYCLE (integer value x,y);
        begin insert x in B(y);
            delete y from A(x)
        end NOCYCLE;
        procedure UNMARK (integer value x);
        begin mark(x) := false;
            for y ∈ B(x) do
            begin insert x in A(y);
                if mark(y) then UNMARK(y)
            end;
            empty B(x)
        end UNMARK;
        logical G;
        mark(v) := true;  f := false;
        insert v in the stack;
        t := number of vertices in the stack;
        position(v) := t;
        if ¬ reach(v) then q := t;
        for w ∈ A(v) do
            if ¬ mark(w) then
            begin CYCLE (w,q,g);
                if g then f := true else NOCYCLE(v,w)
            end
            else if position(w) ≤ q then
                begin output cycle w to v from stack, then w;
                    f := true
                end else NOCYCLE(v,w);
        delete v from stack;
        if f then UNMARK(v);
        reach(v) := true;
        position(v) := N+1
    end CYCLE;
    read the digraph D;
    find the adjacency lists A of the strongly connected components of D;
    for j:=1 step 1 until N do mark(j):= reach(j) := false;
    for each non-trivial strongly connected component do
    begin s := vertex with maximal indegree in this component;
        CYCLE (s, dummy, dummy)
    end
```

3.4 Correctness

Let D(V,E) be an input digraph with no trivial components.

Lemma 3.1:

Every vertex $v \in V$ enters the stack at least once.

Proof:

Let S denote the strongly connected component of D which v belongs to, and $s \in V$ denote the chosen start vertex of S. Clearly the call CYCLE(s, dummy, dummy) occurs and thus s enters the stack. Since all vertices v, $v \neq s$ of S are unmarked at the time of this call; and since s reaches v, by induction it can be shown that every vertex will eventually be added to the stack.

Lemma 3.2:

If $v_1 \ldots v_k$ constitutes the stack at a given moment, and a new elementary cycle is found with v_k then all vertices $v_1, \ldots, v_k$ are unmarked upon leaving the stack.

Proof:

At the time this cycle is detected, variable f is set to true, which ensures that v_k is unmarked upon leaving the stack. Because of the statement <u>if</u> g <u>then</u> f := <u>true</u> ... executed at the return of each recursive call of CYCLE, an inductive argument shows that a call UNMARK(v_i) $1 \leq i \leq k$ occurs when v_i is popped from the stack. Therefore, each v_i is unmarked at that time.

Lemma 3.3

Let $v_1, \ldots, v_k, v_1$ be an elementary cycle, such that $v_1 \ldots v_k$ or a cyclic permutation of it have already appeared in the k top positions of the stack at some earlier time, and at least one of these vertices has been deleted from it before. If $v_1 \ldots v_k$ now occupy the k top positions of the stack, then <u>all</u> $v_1, \ldots, v_k$ have already been deleted from it.

Proof:

If exactly the configuration $v_1 \ldots v_k$ appeared before as the k top positions of the stack, this means that the configuration of the stack below v_1 on that occasion was different from that below v_1 on the present one, because the backtracking search strategy ensures that a given configuration of the stack can never be repeated once its top vertex is deleted. Therefore, all $v_1, \ldots, v_k$ have already left the stack. If instead, a cyclic permutation $v_j \ldots v_k\ v_1 \ldots v_{j-1}$ $(j \neq 1)$ appeared before as the k top positions of the stack, we also conclude that all these vertices later left the stack, since v_1 is above v_j in that configuration and below v_j in the present one.

Lemma 3.4:

Let $z_1, \ldots, z_k$ be an elementary path, $(z_k, v) \in E$, where v is a vertex in the stack that has never been deleted from it. Then if $z_1, \ldots, z_k$ are not in the stack, z_1 is unmarked.

Proof:

By induction on the index k. For z_k the lemma holds, because before the first time z_k is reached, z_k is unmarked (by the initialisation) and because $(z_k, v) \in E$ by lemmas 3.3 and 3.2, we conclude that z_k is unmarked each time it leaves the stack. By the induction hypothesis, if $z_2, \ldots, z_k$ are not in the stack, z_2 is unmarked. Assume now that $z_1, z_2, \ldots, z_k$ are not in the stack. If z_1 has not been explored yet or a new elementary cycle was found with z_1 in its last exploration, then z_1 is unmarked, and the lemma is satisfied. Suppose then that no new cycle was detected with z_1 at the last time z_1 was in the stack. Therefore, the exploration of edge (z_1, z_2) would cause z_1 to be inserted in $B(z_2)$, and at the time z_1 left the stack with the mark on, z_2 was also marked. Hence if $z_1, z_2, \ldots, z_k$ are now not in the stack, we can apply the induction hypothesis and conclude that a call of UNMARK(z_2) occurred for unmarking z_2.

Since $z_1 \in B(z_2)$ a recursive call UNMARK(z_1) also occurred and z_1 is unmarked.

Lemma 3.5:

Let $v_1, \ldots, v_k, v_1$ be a convenient cyclic permutation for an elementary cycle, such that v_1 was the first among v_j, $1 \le j \le k$ to ever enter the stack. Then there exists a configuration of the stack, such that before v_1 leaves the stack for the first time, $v_1 v_2 \ldots v_j$, $1 \le j \le k$ appear in the j top positions of the stack.

Proof:

Induction on j. For j=1 the lemma holds, trivially by its hypothesis. By the induction hypothesis, $v_1 \ldots v_{j-1}$ occupy the j-1 top positions of the stack and v_1 has not yet left the stack. Since $(v_{j-1}, v_j) \in E$ this edge will eventually be reached. Because v_{j-1} can only leave the stack after all the edges from it have been examined, we conclude that when (v_{j-1}, v_j) is going to be examined the j-1 top positions of the stack are still $v_1 \ldots v_{j-1}$ and v_1 has not yet left the stack. Also no v_p, $j \le p \le k$, at that moment is in the stack, because otherwise v_p would be underneath v_1, which contradicts the fact that v_1 entered the stack before v_p and has not left it. In addition, $v_j, \ldots, v_k$ is an elementary path and $(v_k, v_1) \in E$. Therefore, by lemma 3.4 we conclude that v_j is unmarked and hence will be placed on top of v_{j-1}, in the stack.

Comment: Because of lemma 3.1, the hypothesis of lemma 3.5 that v_1 was the first among v_j ever to enter the stack, is consistent.

Lemma 3.6:

If a vertex is in the stack, it is marked.

Proof:

If a vertex enters the stack it becomes marked. We have then to prove that it is not unmarked while in the stack. Note that an unmarking process can only be initiated by a call UNMARK(z) where z is a vertex which is presently being deleted from the stack, and which was involved in a newly detected elementary cycle. Assume this is the case and the problem is to show that UNMARK(z) will not unmark any vertex in the stack. Suppose vertex w_1 is in B(z) at the time of this call. Then when w_1 entered B(z), either w_1 was above z in the stack, or z was marked and not in the stack. The latter alternative cannot occur, since later z entered the stack, which ensures that z was unmarked and its unmarking emptied B(z). Therefore w_1 was not in the stack when it was unmarked. By an inductive argument it can be shown that if the call UNMARK(z) invoked recursive calls UNMARK(w_i), then all w_i entered the stack necessarily after z, and hence are not in the stack at the time the call UNMARK(z) occurs.

Lemma 3.7:

Each elementary cycle of D is listed at least once.

Proof:

Let v_1, ..., v_k, v_1 be an elementary cycle of D, such that v_1 was the first among v_1, ..., v_k ever to enter the stack. By lemma 3.5, v_1 ... v_k will eventually occupy the k top positions of the stack before v_1 leaves the stack for the first time. If v_1 has not yet left the stack, at the start of the computation of CYCLE with $v = v_1$, reach (v_1) = false and therefore q was set to position (v_1). Thus, we can conclude that parameter q passed to the computation of CYCLE with $v = v_k$ satisfies position $(v_1) \leq q$. In addition, by lemma 3.6 we conclude that the examination of edge (v_k, v_1) in this last computation will find mark (v_1) = true. Hence, the cycle v_1, ..., v_k, v_1 is listed.

Lemma 3.8:

Each elementary cycle of D is listed at most once.

Proof:

Let $v_1, \ldots, v_k, v_1$ be an elementary cycle of D which has already been generated and assume $v_1 \ldots v_k$ occupy the k top positions of the stack. By lemma 3.3 we conclude that all $v_1, \ldots, v_k$ have already been deleted from the stack some time before. Therefore, reach (v_1) = true at the start of any of the current computations of CYCLE with $v = v_j$, for $1 \leq j \leq k$. Consequently, position $(v_1) > q$ in any of these computations. Thus, the exploration of the edge (v_k, v_1) will not cause the cycle $v_1, \ldots, v_k, v_1$ to be listed.

Theorem 3.1:

The proposed algorithm for finding all elementary cycles of D, is correct.

Proof:

Lemmas 3.7 and 3.8.

3.5 Performance

Lemma 3.9:

Let D be a strongly connected component of a digraph input to the program. If a vertex v changes from marked to unmarked twice, a new elementary cycle is enumerated.

Proof:

If v is in the stack and a new elementary cycle was found with v the lemma is satisfied. Assume then that v left the stack with the mark on, and let z denote the top most vertex of the stack with which a new elementary cycle was later found, and whose unmarking would eventually invoke a recursive call of UNMARK(v). Assume this call occurred and denote by u_2 and u_1 respectively the top and bottom vertices of the cycle to which z belongs (figure 3.3). Thus, there exists an elementary path v, ..., z, ..., u_2, whose vertives are all unmarked by the return of this call. Therefore, if v enters the stack afterwards, so does u_2. Assume this case, and suppose that u_1 has not left the stack in the meantime. Then the exploration of edge (u_2, u_1) would lead to a new elementary cycle $u_1, \dots, u_2, u_1$. If on the other hand, u_1 had left the stack when u_2 is reached, this means that at least one new edge from a vertex w, below the first position of u_1 in the stack was explored for the first time. Since the processed digraph is strongly connected, a new elementary cycle ...,w,... is detected with this edge.

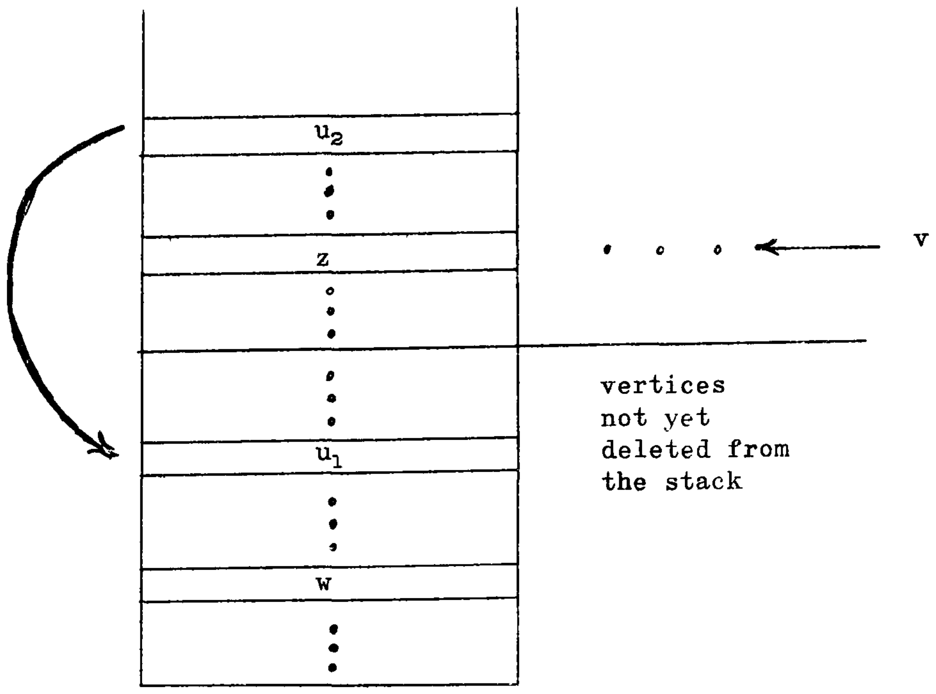


Figure 3.3

Theorem 3.2:

Let D be a directed graph with N vertices, M edges and C elementary cycles, input to the program. Then O(N+(C+1)M) time and O(N+M) space are required to enumerate C elementary cycles.

Proof:

The space bound follows from the fact that the representation of the digraph by adjacency lists requires O(N+M) cells, the B lists require also O(N+M) cells and the remaining data structures require O(N). The time bound follows from lemma 3.9. A vertex can enter the stack at most twice between the output of two new elementary cycles. Consequently, a given edge can be explored at most twice during this time. Also because of lemma 3.9, a recursive call UNMARK(v) from the computation of UNMARK(w), for $(v,w) \in E$, can only occur at most twice between the detection of new elementary cycles. The same results apply for the situations before the first cycle is output and after the last one. Also we observe that any deletion or insertion in lists A and B occurring during the process can be performed in a constant number of steps. Thus a time bound per cycle is O(N+M). If $D_1,\ldots,D_p$ are the strongly connected components of D, having respectively, N_i vertices, M_i edges and C_i elementary cycles, $1 \leq i \leq p$, then an upper bound for the output of the C_i cycles of D_i is $O((N_i + M_i)(C_i + 1))$. If D_i is non-trivial then $M_i \geq N_i$, otherwise $M_i = 0$, and consequently, this bound can be expressed by $O(N_i + M_i C_i)$. Since finding strongly connected components of D, in the initialisation of the process, consumes O(N+M) time, we conclude that the total time bound is O(N+M(C+1)).

Corollary 3.1: A time bound per cycle is O(M) for any elementary cycle, except for the first enumerated, whose bound is O(N+M).

3.6 Critical Remarks

Prabhaker and Deo [PrDe74] have already shown that so far, the most successful cycle-finding algorithms are those based on a backtracking search strategy. Tiernan's algorithm adopts an essentially unconstrained backtracking. The main difference between the algorithms of Tiernan and Tarjan is that the latter has introduced a marking mechanism which avoids the exploration of a vertex if this vertex is found marked when it is reached. This situation can occur even if this vertex does not lie in the path currently under examination. As a result the backtracking becomes constrained. The basic difference between the algorithms by Tarjan and Johnson is that the latter has modified and improved the marking system. If an elementary cycle is found with a certain vertex v, then upon v leaving the stack, Tarjan unmarks v and all vertices of a set Z which is the set of vertices which are marked, not in the stack, and which entered the stack for the last time, after v. Instead, Johnson unmarks v and only such vertices $z \in Z$ for which there exists a path from z to v, involving solely vertices of Z. Also, all N vertices become start vertices in Tarjan's algorithm. In Johnson's method, for each strongly connected component the number of start vertices equals the number of vertices v such that there exists an edge to v, from a descendant of v in a directed rooted tree, obtained by a depth-first search of this component. These conditions represent further constraints to the backtracking.

The principal difference between Johnson's algorithm and the present one is that we detect an elementary cycle, as soon as it appears in the top positions of the stack. Consequently, while exploring a vertex v we do not seek exclusively cycles involving v and the start vertex, but any other new cycle is considered. Since this earlier detection means that the algorithm will not initiate an explicit new search aimed to find this cycle, as [Jo73a] does, this new strategy imposes a further constraint on the backtracking. Also unlike [Jo73a] for each non-trivial strongly connected component the present algorithm considers exactly one start vertex. Another difference between the two strategies lies in the marking system: if w is a vertex that is marked and (v,w)$\in$ E then in the proposed method only one unsuccessful exploration of edge (v,w) can occur whilst w remains marked. In [Jo73a] each time vertex v is found unmarked, an exploration of edge (v,w) certainly occurs. The effect of these differences in the actual manipulation of digraphs may be appreciated in the following examples.

The digraph of figure 3.4 has N vertices, 2N-3 edges and N-2 elementary cycles. It has the property that certain vertices (1,2 and 3 in the example) are involved in every possible existing cycle. Digraphs with this property seem to provide favourable examples for Johnson's algorithm because if one of these special vertices is the start vertex then each elementary cycle is generated only once. In fact, for such digraphs both algorithms([Jo73a] and the present) may perform exactly the same number of steps, for identical adjacency lists. In figure 3.4 the start vertex is vertex 1 for both algorithms, and both would explore each edge exactly once in the search for the N-2 elementary cycles, thus requiring 2N-3 steps, for termination. Note that by number of steps we mean the frequency of execution of a given statement which has the highest frequency among all by the end of the process (this corresponds to the number of edge explorations). If the digraph is re-labelled such that the new vertex 1 is the previous vertex 2, Johnson's algorithm would take 3N-6 steps, because the previous vertex 1 (and the edge from it to the new vertex 1) suffers N-3 additional explorations. Since this vertex is the vertex with maximal indegree, the present algorithm would always consider it as start vertex and consequently would find all elementary

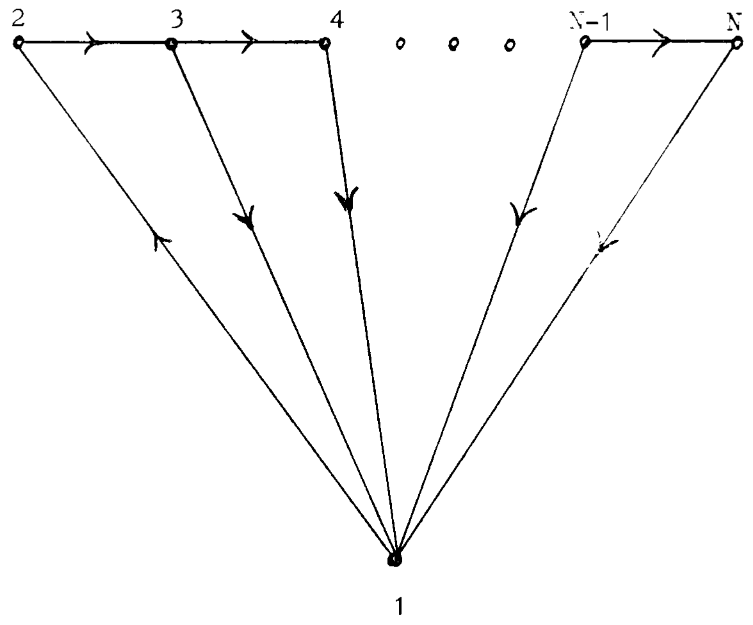


Figure 3.4

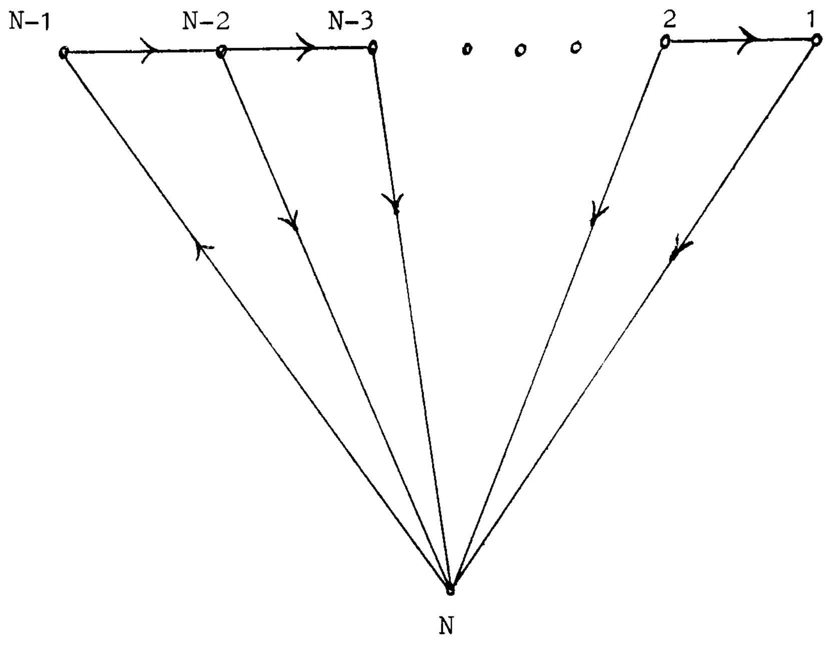


Figure 3.5

cycles in 2N-3 steps. For this class of digraphs, the worst case for Johnson's algorithm occurs when the vertices are labelled as in figure 3.5, in which the subdigraph composed of vertices N, N-1, N-2, is explored N-2 times, the subdigraph composed of N, N-1, N-2, N-3 is explored N-3 times, and so on. A total of N(N-2) steps are required for the enumeration of the elementary cycles of this digraph, using [Jo73a] compared with 2N-3 using the present method.

Concerning the choice of the start vertex, we have adopted a different strategy from [Jo73a] which always chooses the least vertex as start vertex. Our approach is based on the fact that if $v_1, v_2, \ldots, v_k, v_1$ and $v_1', v_2', \ldots, v_k', v_1'$ are elementary cycles involving precisely the same vertices, $v_1 = v_1'$ and there exists an index j, such that $v_j \neq v_j'$, then this information is sufficient to recognise those cycles as non identical (Johnson has imposed as a further condition - following [Ti70]- that v_1 to be the least vertex of $v_1, v_2, \ldots, v_k$). The alternative that has been adopted in the present method consists of choosing for the start vertex, one that is likely not to produce many unfruitful explorations of other vertices, in the search for elementary cycles involving the start vertex. If v_1 is the start vertex and v_j is such that $(v_j, v_1) \in E$, then every exploration of v_j leads to a new elementary cycle, hence is not unfruitful. Therefore, the choice for the start vertex to be a vertex with maximal indegree among the vertices of the considered strongly connected component seems to be perhaps more appropriate. Observe that a similar choice could be made, as to which vertex to explore, among the vertices $v_2, (v_1, v_2) \in E$ and v_1 the start vertex. Also, extend this strategy to which vertex v_j to explore, among the vertices v_j such that $(v_{j-1}, v_j) \in E$, v_{j-1} being the vertex of the top of the stack and not having been deleted from it yet.

Next consider the digraph of figure 3.6, with N vertices, 2N-2 edges and N-1 elementary cycles. Johnson's algorithm would consider vertex 1 as start vertex, explore the path 1, ..., N, generate all elementary cycles of the digraph, but since this algorithm only considers cycles involving the start vertex, only the cycle 1,2,1 is enumerated, at this stage. Next, vertex 1 is deleted and a similar process occurs for the resulting subdigraph, with vertices 2, ..., N. Vertex 2 is the new start vertex, path 2,...,N is again reconsidered, and so on. It takes N(N-1) steps for enumerating all N-1 elementary cycles using the above strategy. The present algorithm would find all such cycles in the course of exploring the paths j,j+1,...,N and j,j-1,...,1, where j is the start vertex, consuming precisely 2N-2 steps, for termination. Digraphs of this class have the additional property that for any start vertex chosen, the present algorithm requires 2N-2 steps, whilst in [Jo73a] there is no possible choice of the start vertex for which the algorithm requires just O(N) steps.

Consider now the complete digraph K_n, with n vertices. Since a new elementary cycle exists with every possible exploration of a given vertex, any vertex is found unmarked, when reached, and this is true for both algorithms. Therefore, in the course of finding the elementary cycles involving the start vertex, all elementary cycles of K_n are generated, but [Jo73a] would only enumerate those with the start vertex. Assume now, a modified version of [Jo73a] with the marking system of the present algorithm incorporated. If T_n is the total number of steps required by the present algorithm to enumerate all elementary cycles of K_n then

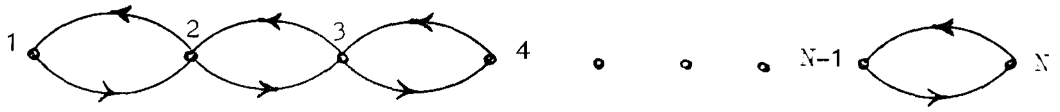


Figure 3.6

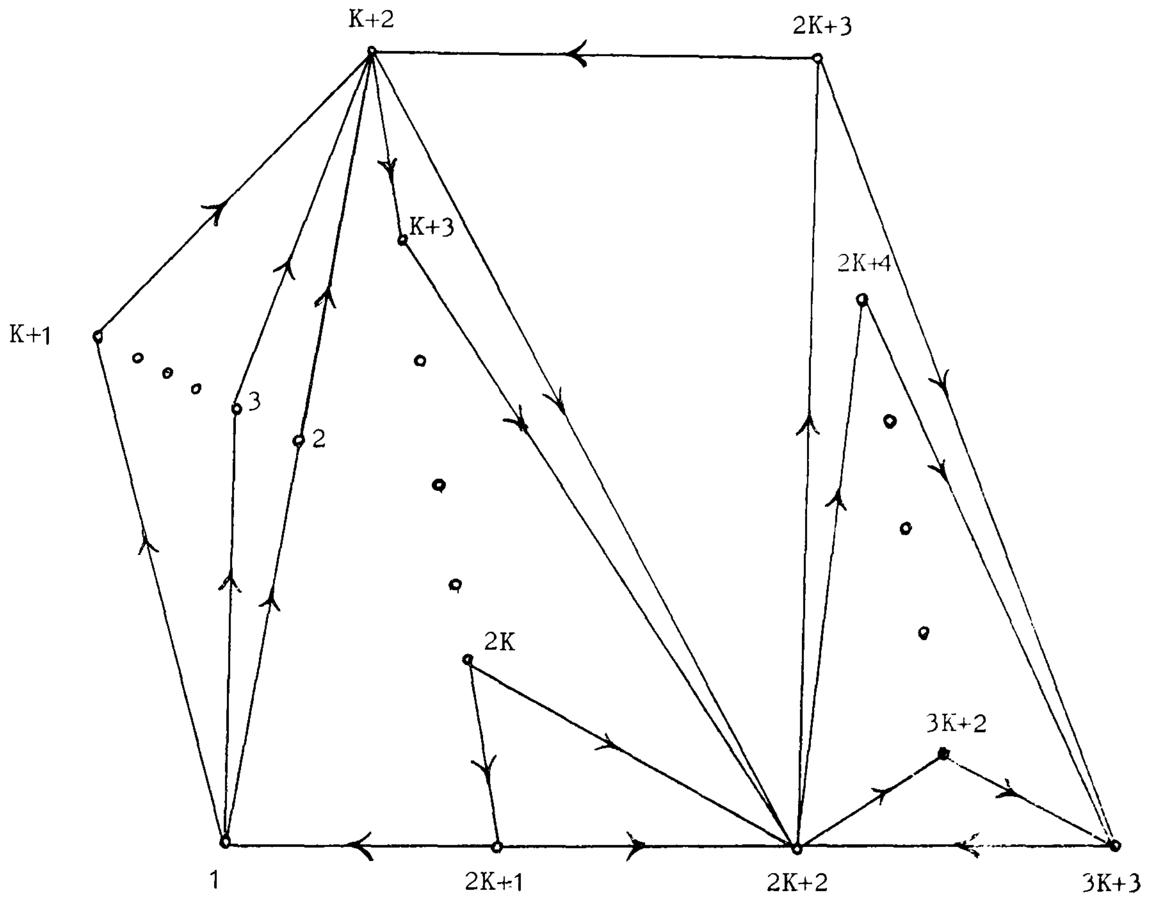


Figure 3.7

this modified version of [Jo73a] would require $\sum_{j=2}^{n} T_j$ steps, for the digraph K_n. Consequently, the total number of steps T_n' required by the actual Johnson's algorithm for enumerating all elementary cycles of K_n satisfies $T_n' > \sum_{j=2}^{n} T_j$, $n > 2$. Observe however, that $\sum_{j=2}^{n} T_j$ tends to T_n as n increases.

It should also be noted that in [Jo73a] it is stated that this algorithm unmarks a vertex v, only if appending v again to some elementary path is sure to lead to finding an elementary cycle, which includes the path followed by v. However, we observe that in both [Jo73a] and in the proposed algorithm, a vertex can be unmarked many times, without being involved in an elementary cycle when explored afterwards as can be seen in the digraph of figure 3.7. The example of figure 3.7 with 3K+3 vertices, 6K+2 edges and 3K elementary cycles, was shown by Johnson to be a worst case for Tarjan's algorithm. We can observe that both algorithms would unmark and explore each vertex of the subdigraph composed by vertices 2K+2, 2K+3, ..., 3K+3, for each elementary cycle existing with vertex 1 as start vertex, although no vertex of this subdigraph is involved in such cycles. For the enumeration of the 3K elementary cycles of this digraph, Johnson's algorithm requires $6K^2+11K-1$ steps, while the present algorithm requires $2K^2 + 6K$ or 7K+1 steps, depending on which vertex, K+2 or 2K+2 respectively, was chosen for the start vertex. Note that in this last fortunate case (vertex 2K+2 the start vertex), each edge of the digraph is explored just once during the entire process, with the exception of edge (3K+3, 2K+2) which is explored K times.

3.7 Fundamental set of cycles

We consider now the problem: Given an undirected graph, find a fundamental set of cycles of this graph. A strategy for solving this problem, commonly adopted by some existing methods, consists of performing the following steps:

(i) Find a spanning forest of the graph

(ii) Obtain the fundamental set of cycles, from the spanning forest, by successively considering edges from the graph, which do not belong to the forest.

Welch [We66] assumes the graph to be represented by an incidence matrix $B=(b_{ij})$. For each column j of the matrix if possible, a row i is chosen, such that i was previously unchosen and $b_{ij}=1$. If row i was chosen label the edge of the j-th column by i, and replace any other row k, such that $b_{kj}=1$, by the sum module 2, of rows i and k. This corresponds to the step (i) mentioned above. Each cycle of the fundamental set (step (ii)) is obtained by combining an unlabelled edge, corresponding to column j, with edges labelled k, such that $b_{kj}=1$. According to the analysis of Welch's algorithm by Gotlieb and Corneil [GoCo67], the time and space bound for this method are $O(N^2M)$ and $O(NM)$, respectively. Gotlieb and Corneil have also presented a modified version of Welch's algorithm, which improved running time whilst adding some extra space. However, the modifications introduced did not alter the most significant figures in the expressions of time and space bounds, and those remained the same: $O(N^2 M)$ and $O(NM)$, respectively.

The algorithm [GoCo67] operates with the graph G, assumed to be connected, represented by an adjacency matrix $A=(a_{ij})$. Like [We66], this algorithm also performs the steps (i) and (ii) explicitly. For obtaining a spanning tree, the algorithm first constructs a N x N matrix $B=(b_{ij})$, such that: for $i \leq j$, $b_{ij}=1$ if $a_{ij}=1$ and $a_{ik}=0$, for all k $i \le k < j$, and otherwise $b_{ij} = 0$. For $i > j$, it is defined $b_{ij} = b_{ji}$. Matrix B corresponds to an adjacency matrix of a forest, which is a partial subgraph of G. Then a T x N matrix $C = (c_{ij})$ is constructed where T is the number of trees in that forest. This matrix is defined as follows: $c_{ij} = 1$ if vertex j belongs to tree i, and $c_{ij} = 0$ otherwise. Vertex 1 is defined as belonging to tree 1 and if vertex k belongs to tree j, then clearly, all vertices reachable from k in B, also belong to tree j. The next step consists of transforming B into an adjacency matrix of a spanning tree of G by adding T-1 edges to it. This is accomplished by examining the T-1 rows of C, with fewest 1's. There must be one edge for each of these T-1 rows which joins the tree corresponding to this row to another tree in the forest. Therefore, when examining row i, if $c_{ij} = 1$ with $a_{jk} = 1$ and $b_{jk} = 0$ then edge (j,k) is added to B, i.e. we set $b_{jk} = b_{kj} = 1$. The spanning tree is now complete. Each edge that is now added to matrix B will produce a cycle in the fundamental set. However, note that the cycles ought actually to be traced back from B since, the simple addition of the edge in the matrix does not make the cycle explicit. Clearly, if the graph is not connected, then the algorithm is applied separately to each of the connected components. Gotlieb and Corneil have presented a detailed analysis of the algorithm from which it is deduced that the time and space bounds are $O(N^3M)$ and $O(N^2)$, respectively.

The algorithm by Paton [Pa69] also utilises the graph represented by an adjacency matrix $A=(a_{ij})$ but constructs the spanning tree using links from each vertex to its ancestor in the rooted version of the tree. Unlike the two previous methods, in [Pa69] step (ii) - obtaining the fundamental cycles - is performed in parallel with step (i) - finding a spanning tree. The idea of the algorithm is as follows: The first vertex of the graph is considered to be the root of the spanning tree and each vertex in this tree is examined once. When one vertex i is examined, all vertices j such that $a_{ij} = 1$ are considered. If j is already in the tree, then a cycle of the fundamental set has been detected. This cycle consists of the path from j to i in the tree, plus the edge (i,j). Otherwise, if j is not in the tree, then j is added to the tree, with the link corresponding to vertex j pointing to i, the ancestor of j. After the examination of edge (i,j), a_{ji} is set to zero to avoid considering this edge twice. After all the edges incident to i have been examined, a new vertex already in the tree but not yet considered, is chosen. Paton points out that the best method for selecting this new vertex is the last element method, which consists of always selecting a non examined vertex which entered the tree last. This method has the advantage of simplifying the task of tracing the cycles which have been detected. From Paton's analysis of the algorithm, we conclude that both the time and space bounds are $O(N^2)$. The time bound includes the input of the graph and the generation of the cycles, but not its output. The actual output requires O(NM) time, since there are O(M) cycles in a fundamental set, each cycle with O(N) vertices. A modification to this algorithm has been suggested by Jovanovich [Jo74] which requires N cells less of memory.

3.8 The proposed algorithm

The basic idea of our algorithm for finding a fundamental set of cycles is similar to the three methods described, namely to find a spanning forest of the graph and obtain the fundamental set of cycles by successively considering edges of the graph which do not belong to the forest. Like Paton's algorithm, we detect the cycles concurrently with the construction of the forest.

The strategy of the present algorithm is based on two main points. First, since any spanning tree corresponding to a given connected component of the graph, may be used to find the fundamental set of cycles of this component, the simpler and more efficient the method for finding such a tree, the better the process. Second, since our aim is to generate a fundamental set of cycles and obtaining the spanning forest is just a step towards that aim, we do not need to construct the forest explicitly. Instead, when a given vertex v of the digraph is being considered, we only store, the information that is relevant to our purpose, namely the path from v to the root of the spanning tree under consideration.

In relation to the first of the above points, observe that a depth-first search of the graph, as stated by Tarjan [Ta72] produces a spanning tree in O(N+M) time. Starting from the algorithm [Ta72] we add a mechanism for detecting the fundamental set of cycles and also, we simplify the process slightly by avoiding the explicit construction of the tree.

Our variation of Tarjan's depth-first search algorithm uses a stack for storing dynamically the paths to the root of the tree. We represent the graph as a set of adjacency lists A, with each edge (v, w) of the graph represented twice, namely vertex v in A(w) and vertex w in A(v). A vector <u>position</u> is also used. Before a vertex v is examined, we have position(v) = 0. If v belongs to the stack and is its t-th vertex from the bottom, then position(v) = t. When v leaves the stack, position(v) is set to N and remains unchanged until the end of the process. The algorithm makes use of a recursive backtracking procedure BASIS which attempts to extend the path kept in the stack. Suppose that vertex v is at the top of the stack and edge (v,w) is reached. The following cases may occur:

(i) If position(w) = 0 then necessarily w is not in the stack and w has not been examined yet. A call BASIS(w) will occur which will insert w in the stack, so extending the path under examination. An edge incident to w is then examined.

(ii) If position(w) = N then w has already been explored before and has been deleted from the stack. Since a vertex only leaves the stack after all the edges incident to it have been explored, we conclude that edge (w,v) has already been considered when w was at the top of the stack and v was underneath w, in the stack. Therefore, we can now disregard this edge and choose another edge incident to v, for examination.

(iii) If position(w) = t-1, where t is the number of vertices in the stack, then w is present in the stack, immediately underneath v. This means that v has been inserted in the stack during the computation of BASIS(v), whose call occurred while exploring edges incident to w. Thus edge (v, w) has already been considered before and can now be disregarded. Another edge incident to v is then selected.

(iv) If 0 < position(w) < t-1 then necessarily w is somewhere in the stack, not immediately below v. Hence this is the first appearance of the edge (v,w) and a cycle of the fundamental set has been detected. This cycle can be considered at once and no extra work is required to trace it. The cycle consists of w, the vertices above it in the stack, then w. Again, the next edge incident to v is selected.

After the examination of all edges incident to v, vertex v is deleted from the stack and the algorithm backtracks to the vertex immediately below v, in the stack. When the vertex at the bottom of the stack is deleted, a fundamental set of cycles, corresponding to the connected component of the graph which this vertex belongs to, has been obtained. Another vertex not yet explored, becomes the root of the spanning tree of the new connected component and so on, until all such components are considered. Observe that case (i) above corresponds to the exploration of an edge that is part of a spanning tree. Case (iv) corresponds to an edge that does not belong to a spanning tree in the forest and which produces a cycle in the fundamental set to be enumerated. Cases (ii) and (iii) correspond to the second instance of the exploration of an edge. Such exploration occurs since every edge is represented twice in the adjacency lists of the digraph. It follows from these observations that if the input graph has N vertices, M edges and K connected components, then exactly N-K edges are explored in case (i), a total of M edges are explored in (ii) and (iii), and exactly M-N+K are examined in case (iv).

Below we present an ALGOL-like formulation of the strategy.

ALGORITHM 3.2

```
begin comment an algorithm for obtaining a fundamental set of cycles
              of an undirected graph;
   procedure  BASIS (integer value v);
   begin      insert v in the stack;
              t := number of vertices in the stack;
              position(v) := t;
              for w ∈ A(v) do
                 if position(w) = 0 then BASIS(w)
                 else if position(w) < t-1 then
                                   output cycle w to v from stack, then w;
              position(v) := N;
              delete v from the stack;
   end;
   read the digraph and construct the adjacency lists;
   for j := 1 until N do position(j) := 0;
   for j := 1 until N do
              if position(j) = 0 then BASIS(j)
end
```

3.9 Correctness

Let G(V, E) be a connected graph with N vertices, input to algorithm 3.2 (if the graph is not connected we can assume, without loss of generality that each connected component is handled separately):

Lemma 3.10

There exists a spanning tree T of G, such that, at any arbitrary point of the computation of the algorithm the content of the stack between any two vertices v and w which belong to the stack, corresponds to the path in T between these vertices.

Proof:

Consider the graph T(X, Y), where X is the subset of V, whose vertices are at sometime inserted in the stack and for $p, q \in X$, $(p,q) \in Y$ iff p and q occupy consecutive positions in any possible configuration of the stack through the process. By inspecting the algorithm, we conclude that a recursive call BASIS(w) can only occur from the computation of BASIS(v) if $w \in A(v)$ i.e. if $(v, w) \in E$. Therefore, $Y \subseteq E$ and T is a partial subgraph of G. Next we note that since the graph is connected, any of its vertices is reachable from the vertex at the bottom of the stack. Because every vertex is inserted in the stack after being reached for the first time, we conclude that all vertices of G are eventually inserted in the stack, i.e. V=X and T spans G. Finally, since the call BASIS(w) can only occur if w has not been present in the stack before and since the content S of the stack between w and a fixed vertex z below it, remains unchanged until w is deleted from it, we conclude that there is an unique path in T between w and z, given by S. Therefore T is a spanning tree of G.

Lemma 3.11:

Let (v,w) be an edge of G, which is not in the considered spanning tree T. Then when (v, w) is first explored, both v and w belong to the stack.

Proof:

Let (v,w) be first reached when v is at the top of the stack. If w does not belong then to the stack and has not been present in it before, we have position(w) = 0. Therefore w will be inserted in the stack, and the path between v and w in T, of lemma 3.10, will include edge (v, w) since these vertices are consecutive in the stack. This contradicts edge (v, w) not belonging to T. On the other hand, if w does not belong to the stack and has already been present in it before we have position(w) = N. Therefore, all edges incident to w had been explored sometime before, which contradicts the fact that edge (v, w) has not been considered yet. Thus w also belongs to the stack.

Theorem 3.3:

Each cycle of the fundamental set of G, corresponding to the considered spanning tree T, is enumerated exactly once.

Proof:

Let (v, w) be an edge of the digraph which does not belong to T and suppose (v, w) is first reached when exploring the edges of v. Then when this edge is examined, v is at the top of the stack and by lemma 3.11, w is somewhere underneath v, in the stack. Therefore, position(v) = t and 0 < position(w) < t, where t is the number of vertices in the stack. Since we are considering the first exploration of (v, w), we conclude that w is not immediately below v in the stack and therefore position(w) < t-1. Thus, the cycle of the fundamental set, corresponding to edge (v, w) is enumerated at least once. Now, after all edges incident to v have been explored, exactly once each, v is deleted from the stack and position(v) is set to N. Since w still belongs to the stack at that moment, w will eventually occupy the top position and another exploration of edge (v,w) occurs. However, because position(v) = N and N ≥ t, it follows position(v) > t-1 and therefore the cycle containing edge (v,w) is not enumerated again. When w leaves the stack, position(w) is set to N and since position(v) and position(w) are now both different from zero, v and w can not be explored again. Therefore, no re-exploration of edge (v,w) can occur and we conclude that the cycle of the fundamental set, corresponding to edge (v,w) is enumerated exactly once.

Theorem 3.4:

Only cycles belonging to the fundamental set, corresponding to a spanning tree T, are enumerated by the algorithm.

Proof:

Let w,...,v,w be a cycle enumerated by the algorithm. Since w,...,v is taken from the stack, by lemma 3.10 we conclude that w,...,v is the path in T, between w and v. Therefore all edges of w,...,v belong to the spanning tree. On the other hand, the edge (v,w) can not belong to T, because T has no cycles. Thus, w,...,v,w is a cycle with exactly one edge of the graph which does not belong to the spanning tree. Hence w,...,v,w belongs to the fundamental set of cycles, corresponding to T.

3.10 Performance:

Theorem 3.5:

Let G(V,E) be a connected graph with N vertices and M edges, input to algorithm 3.2. Then a total of O(N+M) space and O(N+M) time are required for the enumeration (without listing) of the cycles belonging to a fundamental set of the graph G.

Proof:

The space bound follows immediately from the fact that the representation of the graph by adjacency lists requires O(N+M) space and the remaining data structures require O(N) space. For the time bound, observe that a vertex v can only be explored if it has not been reached before, i.e. if position(v) = 0. Once v is reached, position(v) becomes different from zero and since there is no way of re-setting position(v) to zero, v can not be explored again. Since the exploration of a vertex produces the exploration of all edges incident to it, we conclude that the N exploration of vertices (once each), produces 2M explorations of edges (twice each). Therefore a total of O(N+M) time is required for enumerating (without listing) the cycles of a fundamental set.

As already mentioned in section 3.7 the explicit output of the cycles of the fundamental set requires O(NM) time. Therefore, if this explicit output is desired, then algorithm 3.2 takes O(NM) time for termination, although the actual generation of the cycles requires only O(N+M) time.

Alternatively, we can modify algorithm 3.2, so as to produce a reduced output of the cycles, with the aim of reducing the output time. Consider the graph of figure 3.8. If we assume that its vertices are in ascending order in the adjacency lists representation of the graph then algorithm 3.2 would implicitly find the tree of figure 3.9, as a spanning tree of the graph. Now let us consider the following alteration to algorithm 3.2, in which each vertex is output by the time it leaves the stack: if vertex v is being deleted from the stack and it occupied a position in the stack lower than vertex w (w being the vertex output immediately before v), then vertex v is output underlined ($\underline{v}$): if vertex v occupied a position in the stack higher than w, then vertex v is output with a bar ($\bar{v}$); if v and w occupied the same relative position in the stack or v is the first vertex in the sequence, then vertex v is output neither underlined or barred. With such a scheme, the output corresponding to the graph of figure 3.8 would be:

$$5 \quad 8 \quad \underline{4} \quad \bar{9} \quad \underline{7} \quad \underline{3} \quad 6 \quad \underline{2} \quad \underline{1}$$

The above sequence uniquely determines the tree of figure 3.9. The ancestor of any given vertex v (v not being the root) is the first underlined vertex w to the right of v in the sequence, such that the number of underlines - from the right of v until and including w - exceeds the number of bars. Now, let us consider the information concerning the cycles. If $v_1, v_2, \ldots, v_k, v_1$ is a cycle in the fundamental set, with v_i preceding v_{i+1} in the output sequence ($1 \le i < k$) then this cycle is perfectly characterized by that output sequence with underlines and bars and by the pair v_1, v_k. Therefore, we can simply add (v_k) to the sequence, immediately after v_1, with the parenthesis distinguishing the notation of the cycle from the occurrence of the actual vertex v_k, in the sequence. With this scheme of output, the fundamental set of cycles of the graph of figure 3.8 is represented by the sequence:

$$5(1)8(3)\underline{4}(2,1)\bar{9}(3)\underline{7}(2)\underline{3}(1)6(1)\underline{2}\ \underline{1}$$

Each vertex between parenthesis in the sequence corresponds to a cycle in the fundamental set. In the example above, therefore there are 8 cycles in the set. For obtaining the explicit form of say the first cycle in that example, we proceed as follows: the 5(1) in the beginning of the sequence, indicates that there is a cycle whose "first" vertex is 5 and "last" vertex is 1. Consequently, starting from 5 we successively find the ancestors of the vertices in

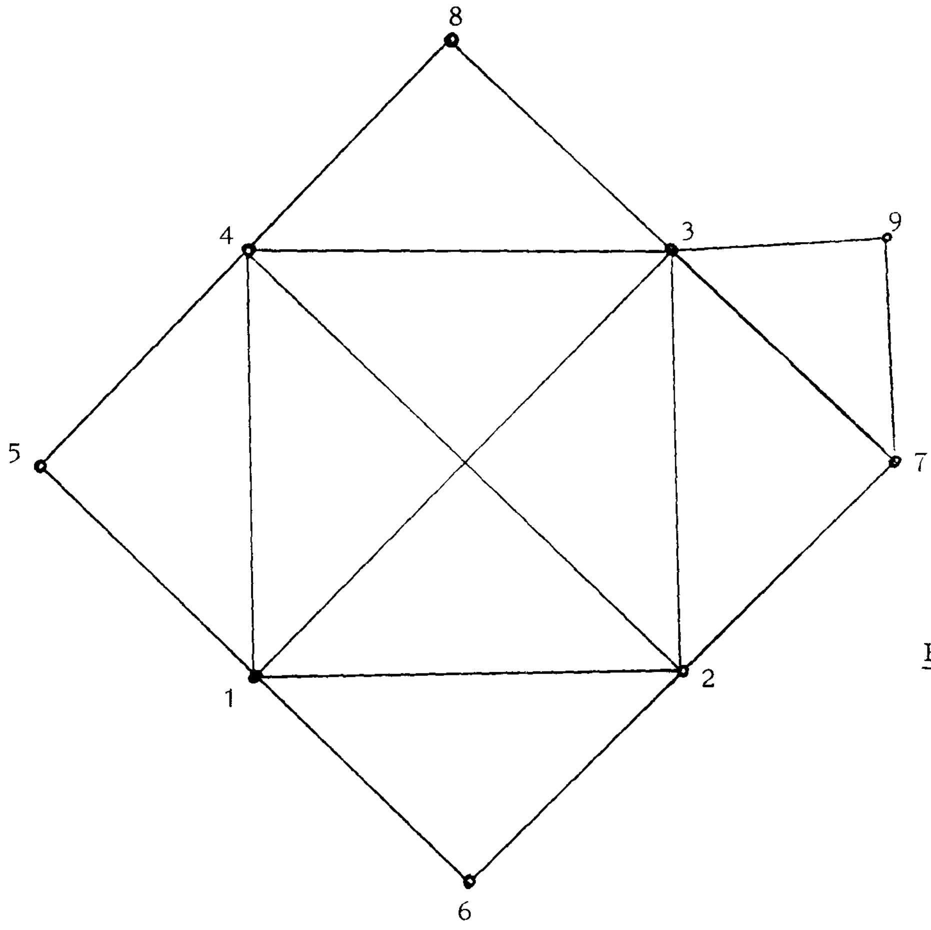


Figure 3.8

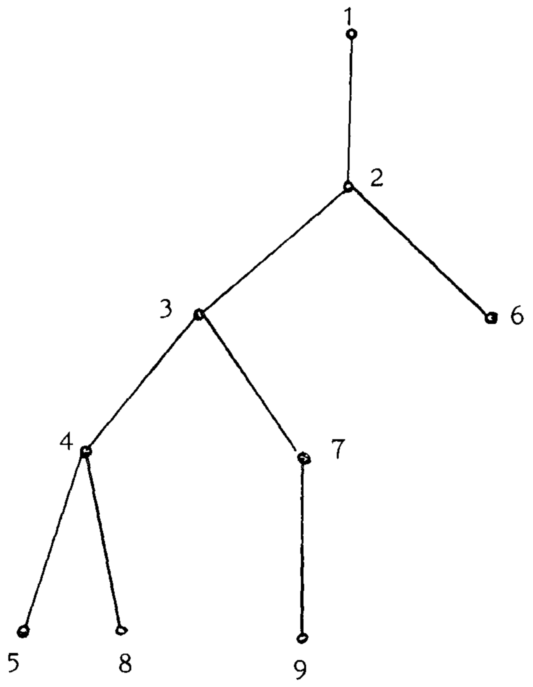


Figure 3.9

the cycle, until 1 is reached. The first vertex after 5, such that the number of underlines exceeds the number of bars is 4 and therefore 4 is the ancestor of 5. Similarly, we find 3 as the ancestor of 4; 2 the ancestor of 3; and 1 the ancestor of 2. Hence the desired cycle is 543215.

Using this technique, we can output a fundamental set of cycles in O(N+M) time. Alternatively, we can decide to eliminate from the output, the vertices that are involved in no cycles, which brings the output time bound to O(M). Consequently, the whole process of finding a fundamental set of cycles can be accomplished in O(N+M) time, which makes the proposed strategy optimal within a constant factor.

3.11 Elementary cycles in undirected graphs

Given an undirected graph G(V,E) we consider now the problem of finding the elementary cycles of the graph.

One possible approach to this problem consists of modifying an algorithm for finding the elementary cycles of a digraph (sections 3.2 to 3.6) to operate for undirected graphs. Basically, any elementary cycle finding algorithm for digraphs can be adapted to handle undirected graphs. This method is discussed in the next section.

Another way of solving the present problem consists of finding the elementary cycles of the graph, by computing the elements of the cycle vector space of the graph. The basic idea is first to obtain a fundamental set of cycles (sections 3.7 to 3.10). Next, the elements of the cycle vector space are computed, starting from that set. A test is performed to verify whether a newly computed element of the space is an elementary cycle and if so the cycle can be output. Welch [We66] has attempted to produce all elementary cycles of the graph, without considering all possible elements of the vector space, by conveniently ordering the cycles of the fundamental set. Gibbs [Gi69] has shown however, that the ordering of [We66] does not necessarily exist and as a consequence, Welch's method fails to enumerate all elementary cycles, in some cases. Hsu and Honkanen [HsHo72] have described a method for finding the cycles that [We66]misses. Prabhaker and Deo [PrDe74] also find the elementary cycles by trying to reduce the number of computations of elements of the vector space. However, as it is shown in [PrDe74], there exist worst cases for which this algorithm has an exponential time bound. In this same paper, it is pointed outthat all known methods for finding elementary cycles through the cycle vector space, have also exponential time bounds.

3.12 The proposed algorithm

Our strategy consists of modifying algorithm 3.1 for finding the elementary cycles of a digraph, to handle undirected graphs. This approach is justified by the fact that so far all atempts to produce a "pure undirected graph cycle finding algorithm", based upon computations of elements of the cycle vector space, have been shown to have an exponential time bound, as mentioned above.

Let G(V,E) be an undirected graph with N vertices and M edges. Consider the <u>digraph version</u> D of G, which is obtained by replacing each undirected edge of G, by two directed edges having opposite directions. Figure 3.11 illustrates the digraph version of the undirected graph of figure 3.10. Comparing the cycles of G and D, we observe that D contains all cycles of G, plus two additional classes of cycles, which are not present in G: The first is composed by

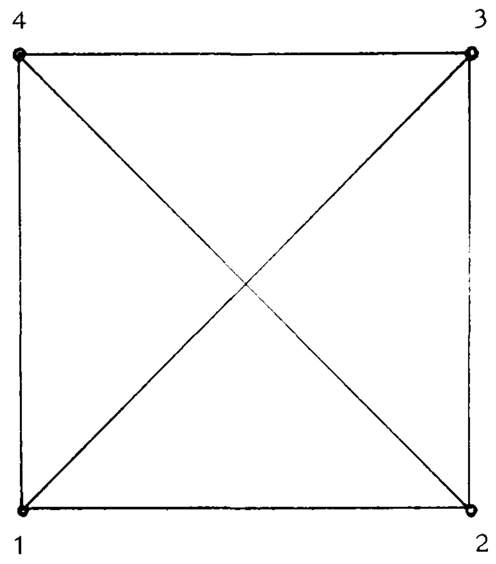


Figure 3.10

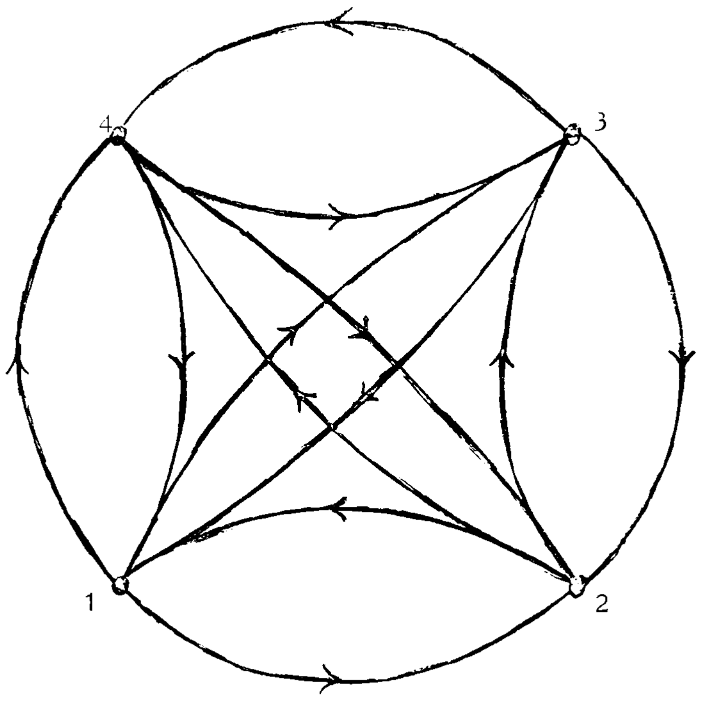


Figure 3.11

the cycles of D having exactly two edges. These cycles correspond to single edges in G and clearly there are precisely M such cycles. For characterizing the other mentioned class, consider an elementary cycle $c = v_1, v_2, \ldots, v_{k-1}, v_k, v_1$ $(k > 2)$, of G. Clearly, D also contains the cycle $c_1' = v_1, v_2, \ldots, v_{k-1}, v_k, v_1$ and since (v_i, v_j) and (v_j, v_i), $1 \leq i, j \leq k$ are distinct edges in D, $c_2' = v_1, v_k, v_{k-1}, \ldots, v_2, v_1$ and c_1' are distinct cycles in D. However, both cycles c_1' and c_2' of D correspond to the single cycle c of G. We name c_1' and c_2' as <u>opposite cycles</u> each to the other. For example, the cycle 12341 of the undirected graph of figure 3.10 corresponds to the opposite cycles 12341 and 14321 of the digraph of figure 3.11. Thus, if G contains C elementary cycles D will contain the total of $C' = 2C + M$.

Now let us suppose that an algorithm for finding the elementary cycles of a digraph is applied to D, aiming to obtain the cycles of G. The problem that arises when proceding so is concerned with those two classes of cycles which are in D, but not in G. The basic alterations required, in order to make the algorithm operate correctly, consist of finding a suitable way of detecting these classes of cycles and avoiding their output. The cycles composed by exactly two edges can be easily recognised, simply by testing the number of edges in each newly generated cycle. As for the other class of unwanted cycles, we wish to find a way of detecting that a newly generated cycle c_2' of D is the opposite cycle of another cycle c_1' of D, which either has already been generated or which eventually will be generated. The constraint that we impose on the method of checking opposite cycles, is that the mechanism of detection should <u>not</u> increase the time bound, i.e. the time bound of the algorithms that find elementary cycles in directed and undirected graphs should be the same. In order to describe this mechanism of detection, it is convenient to represent every cycle of G as $v_1, v_2, \ldots, v_k, v_1$ $(k>2)$, with v_1 fixed and $v_2<v_k$. The elementary cycles of the graph of figure 3.10, using this representation, are 12341, 12431, 13241, 1231, 1241, 1341 and 2342. Using this representation we can divide all elementary cycles $v_1, v_2, \ldots, v_k$, v_1 of D, with more than two edges, into two sub-classes: those with $v_2 < v_k$ and those with $v_2 > v_k$, since $v_2 \neq v_k$ because $k > 2$. Clearly, the opposite cycle of a cycle of one of these sub-classes, belongs necessarily to the other sub-class. Therefore our problem of generating duplicate cycles can be solved simply by testing whether $v_2 < v_k$ or $v_k > v_2$ in each newly generated cycle and rejecting it if the latter is satisfied, for instance. Clearly to each accepted cycle there corresponds a rejected one and a cycle can be accepted either before or after the generation of its opposite rejected one.

The implementation of this mechanism in algorithm 3 .1 is straightforward. We replace the following block of that algorithm

```
begin
          output cycle w to v from stack, then w;
          f := true
end
```

by the following:

```
begin
          z := vertex immediately above w in the stack;
          if z < v then output cycle w to v from stack, then w;
          f := true
end
```

The above implementation is justified by the fact that every newly generated cycle $v_1, v_2, \ldots, v_k, v_1$ corresponds to the configuration of the stack being $\ldots v_1 v_2 \ldots v_k$, with v_k at the top. Therefore, in terms of variables of algorithm 3.1, variable v contains the value of v_k, variable w contains the value of v_1 and consequently the vertex above w in the stack which is assigned to variable z, corresponds to v_2. Observe also that storing the stack in sequential allocation form, seems to be advantageous since z can be immediately determined in this case, by z := stack(position(w) +1). Finally, we mention that in practical terms, one single comparison has been added to algorithm 3.1. This single test also solves the problem of rejecting the cycles which have exactly two edges, because for these cycles $v_2 = v_k$ and consequently $z = v$.

3.13 Correctness

The correctness of the proposed strategy follows directly from the correctness of algorithm 3.1 and from the observations of the previous section, concerning the introduction of the mechanism for detecting and rejecting cycles of the digraph version of the given undirected graph G, which do not belong to G.

3.14 Performance

Theorem 3.6:

Let G be an undirected graph with N vertices, M edges and C elementary cycles. Then algorithm 3.1, with the modifications of section 3.12 incorporated, enumerates the elementary cycles of G in $O(N+(C+1)M)$ time and $O(N+M)$ space.

Proof:

The space bound is obvious. For the time bound, consider the digraph version D of G. According to sections 3.5 and 3.12, the C' = 2C+M elementary cycles of D are generated in at most O(N+(C'+1)M) time. Now, let us divide the time required for generating the C' = 2C + M cycles of D, into two parts: the time needed for generating those 2C elementary cycles and that needed for those M cycles. Because the latter M cycles correspond to the cycles of D which have exactly two edges, the time required for generating them is not greater than O(N+M). On the other hand, O(N+(C+1)M) time is required for the generation of those 2C elementary cycles. Consequently we conclude that the total time bound is O(N+(C+1)M).

3.15 Conclusions

We have presented in this chapter, algorithms for finding the elementary cycles in directed and undirected graphs as well as an algorithm for finding a fundamental set of cycles of an undirected graph.

The latter algorithm consists basically of performing a depth-first search of the graph and requires O(N+M) time for generating the cycles, excluding the time required for their output. The explicit output of the cycles consumes O(NM) time. However, an alternative reduced form of listing the cycles has been also presented, which requires O(N+M) time. The proposed solution is optimal, within a constant factor. The algorithm by Paton [Pa69] - which is the best extant algorithm for solving this problem - although it can easily be modified for generating the cycles (with no output) in O(N+M) time, as presented in [Pa69] consumes $O(N^2)$ time for this task, plus the usual O(NM) time for the explicit output of the cycles. Also another difference between our algorithm and [Pa69] is that the latter is not a depth-first search algorithm. This contradicts what is reported in [Jo74].

The proposed strategy for finding the elementary cycles of an undirected graph was obtained by adapting our algorithm for the elementary cycles of a digraph. It should be noted that the modifications introduced in the latter, in order to make it operate for undirected graphs, did not increase the overall time bound.

The proposed algorithm for the elementary cycles of a digraph is based on work done by Tiernan-Tarjan-Johnson. Although its (worst case) time bound is similar to that achieved by Johnson, namely O(N + M) per cycle, we believe that the techniques for detecting an elementary cycle anywhere in the path under consideration and its enumeration as soon as the cycle is contained in this path - which were used in our proposed algorithm - represent some important features for cycle finding methods.

The present chapter has shown examples where unnecessary work was done by some existing algorithms for finding elementary cycles in digraphs. The question that arises is : what about inefficiencies of our proposed algorithm? Clearly they still exist because a vertex or an edge may be unsuccessfully explored many times during the process. However, these <u>same</u> inefficiencies are also present in the existing backtracking methods. Since we have eliminated some of the inefficiencies of those methods, we believe that our proposed algorithm compares favourably with them.

It should be noted that a previous version of algorithm 3.1 [SzLa74] was an unsatisfactory attempt to devise a method that would explore unsuccessfully any vertex, at most once during the entire process. An open question still remains about the existence of an algorithm that would find all elementary cycles of a digraph, in such a way that any edge or vertex would be unsuccessfully explored at most a constant number of times during the entire process. Such an algorithm would have an optimal time bound.

CHAPTER 4

SHORTEST PATH PROBLEMS IN ACYCLIC DIGRAPHS

4.1 Introduction

Shortest path problems constitutes an important area in graph algorithms, mainly because there is a wide range of different applications which make explicit use of such algorithms. Probably due to this fact it has received much research attention. In fact, an efficient solution for the shortest path between two given vertices of a digraph was devised as early as 1959 by Dijkstra [Di59]. Observe that this problem has no interest from the "pure mathematical" point of view, where efficiency is not considered. For, the following simple algorithm solves the problem:

```
begin
    consider all paths between the two given vertices;
    choose the path with minimal length
end
```

In this chapter we deal with shortest paths problems in acyclic digraphs, with weighted edges. Obviously, algorithms for solving these problems in general (not necessarily acyclic) digraphs, would also operate correctly the acyclic ones. However, if the digraph contains no cycle some shortest path problems admit of more efficient algorithms. Furthermore, acyclic digraphs consitute an important class of digraphs, with many specific applications.

Each of the sections of this chapter handles a different problem related with finding shortest paths in acyclic digraphs. Section 4.2 contains an algorithm for solving the shortest path problem between two given vertices. The extensions of this algorithm to find the shortest paths from all vertices to a fixed vertex and from a fixed vertex to all other vertices are described in sections 4.3 and 4.4, respectively. A further extension to find the shortest paths between every pair of vertices is presented in section 4.5. The problem of finding a shortest path between two given vertices, visiting some specified vertices is the subject of 4.6. Section 4.7 contains an algorithm for the k-shortest paths from all vertices to a fixed vertex. k-shortest paths from one vertex to all others, between every pair of vertices and between two specified vertices are handled in sections 4.8, 4.9 and 4.10, respectively. An algorithm for the longest path in an acyclic digraph is presented in section 4.11 and the k-longest path in such a digraph is considered in 4.12. Some further remarks form the last section.

The strategy for solving a shortest path problem - and the time bound of the corresponding algorithm - may vary according to whether or not the weights assigned to the edges assume negative values. For instance, there exists an algorithm to find the shortest path between two given vertices of a digraph (possibly with cycles) in $O(N^2)$ time, only if all weights are non-negative. A corresponding algorithm that operates for digraphs with negative weights allowed requires O(NM) time. If the digraph contains no cycles such a difference is known not to exist. Therefore, unless otherwise stated (sections 4.11 and 4.12), the weights of the considered acyclic digraphs may assume any real value.

4.2 Shortest path between two given vertices

Given a directed graph D(V,E) with weights d_{ij} assigned to its edges, the problem consists of finding a path from a to b which minimizes the sum of the weights of their edges. Dreyfus [Dr69] has surveyed and discussed a number of algorithms for solving this and other related problems. The method by Dijkstra has a time bound of $O(N^2)$ and was devised for digraphs with non-negative weights. Algorithms were also presented or discussed by Nicholson [Ni66], Boothroyd [Bo67], Dantzig [Da63], among others.

An algorithm for specifically finding a shortest path between two vertices of an acyclic digraph was presented by Elmaghraby [El70]. It uses a "distance matrix" (a_{ij}) for representing the digraph, where if $(i,j) \in E$ then $a_{ij} = d_{ij}$ otherwise a_{ij} is infinite. A pre-pass is performed when a topological sorting arrangement of the vertices of the digraph is obtained. This topological sorting is used for rearranging the distance matrix, so that it becomes upper-triangular. Now the length of the shortest path from vertex 1 to vertex k is computed as follows: label vertex 1 as $\alpha_1 = 0$ and at any step j consider the set of vertices i, such that $(i,j) \in E$; the label α_j of vertex j is found by calculating

$$\alpha_j = \min(\alpha_i + d_{ij}).$$

When vertex k is finally labelled, α_k is the length of the shortest path. For determining the shortest path itself, another pass is performed as follows: for each vertex j, determine vertex i, such that $(i,j) \in E$ and

$$\alpha_i + d_{ij} = \alpha_j \quad , \quad \text{for } j = k,k-1,\ldots,2.$$

The analysis of this algorithm is straightforward. As it stands, the algorithms requires $O(N^2)$ time and space for termination. This follows from the fact that $O(N^2)$ time is required for each of the three distinct passes of the algorithm, namely, the topological sorting pass, the computation of the length of the shortest path and the tracing back pass. A similar algorithm can be found in [Wa70]. It should be pointed out that a simple change in the representation of the digraph - by adopting the adjacency lists representation - can alter the time and space bounds to O(N+M), if the corresponding change in the strategy is performed. Such an algorithm, with the latter bound, was presented by

Johnson [Jo73]. Observe that the bound O(N+M) is realized in each of the three distinct passes of the method.

Our present algorithm uses a backtracking recursive procedure that performs a depth-first search of the digraph. At the end of this search, the shortest path from vertex a to vertex b is determined. The adjacency lists representation is used and the weights are also stored in this list: d_{ij} is supposed to be part of the node which contains vertex j, in A(i). The vectors route and length are also used, so is the boolean vector mark, all of size N. At the beginning mark(v) is set to false, for all v,$1 \leq v \leq N$. When vertex v is reached mark(v) becomes true, remaining so until the end of the process. The content of length(v) equals finally the length of the shortest path from v to vertex b, if there is one, and infinity, otherwise. The vector route is used to keep an updated version of the shortest path itself, so that the tracing back of the path does not require the examination of the whole digraph again. If vertex v reaches b and v is reachable from a then at the end route(v) contains a link to a vertex z,such that a shortest path from v to b is v,z,...,b.

The algorithm proceeds as follows: consider the case in which vertex v,$v \neq b$, has been reached for the first time. Then all edges from v will be explored. Assume edge (v,w) is reached.

(i) If w has not been explored yet then mark(w) = false and a call of the recursive procedure PATH(w) occurs. On returning of this computation, the content of length(w) equals the length of the shortest path from w to b and route(w) contains the vertex following w in this path if there is one or zero otherwise. Therefore if length(w) + $d_{vw} <$ length(v) then length(v) is set to length(w) + d_{vw} and route(v) is set to w.

(ii) If w has already been explored before then mark(w) = true and an action similar to (i) is undertaken except that no call of PATH(w) is invoked.

After the last edge from v is explored, length(v) and route(v) contain respectively the length of the shortest path from v to b and the value of the vertex following v in this path or zero if no such path exists. The algorithm then backtracks to the vertex in whose exploration the call PATH(v) was invoked and so on. In the initialization of the process mark(b) is set to true. Therefore no edge from b can be explored as it is known that they do not lead to the shortest path from a to b. Thus the depth-first search is not necessarily completed at the end of the process.

The following is an ALGOL-like formulation of the algorithm. The length of the desired shortest path is stored at the end, in length(a). The shortest path itself is contained in the route vector: the first vertex is a; the vertex following any vertex v, $v \neq b$, is route(v) and the final vertex is b.

ALGORITHM 4.1

```
begin comment an algorithm for the shortest path from vertex a to
          vertex b is an acyclic digraph;
    procedure PATH (integer value v);
    begin mark(v):=true;
        for w ∈ A(v) do
        begin if ¬mark(w) then PATH(w);
            if length(w) + $d_{vw}$ < length(v) then
            begin length(v):=length(w) + $d_{vw}$;
                route(v):=w
            end
        end
    end  PATH;
    integer a,b;
    read  the digraph and construct the adjacency lists A;
    read the values of a and b;
    for j:=1 until N do
    begin mark(j):=false;
        length(j):=infinity;
        route(j):=0
    end
    mark(b):=true;
    length(b):=0;
    PATH(a)
end
```

The correctness of the proposed method can be verified by the following lemmas:

Let $D(V,E)$ be an acyclic digraph with weights d_{ij} assigned to its edges and $a,b \in V$. Assume D is input to algorithm 4.1:

<u>Lemma 4.1</u>

If $v_1, v_2, \ldots, v_k$ ($a=v_1$, $b=v_k$) is a shortest path from a to b in D, then at the end of the process, length(v_i) contains the length of the shortest path from vertex v_i to v_k, $1 \leq i \leq k$.

<u>Proof</u>:

By induction on decreasing i. For i=k the initialization of the algorithm sets length(v_k)=0 and mark(v_k)=<u>true</u>. The former is the correct value of the length of the shortest path from v_k to itself. The latter prevents length(v_k) to be altered during the process, which completes the proof for the base. By the induction hypothesis we assume that at the end of the process length(v_i) $2 \leq i \leq k$ contains the value of the shortest path from v_i to v_k. Now consider the exploration of vertex v_1. When edge (v_1, v_2) is eventually reached, if mark(v_2)=<u>false</u> a call of PATH(v_2) occurs. On returning, length(v_2) contains its final value in the process, hence the length of the shortest path from v_2 to v_k. The algorithm then compares length(v_2) + $d_{v_1 v_2}$ with length(v_1), which contains the value of the length of a previous path from v_1 to v_k. If the first of these values is the smallest the algorithm assigns it to length(v_1). Otherwise these values are equal and no action is taken. If mark(v_2)= <u>true</u> no call of PATH(v_2) is invoked and length(v_2) contains already its final value, since the digraph is acyclic. A similar comparison and action as above is undertaken. In any case, after the exploration of edge (v_1, v_2), length(v_1) contains the value of length(v_2) +$d_{v_1 v_2}$, hence the length of the shortest path from v_1 to v_k. Also because of this fact, length(v_1) is not altered anymore after the exploration of (v_1,v_2) which completes the proof.

Lemma 4.2:

At the end of the process, route(v_1)=v_2, route(v_2)=v_3,..., route(v_{k-1})=v_k and route(v_k)=0, where $v_1,v_2,\ldots,v_k$ ($a=v_1$, $b=v_k$), is a shortest path from a to b, in D.

Proof:

The proof is similar to that of lemma 4.1.

Theorem 4.1:

Algorithm 4.1 is correct.

Proof:

Lemmas 4.1 and 4.2.

The performance of the algorithm is verified by the following theorem.

Theorem 4.2:

Let D(V,E) be an acyclic digraph, having N vertices and M weighted edges, input to algorithm 4.1. Then it is required O(N+M) space and time, for finding a shortest path from a to b, $a,b \in V$.

Proof:

The space bound is obvious. For the time bound observe that the marking mechanism ensures that a vertex is explored at most once. Since the exploration of an edge (v_i,v_j) can only occur when vertex v_i is being explored, we conclude that any edge is also explored at most once. Therefore, O(N+M) is a time bound, for the method.

Corollary 4.1:

Let D(V,E) be an acyclic digraph, with weighted edges, having N vertices and $a,b \in V$. Define $Z_{a,b} \subseteq V$ and $W_{a,b} \subseteq E$, by:

$Z_{a,b} = \{v_i \in V$, such that $v_i \neq b$ and v_i is reachable from a through a path that does not contain b$\}$

$W_{a,b} = \{(v_i, v_j) \in E$, such that $v_i, v_j \in Z_{a,b}\}$.

Then, excluding the input of D, the program requires $O(N+|W_{a,b}|)$ time for termination and this bound is attained.

Proof:

This can be verified by the following: the backtracking search ensures that all vertices and edges which are not reachable from a are not explored. Also, all vertices and edges which are reachable from a only by a path containing b, are not explored because mark(b) is set to true in the initialization which prevents their exploration. Therefore, $O(|Z_{a,b}|+|W_{a,b}|)$ time is required for the computation of procedure PATH. Since the initialization of the process requires O(N) time, we conclude that $O(N+|W_{a,b}|)$ is the total time bound. Since all vertices of $Z_{a,b}$ are explored, we conclude that this bound is attained.

The present algorithm approaches the problem in a different way from the other algorithms mentioned: when computing the shortest path from a to b, the paths are constructed from b backwards a, i.e. if v is a vertex reachable from a, such that v reaches b, then the algorithm computes the length of the shortest path from v to b, instead of computing it from a to v. Also, note that the present method avoids the computation of any additional pass. In particular, no computation for topological sorting is required.

4.3 Shortest paths from all vertices to a given vertex

Let D(V,E) be an acyclic digraph, with weighted edges d_{ij} and b a chosen vertex. The problem consists of finding the shortest paths from all vertices, to vertex b.

Only small changes are required in the algorithm of the last section which finds the shortest path from a vertex a to a vertex b to transform it into an algorithm for finding the shortest paths from all vertices to vertex b.

The modification consists of maintaining the same procedure PATH, as in algorithm 4.1, but with a different invoking system. We compute the set of source vertices and afterwards find the shortest path from each vertex of this set to vertex b.

Suppose $\{s_1,\dots,s_k\}$ is the set of source vertices. We first find the shortest path from s_1 to b, using a process similar to that described in the previous section. Next, vertex s_2 is considered and the objective is to find the shortest path from s_2 to b. Suppose vertex v is a vertex reachable from both, s_1 and s_2. Then, at that stage, v would have already been explored (mark(v)=<u>true</u>). This means that the shortest path from v to b has already been calculated and there is no need to recompute it again. Clearly, the same applies to all vertices reachable from v. Therefore, at each stage j, when computing the shortest path from s_j to b, the only vertices that ought to be explored are those reachable from s_j through a path that does not contain b, but which are <u>not</u> reachable from s_i, for $1 \leq i < j$, also through a path that does not contain b.

The following is an ALGOL-like description of this method:

ALGORITHM 4.2

```
begin comment an algorithm for finding the shortest paths from all
          vertices to vertex b in an acyclic digraph;
    procedure PATH (integer value v);
    begin
        ...
        ...      the same as in algorithm 4.1
        ...
    end PATH;
    integer b;
    read the digraph and construct the adjacency lists;
    read vertex b;
    for j:=1 until N do
    begin mark(j):=false;
          length(j):=infinity;
          route(j):=0;
    end
    length(b):=0;
    mark(b):=true;
    find the set $S_0$ of source vertices;
    for j $\in$ $S_0$ do PATH(j)
end
```

At the end of the process, length(v) contains the length of the shortest path from v to b, for all v,$1 \leq v \leq N$. Also, route(v) contains the vertex following v, in a shortest path from v to b. If there is no path from v to b, then length(v)=infinity and route(v)=0. Observe that all shortest paths are stored in the single vector route, which corresponds in fact to a representation of a rooted tree, with each vertex having a pointer to its ancestor.

The proofs of correctness are similar to those of the previous section. The same applies to the proof of performance - and algorithm 4.2 is bounded by O(N+M) space and time, being optimal to within a constant factor.

4.4 Shortest paths from a given vertex to all vertices

Let D(V,E) be an acyclic digraph, with weighted edges and $a \in V$, a chosen vertex. The problem is to compute the shortest paths from a to all vertices of D. This problem is similar to that of the previous section and in fact, it can be reduced to it, by adopting the following strategy.

Define the converse digraph $\overline{D}$ of D, by inverting the directions of the edges of D, i.e. $\overline{D}(V,E')$ has $(v,w) \in E'$ iff $(w,v) \in E$, for all $v,w \in V$ and the weight of (v,w) in $\overline{D}$ is the same as the weight of (w,v) in D. Now, apply the algorithm of the previous section for finding the shortest paths from all vertices to vertex a, in $\overline{D}$. This solves the problem because the shortest paths from all vertices to vertex a in $\overline{D}$ correspond to the shortest paths from a to all vertices in D. The following lemma proves the correctness of this assertion.

Lemma 4.3:

Let D(V,E) be an acyclic digraph with weighted edges, $\bar{D}$ its converse digraph and a $\in$ V. If $v_1, v_2, \ldots, v_{k-1}, v_k$ (v_k=a) is a shortest path from v_1 to v_k in $\bar{D}$, then $v_k, v_{k-1}, \ldots, v_2, v_1$ is a shortest path from v_k to v_1 in D.

Proof:

The length of the path $v_1, v_2, \ldots, v_{k-1}, v_k$ in $\bar{D}$ is the same as the length of $v_k, v_{k-1}, \ldots, v_2, v_1$ in D. Therefore, if there exists another path $v_k, w_j, \ldots, w_1, v_1$ in D, with a smaller length, then the path $v_1, w_1, \ldots, w_j, v_k$ in $\bar{D}$ is shorter than $v_1, v_2, \ldots, v_{k-1}, v_k$, which contradicts the hypothesis.

As for the performance of the present solution, observe that inverting the directions of the edges of a digraph is an O(N+M) time operation - if adjacency lists are used. Therefore the space and time bounds remain O(N+M). Note also that if no copy of the representation of the digraph D is required, we could construct directly the digraph $\bar{D}$ from the input. In this case no pre-pass would be required.

4.5 Shortest paths between every pair of vertices

Given an acyclic digraph D(V,E), with weighted edges the problem consists of finding the shortest path between every pair of vertices of D.

Several algorithms are known that solve the problem for general (not necessarily acyclic) digraphs. Floyd [F162] and Dantzig [Da66] have presented solutions which require O(N^3) time. A commonly accepted form of measuring efficiency of algorithms for shortest paths consists of computing the total number of additions and comparisons performed with the weights of a complete digraph, when this complete digraph is input to the algorithm. When computing shortest paths between every pair of vertices in a complete digraph with non-negative weights, [Fl62] and [Da66] are known to require N(N-1)(N-2) additions and comparisons. Yen [Ye72] has presented an algorithm for finding all shortest paths from a single vertex to all others, which requires $\frac{1}{2}N^2$ additions and N^2 comparisons, for a complete digraph with non-negative weights. This method constitutes a variation of Dijkstra's strategy, and by applying it iteratively N times, Yen could solve the all shortest paths problem in $\frac{1}{2}N^3$ additions and N^3 comparisons. A necessary correction to [Ye72] has been given by Williams and White [WiWh73]. The algorithm by Spira [Sp73] requires $O(N^2\log^2 N)$ time in average, for a digraph with non-negative weights. However, as mentioned in [Sp73] this algorithm has a worst case of $O(N^3 \log N)$ time. The algorithm which presents the best time bound - which we know so far - is given by Johnson [Jo73]. It requires $O(N^{2+\frac{1}{k}} + NM)$ time - where $k \geq 1$ is independent of N - for finding all shortest paths in a general digraph with N vertices and M edges.

Now let us consider restricting this problem to acyclic digraphs. A first approach to the problem could consist of applying the strategy of section 4.3 (for finding the shortest paths from all vertices of the digraph to a fixed vertex b) iteratively, N times, for b=1,...,N. After the last iteration the problem would have been solved. Since O(N+M) time is required per iteration, the total time bound for this method is O(N(N+M)).

However, we can improve this method so that in the worst case we take a smaller total number of additions and comparisons. Basically the idea consists of choosing a sink vertex v_1 and applying algorithm 4.2 for finding the shortest paths from all vertices to vertex v_1 and output them. This operation is performed in the given digraph $D=D_1$. Next, since v_1 is a sink vertex, it certainly does not belong to any of the remaining desired paths and therefore v_1 - and all edges leading to v_1 - can be deleted. Let D_2 denote the new digraph so obtained and choose a sink vertex v_2 of D_2. Apply algorithm 4.2 for finding all shortest paths from all vertices of D_2 to v_2 and output them. Delete vertex v_2 and all edges leading to it. D_3 is the new digraph, and so on. A total of N-1 iterations are necessary for determining all shortest paths and at the end of the (N-1)-th iteration the digraph is reduced to a single vertex. Observe that the order in which the vertices are being deleted from the digraph corresponds to a reverse topological ordering.

In order to maintain and update efficiently the information concerning which of the vertices become sink vertices, we would require some additional data structures. First, a vector containing the outdegrees of all vertices. It would be updated each time a vertex is deleted, simply decreasing by 1, the values corresponding to the vertices for which there exist edges to the newly deleted vertex. Second, a list for storing the sink vertices. However, since we are deleting vertices from the digraph, the information that a vertex has become a sink vertex can be obtained from the representation of the digraph. This is indicated by the fact that the adjacency list of a sink vertex is an empty list. This avoids the definition of that vector of outdegrees. As for the list of sink vertices note that when a vertex is deleted its adjacency list is empty and therefore the existing pointer to it becomes idle. Therefore, these pointers can be used for storing the list of sink vertices - and no additional storage is required for those structures. However, for efficiently deleting an edge (w,v) to a sink vertex v, we need to access the node v in the adjacency list of w. We then use the representation by adjacency lists of the converse digraph $\bar{D}$ of D with each node w, in the adjacency list of v - corresponding to the edge (v,w) of $\bar{D}$ - pointing to the location of edge (w,v) in D.

The following is the algorithm for solving the present problem. R contains the set of vertices not yet deleted in the digraph and S_o contains the subset of R whose elements are source vertices. Each deletion that occurs in sets R and S_o as well as each deletion of an edge of the digraph, can be performed in a constant number of steps.

ALGORITHM 4.3

```
begin comment an algorithm for determining the shortest paths between
          every pair of vertices of an acyclic directed graph;
    procedure PATH (integer value v);
    begin
        ...
        ...     the same as in algorithm 4.1
        ...
    end PATH;
    procedure INITIATE (integer value v);
    begin for z ∈ R do
        begin mark(z):=false;
            length(z):=infinity;
            route(z):=0
        end;
        mark(v):=true;
        length(v):=0;
    end INITIATE;
    integer b;
    read the digraph and construct the adjacency lists;
    $S_0$ :=set of source vertices of D;
    R :=set of vertices of D;
    for j:=1 until N-1 do
    begin b:= any sink vertex of D;
        if b $\notin S_0$ then
        begin INITIATE(b);
            for j $\in S_0$ do PATH(j);
        end
        else delete b from $S_0$;
        delete b from R and from the digraph;
        output all shortest paths to vertex b;
    end
```

As an alternative, we could initially determine a reverse topological sorting arrangement of the vertices of the diggraph, $v_1 v_2 \dots v_k$, and iteratively set $b=v_1, v_2, \dots, v_k$. This would slightly simplify the data structures used in an implementation of algorithm 4.3.

The correctness of this method follows directly from the correctness of algorithm 4.2 and from the observation that a vertex that is deleted in an iteration j would not have been involved in any shortest path to be found in iterations k, k>j.

As for the performance, note that the algorithm requires O(N+M) space and the time bound is $\sum_{i=1}^{N-1} O(N_i + M_i)$, where N_i and M_i are the number of vertices and edges of the digraph D_i, immediately before the deletion of the i-th vertex. Since we delete one vertex at each iteration, we have

$$\sum_{i=1}^{N-1} N_i = \frac{(N+2)(N-1)}{2} .$$

The contribution of the edge explorations, in the worst case - a complete acyclic digraph with $M=N(N-1)/2$ edges - is $\frac{1}{2}\sum_{i=1}^{N-1} N_i (N_i - 1) = \frac{1}{2}\sum_{j=1}^{N-1} (j+1)j = \frac{1}{6}(N^3-N)$, since at each iteration i with the digraph D_i, we explore $N_i (N_i - 1)/2$ edges. Therefore,

$$0 \leq \sum_{i=1}^{N-1} M_i \leq \frac{1}{6} (N^3-N).$$

In terms of number of operations performed with the weights of an input complete acyclic digraph, we therefore conclude that exactly $\frac{1}{6}(N^3-N)$ additions and comparisons are required.

4.6 Shortest path visiting a specified subset of vertices

Given an acyclic digraph D(V,E), with weighted edges, given vertices $a,b \in V$ and a set $H \subset V$, the problem consists of finding the shortest path from a to b, passing through all vertices of H. We assume that $a,b \notin H$.

Dreyfus [Dr69] discusses this problem for general digraphs and presents an algorithm for solving it. However in [Dr69] it is pointed out that the travelling-salesman problem is a particular case of the present one, and since no efficient solution is known to the former, the same is true for the latter.

If the digraph is acyclic however, we show that the problem is considerably simplified - and, in fact, a simple and efficient solution is presented in this section. This solution is optimal within a constant factor.

We first find a topological sorting arrangement $v_1 v_2 \ldots v_N$ of the vertices of the digraph. Clearly, since any path from v_i to v_j $i<j$, contains possibly only vertices v_k such that $i \leq k \leq j$, we conclude that a necessary condition for the existence of a solution is that every vertex $u \in H$ is such that u lies between a and b, in a topological sorting arrangement. Furthermore, if v_p and v_q are vertices such that v_p precedes v_q in a path from a to b, then there exists no path from a to b, which contains v_p and v_q, with v_q preceding v_p. Therefore, if the digraph is acyclic, the ordering in which the vertices of set H may be visited, in any path from a to b, is unique - and it corresponds to the ordering in which the vertices of H appear in a topological sorting sequence. Observe that if two vertices $u_1, u_2 \in H$ are such that u_1 precedes u_2 in a certain topological sorting sequence, and there exists another topological sorting sequence such that u_2 precedes u_1 in it, then there exists no solution to the present shortest path problem, since u_1 and u_2 are mutually non-reachable, one from the other. Note also that what causes the present problem to admit an efficient solution for acyclic digraphs - in contrast with general digraphs - is precisely this uniqueness in the ordering in which the vertices of H may be visited. Clearly, this does not hold if the digraph contains cycles.

Let $u_1u_2\ldots u_k$ be an ordering of the vertices of H, such that this ordering is embedded in some topological sorting arrangement $v_1v_2\ldots v_N$ of the vertices of D (i.e. for $u_i=v_{i'}$, $u_j=v_{j'}$, we have: if $i<j$ then $i'<j'$). If there exists a path from a to b, visiting all vertices of H, this path has the form $u_0,P_0,u_1,P_1,u_2,\ldots,u_k,P_k,u_{k+1}$ $(u_0=a, u_{k+1}=b)$, where $P_0,P_1,\ldots,P_k$ are (possibly empty) paths in D, such that P_j contains only vertices that lie between u_j and u_{j+1}, in that topological sorting arrangement, of the vertices of D. P_j cannot contain any vertex $u_i \in H$, since the digraph is acyclic. Among all such paths from a to b, the shortest is precisely that which contains the shortest P_j, for all j, $0\leq j\leq k$. In other words, the shortest path from a to b, visiting $u_1,u_2,\ldots,u_k$, in that order, consists of the shortest path from a to u_1, followed by the shortest path from u_1 to u_2, and so on, until the shortest path from u_k to b is considered. The problem therefore, can be reduced to k+1 shortest paths problems $(k=|H|)$. Since O(N+M) time is required for solving each of these problems, the total time bound would be O((N+M)k).

However, by slightly modifying the strategy and applying adequate data structures, we can reduce the time to just O(N+M). For observe that only vertices that lie between u_j and u_{j+1}, in a topological sorting, ought to be explored in the computation of the shortest path from u_j to u_{j+1}. To restrict the vertices that could be explored during that computation, we need to manipulate properly the information given by the mark vector of procedure PATH, in algorithm 4.1: At the beginning of the process, all vertices v are initialised with mark(v)=<u>true</u>. Before the call of that procedure for computing the shortest path from u_j to u_{j+1} we set mark(v)= <u>false</u>, thus allowing the exploration at this stage of the vertices v that lie between u_j and u_{j+1} in the considered topological sorting arrangement (we recall that the exploration of a vertex w is a call of PATH(v) with v=w). If a certain vertex w lies after u_{j+1}, in that topological sorting arrangement, or has already been explored before, then mark(w)=true and therefore will not be explored. This strategy ensures that any vertex - and consequently the edges from it - is explored at most once, during the entire process.

The following is an ALGOL-like formulation of the algorithm. The visit vector maintains the information of which are the vertices v, such that a shortest path to v has to be computed: if $v \in H$ or v=b then visit(v)=true, otherwise visit(b)=false. The boolean variable solution is, at the end of the process, true if $a,b \notin H$ and all computed shortest paths have non-infinite length. Otherwise solution is false. The existence of a solution to the problem is guaranteed when, at the end of the process, the variable solution has the value true and, in addition, a total of k+1 shortest paths were computed. The number of times the computation of a shortest path problem is invoked is stored in the variable count. The variable total contains the desired length of the shortest path from a to b, passing through the vertices of H. Clearly, this length is the sum of the lengths of the k+1 intermediate shortest paths, which are computed. The final shortest path itself can efficiently be obtained as before from the route vector, which is properly set within the scope of procedure PATH.

ALGORITHM 4.4:

```
begin comment an algorithm for finding the shortest path, in an acyclic
              digraph D(V,E), from vertex a to b, visiting all vertices
              of a set H, H ⊂ V and a,b ∉ H.
      procedure PATH (integer value v);
      begin
           . . .
           . . .     the same as in algorithm 4.1
           . . .
      end PATH;
      logical solution;
      integer total, count, i, j;
      read the digraph D and construct the adjacency lists A;
      read the k vertices of set H and vertices a,b;
      find a topological sorting arrangement $v_1 v_2 \ldots v_N$, of the vertices of D;
      for w:=1 until N do
      begin mark(w):=true;
            visit(w):=false;
            length(w):=infinity;
            route(w):=0;
      end
      for w ∈ H do visit(w):=true;
      solution:=¬ (visit(a) or visit(b));
      visit(b):=true;
      total:=count:=0;
      j:=index of $v_j$ in $v_1 v_2 \ldots v_N$, such that $v_j$=a;
      while $v_j \neq$b and solution do
      begin i:=j;
            repeat mark($v_j$):=false;
                   j:=j+1
            until visit($v_j$);
            length($v_i$):=infinity;
            length($v_j$):=0;
            count:=count+1;
            PATH($v_i$);
            if length($v_i$)=infinity then solution:=false
            else total:=total + length ($v_i$)
      end
      if solution and count=k+1 then output the desired shortest path
      else output 'NO SOLUTION EXISTS(INFINITE PATH LENGTH)';
end
```

The correctness of the presented strategy follows from the lemmas enunciated below.

Let $D(V,E)$ be an acyclic digraph with weighted edges, $a,b \in V$ and $H \subset V$, with $a,b \notin H$.

Lemma 4.4:

A shortest path from a to b, visiting all vertices of H, with non-infinite length has the form

$$u_0, P_0, u_1, P_1, u_2, \ldots, u_k, P_k, u_{k+1}$$

where: $u_0=a$; $u_{k+1}=b$; $\{u_1, u_2, \ldots, u_k\} = H$; $u_1, u_2, \ldots, u_k$ are such that if $i<j$ then u_i precedes u_j in a topological sorting arrangement. $P_0, P_1, \ldots, P_k$ are (possibly empty) paths such that $u_i P_i u_{i+1}$ is the shortest path from u_i to u_{i+1}.

Lemma 4.5:

Let $u_0, P_0, u_1, P_1, u_2, \ldots, u_k, P_k, u_{k+1}$ represent a shortest path from a to b, visiting all vertices of H, as above. Then the only possible vertices that could lie in any P_j, $0 \leq j \leq k$, are those which are between u_j and u_{j+1}, in a topological sorting arrangement of the vertices of D.

The performance of the presented method can be evaluated by the following theorem.

Theorem 4.3:

Let $D(V,E)$ be an acyclic digraph with weighted edges, $a,b \in V$, $H \subset V$ and $a,b \notin H$. Then algorithm 4.4, for finding a shortest path from a to b visiting all vertices of H, requires $O(N+M)$ space and time.

The arguments in which the proofs of the theorem and lemmas above are based were informally given through this section.

4.7 k-shortest paths from all vertices to a given vertex

Let D(V,E) be an acyclic digraph, with weights d_{ij} assigned to its edges and b $\in$ V. The problem consists of finding paths from all vertices to b, such that each desired path from vertex v to b, has the k-th smallest length, among all paths from v to b.

Elmaghraby [El70] has presented an algorithm for finding the length of the k-shortest path from a given vertex a to a given vertex b in an acyclic digraph. The method [El70] is simple and short, although its efficiency can be well improved. Actually, it finds the lengths of the k-shortest paths from all vertices to vertex b. It proceeds as follows: let α_v^j denote the length of the j-shortest path from v to b; let $v_1 v_2 \ldots v_p$ (v_1=b) represent a reverse topological sorting arrangement of the vertices of D, up to vertex b; initialise $\alpha_b^1=0$ and $\alpha_b^2=\alpha_b^3=\ldots=\alpha_b^k$=infinity. Then the desired k-shortest paths are obtainable from

$$\alpha_v^j = \min_j\{\alpha_w^r + d_{vw}\}, \qquad \begin{array}{l} v=v_2,v_3,\ldots,v_p \\ j=1,2,\ldots,k \\ r=1,2,\ldots,j \\ \text{for all } w \text{ - where } (v,w) \in E, \end{array}$$

with $\min_j$ representing the j-th minimum.

Consider the worst case analysis of this algorithm namely a complete acyclic digraph with b being the sink vertex. At each step i (i=2,3,...,p), all k-shortest paths from v_i to b are calculated. Next vertex v_{i+1} is considered and so on. Therefore, for each i a total of (i-1)k additions are performed. Consequently the total number of additions of weights required is $\sum_{i=2}^{N}(i-1)k=kN(N-1)/2$. As for the number of comparisons, note that α_v^1 may be obtained from $\alpha_v^1=\min\{\alpha_w^r + d_{vw}\}$.

Also $\alpha_v^2=\min[\{\alpha_w^r + d_{vw}\}\setminus\{\alpha_v^1\}]$, $\alpha_v^3=\min[\{\alpha_w^r + d_{vw}\}\setminus\{\alpha_v^1,\alpha_v^2\}]$, etc. Clearly, this is more efficient than $\alpha_v^1=\min_1\{\alpha_w^r + d_{vw}\}$, $\alpha_v^2=\min_2\{\alpha_w^r + d_{vw}\}$, etc., of the original algorithm. Using the first of these two schemes, at each stage i, $\alpha_{v_i}^j$ requires one minimization of a set of $(i-2)j+1$ elements, which corresponds to $(i-2)j+1$ comparisons of weights. Therefore for computing k minimizations, for all $\alpha_{v_i}^j$ of a fixed vertex v_i, we require $\sum_{j=1}^{k}(i-1)j+1=k(ki-2k+i)/2$ comparisons. Thus the total number of comparisons necessary to obtain the lengths of the k-shortest paths from all vertices to vertex v_1 is $\sum_{i=2}^{N}k(ki-2k+i)/2=k(N-1)[(k+1)(N+2)-4k]/4$, i.e. $O(N^2k^2)$. Alternatively, instead of performing the comparisons step by step as indicated, we can compute all additions necessary to find the k-shortest paths and produce all $\alpha_{v_i}^j$, for a fixed v_i, by finding the k smallest values of the set composed by those additions. Spira [Sp73] has shown that the minimum k values of a set with S elements can be computed using $S-1 + (k-1)\lceil\log_2 S\rceil$ comparisons. Therefore by adopting this strategy, a total of $(i-1)k - 1 + (k-1)\lceil\log_2(i-1)k\rceil$ comparisons are required for obtaining the k-shortest paths from a fixed vertex v_i to v_1. Consequently for the entire process $O(kN(N+\log k))$ comparisons are required.

The algorithm that we propose in this thesis, for finding the lengths of the k-shortest paths from all vertices to vertex b uses a recursive procedure LENGTHQ and convenient data structures for decreasing the total number of additions and comparisons required. It prevents the computation of a j-shortest path from a vertex to vertex b, if this path is known to have infinite length and avoids the exploration of an edge (v,w), in the computation of the j-shortest path from v to b, if the (non-infinite) longest path from w to b had been used before, in a i-shortest path from v to b, $i<j$. With each edge (v,w) we associate two variables:

t_{vw} and y_{vw}. After the computation of the j-shortest path ($j>1$) from v to b, t_{vw} equals one plus the number of i-shortest paths from v to b, $1\le i<j$, which contain edge (v,w); y_{vw} equals the weight of edge (v,w) plus the length of the t_{vw}-shortest path from w to b. Now, if we denote by short(v,j) the length of the j-shortest path from v to b, then the value of short(v,j) can be calculated simply by

$$\text{short}(v,j) = \min\{y_{vw},\ w \in A(v)\}$$

A vector <u>way</u> is also used, with way(v) containing the vertex following v in the shortest path from v to w. Thus the problem consists basically in keeping and manipulating efficiently these quantities through the process.

Initially we find the lengths of the shortest paths from all vertices to vertex b, using algorithm 4.2. During this phase we can delete from the digraph all vertices whose shortest path length to vertex b is infinite. Next, we initialise variables as follows: $t_{vw}=1$ for all edges (v,w); short(v,1)=length of the shortest path from v to b; y_{vw}=short(w,1) + d_{vw} and way(v) is initialised as mentioned above. Next we pass to the actual computation of the k-shortest paths. The vertices are processed in reverse topological ordering starting from the vertex immediately succeeding b in this sequence. Vertex b is not processed and short(b,2) is set to infinity. When returning from a call LENGTHQ(v,2,way(v)), invoked from the outside of the procedure, the length of the k-shortest paths from v to b have been determined. So, if $u_1u_2\ldots u_p$ (u_p=b) is a topological sorting arrangement - up to b - of the vertices of the digraph, we first calculate all the desired shortest paths from vertex u_{p-1} to b, then we consider vertex u_{p-2}, and so on. This strategy can be adopted because if the digraph is acyclic any j-shortest path from a vertex u_q to vertex b depends only on i-shortest paths $(1 \leq i \leq j)$ from vertices u_r to b, where $r > q$.

Now assume that a recursive call of the procedure was invoked and LENGTHQ(v,j,q) is being computed. Then the strategy ensures that all short(v,i), $1 \leq i < j$, have already been calculated, that short $(v,j-1) <$ infinity and that all k-shortest paths from all vertices succeeding v in the topological sorting arrangement have already been determined. The parameter q corresponds to the vertex immediately following v in the (j-1)-shortest path from v to b. Since edge (v,q) was used in this last computed (j-1)-shortest path from v to b, t_{vq} must be incremented by one, and y_{vq} updated. The new y_{vq} will contain the length of the (new) t_{vq}-shortest path from q to b, plus d_{vq}. If, however, short(q,t_{vq}) is now infinite, this means that q will never again be part of any i-shortest path $(i \geq j)$ from v to b and therefore edge (v,q) can be deleted to avoid unsuccessful searches. If the adjacency list of vertex v contained the sole edge (v,q), and this edge has been deleted, then A(v) is now empty, which means that there are no more unused paths from v to b, i.e. short(v,j) is infinite. In this case, no calls of the procedure will occur to compute the (j+1)-shortest path from v to b, since its length is known to be infinite. In the case that A(v) is not empty, the length of the j-shortest path from v to b is clearly the minimum of all y_{vw}, for $w \in A(v)$. By adopting this strategy, we do not need to re-compute the value of the y_{vw}'s which were not minimum. They remain and are eventually used in an i-shortest path $(i > j)$ from v to b.

The following is an ALGOL-like formulation of this algorithm for computing the lengths of the k-shortest paths from all vertices of an acyclic digraph, to the fixed vertex b.

ALGORITHM 4.5:

```
begin comment an algorithm for finding the lengths of the k-shortest paths
            from all vertices of an acyclic digraph D(V,E) to a vertex b;
    procedure LENGTHQ(integer value v,j,q);
    begin comment v,q,...,b was the (j-1)-shortest path from v to b;
          $t_{vq}$:=$t_{vq}$ + 1;
          if short (q,$t_{vq}$) < infinity then $y_{vq}$:=short(q,$t_{vq}$) + $d_{vq}$
          else delete edge (v,q) from A(v);
    $\alpha$:   if A(v) non-empty then
    $\beta$:   begin short(v,j):=min{$y_{vw}$, w $\in$ A(v)}
                comment let z denote the minimizing w;
                if j<k then LENGTHQ(v,j+1,z)
          end
          else short(v,j):=infinity
    end LENGTHQ;
    read the digraph and construct the adjacency lists A;
    read the value of k and vertex b;
    find the shortest paths from all vertices to vertex b;
    for v:=1 until N do
       if length shortest path from v to b = infinity then delete vertex v
       else begin short(v,1):=length shortest path from v to b;
                  way(v):=vertex following v in a shortest path
                          from v to b (v$\neq$b)
            end
    for (v,w) $\in$ E do
    begin $t_{vw}$:=1;
          $y_{vw}$:=short(w,1) + $d_{vw}$
    end
    find a topological sorting arrangement $u_1 u_2 \ldots u_p$ ($u_p$=b) of the
          remaining vertices of the digraph;
    short(b,2):=infinity;
    for i:=p-1 step-1 until 1 do LENGTHQ($u_i$,2,way($u_i$))
```

The implementation of this algorithm is simple. The shortest paths from all vertices to vertex b can be found by algorithm 4.2 of section 4.3; a topological sorting arrangement can be obtained by the algorithm [Kn68]; the t_{vw} and y_{vw} quantities may be stored in the adjacency lists, i.e., each node of the A(v) list, corresponding to edge (v,w), would contain the triple (w,t_{vw},y_{vw}). The short (v,j) quantities can be stored either as a Nxk matrix or as a set of linked lists, one list B(v) per vertex v, of the digraph. In the latter more economical scheme, the above t_{vw} variables, are replaced by pointers p_{vw}, to the location of the t_{vw}-th node, corresponding to short(v,t_{vw}), in B(v); the statement corresponding to $t_{vq}:=t_{vq}+1$ is replaced by p_{vq}:=location of the next node in B(v) list, and so on.

The correctness of the proposed method is based on the following lemmas:

Let D(V,E) be an acyclic digraph, with weights d_{ij} associated with its edges, $b \in V$, $k>1$, j such that $2 \leq j \leq k$ and D' a digraph obtained from D, by deleting all vertices of D, whose shortest path lengths to b are infinite. Let $u_1 u_2 \ldots u_p$ $(u_p=b)$ be a topological sorting arrangement of the vertices of D'. Let short(v,j) denote the length of the j-shortest path from v to b in D'. Let D be input to algorithm 4.5.

Lemma 4.6:

If s<infinity and q is the vertex succeeding u_i $1 \leq i < p$, in the (j-1)-shortest path from u_i to b, then both:

(i) At the point α of the computation of LENGTHQ (u_i,j,q) $A(u_i)$ is not empty and each $t_{u_i w}$ contains one plus the number of r-shortest paths from u_i to b that contained w $(1 \leq r \leq j-1)$. Each y_{vw} contains the length of the shortest path from v to b through w not yet used in any computation of a r-shortest path from v to b , $1 \leq r \leq j-1$.

(ii) At the point β of the same computation short (u_i,j) is set to the length of the j-shortest path from u_i to b.

Lemma 4.7:

If s=infinity, then either:

(i) If the length of the (j-1)-shortest path from u_i to b is non-infinite, then a call LENGTHQ(u_i,j,q) eventually occurs with q as above. At the point α of that computation the list $A(u_i)$ is empty and therefore short(u_i,j) is set to infinity.

(ii) If the length of the (j-1)-shortest path from u_i to b is infinite, then no call of LENGTHQ(u_i,j,q) is ever invoked and short (u_i,j) is not referenced at any part of the process.

Proof (lemma 4.6):

Before the first call of the procedure is invoked, the algorithm finds a topological sorting $u_1u_2\ldots u_p$ of D'. We now proceed by induction on decreasing i, increasing j. Let h=max(i), such that short$(u_h,2)$<infinity. For j=2, all t_{u_hw} and y_{u_hw} contain their initial values at the entry to LENGTHQ$(u_h,2,q)$ because this is the first call with parameter $v=u_h$. For the same reason q=way(u_h) is the vertex following u_h in the shortest path from u_h to b. Since the digraph is acyclic, $q=u_t$, for some $h+1\leq t\leq p$ and therefore short$(u_t,2)$=infinity and edge (u_h,q) is deleted from $A(u_h)$. Because there exist more than one path from u_h to b, $A(u_h)$ is not empty at the point α of that computation, and therefore at point β short$(u_h,2)$ is set to min$\{y_{vw},\ w \in A(u_h)\}$ which corresponds to the length of the second shortest path from u_h to b. The induction step, for LENGTHQ(u_h,j,q) is similar to the case j=2, except that at the point α of the computation

of LENGTHQ(u_h,j,q), A(u_h) contains j-2 fewer edges than at the same point of LENGTHQ(u_h,j,q). This follows from the fact that in each computation of LENGTHQ(u_h,j',q), for all $2 \le j' \le j$, one edge is deleted, since each vertex in A(u_h) has only one possible path to b. Now suppose the lemma holds for vertex u_{i+1}, for some i, $1 \le i < h$, and let us verify the case u_i. The proof for vertex u_i with j=2, is similar to that for u_h with j=2, except that the deletion of edge (u_i,q) does not necessarily occur. Suppose then that the lemma holds for vertex u_i with j-1 and let us verify the case u_i, with $j \le k$. By this induction hypothesis (i) and (ii) of the lemma are satisfied for LENGTHQ(u_i,j-1,q), and a recursive call LENGTHQ(u_i,j,q) occurs with q being the vertex following u_i in the (j-1)-shortest path from u_i to b. Consider now the computation of LENGTHQ(u_i,j,q). The algorithm sets $t_{u_i q} := t_{u_i q} + 1$. The value short ($q, t_{u_i q}$) has already been computed because since the digraph is acyclic $q = u_t$ for some t, $i+1 \le t \le p$, and $t_{u_i q} + 1 \le j$. Now if short($q, t_{u_i q}$) is infinite then all remaining paths from u_i to b through q have infinite length and therefore edge (u_i,q) can be deleted. Otherwise $y_{u_i q}$ is updated to its appropriate value, i.e. $y_{u_i q} := \text{short}(q, t_{u_i q}) + d_{u_i q}$. In any case, the only difference between the values of $y_{u_i w}$'s at points α and β of the computations LENGTHQ(u_i,j-1,q') and LENGTHQ(u_i,j,q) is in $y_{u_i q}$, which either was correctly updated or whose edge (u_i,q) has been deleted. Therefore is s<infinity there exists at least one value of $y_{u_i w}$ which has not been used yet, and consequently at point α of this last computation, A(u_i) is not empty, $t_{u_i w}$ and $y_{u_i w}$ satisfy (i) and short(u_i,j) is set to the length of the j-shortest path from u_i to b at point β.

The proof of lemma 4.7 can be established similarly.

The performance of the present method can be evaluated by the following theorem.

Theorem 4.4:

Let D(V,E) be anacyclic digraph with weighted edges, input to algorithm 4.5 and $b \in V$. Then for calculating the lengths of the k-shortest paths from all vertices to vertex b, O(Nk+M) space and O((N+M)k) time are required, where N and M are respectively the number of vertices and edges of the digraph. The number of additions and comparisons of weights, performed within the scope of LENGTHQ are O(Nk) and O(Mk), respectively.

Proof:

The representation of the digraph by adjacency lists requires O(N+M) space. Storing the short(v,j) quantities as a matrix requires O(Nk) space and the remaining data structures require O(N+M). Therefore O(Nk+M) space is needed. For the time bound, observe that, in each computation of LENGTHQ(v,j,q), at most one addition of weights and at most one minimization are performed. This minimization consists of finding the minimum of $\{y_{vw}, w \in A(v)\}$. Therefore, at most outdegree(v) comparisons are required per call of the procedure. Since LENGTHQ is invoked at most k times per vertex v, we conclude that O(Nk) additions and O(Mk) comparisons are required. Since the part of the algorithm outside LENGTHQ requires O(N+M) time - finding the shortest paths from all vertices to vertex b; initialising the variables; obtaining a topological sorting arrangement; all require O(N+M) time - we conclude that the total time bound is O((N+M)k).

Now let us examine in more detail the behaviour of the algorithm in the worst case, namely a complete acyclic digraph, with b being the sink vertex. We wish to find the length of the k-shortest paths from all vertices to vertex b. Assume, without loss of generality, that the numbering of the vertices $\{1,..,N\}$ of this digraph is such that the topological sorting arrangement corresponds to the decreasing ordering of the vertices (then b=1 and N=the source vertex). For any vertex v, $v \neq 1$, a maximal number of computations of LENGTHQ(v,j,q) equals the number of different paths that exist from v to b. Let p_v represent this number of different paths. There exist exactly p_w different paths from v to b through vertex w, where $(v,w) \in E$. Therefore $p_v = \sum_{i=1}^{v-1} p_i$, with $p_1=1$. Hence $p_v = 2^{v-2}$. Consequently, another upper bound for the total number of calls of LENGTHQ is $\sum_{i=2}^{N} p_i = 2^{N-1} -1$. This bound is attained only when we desire to obtain the lengths of all possible paths, from all vertices to vertex b.

Since at most one addition of weights is performed per call of LENGTHQ, we conclude that

$$\text{total number of additions} \leq \min \{Nk,\ 2^{N-1} -1\}.$$

For the number of comparisons, we recall that each minimization at the point β of algorithm 4.5, in the computation of LENGTHQ(v,j,q) is performed in a set of at most outdegree(v) elements. If we disregard the deletions of edges, this number is exactly outdegree(v)=v-1. In this case for each vertex $v \neq 1$ there are at most $(v-1)2^{v-2}$ comparisons. Therefore, for the entire process we have at most $\sum_{i=2}^{N} (i-1)2^{i-2} = (N-2)2^{N-1} +1$ comparisons. Since M=N(N-1)/2, we conclude that

$$\text{total number of comparisons} \leq \min\{N(N-1)k/2,\ (N-2)2^{N-1}+1\}.$$

Now let us consider the deletions of edges. The problem that arises when considering them is that the actual number of comparisons becomes dependent on the particular values assigned to the weights. This happens because of the fact that a deletion of an edge (v,w) occurs precisely in the computation of a j-shortest path from v to b, such that the (j-1)-shortest path from v to b was found to be v,w,..,b, and this path is the last unused path from v to b through w. Consequently the more the paths through w are used, the more likely it is that the edge (v,w) will eventually be deleted - and this depends on the relative values of the weights. Clearly the sooner edges are deleted from the digraph the smaller the number of comparisons. Therefore we can consider a "worst worst case" to be a complete acyclic digraph in which the values of the weights are such that the deletions of the edges occur as late as possible. Figure 4.1 is such an example, with N=5. Underlined numbers in this figure correspond to the weights of the edges and the remaining numbers correspond to the vertices. In this digraph, for each vertex v, $v \neq 1$, the v-1 longest (non-infinite) paths from v to 1 are of the form: v,(v-1),...,1 ;v,(v-2),...,1 ;v,(v-3),...,1 ;... ;v,1. Consequently, since every edge from v is involved in one of the v-1 last shortest paths from v to b, they cannot be deleted before these paths are considered. In fact the v-1 edges from v are deleted, respectively only in the last v-1 possible computations of LENGTHQ(v,j,q). We recall that a total of 2^{v-2} calls of the procedure are invoked for each vertex v (clearly we are considering the extreme case where all shortest paths are desired). Hence, in each of the first $2^{v-2}-(v-1)$ computations of LENGTHQ(v,j,q), v-1 comparisons occur. Subsequently one new edge is deleted in each of the following v-1 computations of the procedure. Therefore, for each vertex v, at most $(v-1)[2^{v-2}-(v-1)]+\sum_{i=1}^{v-1} i = (v-1)2^{v-2}-(v-1)(v-2)/2$ comparisons can occur. Hence the maximum number of comparisons that may be performed for all vertices during the entire process is

$$\sum_{i=2}^{N}[(i-1)2^{i-2}-(i-1)(i-2)/2] = (N-2)[2^{N-1}-N(N-1)/6]+1 .$$

Thus the following is satisfied for the "worst worst case":

total number of comparisons$\leq$min$\{N(N-1)k/2,(N-2)[2^{N-1}-N(N-1)/6]+1\}$.

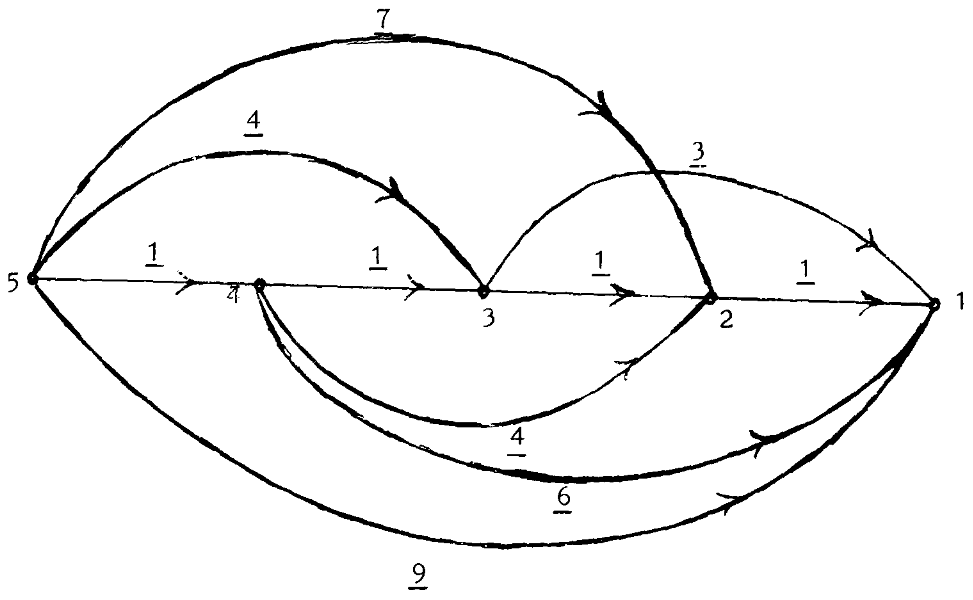


Figure 4.1

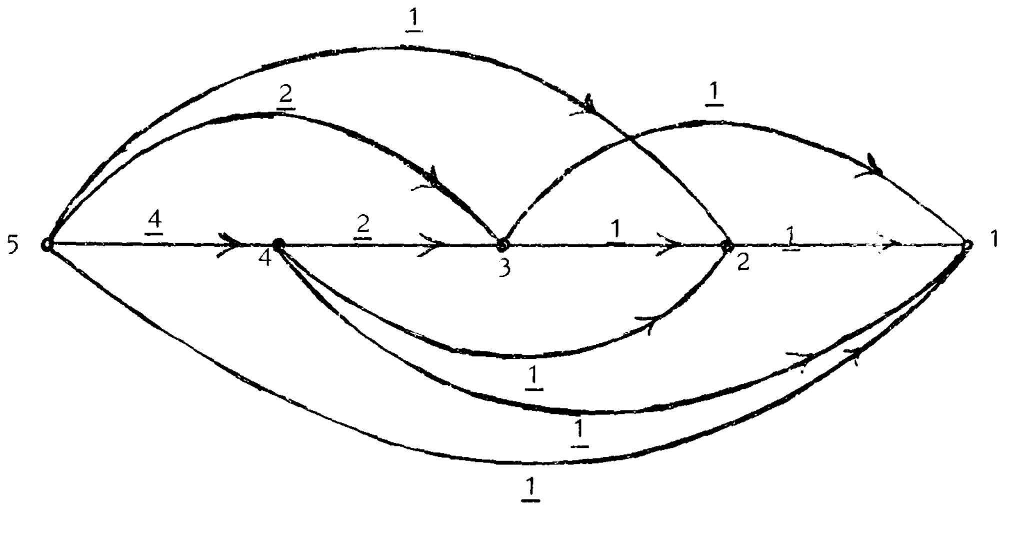


Figure 4.2

There exists a "best worst case" corresponding to a complete acyclic digraph in which the deletions of edges are performed in the earliest possible time. The digraph of figure 4.2 is such an example with $N=5$. It has the property that, for every vertex v, $v\neq 1$, if $v,w_1,\ldots,1$ and $v,w_2,\ldots,1$ are two paths from v to 1, then the length of $v,w_1,\ldots,1$ is smaller than the length of $v,w_2,\ldots,1$, when $w_1<w_2$. Since edge (v,w_p) can be deleted after all paths $v,w_p,\ldots,1$ have been used, we conclude that it can be deleted before the consideration of any path $v,w_q,\ldots,1$, with $q>p$.

If the k-shortest paths themselves are required in addition to their lengths, it is not recommended trying to trace them back starting from the obtained lengths. Instead, the paths can be found during the actual process of finding the lengths. Two Nxk matrices, vertex and order would be required. For a certain vertex v, vertex (v,j) would contain the vertex w, which follows v, in the j-shortest path from v to b. The content of order(v,j) would be the integer i, such that the path w,...,b, in the j-shortest path v,w,...,b from v to b, is the i-shortest path from w to b. If the j-shortest path from v to b is infinite then vertex(v,j)=0 and order(v,j) is undefined. The implemention of this strategy is simple: vertex(v,1) and order(v,1) are initialised according to the results obtained in the step of finding the shortest paths from all v to b with order(b,1)=0. Also, vertex(v,j) is initially zero for all v and for all j, $2\leq j\leq k$. Now in algorithm 4.5 after the line

```
        comment let z denote the minimizing w;
```

insert the following statements:

```
        vertex(v,j):=z;
        order(v,j):=$t_{vz}$;
```

and this is sufficient for the purpose. The output of the j-shortest path from vertex v to b at the end of the process can be performed as follows:

```
begin integer s;
      repeat output v;
             s:=v;
             v:=vertex(s,j);
             j:=order(s,j)
      until  v=0
end
```

Clearly printing any path with such a method requires a number of steps equal to the number of vertices in the path.

Instead of using the matrices vertex and order an alternative scheme for obtaining the k-shortest paths can be proposed, which utilises a linked list and one matrix. Each node q in the list consists of two fields: vertex and link. The content of $vertex_q$ is the label v of a vertex in some j-shortest path. The field $link_q$ points to the location of the node in this list which contains the vertex following v in that j-shortest path. The Nxk matrix path is also defined, with path(v,j) pointing to the node in the list whose vertex is the first in the j-shortest path from v to b. If this path has infinite length then path(v,j) should contain a special symbol indicating this situation. An implementation of this scheme can be easily accomplished, by a slight modification of the proposed algorithm. Observe that such a list constitutes a rooted tree with the node containing vertex b being the root. The link fields correspond to pointers to ancestors in a represention of the tree. The tree pictured in figure 4.4 shows all j-shortest paths (j=1,2,3) from all

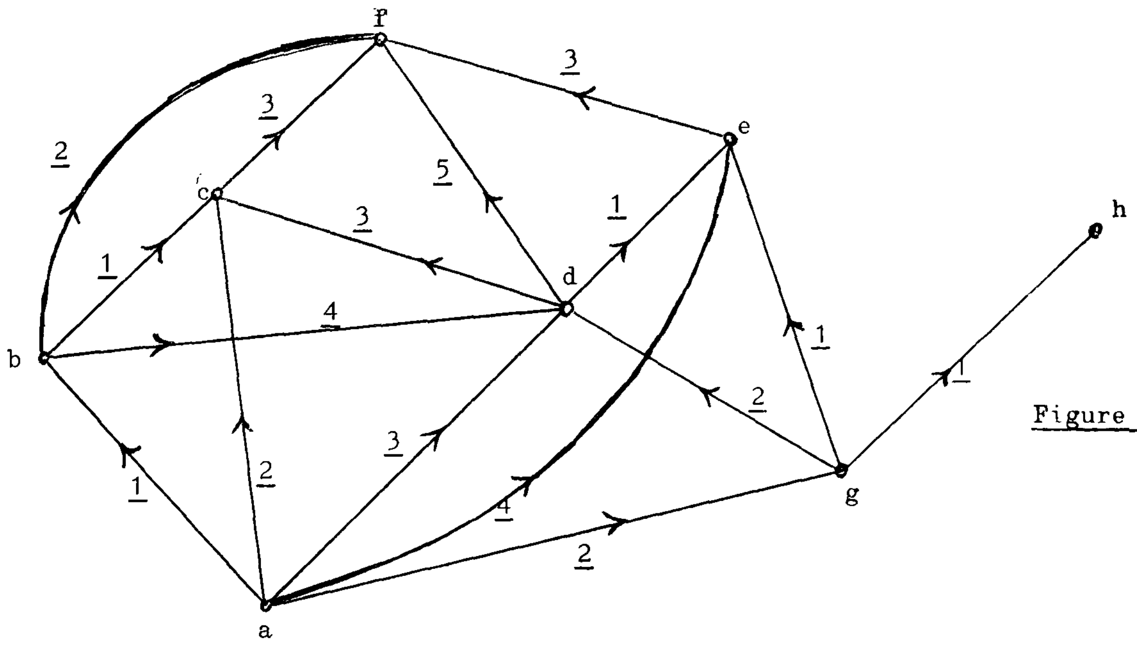


Figure 4.3

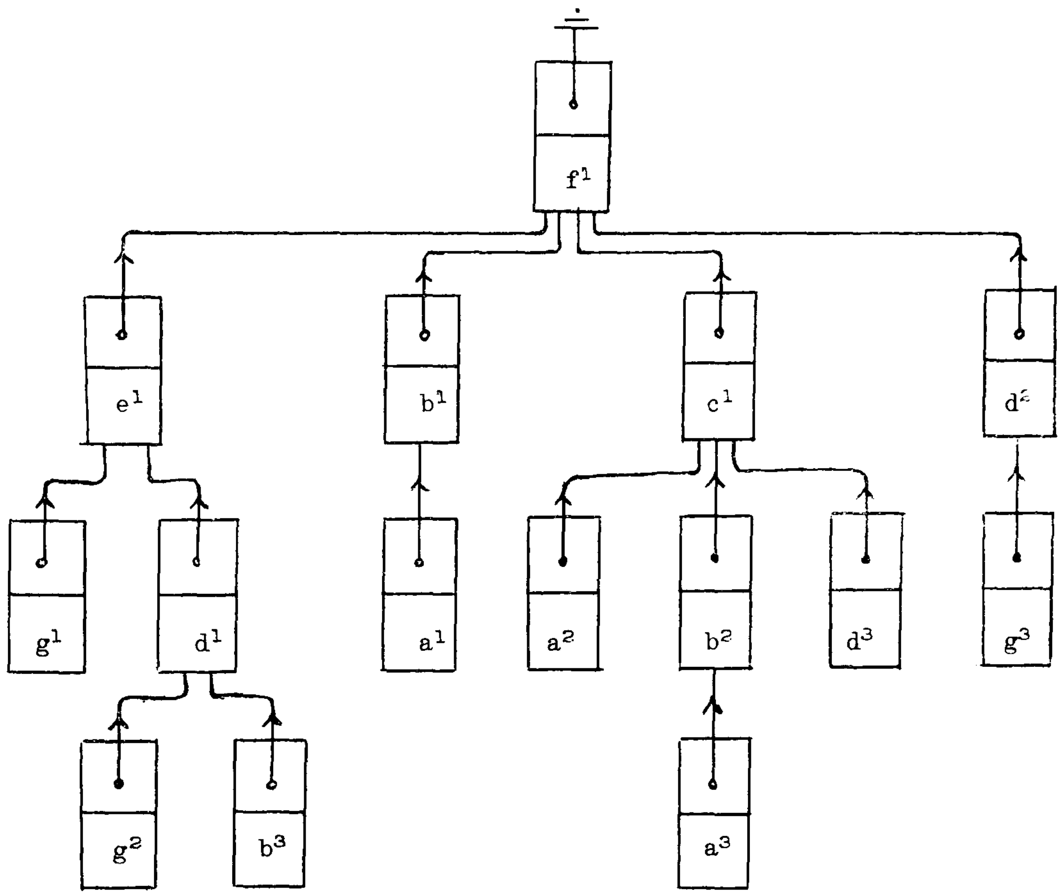


Figure 4.4

vertices of the digraph of figure 4.3 to vertex f. The superscript i, which appears in the vertex field of a node in figure 4.4, simply denotes that this node would be referenced from path(v,i), as containing the first vertex in the i-shortest path from v to f. For example, path(g,2) would point to the node whose vertex field is g^2, meaning that a second shortest path from g to f is gdef.

We can alter the proposed algorithm so that the total number of comparisons of weights performed during the execution of the recursive procedure is in general less than that of the original version presented - but an overhead is added to the part of the algorithm outside the procedure. Examining algorithm 4.5 we observe that in each call of LENGTHQ(v,j,q) a minimization of y_{vw} occurs. In any two consecutive recursive calls the corresponding sets of y_{vw}'s differ by at most one element. This fact suggeststhat the nodes of the adjacency list of vertex v may be kept sorted according to increasing values of y_{vw}. Therefore the vertex w for which $\{y_{vw}, w \in A(v)\}$ is minimized will always correspond to the <u>first</u> node of the adjacency list.

In order to obtain the adjacency lists permanently sorted as required, two actions are necessary: first, in the initialization of the process, i.e. before the first call of the procedure, every A(v) list must be sorted so that y_{vw} values are in increasing order. During this process another slight improvement can be made. Let $y_{vw_1}, \ldots, y_{vw_q}$ be the y_{vw}'s of A(v) in increasing order and $q>k$. Clearly in any j-shortest paths $(2\le j\le k)$ from v to b, at most the first k values of y_{vw} are actually used for computing the lengths of the paths. Therefore all y_{vw_t}, $t>k$, may be deleted from A(v) in the initialisation of the algorithm since they are not involved in the computation of any j-shortest path, $2\le j\le k$. The pruning of the A(v) lists will contribute to decrease the number of comparisons of weights performed later in the execution of the recursive procedure. The second action required is to maintain the lists adequately sorted during the actual process of finding the j-shortest paths. Assume that at the start of a given call of the recursive procedure for finding the j-shortest path from v to b, the A(v) list is correctly sorted. Since q is the vertex following w in the (j-1)-shortest path, q is the first vertex of A(v). After increasing t_{vq} by 1, if short(v,t_{vq})< infinity then the sum y_{vq}:=short(q,t_{vq})+d_{vq} is performed. In this case the list A(v) has to be rearranged because the new first value y_{vq} of the y_{vw}'s is not necessarily the smallest among all y_{vw}'s. However, since A(v) is necessarily sorted from its second node until the last the rearranging of A(v) is equivalent to the problem of adequately inserting a new element in a sorted list in such a way that the appropriate ordering is maintained. This can be accomplished in a number of comparisons, which is on average less than $|A(v)|$, the number of nodes of A(v) - we recall that the minimization in algorithm 4.5 requires exactly $|A(v)|$ comparisons. On the other hand, if short(v,t_{vq})=infinity then the edge (v,q) is deleted from A(v). In this case no rearrangement of A(v) is necessary since the sorting is preserved after the deletion.

The following is an ALGOL-like description of this new variation. Note that the (third) parameter q of the recursive procedure has been deleted. This is because the information represented by q in algorithm 4.5 can be obtained from the first node of A(v) when it is sorted. For the same reason the use of vector way of algorithm 4.5 can be avoided in this case.

ALGORITHM 4.6:

```
begin comment an algorithm for finding the lengths of the k-shortest
              paths from all vertices of an acyclic digraph D(V,E) to
              vertex b;
      procedure LENGTH(integer value v,j);
      begin comment q denotes the first vertex in the A(v) list;
            $t_{vq}:=t_{vq}+1$;
            if short(q,$t_{vq}$)<infinity then
            begin $y_{vq}$:=short(q,$t_{vq}$)+$d_{vq}$;
                  rearrange list A(v) - by possibly moving its first
                     vertex - so that A(v) remains sorted in non-
                     decreasing values of $y_{vw}$'s;
            end
            else delete edge (v,q) from A(v);
            if A(v) non-empty then
            begin comment z denotes the new first vertex of A(v);
                  short(v,j):=$y_{vz}$;
                  if j<k then LENGTH(v,j+1)
            end
            else short(v,j):=infinity;
      end LENGTH;
      read the digraph and construct the adjacency lists A;
      read the value of k and vertex b;
      find the shortest paths from all vertices to vertex b;
      for v:=1 step 1 until N do
         if length shortest path from v to b = infinity then delete vertex v
         else short(v,1):=length of shortest path from v to b;
      for (v,w) $\in$ E do
      begin $t_{vw}$:=1;
            $y_{vw}$:=short(w,1) + $d_{vw}$
      end
      sort each A(v) list according to non-decreasing values of $y_{vw}$'s;
      for v:=1 until N do
         if |A(v)|>k then delete the last |A(v)|-k nodes of A(v);
      find a topological sorting arrangement $u_1u_2...u_p$ ($u_p$=b), of the
           remaining vertices of the digraph;
      short(b,2):=infinity;
      for i:=p-1 step -1 until 1 do LENGTH($u_i$,2)
end
```

The space bound for algorithm 4.6 is the same as for algorithm 4.5 namely O(Nk+M). For the time bound observe that the number of additions of weights performed in the recursive procedure is also the same in both cases: O(Nk). The average number of comparisons performed in the recursive procedure is less for algorithm 4.6. However if linear search is adopted for rearranging the A(v) lists in algorithm 4.6 the worst case is also the same for both algorithms: O(Mk) comparisons. The part of the algorithm outside procedure LENGTH is bounded by $O(\min\{M\log M, N^2\log N\})$ time, because of the sorting of the A(v) lists. Therefore, the total time bound for algorithm 4.6 is $O(\min\{M\log M, N^2\log N\} + (N+M)k)$. However, depending on the particular input digraph, this algorithm can be faster. For example, in the digraph of figure 4.2, all the weights are such that when the value of y_{vq} (which is contained in the first node of A(v)) is altered by the sum y_{vq}:=short(q,t_{vq}) + d_{vq}, then the new value of y_{vq} is still the smallest of all y_{vw}'s. Therefore the rearranging of the A(v) list will not alter the ordering of the nodes. Consequently in each call of LENGTH at most a constant number of comparisons of weights is performed. Hence, the total number of comparisons, performed inside procedure LENGTH, with the input digraph of figure 4.2 is O(Nk) and the total time spent inside this procedure is also O(Nk).

Now we add some further short comments in relation to the algorithms presented in this section. If we examine algorithms 4.5 and 4.6 we observe that the progress of the computation is as follows: first, all j-shortest paths, $2\leq j\leq k$, from vertex u_{p-1} to u_p=b are found, where u_{p-1} is the vertex immediately preceding u_p in the topological sorting. Next all j-shortest paths from u_{p-2} to u_p are found, where u_{p-2} is the vertex immediately preceding u_{p-1} in the topological sorting. Next

u_{p-3} is considered and so on. Alternatively those algorithms can be modified to compute all second shortest paths from all vertices to vertex b; after all third shortest paths, all fourth shortest paths, and so on. As it is shown later , this alteration makes it possible to process the vertices in any order so avoiding the topological sorting pass.

In the presented algorithms there is no mention of special procedures for resolving ties between different paths having the same lengths from a vertex v to b. Some existing algorithms for finding the k-shortest paths in (general) digraphs, require all paths to have different lengths. When this is not satisfied a special treatment for tie resolution is necessary in these algorithms.

4.8 k-shortest paths from a given vertex to all others

The problem consists of: Given an acyclic digraph D(V,E) , vertex a $\in$ V and an integer k, k>1, find the lengths of the k-shortest paths from a to all vertices of the digraph. The strategy to be adopted is similar to that of section 4.4. From the digraph D obtain the converse digraph $\bar{D}$. Now simply apply the algorithms of section 4.7 with $\bar{D}$ as input digraph and b=a. Thus we obtain the lengths of the k-shortest paths from all vertices to a in $\bar{D}$, which is equivalent to obtaining the lengths of the k-shortest paths from a to all vertices in D.

Clearly the remarks of the previous section concerning time and space bounds, finding the actual k-shortest paths in addition to their lengths, discussion of alternative algorithms and so on - are all valid in the present case.

4.9 k-shortest paths between every pair of vertices

Given an acyclic digraph D and an integer k, k>1, the problem is to find the lengths of the k-shortest paths between every pair of vertices of the digraph.

A straightforward and, in this case, efficient way of solving this problem consists of applying at most N-1 times the method of section 4.7, for finding the k-shortest paths from all vertices to a fixed vertex of the digraph. When applying that method we recall that deletions of certain edges of the digraph may occur. However, an edge that is deleted in the middle of the computation for the k-shortest paths to vertex v may be needed later in the computation for the k-shortest paths to vertex w, $w \neq v$. Because of this fact we have to store the input digraph D', and use an auxiliary digraph D. Let A' and A represent the adjacency lists of digraphs D' and D, respectively. Initially we define A, as being A=A'. The topological sorting arrangement $u_1 u_2 \dots u_N$ of the vertices of D' is obtained. Next for each p, p=N,N-1,...,2 such that u_p is <u>not</u> a source vertex, the method of section 4.7 is applied for finding the lengths of the k-shortest paths from $u_1, u_2, \dots, u_{p-1}$ to vertex u_p. Note that the digraph to be used by procedure LENGTHQ of algorithm 4.5 is digraph D with adjacency list A and not the input digraph. Therefore, the deletion of edges by procedure LENGTHQ is performed in digraph D. When this step is completed vertex u_p may be deleted from D' since it is sure that no paths can exist to u_p from vertices after u_p in the topological sorting arrangement. D' is re-assigned to D, vertex u_{p-1} is now considered and so on.

If the lengths of the k-shortest paths obtained are to be used only for output, no storing of <u>all</u> lengths is required. In fact when the algorithm is computing the k-shortest paths from all vertices to vertex v there is no reference to the length of the j-shortest path, $1 \leq j \leq k$, to any other vertex w, $w \neq v$.

The following is an ALGOL-like formulation of the algorithm.

ALGORITHM 4.7:

```
begin comment an algorithm for finding the lengths of the k-shortest
              paths between every pair of vertices of an input acyclic
              digraph;
     procedure LENGTHQ(integer value v,j,q);
     begin
          . . .
          . . .     the same as in algorithm 4.5
          . . .
     end LENGTHQ;
     read the digraph D' and construct its adjacency lists A';
     find a topological sorting arrangement $u_1 u_2 \ldots u_N$ of the
          vertices of the digraph;
     for p:=N step -1 until 2 do
     begin if $u_p$ is not a source vertex then
             begin A:=A';
                   comment  A are the adjacency lists of an auxiliary
                            digraph D(V,E) to be used by procedure LENGTHQ;
                   find the shortest paths from all vertices to vertex $u_p$
                        in digraph D;
                   for r:=1 until p do
                        if length shortest path from $u_r$ to $u_p$ = infinity
                           then delete vertex $u_r$ from D
                        else begin short($u_r$,1):=length shortest path from
                                                     $u_r$ to $u_p$;
                                   way($u_r$):=vertex following $u_r$ (r$\neq$p) in
                                               a shortest path from $u_r$ to $u_p$
                           end
                   for (v,w) $\in$ E do
                   begin $t_{vw}$:=1;
                         $y_{vw}$:=short(w,1) + $d_{vw}$
                   end;
                   short($u_p$,2):=infinity;
                   for r:=p-1 step -1 until 1 do LENGTHQ($u_r$,2,way($u_r$));
                   output the values of the k-shortest paths to $u_r$;
             end;
             delete vertex $u_p$ from D'
     end
end
```

The correctness of the algorithm follows directly from the correctness of the method for finding the lengths of the k-shortest paths from all vertices to a fixed vertex. The same applies to its performance as can be seen from the theorem below.

Theorem 4.5:

Let D'(V,E) be an acyclic digraph with weighted edges input to algorithm 4.7. Then for calculating the lengths of the k-shortest paths between every pair of vertices of D', O(Nk+M) space and O((N+M)Nk) time are required. The total number of additions and comparisons performed within the scope of procedure LENGTHQ are $O(N^2k)$ and O(NMk), respectively.

Proof:

The space bound follows directly from theorem 4.4 because the only additional structures that algorithm 4.7 requires, in relation to algorithm 4.5, are the adjacency lists A of the auxiliary digraph D, which require O(N+M) space. Therefore algorithm 4.7 requires O(Nk+M) space. The time bound also follows from theorem 4.4. Algorithm 4.7 is basically an interation of algorithm 4.5 at most N-1 times. Hence, procedure LENGTHQ is invoked $O(N^2k)$ times, with $O(N^2k)$ additions and O(NMk) comparisons of weights performed. Therefore the time bound is O((N+M)Nk).

Now let us consider the evaluation of the algorithm in the worst case, namely, when the input digraph is a complete acyclic digraph with N vertices. Assume the weights of the digraph to be such that the deletions of edges which occur during the computation of the LENGTHQ procedure are performed at the latest possible time , i.e., a "worst worst case", as in the digraph of figure 4.1. It follows from algorithm 4.7, that finding the lengths of the k-shortest paths between every pair of vertices of a complete acyclic digraph with N vertices corresponds to solving the problem of finding the lengths of the k-shortest paths from all vertices to the sink vertex for a complete acyclic digraph with N vertices, then the same problem for a complete acyclic digraph with N-1 vertices, then N-2 vertices and so on. Consequently we can apply to this case the results of section 4.7 for determining upper bounds for the number of additions and comparisons performed in the LENGTHQ procedure. From section 4.7 we know that the total number of additions required for finding the lengths of the k-shortest paths from all vertices to the sink vertex in a complete acyclic digraph is less than or equal to $\min\{Nk, 2^{N-1}-1\}$. Consequently, the total number of additions performed within LENGTHQ, for the all pairs of vertices problem, is less than or equal to $\min\{\sum_{i=2}^{N} ik, \sum_{i=2}^{N} 2^{i-1}-1\}$. Hence

$$\text{total number of additions} \leq \min\{(N-1)(N+2)k/2,\ 2^N-N\}.$$

Similarly, we conclude that the total number of comparisons performed within LENGTHQ, for the all k-shortest paths, is less than or equal to $\min\{\sum_{i=2}^{N} i(i-2)k/2, \sum_{i=2}^{N} (i-2)[2^{i-1}-i(i-1)/6]+1\}$. Hence

$$\text{total number of comparisons} \leq \min\{(N+1)N(N-1)k/6, (N-3)2^N-(N+1)N(N-1)(N-2)/24+N+3\}.$$

Finally, we observe that it is also possible to present a variation of algorithm 4.7, which would make use of procedure LENGTH of algorithm 4.6 (which assumes the adjacency lists to be sorted in increasing values of y_{vw}'s), instead of procedure LENGTHQ.

4.10 k-shortest path between two given vertices

Given an acyclic digraph D(V,E) with weights d_{ij} associated with its edges, the problem consists of finding the length of the k-shortest path from a given vertex a to another given vertex b. The methods presented in section 4.7 for finding the lengths of the k-shortest paths from all vertices to the fixed vertex b would find in particular the k-shortest path from a to b and hence may be used for solving the present problem. However in this case a more efficient algorithm can be devised that takes advantage of the simpler nature of this problem.

The following definition was first given by Hoffman and Pavley [HoPa59]: A deviation from a shortest path from a to b, in a given digraph, is a path that coincides with this shortest path, from a up to some vertex v on the path ($v=a$ or $v=b$ are also possible); afterwards deviates to some vertex w, such that $(v,w) \in E$ and w is not the vertex that follows v in the shortest path; and finally proceeds from w to b, via the shortest path from w to b. In [HoPa59] it is shown that the second shortest path from a to b is a deviation from the shortest path. Similarly, the third shortest path is a deviation either from the shortest path or from the second shortest path, and so on. Therefore if $v_1, v_2, \ldots, v_p$ ($v_1=a$, $v_p=b$) is a shortest path from a to b, in D, in order to compute the second shortest path from a to b we need just to compute the second shortest paths to b, from all v_i, $1 \leq i < p$. Hence, if $w \neq v_i$, $1 \leq i < p$, there is no need to calculate the second shortest path from w to b. Similarly for the third shortest path, and so on.

Our problem is to devise an algorithm that would efficiently take advantage of this property and therefore reduce the total number of computations required to solve the problem. An algorithm on the lines of algorithm 4.5 or 4.6 would be inadequate because those methods compute all k-shortest paths from a vertex v to b in the iteration corresponding to vertex v.

Instead the algorithm that is proposed in this section initially seeks the computation of the second shortest path from a to b. In this process it also calculates the second shortest paths to b from the other vertices that lie on the shortest path from a to b. Afterwards, it seeks the computation of the third shortest path from a to b. In this process it also computes the third shortest paths to b from the vertices that belong to both the shortest path and the second shortest path from a to b. Also during this process the second shortest paths to b are calculated from the vertices that belong to the second shortest path from a to b, but which do not belong to its shortest path. The process is iterated, until the desired k-shortest path is computed. This strategy is similar to that used by Hoffman and Pavley. However, our algorithm possesses a better time bound than [HoPa59] and avoids much of the book-keeping existing in it. For instance we do not need to sort and merge paths as [HoPa59] requires.

Basically the same data structures used in algorithm 4.5 are required in the present one. The short(v,j), t_{vw} and y_{vw} quantities have the same meaning as before. However, way(v) has now a different interpretation; it now represents the vertex following v in the <u>last</u> j-shortest path from v to b so far computed. For example if the fourth shortest path from v to b has already been computed but the fifth has not, then way(v) contains the vertex that follows v in the fourth shortest path from v to b. The information short(v,j) is considered <u>processed</u> when the j-shortest path from v to b has been calculated (i.e. if a recursive call of the procedure occurred, for computing this j-shortest path, or the content of short(v,j) was set in the initialisation of the process).

Otherwise if the j-shortest path from v to b has not yet been computed, the information short(v,j) is said to be not processed. In an actual implementation of this method the not processed state would be indicated by storing in short(v,j) a convenient special symbol distinguishable from any path length.

We use a recursive procedure ABLENGTHQ. An interesting aspect of it is that it naturally finds the vertices v and the integers j for which the j-shortest path from v to b ought to be calculated in order to find the k-shortest path from a to b. The way of finding these values consists of testing whether short(q,t_{vq}) has already been processed, in the course of the computation of ABLENGTHQ(v,j). If it has not yet been processed then a recursive call ABLENGTHQ(q,t_{vq}) occurs that eventually computes short(q,t_{vq}). Vertex q denotes as before the vertex following v in the (j-1)-shortest path from v to b. We do not need to pass it as a parameter of the procedure because in this case q is precisely way(v).

The following is an ALGOL-like formulation of the algorithm. Note that the topological sorting pass no longer exists, since the ordering in which the j-shortest paths are computed now is determined "automatically" by the actual recursive procedure.

ALGORITHM 4.8:

```
begin comment an algorithm for finding the length of the k-shortest
          paths from vertex a to vertex b, in an acyclic digraph D(V,E);
    procedure ABLENGTHQ(integer value v,j);
    begin integer q;
          q:=way(v);
          $t_{vq}$:=$t_{vq}$ +1;
          if short(q,$t_{vq}$)=not processed then ABLENGTHQ(q,$t_{vq}$);
          if short(q,$t_{vq}$)<infinity then $y_{vq}$:=short(q,$t_{vq}$) + $d_{vq}$
          else delete edge (v,q) from A(v);
          if A(v) non-empty then
          begin short(v,j):=min{$y_{vw}$,w $\in$ A(v)};

                comment let z denote the minimizing w;
                way(v):=z;
                short(v,j+1):=not processed
          end
          else short(v,j):=infinity
          comment short(v,j) is now processed;
    end ABLENGTHQ;
    integer i;
    read the digraph D(V,E) and construct the adjacency lists A;
    read the value of k, and vertices a,b;
    find the shortest paths from all vertices to vertex b;
    short (a,1):=infinity;
    for v:=1 step 1 until N do
        if length shortest path from v to b = infinity then delete vertex v
        else begin short(v,1):=length shortest path from v to b;
                   short(v,2):=not processed;
                   way(v):=vertex following v in the shortest path
                           from v to b (v$\neq$b)
             end
    for (v,w) $\in$ E do
    begin $t_{vw}$:=1;
          $y_{vw}$:=short(w,1) + $d_{vw}$
    end
    short(b,2):=infinity;  comment short(b,2) is now processed;
    i:=1;
    while i<k do
          if short(a,i) < infinity then
          begin i:=i+1;
                ABLENGTHQ(a,i)
          end
          else i:=k;
end
```

The correctness of this strategy is based on the correctness of algorithm 4.5 and on the fact that the k-shortest path from a to b is a deviation from a j-shortest path from a to b, for some j, $1 \leq j < k$.

For evaluating the performance of algorithm 4.8, we observe that although the number of computations of the intermediate j-shortest paths has been lowered, in terms of upper bounds there exists a worst case which is similar to the worst case of section 4.7. If D(V,E) is a complete acyclic digraph, with a and b being respectively the source and sink vertices, then for a certain assignment of weights to the edges the computation of the k-shortest paths from a to b may be equivalent to the computation of the k-shortest paths from all vertices to vertex b as performed by algorithm 4.5. In fact O(outdegree(v)) steps are performed per call of ABLENGTHQ(v,j) corresponding to the number of comparisons required for the minimization of $\{y_{vw}, w \in A(v)\}$; there are at most k-1 calls of ABLENGTHQ from outside its body (the calls ABLENGTHQ(a,i)); there are, at most, (k-1)(N-2) recursive calls of ABLENGTHQ; the part of the algorithm outside the recursive procedure requires O(N+M+k) time. Therefore, an upper bound for algorithm 4.8 is O((N+M)k) time. The space bound is also equivalent to algorithm 4.5, namely O(Nk+M).

Finally, we add some more remarks about this method. The alternative strategy of maintaining the nodes of each A(v) list, sorted according to increasing values of y_{vw}'s (as in algorithm 4.6), can also be applied to this case. Also, it is obvious that (i) an algorithm for finding the k-shortest paths from all vertices to the fixed vertex b, (ii) an algorithm for the k-shortest paths from a fixed vertex to all others, and (iii) an algorithm for all k-shortest paths, can be devised based on the strategy of the present algorithm 4.8. These algorithms would have bounds similar to those previously described.

4.11 The longest path

Given an acyclic digraph D(V,E) with non-negative weights d_{ij} associated with its edges, the problem consists of finding a path which has the longest non-infinite length among all possible paths in the digraph. When only positive weights are considered, such a path necessarily starts with a source vertex and ends with a sink vertex.

The problem is directly related to PERT (Project Evaluation and Review Technique) networks, for a critical path in such a network is precisely the longest path in the corresponding digraph. Therefore the present problem is also handled in the vast literature of PERT, CPM (Critical Path Method) and scheduling project networks. Klein [Kl67], Lass [La65], Chen and Wing [ChWi66], Leavenworth [Le61], Eisenman and Shapiro [EiSh62], Elmaghraby [El70a], Charnes and Cooper [ChCo62], Furtado [Fu73], Even [Ev73], Price [Pr71], among many others, have approached the critical path (or longest path) problem and solutions have been presented, which vary from efficient algorithms - such as presented in [Fu73] or [Ev73] - to less efficient methods, as presented in [ChWi66]. The algorithms [Fu73, Ev73] require $O(N^2)$ time for finding the longest path in an acyclic digraph, but a minor alteration (basically adapting them to adjacency lists) transforms them into $O(N+M)$ methods. The algorithm [ChWi66] for finding the longest path in the digraph computes initially the lengths of all longest paths between every pair of vertices. In addition it requires the computation of the reachability matrix of the digraph. We can also mention that some of the methods for finding the longest path in a digraph require a topological sorting of its vertices to be performed before the actual computation of the longest path.

Our proposed method uses a recursive backtracking procedure PATHL which - besides some minor differences - is essentially similar to procedure PATH of algorithm 4.1. At the end of the computation of a call PATHL(v), the longest path of the digraph, starting from vertex v has been calculated. The method does not require a topological sorting to be performed. It starts by computing the sets S_1 and S_0 of sink and source vertices, respectively. The vector mark is used for preventing the exploration of each vertex more than once. The length and route vectors are used for storing, for each vertex v, the length of the longest path in the digraph, starting with v and the vertex that follows v in such a path, respectively. The initialisation is executed as follows: If v is a sink vertex then mark(v) is initialised with true, length(v) and route(v) are initialised with zero. Otherwise, mark(v) is initially set to false and length(v) to infinity. As before, the special symbol "zero" is used in the route vector to indicate the occurrence of a last (sink) vertex in a longest path. For each source vertex u, a non-recursive call PATHL(u) occurs, which will compute the longest path which starts from u. The longest path in the digraph is clearly the longest of all such paths from these source vertices. In each computation of an invoked PATHL(v), mark(v) is set to true and all edges from v are explored. Assume the computation of PATHL(v) and the exploration of an edge(v,w).

(i) If w is found unmarked, this means that w has not been explored yet and a call PATHL(w) occurs. On returning of this call, length(w) contains the length of the longest path in the digraph, starting from w. A test is therefore made as to whether the path starting with v - and proceeding by the longest path from w - is longer than the so far computed longest path from v. In the affirmative case, this path from v through w, becomes the new longest path from v.

(ii) If w is found to be marked then it will not be explored again and no recursive call of PATHL(w) is invoked. In this case, length(w) contains already the length of the longest path in the digraph, starting with w. A similar comparison and action as in (i) is therefore undertaken.

At the end of the whole process variable total contains the length of the longest path in the digraph and variable first points to the first vertex in the longest path. This path can be obtained in the usual way: if $v_1, v_2, \ldots, v_k$ is the longest path, then v_1 = first, v_{j+1}=route(v_j) for $1 \leq j < k$, and route(v_k)=0.

The following is the algorithm for computing the longest path in an acyclic digraph in an ALGOL-like notation.

ALGORITHM 4.9:

```
begin comment  an algorithm for finding the longest path in an acyclic
               digraph D(V,E);
      procedure PATHL(integer value v);
      begin mark(v):=true;
            for w $\in$ A(v) do
            begin if $\neg$ mark(w) then PATHL(w);
                  if length(w) + $d_{vw}$ > length(v) then
                  begin length(v):=length(w) + $d_{vw}$;
                        route(v):=w
                  end
            end
      end PATHL;
      integer first, total;
      read the digraph and construct the adjacency lists A;
      for j:=1 until N do
      begin mark(j):=false;
            length(j):=-infinity
      end
      $S_1$:=set of sink vertices;
      $S_0$:=set of source vertices;
      for u $\in$ $S_1$ do
      begin mark(u):=true;
            length(u):=0;
            route(u):=0
      end
      total:=-infinity;
      for u $\in$ $S_0 \setminus S_1$ do
      begin PATHL(u);
            if length(u) > total then
            begin total:=length(u);
                  first:=u
            end
      end
end
```

The correctness of the proposed strategy follows from the lemma below whose proof can be basically established by induction on the computations of PATHL.

Lemma 4.8:

Let D(V,E) be an acyclic digraph with non-negative weights d_{ij} assigned to its edges. Consider D input to algorithm 4.9 and let $u \in V$ be such that u is not a sink vertex. Then, during the execution of this algorithm a call PATHL(u) occurs, and by the end of this computation length(u) contains the length of the longest path of the digraph starting from u.

The performance of the algorithm can be evaluated by the following theorem, which also ensures that the present strategy is optimal within a constant factor.

Theorem 4.6:

Let D(V,E) be an acyclic digraph with non-negative weights d_{ij} assigned to its edges. Consider D input to algorithm 4.9. Then the longest path of the digraph is computed in O(N+M) time, using O(N+M) space. A total of M additions and $M + |S_0 \setminus S_1|$ comparisons of weights are performed, where S_0 and S_1 are the sets of source and sink vertices respectively.

The time bound mentioned in the theorem above follows directly from the fact that if $u \notin S_1$ then precisely one call PATHL(u) is invoked, and otherwise no such call occurs. For each computation of PATHL(v) precisely outdegree(v) additions and comparisons of weights are performed. For each vertex v, such that $v \in S_0 \setminus S_1$, one extra comparison of weights is made outside PATHL. Furthermore, O(N) time is spent in the part of the algorithm outside the recursive procedure, beyond the O(N+M) time required for the input of the digraph.

4.12 The k-longest path

Given an acyclic digraph D(V,E) with non-negative weights d_{ij} assigned to its edges and an integer k, $k>1$, the problem consists of finding the length of the k-longest (non-infinite) path, from a source to a sink vertex, in the digraph.

Our approach to the problem consists of applying results from section 4.11 in which a strategy for finding the longest path in the digraph was presented, combined with results from section 4.10, which contains a method for finding the length of the k-shortest path between two given vertices of the digraph.

The data structures of the proposed method are the following: we use the way(v), t_{vw} and y_{vw} quantitites of algorithm 4.8 with similar purposes, except that they now refer to j-longest paths from v to sink vertices, instead of j-shortest paths. The short(v,j) quantitites are replaced by long(v,j), which contain the lengthsof the j-longest path in the digraph, starting from v and ending with a sink vertex. If there exists such a j-longest path, but there is no (j+1)-longest path, then long(v,j+1) is defined to be equal to -infinity. In addition, we use the length and index vectors. Denoting by S_0 and S_1 respectively the set of source and sink vertices of the digraph we have for each $v \in S_0 \setminus S_1$ length(v) storing the j-longest path from v to any sink vertex so far computed and index(v)=j.

A recursive procedure LENGTHQL(v,j) is used, which is similar to procedure ABLENGTHQ of algorithm 4.8, except that the minimization is now replaced by a maximization, and the absence of a j-longest path from v to any sink vertex (which occurs when A(v) becomes empty) is now indicated by setting long(v,j) to -infinity, whilst in algorithm 4.8 short(v,j):=infinity was used in the corresponding case.

The process is initiated by finding the longest path in the digraph using algorithm 4.9. Next the variables are set to their initial values in a similar way as in algorithm 4.8 except that they should refer to longest paths. The long(v,2) quantities are set to "not processed", which has a similar meaning as in algorithm 4.8. For all vertices $v \in S_1$, the following additional initialisations occur, length(v) is set to the longest path from v, index(v) is set to 1 and long(v,2) is set to -infinity therefore becoming "processed". The first call of the procedure is the call LENGTH(u,2) where u is the first vertex in the longest path of the digraph. At the end of this computation the length of the second longest path from u to a sink vertex is stored in long(v,2). Now, if we assign this value to length(u), then the length of the second longest path in the digraph from a source to a sink vertex is calculated simply by maximizing $\{length(v), v \in S_0\}$. To calculate the length of the third longest path from a source to a sink vertex of the digraph assign to u the value of the first vertex of the second such longest path, call LENGTHL(u,index(u)) and repeat the process. The iteration is performed until the length of the k-longest path is obtained, or it is detected that no such path exists.

The following is an ALGOL-like formulation of this strategy. At the end of the process, length(u) contains the length of the k-longest path from a source to a sink vertex if there is one. If it does not exist length(u) contains the length of a j-longest path, where j is the greatest integer $(j \geq 1)$ such that the digraph admits a j-longest path from a source to a sink vertex.

ALGORITHM 4.10:

```
begin comment an algorithm for finding the k-longest path from a source
          to a sink vertex, in an acyclic digraph D(V,E);
   procedure LENGTHQL(integer value v,j);
   begin integer q;
            q:=way(v);
            $t_{vq}$:=$t_{vq}$ + 1;
            if long(q,$t_{vq}$)=not processed then LENGTHQL(q,$t_{vq}$);
            if long(q,$t_{vq}$) > -infinity then $y_{vq}$:=long(q,$t_{vq}$) + $d_{vq}$
            else delete edge (v,q) from A(v);
            if A(v) non-empty then
            begin long(v,j):=max{$y_{vw}$, w $\in$ A(v)};
                  comment let z denote the maximizing w;
                  way(v):=z;
                  long(v,j+1):=not processed
            end
            else long(v,j):=-infinity;
            comment long(v,j) is now processed;
   end LENGTHQL;
   integer i,u;
   read the digraph and construct the adjacency lists A;
   read the value of k;
   $S_1$ :=set of sink vertices;
   $S_0$ :=set of source vertices;
   find the longest path in the digraph, from a source vertex;
   u:=first vertex in the longest path of the digraph;
   for v:=1 until N do
      if v $\in$ $S_1$ and  v $\in$ $S_0$ then delete vertex v
      else begin long(v,1):=length of the longest path from v;
                 long(v,2):=not processed;
                 way(v):=vertex following v in the longest path
                         path from v (v $\notin$ $S_1$ )
```

```
        for (v,w) ∈ E do
        begin $t_{vw}$:=1;
                $y_{vw}$:=long(w,1) + $d_{vw}$
        end
        for v ∈ $S_o$ do
        begin length(v):=long(v,1);
                index(v):=1
        end
        for v ∈ $S_1$ do long(v,2):=-infinity;
                comment if v ∈ $S_1$ then long(v,2) is processed;
        i:=1;
        while i<k do
        begin i:=i+1;
                index(u):=index(u) + 1;
                LENGTHQL(u,index(u));
                length(u):=long(u,index(u));
                if long(u,index(u))=-infinity then
                        delete u from $S_o$;
                if $S_o$ non-empty then
                        u:=maximizing v of max{length(v), v ∈ $S_o$}
                else i:=k
        end
end
```

The correctness of the strategy follows from the correctness of algorithms 4.8 and 4.9. Its space requirements are $O(Nk+M)$ cells and the time bound is $O((N+M)k)$. For each pair (v,j), $v \in V$, $2 \leq j \leq k$, at most one call LENGTHQL is invoked. For each of these calls at most one addition and maximization of weights are performed. This maximization consists of at most outdegree(v) comparisons corresponding to the number of nodes in the A(v) list. Therefore in relation to the procedure LENGTHQL the following are valid:

number of additions of weights $\leq Nk$

number of comparisons of weights $\leq Mk$

Outside the recursive procedure, at most M additions of weights (for initializing the y_{vw}'s) and $|S_0|k$ comparisons of weights (when returning from a non-recursive call) are performed.

The k-longest path itself can be obtained by employing techniques similar as described in section 4.7. Also the method of decreasing the number of comparisons performed inside the body of the recursive procedure, as described in that same earlier section, can be applied for this case.

4.13 Conclusions

We have presented algorithms for solving some different shortest paths problems in acyclic digraphs with weighted edges. The justification for developing a set of algorithms restricted to acyclic digraphs is that these structures represent an important class of digraphs and constitute of mathematical models for some important practical problems. Furthermore the algorithms which were presented in this chapter have better time bounds than corresponding algorithms which apply to digraphs in which cycles may exist. These remarks do not apply to the longest and k-longest path algorithms, since these problems are normally restricted to acyclic digraphs.

The implementation of the methods presented in this chapter is simple. The algorithms are based on the execution of recursive procedures which can be considered as short and simple. Furthermore, many of the algorithms presented apply procedures defined in other algorithms to different control structures, which simplifies the implementation of the whole set of algorithms.

CHAPTER 5

k-SHORTEST PATHS

5.1 Introduction

This chapter is devoted to the discussion and proposal of a strategy for solving problems of finding k-shortest paths in digraphs with weighted edges. As opposed to the previous chapter, the digraphs now considered may contain cycles. No restriction is made for the values that the weights can assume, except that no cycles with negative length are allowed. Note that there is no solution for the problem if the digraph contains such a cycle.

k-shortest path problems have been the subject of research for some time. For instance, an efficient algorithm for the k-shortest paths between two specified vertices has been known since 1959 [HoPa59]. Dreyfus [Dr69] has discussed this algorithm and extended it for finding the k-shortest paths from all vertices to a fixed vertex. Dreyfus has also improved the algorithm by Bellman and Kalaba [BeKa60] which also solves the k-shortest path problem from all vertices to a fixed one. These two extensions were shown in [Dr69] to be equivalent in time requirements. Bellman and Kalaba have actually stated their algorithm for the case k=2, i.e. finding second best paths. The generalization of it again appears in [Dr69]. However, Dreyfus' algorithm for an arbitrary k, is better than the strict generalization of the method by Bellman and Kalaba, since it requires fewer comparisons of weights. A survey paper has also been published on the subject by Pollack [Po61].

As for the problem of finding the k-shortest paths between every pair of vertices, Minieka [Mi74] has presented two solutions, corresponding respectively to generalizations of the algorithms by Floyd [Fl62] and Dantzig [Da66], which find all shortest paths in a digraph. The total number of additions and comparisons required by both of Minieka's algorithms are $2N^3k^2$ and $2N^3(k^2+k)$ respectively, as stated in [Mi74]. Another algorithm was presented by Beilner [Be72], based on the solutions given for the all shortest paths problem by Hoffman and Winograd [HoWi71] and Floyd [Fl62]. Beilner's algorithm requires $\frac{1}{3}N^{5/2}k^{5/2} + 5N^{5/2}k^{3/2} + O(N^{3/2}k^{5/2})$ additions/subtractions and $O(N^3k^3)$ comparisons, as mentioned in [Be72].

All these algorithms refer to the problem of finding k-shortest paths such that cycles may be part of the paths. Note that every shortest path in a digraph contains no cycle . However, a k-shortest path k>1, may contain one. For instance, the second shortest path from a vertex to itself is a cycle. If only cycle-less paths are desired, other algorithms ought to be used: Clarke, Krikorian and Rausen [ClKrRa63], Pollack [Po61a], Yen [Ye71], Lawler [La72].

The problem of finding the k-shortest path between two given vertices is the subject of section 5.2. Finding the k-shortest paths from all vertices to a fixed one, from one fixed vertex to all others and between every pair of vertices constitute sections 5.3, 5.4 and 5.5 respectively. Some further remarks form the last section. The problems that we have considered involve finding paths which may contain cycles.

5.2 k-shortest paths between two vertices

Given a digraph D(V,E) with weights d_{ij} assigned to its edges, vertices a,b $\in$ V and an integer k>1, the problem consists of finding the k-shortest path from a to b in D.

Our approach consists of adapting algorithm 4.8, which finds the k-shortest path from a to b in an acyclic digraph, to an algorithm for handling digraphs possibly with cycles. Observe that the strategy in which are based the other k-shortest paths algorithms of Chapter 4 (algorithms 4.5, 4.6 and 4.7) is inadequate for manipulating digraphs with cycles.

This follows from the fact that in those algorithms the progress of the computation is such that in each iteration corresponding to each vertex of the digraph all p-shortest paths ($2 \leq p \leq k$) from the considered vertex to vertex b are calculated. This strategy is satisfactory when the digraph is acyclic, but it does not produce the correct solution for digraphs with cycles, because in the latter case if the j-shortest path from vertex v to b contains the i-shortest path from vertex w to b, it is now possible that the j'-shortest path from w to b contains the i'-shortest path from v to b ($i \leq j$; $i' \leq j'$; and $i,j,i',j' > 1$). Therefore, we can not compute the j-shortest path from v before the computation of the i-shortest path from w. Similarly, the j'-shortest path from w can not be computed before the i'-shortest path from v. Algorithm 4.8 however iterates p for $2 \leq p \leq k$, and within each iteration of p the vertices are recursively considered, for computing j-shortest paths, $j \leq p$.

The basic alteration required in algorithm 4.8 for handling digraphs with cycles is that the lengths of the second, third, etc. shortest paths from b to itself are no longer necessarily infinite, and therefore they need to be computed. In fact, the computation of the second shortest path from b to itself ought to be the first among all computations for the second shortest paths to b, since the second shortest path from any vertex to itself depends only on shortest paths. Another alteration that is obviously required is that we should not apply algorithm 4.2 for solving the step of finding the shortest paths from all vertices to b since algorithm 4.2 only manipulates acyclic digraphs. Clearly an appropriate algorithm (which unfortunately has a greater time bound) has to be used for finding all shortest paths to vertex b in a digraph which may contain cycles. We recall that finding all such shortest paths constitutes one of the steps of algorithm 4.8.

The following is an ALGOL-like notation of the algorithm for finding the k-shortest paths from vertex a to vertex b in a digraph D where cycles may occur. The data structures that appear in it are the same - and have similar interpretations - as those used in algorithm 4.8.

ALGORITHM 5.1

```
begin comment an algorithm for finding the length of the k-shortest
              path from vertex a to vertex b, in a digraph D(V,E):
      procedure ABLENGTHQ(integer value v,j);
      begin
           . . .
           . . .    as in algorithm 4.8
           . . .
      end ABLENGTHQ;
      integer i;
      read the digraph D(V,E) and construct the adjacency lists A;
      read the value of k, and vertices a,b;
      short(a,1):=infinity;
      find the shortest paths from all vertices to vertex b;
      for v:=1 step 1 until N do
           if length shortest path from v to b = infinity then delete
              vertex v
           else begin short(v,1):=length shortest path from v to b;
                      short(v,2):=not processed;
                      way(v):=vertex following v in the shortest path
                              from v to b ($v \neq b$)
                end
      for (v,w) $\in$ E do
      begin $t_{vw}$:=1;
            $y_{vw}$:=short(w,1) + $d_{vw}$
      end
      if A(b) non-empty then
      begin short(b,2):=min $\{y_{bw}, w \in A(b)\}$;
            comment let z denote the minimizing w;
            way(b):=z;
            short(b,3):=not processed
      end
      else short(b,2):=infinity;
      comment short(b,2) is now processed;
      i:=1;
      while $i<k$ do
            if short(a,i)$<$infinity then
            begin i:=i+1;
                  ABLENGTHQ(a,i)
            end
            else i:=k
end
```

The correctness of this method is based on the correctness of algorithm 4.8 and on the lemma below:

Lemma 5.1:

Let D(V,E) be a directed graph with weighted edges, $v_1, b \in V$ and integer $j>1$. Let D be input to algorithm 5.1. Then the computation of ABLENGTHQ(v_1,j) for finding the j-shortest path from v_1 to b does not cause the recursive call ABLENGTHQ(v_1,j) to be eventually invoked.

Proof:

It follows from the examination of the algorithm that a recursive call ABLENGTHQ(v_2,j'), $v_2 \in V$, can only be invoked from the computation of ABLENGTHQ(v_1,j) if $j' \leq j$ and $v_2 \in A(v_1)$. Therefore a circularity in this computation can only occur if there exists a cycle $v_1, v_2, \ldots, v_p, v_1$ $(p>1)$, such that ABLENGTHQ(v_i,j) invokes ABLENGTHQ(v_{i+1},j) for all $1 \leq i < p$ and ABLENGTHQ(v_p,j) invokes ABLENGTHQ(v_1,j). On the other hand if ABLENGTHQ(v_1,j) invokes ABLENGTHQ(v_2,j) this means that all i-shortest paths from v_1 to b, $1 \leq i \leq j-1$, contain v_2. Consequently by an inductive argument we conclude that if the circularity in the computation of ABLENGTHQ(v_1,j) occurs then all i-shortest paths from v_1 to b, $1 \leq i \leq j-1$, contain v_p and all such i-shortest paths from v_p to b contain v_1. This contradicts the fact that a shortest path between two vertices in D contains no cycle.

Observe that a corresponding lemma for algorithm 4.8 would be trivially true since the input digraph in that case is supposed not to contain cycles.

As for the performance, there can be at most O(Nk) calls of ABLENGTHQ which correspond to O(Nk) additions, and O(Mk) comparisons of weights. Therefore the total time spent in the computation of the procedure is $O((N+M)k)$. The part of the algorithm outside the scope of the recursive procedure requires $O(N+M+k)$ time, beyond that time required for finding the shortest paths from all vertices to vertex b. This last step requires $O(N^2)$ time if we assume that only non-negative weights are allowed. Otherwise, if negative weights may also occur $O(NM)$ time is required for this step. Therefore the total time bound is $O(N^2+(N+M)k)$ or $O(NM+(N+M)k)$, corresponding to each of these two cases respectively. However we emphasize that in addition to calculating the shortest paths from all vertices to vertex b, the present method requires not more than $O((N+M)k)$ time and this bound is not necessarily attained. The space bound is $O(Nk+M)$.

5.3 k-shortest paths from all vertices to a fixed vertex

Given a digraph $D(V,E)$ with weights d_{ij} assigned to its edges and a vertex $b \in V$, the problem consists of finding the k-shortest paths from all vertices to vertex b.

This problem can be solved by slightly modifying algorithm 5.1. In fact, we only need to alter that part of the algorithm corresponding to the control of the non-recursive calls of the procedure, i.e. those calls invoked outside its body. We want to ensure that calls of ABLENGTHQ for computing the length of the j-shortest path from v to b only occur if this length has not been previously computed and the length of the (j-1)-shortest path from v to b is known to be finite. Therefore, in algorithm 5.1, replace the (entire) statement

```
while i<k do
        . . .
        . . .
        . . .
```

by:

```
S:=V;
while i<k do
      if S = empty then i:=k
      else begin i:=i+1;
                 for u ∈ S do
                     if short(u,i-1) = infinity then
                     begin delete u from S;
                           short(u,k):=infinity
                     end
                     else if short(u,i) = not processed then ABLENGTHQ(u,i)
           end
```

The newly introduced set S contains initially the subset of vertices of the digraph for which the lengths of their shortest paths to b are finite. The deletions in S are to avoid unnecessary iterations when computing j-shortest paths whose lengths are already known to be infinite.

Since the time required for the execution of the altered while statement is O(Nk) and not more than O(N) space has been added to the algorithm, we conclude that the time and space bounds of algorithm 5.1 have been maintained in the present case.

5.4 k-shortest paths from a fixed vertex to all vertices

Given a digraph D(V,E) with weights d_{ij} assigned to its edges and a vertex a $\in$ V, the problem consists of finding the k-shortest paths from vertex a to all vertices of the digraph.

As before, the problem can be solved by applying the strategy of section 5.3 for finding the k-shortest paths from all vertices to a certain vertex b to the converse digraph $\bar{D}$ of D, with b=a. The time and space bounds are the same as those of section 5.2.

5.5 k-shortest paths between every pair of vertices

Given a digraph D(V,E) with weights d_{ij} assigned to its edges, we wish to obtain the k-shortest paths between every pair of vertices of the digraph.

We can solve this problem by applying the strategy of section 5.3 - for finding the k-shortest paths from all vertices to a fixed vertex b - iteratively, at most N times, varying vertex b. The execution of the recursive procedure ABLENGTHQ in this case corresponds to $O(N^2k)$ additions and O(NMk) comparisons. The part of the algorithm outside the procedure requires $O(N^3+N^2k)$ time, corresponding to $O(N^3)$ time for finding the shortest paths between every pair of vertices and $O(N^2k)$ for the <u>while</u> $i<k$ <u>do</u> loop (of section 5.3), which controls the non-recursive calls of the procedure. The other steps involved in the initialization of the process require less than $O(N^3+N^2k)$ time. Therefore the total time bound is $O(N^3+N(N+M)k)$.

If D contains negative weights it is advantageous for the step of finding the shortest paths to vertex b, to apply once an all shortest paths algorithm with overall time bound $O(N^3)$. In this case, $O(N^2)$ extra space ought to be added for storing the matrix of all shortest paths. Therefore, since the strategy of section 5.3 requires O(Nk+M) space we conclude that the space bound for the present case is $O(N^2+Nk)$. If D contains only non-negative weights we may choose not to apply an all shortest paths algorithm and instead calculate all shortest paths to vertex b in each iteration of b. In this last case the space bound of $O(Nk+M)$ can be maintained.

In algorithm 5.1 and in the solution of the subsequent problems of this chapter the decision as to whether or not a call ABLENGTHQ(v,j) should be invoked, for computing the j-shortest path from v to b is taken by testing whether short(v,j) is "processed". Alternatively we can adopt the following strategy described in [Dr69]. After finding the shortest paths from all vertices to vertex b, we find an ordering $v_1 v_2 \ldots v_N$ of the vertices of the digraph, such that if the shortest path from v_p to b contains less vertices than the shortest path from v_q to b, then $p<q$. If we process the vertices of the digraph in the above ordering $v_1 v_2 \ldots v_N$, for each j $2 \leq j \leq k$, then we can disregard the "processed" and "not processed" information of short(v,j), since the computation of each j-shortest path from v to b depends only on i-shortest paths that have already been computed. Hence if this ordering is used no recursive call of ABLENGTHQ would occur, since short(q, t_{vq}) would always be found "processed" in the corresponding test inside the procedure.

5.6 Conclusions

We have presented in this chapter solutions to k-shortest paths problems in digraphs. These solutions were obtained by slightly modifying strategies presented in Chapter 4 for solving such problems in acyclic digraphs.

One interesting aspect of the k-shortest path problem is that, unlike the shortest path problem, the following property does not hold:

> "If (v,w) is an edge of the digraph with weight d_{vw}, and short(v,k) denotes the length of the k-shortest path from v to a certain certex b, then short(v,k) = min $\{$short(w,k) + d_{vw} , for $w \in A(v)\}$."

If $k=1$ this assertion is true, but in general it does not hold for $k>1$. There may exist an integer j, $1 \leq j \leq k$ and vertex $z \in A(v)$ such that

$$\text{short}(v,k-1) < \text{short}(z,j) + d_{vz} < \min \{\text{short}(w,k) + d_{vw}, \text{ for } w \in A(v)\}.$$

In this case, clearly

$$\text{short}(v,k) < \min \{\text{short}(w,k) + d_{vw}, \text{ for } w \in A(v)\}.$$

The corresponding correct expression which holds for $k \geq 1$ is:

$$\text{short}(v,k) = \min_k \{\text{short}(w,j) + d_{vw}, \text{ for } w \in A(v) \text{ and } 1 \leq j \leq k\},$$

with $\min_k$ denoting the k-th minimum.

CONCLUSIONS

This thesis has presented algorithms for solving certain computational graph theoretic problems. We have largely employed backtracking as the basic strategy in most of the algorithms. The use of backtracking as a convenient tool for treating graph problems has again been emphasized. Its importance can be assessed by the fact that in recent years a wide variety of graph problems have been successfully solved by algorithms based upon backtracking strategies. Furthermore it provides a methodical way of approaching a possible solution for a given problem and also tends to produce algorithms that in general resemble one another. This uniformity and resemblance of the algorithms, we suppose, are factors that contribute to the elegance of the solutions as a whole, and also in certain cases, may provide a means of ranking the difficulty of some different problems through their backtracking algorithmic solutions. This evaluation would be undertaken by a simple examination of the algorithms, since their resemblance to a certain extent, facilitates the task of "comparing" these algorithms.

Examining the solutions to the problems considered in this thesis, we note that the backtracking algorithms are based on recursive procedures and most of them may be fitted essentially into the following formulation. Each of the symbols B1, B2,...,B9 denotes a (possibly empty) sequence of statements, which varies according to which particular algorithm is to be fitted into the formulation.

```
begin
    procedure X (integer value v; ...);
    begin mark vertex v;
          insert vertex v in the stack;
          B1;
          for w ∈ A(v) do
          begin B2;
                if w is not marked then
                begin B3;
                      X(w,...);
                      B4
                end
                else B5
          end;
          B6;
          delete vertex v from the stack
    end X;
    B7;
    read the digraph and construct the adjacency lists A:
    B8;
    X(v,...);
    B9
end
```

Another point on which we would like to comment is the way we chose to discuss the correctness of the different strategies throughout the thesis. Essentially, we have proposed proofs by induction which were derived directly from the recursiveness of the procedures. In fact there is a relation between them. The assumptions that are made, concerning the states of different variables of a recursive procedure, at the start of an arbitrary computation of it, may be associated with the assumptions corresponding to the inductive hypothesis of a proof by induction for the correctness of this procedure. An equivalent statement could be made for procedures using iteration instead of recursion. However the generally shorter and clearer description of a strategy achieved by employing recursion tends to make such proofs more transparent.

The technique of deriving proofs by induction from recursive algorithms has been commonly used through the years. However, much less common is the converse technique: the derivation of a recursive algorithm from a suitable proof by induction of a certain theorem related to an algorithmic problem. The application of this technique is often not convenient nor perhaps possible, but an example where a recursive algorithm was obtained by conveniently translating a proof by induction is presented in the appendix to this thesis. The example has considered the proof by induction of Dilworth's decomposition theorem for partially ordered sets. This theorem states that the minimum number of chains which cover a poset equals the maximum number of elements in an antichain. The derived algorithm finds a minimal chain covering for the poset from maximal antichains. This derivation was possible because of the fact that the chosen proof implicitly considered the construction of a minimal chain covering. However the fact that it may be possible to translate directly a proof by induction into an algorithm by recursion emphasizes the relationship between them.

To the best of our knowledge all algorithms described in Chapters 2 to 5 of this thesis have performances, in terms of time and space bounds, at least as good as existing algorithms for equivalent tasks. The bounds of the algorithms we have proposed have been calculated through the thesis and appear in the following table, which summarizes the list of the specific problems that we have considered.

Two methods mentioned in the thesis have been left out of this summary table. The first is a method for obtaining a topological sorting arrangement, which can be derived from the results described in Chapter 1, which showed a relationship existing between ternary search trees and topological sortings in a partially ordered set. Such a method would be in general worse for topological sorting than known methods. Therefore we do not consider the practical proposal of a topological sorting algorithm based on results of Chapter 1. However, the topological and quasi-topological sorting properties of ternary search trees are interesting characteristics of ternary trees and are therefore worth describing. The second method which has not been listed in the table is the algorithm for the problem related with Dilworth's theorem described in the appendix. As already mentioned our purpose in describing that method was to provide an example for illustrating how a proof by induction may be directly translated into a recursive algorithm.

SECTION NUMBER	PURPOSE OF THE ALGORITHM	TIME BOUND	SPACE BOUND
2.2	Finding the T topological sorting arrangements of an acyclic digraph	O((N+M)T)	O(N+M)
3.3	Finding the C elementary cycles of a digraph	O(N+M(C+1))	O(N+M)
3.8	Finding a fundamental set of cycles of an undirected graph (with explicit output)	O(N.M)	O(N+M)
3.10	Finding a fundamental set of cycles of an undirected graph (with reduced edited output)	O(N+M)	O(N+M)
3.12	Finding the C elementary cycles of an undirected graph	O(N+M(C+1))	O(N+M)
4.2	Finding the shortest path between two vertices of an acyclic diagraph	O(N+M)	O(N+M)
4.3	Finding the shortest paths from all vertices of an acyclic digraph to a fixed vertex	O(N+M)	O(N+M)
4.4	Finding the shortest paths from a fixed vertex to all others in an acyclic digraph	O(N+M)	O(N+M)
4.5	Finding the shortest paths between every pair of vertices in an acyclic digraph	O((N+M)N)	O(N+M)
4.6	Finding the shortest path between two given vertices of an acyclic digraph visiting a given subset of vertices	O(N+M)	O(N+M)
4.7	Finding the k-shortest paths from all vertices of an acyclic digraph to a fixed vertex	O((N+M)k)	O(Nk+M)
4.8	Finding the k-shortest paths from a fixed vertex of an acyclic digraph to all others	O((N+M)k)	O(Nk+M)
4.9	Finding the k-shortest paths between every pair of vertices of an acyclic digraph	O((N+M)Nk)	O(Nk+M)

SECTION NUMBER	PURPOSE OF THE ALGORITHM	TIME BOUND	SPACE BOUND
4.10	Finding the k-shortest paths between two given vertices of an acyclic digraph	$O((N+M)k)$	$O(Nk+M)$
4.11	Finding the longest path in an acyclic digraph (non-negative weights)	$O(N+M)$	$O(N+M)$
4.12	Finding the k-longest path in an acyclic digraph (non-negative weights)	$O((N+M)k)$	$O(Nk+M)$
5.2	Finding the k-shortest paths between two given vertices of a digraph (non-negative weights)	$O(N^2+(N+M)k)^*$	$O(Nk+M)^{**}$
5.3	Finding the k-shortest paths from all vertices of a digraph to a fixed vertex (non-negative weights)	$O(N^2+(N+M)k)^*$	$O(Nk+M)^{**}$
5.4	Finding the k-shortest paths from a fixed vertex of a digraph to all others (non-negative weights)	$O(N^2+(N+M)k)^*$	$O(Nk+M)^{**}$
5.5	Finding the k-shortest paths between every pair of vertices of a digraph (non-negative weights)	$O(N^3+(N+M)Nk)^{**}$	$O(Nk+M)^{***}$

* $O(NM+(N+M)k)$ when negative weights are allowed

** Remains the same when negative weights are allowed

*** $O(N^2+Nk)$ when negative weights are allowed

APPENDIX

ON DILWORTH'S PROBLEM

The following theorem is due to Dilworth [Di50].

Dilworth's decomposition theorem:

Given a poset $(P, \preccurlyeq)$, the minimal number of disjoint chains which cover P is equal to the maximal number of elements in an antichain.

Several proofs for this theorem have been published, since it was first formulated: Dilworth [Di50], Fulkerson [Fu56], Dantzig and Hoffman [DaHo56], Perles [Pe63, Pe63a], Tverberg [Tv67], among others. A dual of this theorem, obtained by interchanging the roles of chains and antichains has been established by Mirsky [Mi71].

A problem related to Dilworth's theorem can be formulated as: given a finite poset $(P, \preccurlyeq)$ find a covering of P by a minimal number of disjoint chains. The poset represented by figure A.1, for example, has {4,5,2} as a maximal antichain and a minimal chain covering is {1,7,4}, {9,5,8,6} and {2,3}. Dantzig and Hoffman [DaHo56] have solved this problem by employing linear programming techniques.

In this appendix, we seek a solution for a similar problem except that we employ maximal antichains in order to obtain the minimal chains. The algorithm presently described was obtained by deriving an algorithmic translation from Perles' proof [Pe63] of Dilworth's theorem. We wish to emphasize that our principal aim in this appendix is not the proposal of an algorithm for solving the minimal chain problem, but to illustrate with an example how proofs employing induction can motivate

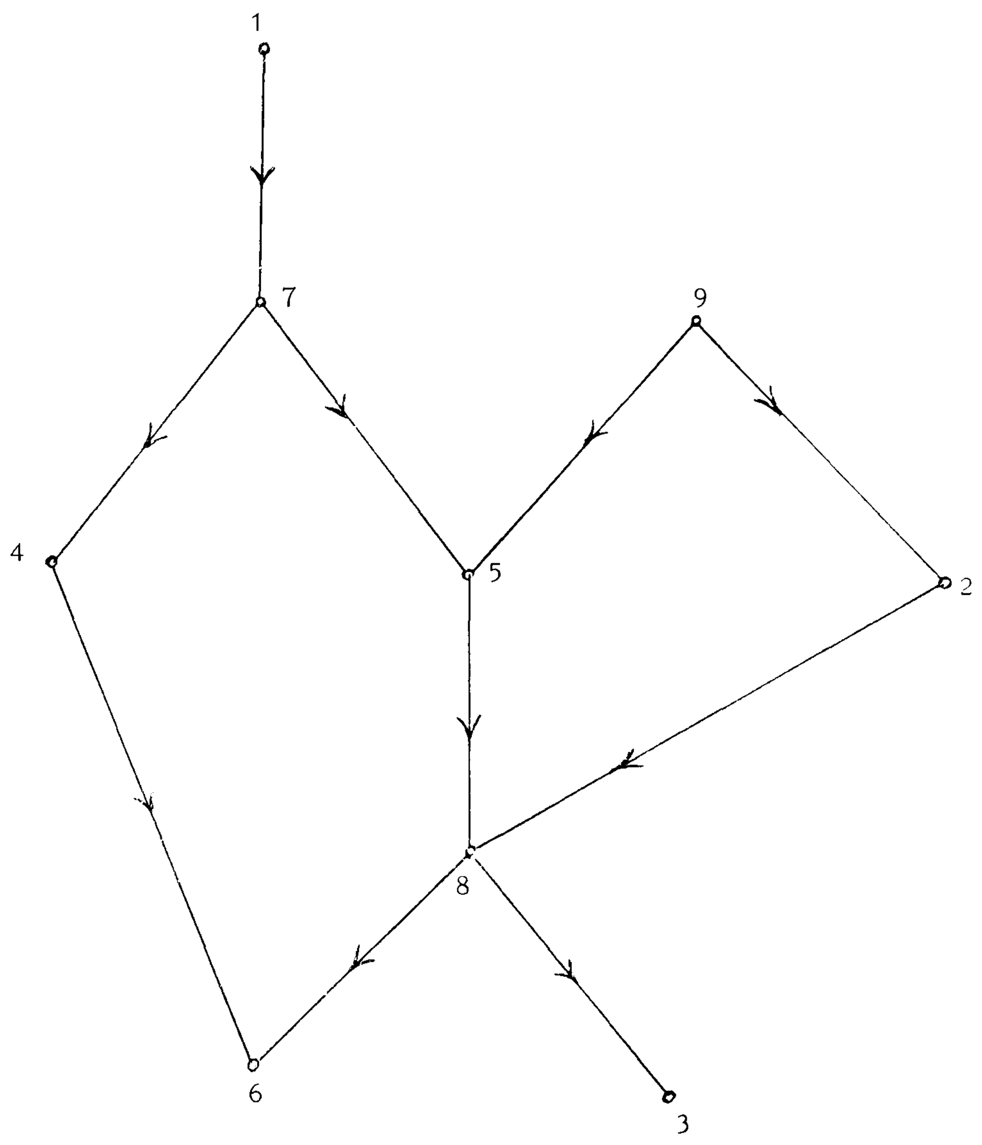


Figure A.1

algorithms employing recursion. Therefore we note that efficiency is not our main concern in this illustration.

The following is essentially the formulation and proof of Dilworth's theorem as given by Perles:

Theorem:

Let $(P, \preccurlyeq)$ be a partially ordered set. If the maximal number of elements in an antichain of $(P, \preccurlyeq)$ is k, then P is a union of k chains.

Proof:

The proof proceeds by induction on $|P|$, for all k simultaneously. If $|P| = 1$, there is nothing to prove. Assume, therefore, that the theorem holds for $|P| < n$, and let $|P| = n$. Denote by S_i and S_o the sets of sinks and sources of P, respectively.

case 1: P contains an antichain Q of k elements, different from both S_i and S_o. Define

$P_1 = \{p \in P$ such that $q \preccurlyeq p$, for some $q \in Q\}$,

$P_2 = \{p \in P$ such that $p \preccurlyeq q$, for some $q \in Q\}$.

It is easily verified that $P_1 \cap P_2 = Q$, $P_1 \cup P_2 = P$, $P_1 \neq P$ and $P_2 \neq P$ (the first relation follows from the fact that Q is an antichain, the second from the maximality of Q, the third from $Q \neq S_o$ and the fourth from $Q \neq S_i$).

Now, $|P_1| < |P|$, $|P_2| < |P|$. By induction hypothesis, P_1 and P_2 decompose into k chains:

$$P_1 = \bigcup_{i=1}^{k} U_i, \qquad P_2 = \bigcup_{i=1}^{k} L_i$$

The elements of Q, being the sources of P_1 and the sinks of P_2, are the sources of the chains U_i and the sinks of the chains

L_i. Let $Q = \{q_1, \ldots, q_k\}$ and assume without loss of generality that q_i is the source of U_i and the sink of L_i $(1 \leq i \leq k)$. Define $C_i = L_i \cup U_i$. C_i is a chain and we have

$$P = P_2 \cup P_1 = \bigcup_{i=1}^{k} C_i$$

<u>case 2:</u> Every independent subset of P containing k elements coincides with S_1 or with S_0. Take some $a \in S_0$ and choose $b \in S_1$, such that $b \not> a$ (b may equal a). Define $C_k = \{a, b\}$ and $P_3 = P \setminus \{a, b\}$. C_k is a chain, $|P_3| < |P|$ and P_3 contains $k - 1$, but not k mutually incomparable elements. Therefore we have, by induction hypothesis, $P_3 = \bigcup_{i=1}^{k-1} C_i$, where the C_i are chains, and

$$P = P_3 \cup \{a, b\} = \bigcup_{i=1}^{k} C_i .$$

From the proof above we derive the following algorithm employing the recursive procedure CHAIN. We assume that $P = \{1, \ldots, N\}$ and the desired minimal chains are stored, at the end of the process, in vector <u>link</u> defined as follows: if $x \lessdot y$ and x immediately precedes y in a chain then link(y) = x; if x is the first element in a chain, then link(x) = x. The boolean variable <u>case1</u> is used for distinguishing between cases 1 and 2 of the proof. The meaning of the remaining data structures employed in the algorithm follows directly from the proof.

ALGORITHM A.1

```
begin comment  an algorithm for finding a minimal chain covering of a
               poset, using a procedure ANTICHAIN, for obtaining
               maximal antichains in the poset;
      procedure CHAIN (integer set P);
      begin integer a, b;
            logical case1;
            if P not empty then
            begin ANTICHAIN (P, Q, case1);
                  comment if P contains a maximal antichain different from
                          both the subsets of sources and sinks of P, then
                          ANTICHAIN assigns it to Q and case1 is assigned
                          to true - otherwise, case1 is assigned to
                          false;
                  if  case1 then
                  begin  $P_1$ := $\{p \in P$ such that $q \preccurlyeq p$, for some $q \in Q\}$;
                         $P_2$ := $\{p \in P$ such that $p \preccurlyeq q$, for some $q \in Q\}$;
                         CHAIN (P1);
                         CHAIN (P2)
                  end
                  else
                  begin  a := any source element in P;
                         b := sink element in P, such that $b \succ a$;
                         link(b) := a;
                         P3  := $P \setminus \{a, b\}$;
                         CHAIN (P3)
                  end
            end
      end CHAIN;
      read the poset $(P, \preccurlyeq)$;
      for $p \in P$ do link (p) := p;
      CHAIN (P);
      output the minimal chain covering from link vector
end
```

The problem solved by the above algorithm can be enunciated as: given a poset $(P, \preccurlyeq)$ and a procedure ANTICHAIN for finding maximal antichains in posets, obtain a minimal covering by disjoint chains. The procedure ANTICHAIN itself is not presented, since we consider it as being out of scope of this appendix, in which our objective is to illustrate recursive algorithmic translations from inductive proofs.

The proofs of correctness of algorithm A.1 are a direct and simple consequence of the proof by Perles of Dilworth's theorem. Finally we mention the fact that besides the computation of ANTICHAIN procedure, all the remaining operations that appear in the description of the algorithm are simple and can be easily implemented in a computer.

REFERENCES

[Be71] A.T. Berztiss, Data Structures: Theory and Practice, Academic Press, New York, N.Y., 1971.

[Be72] H. Beilner, Ein Algorithmus zur Ermittlung K-Beste Kantenfolgen Zwischen Allen Ecken Eines Graphen. Computing, 10, pp.205-220, 1972.

[Be73] A.T. Berztiss, A k-Tree Algorithm for Simple Cycles of a Directed Graph, Tech. Rep. 73-6, Department of Computer Science, University of Pittsburgh, Pittsburgh, Penns., 1973.

[BeKa60] R. Bellman and R. Kalaba, On k-th Best Policies, SIAM J., 8, pp.582-588, 1960.

[Bo67] J. Boothroyd, Shortest Path Between Start Node and End Node of a Network, Computer J., 10, pp.306-307, 1967.

[ChCo62] A. Charnes and W.W. Cooper, A Network Interpretation and a Directed Subdual Algorithm for Critical Path Scheduling, J. Indust. Engin., 13, pp.213-219, 1962.

[ChWi66] Y.C. Chen and O. Wing, Some Properties of Cycle-Free Directed Graphs and the Identification of the Longest Path, J. Franklin Inst., 281, pp.293-301, 1966.

[ClKrRa63] S. Clarke, A. Krikorian and J. Rausen, Computing the N best Loopless Paths in a Network, SIAM J., 11, pp.1096-1102, 1963.

[Co74] D.G. Corneil, The Analysis of Graph Theoretical Algorithms, Tech. Rep. 65, Department of Computer Science, University of Toronto, Toronto, 1974.

[Da63] G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, N.J.,1963.

[Da66] G.B. Dantzig, All Shortest Routes in a Graph, in Théorie de Graphes, Proceedings of the International Symposium (Rome), pp.91-92, published by Dunod, Paris, 1966.

[DaDiHo72] O.-J. Dahl, E.W.Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, London, 1972.

[DaHo56] G.B. Dantzig and A.J. Hoffman, Dilworth's Theorem on Partially Ordered Sets, in Linear Inequalities and Related Systems, H.W. Kuhn and A.W. Tucker, eds., Princeton University Press, Princeton, N.J., 1956.

[Di50] R.P. Dilworth, A Decomposition Theorem for Partially Ordered Sets, Ann. Math., 51, pp.161-166, 1950.

[Di59] E.W. Dijkstra, A Note on Two Problems in Connexion with Graphs, Numer. Math., 1, pp.269-271, 1959.

[Dr69] S.E. Dreyfus, An Appraisal of Some Shortest Path Algorithms, Oper. Res., 17, pp.395-412, 1969.

[EhFoOs73] A. Ehrenfeucht, L.D. Fosdick and L.J. Osterweil, An Algorithm for Finding the Elementary Circuits of a Directed Graph, Tech. Rep. #CU-UC-024-73, Department of Computer Science, University of Colorado, Colorado, 1973.

[EiSh62] B. Eisenman and M. Shapiro, Evaluation of a Pert Network, Comm. ACM, 5, pp.436-437, 1962.

[El70] S.E. Elmaghraby, The Theory of Networks and Management Science (part I), Manage. Sci., 17, pp.1-34, 1970.

[El70a] S.E. Elmaghraby, The Theory of Networks and Management Science (part II), Manage. Sci., 17, pp. B 54-70, 1970.

[Ev73] S. Even, Algorithmic Combinatorics, The Macmillan Co., New York, N.Y., 1973.

[Fl62] R.W. Floyd, Shortest Path, Comm. ACM, 5, pp.345, 1962.

[Fl67] R.W. Floyd, Nondeterministic Algorithms, J. ACM., 14, pp.636-644, 1967.

[Fu56] D.R. Fulkerson, Note on Dilworth's Decomposition Theorem for Partially Ordered Sets, Proceedings Am. Math. Soc., 7, pp.701-702, 1956.

[Fu73] A.L. Furtado, Teoria dos Grafos - Algoritmos, Livros Técnicos e Cientificos Editora S.A., Rio de Janeiro, R.J., 1973.

[Gi69] N.E. Gibbs, A Cycle Generation Algorithm for Finite Undirected Linear Graphs, J. ACM, 16, pp.564-568, 1969.

[GoBa65] S.W. Golomb and L.D. Baumert, Backtrack Programming, J. ACM, 12, pp.516-524, 1965.

[GoCo67] C.C. Gotlieb and D.G. Corneil, Algorithms for Finding a Fundamental Set of Cycles for an Undirected Linear Graph, Comm. ACM, 10, pp.780-783, 1967.

[Ha69] F. Harary, Graph Theory, Addison-Wesley Publ. Co., Reading, Mass., 1969.

[Ha73] M.C. Harrison, Data Structures and Programming, Scott, Foresman and Co., Glenview, Illin., 1973.

[HaNoCa65] F. Harary, R.Z. Norman and D. Cartwright, Structural Models: An Introduction to the Theory of Directed Graphs, John Wiley & Sons Inc., New York, N.Y., 1965.

[Hi62] T.N. Hibbard, Some Combinatorial Properties of Certain Trees with Application to Searching and Sorting, J. ACM, 9, pp.13-28, 1962.

[HoPa59] W. Hoffman and R. Pavley, A Method for the Solution of the Nth Best Path Problem, J. ACM, 4, pp.506-514, 1959.

[HoTa73] J. Hopcroft and R. Tarjan, Efficient Algorithms for Graph Manipulation, Comm. ACM, 16, pp.372-378, 1973.

[HoTa73a] J. Hopcroft and R. Tarjan, Dividing a Graph into tri-Connected Components, SIAM J. Comput., 2, pp.135-158, 1973.

[HoTa74] J. Hopcroft and R. Tarjan, Efficient Planarity Testing, J. ACM, 21, pp.549-568, 1974.

[HoWi71] A.J.Hoffman and S. Winograd, On Finding all Shortest Distances in a Directed Network, IBM Research Report RC3613, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 1971.

[HsHo72] H.T. Hsu and P.A. Honkanen, A Fast Minimal Storage Algorithm for Determining the Elementary Cycles of a Graph, Computer Science Department, The Pennsylvania State University, Pennsylvania, 1972.

[Jo73] D.B. Johnson, Algorithms for Shortest Paths, Tech. Rep., 73-169, Department of Computer Science, Cornell University, Ithaca, N.Y., 1973.

[Jo73a] D.B. Johnson, Finding All the Elementary Circuits of a Directed Graph, Tech. Rep. 145, Computer Science Department, The Pennsylvania State University, Pennsylvania, 1973 (Revised Feb. 1974) - to appear in SIAM J. Comput.

[Jo74] A.D. Jovanovich, Note on a Modification of the Fundamental Cycles Finding Algorithm, Inf. Process. Lett., 3, pp.33, 1974.

[Ka62] A.B. Kahn, Topological Sorting for Large Networks, Comm. ACM, 5, pp.558-562, 1962.

[Ka63] R.H. Kase, Topological Sorting for PERT Network, Comm. ACM,6, pp.738-739, 1963.

[Kl67] M.M. Klein, Scheduling Project Networks, Comm. ACM, 10, pp.225-231, 1967.

[Kn68] D.E. Knuth, Fundamental Algorithms, The Art of Computer Programming 1, Addison-Wesley Publ. Co., Reading, Mass., 1968 (second ed. 1973).

[Kn73] D.E. Knuth, Sorting and Searching, The Art of Computer Programming 3, Addison-Wesley Publ. Co., Reading, Mass., 1973.

[Kn74] D.E. Knuth, Structured Programming with go to Statements, Comp. Surv., 6, pp.261-301, 1974.

[Kn74a] D.E. Knuth, Private communication, March,5, 1974.

[Kn75] D.E. Knuth, Estimating the Efficiency of Backtrack Programs, Mathematics of Computation, 29, pp.121-136, 1975.

[KnSz74] D.E. Knuth and J.L. Szwarcfiter, A Structured Program to Generate all Topological Sorting Arrangements, Inf. Process., Lett., 2, pp.153-157, 1974.

[Ku30] K. Kuratowski, Sur le Problème des Courbes Gauches en Topologie, Fund. Math., 15, pp.271-283, 1930.

[La61] D.J. Lasser, Topological Ordering of a List of Randomly Numbered Elements of a Network, Comm. ACM, 4, pp.167-168, 1961.

[La65] S.E. Lass, PERT Time Calculation Without Topological Ordering, Comm. ACM, 8, pp.172-174, 1965.

[La72] E.L. Lawler, A Procedure for Computing the k Best Solutions to Discrete Optimization Problem and its Application to the Shortest Path Problem, Manage. Sci., 18, pp.401-405, 1972.

[La73] P.E. Lauer, The Perils of Indirect Proof or Another Efficient Search Algorithm to Find the Elementary Circuits of Directed Graphs, Tech. Rep. 42, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, 1973 (Revised Sep. 1973).

[Le61] B. Leavenworth, Critical Path Scheduling, Comm. ACM, 4, pp.152-153, 1961.

[Le74] P.G.H. Lehot, An Optimal Algorithm to Detect a Line Graph and Output its Root Graph, J. ACM, 21, pp.569-575, 1974.

[MaBi67] S. MacLane and G. Birkhoff, Algebra, The MacMillan Co., New York, N.Y., 1967.

[Mi71] L. Mirsky, A Dual of Dilworth's Decomposition Theorem, Am. Math. Monthly, 78, pp.876-877, 1971.

[Mi74] E. Minieka, On Computing Sets of Shortest Paths in a Graph, Comm. ACM, 17, pp.351-353, 1974.

[Ni66] T.A.J. Nicholson, Finding the Shortest Route between Two Points in a Network, Comp. J., 9, pp.275-280, 1966.

[Pa69] K. Paton, An Algorithm for Finding a Fundamental Set of Cycles of a Graph, Comm. ACM, 12, pp.514-518, 1969.

[PaWi73] E.S. Page and L.B. Wilson, Information Representation and Manipulation in a Computer, Cambridge University Press, London, 1973.

[Pe63] M.A. Perles, A Proof of Dilworth's Decomposition Theorem for Partially Ordered Sets, Is. J. Math. 1, pp.105-107, 1963.

[Pe63a] M.A. Perles, On Dilworth's Theorem in the Infinite Case, Is. J. Math., 1, pp.108-109, 1963.

[Po61] M. Pollack, Solutions of the k-th Best Route Through a Network - a Review, J. Math. Anal. and Appl., 3, pp.547-559, 1961.

[Po61a] M. Pollack, The k-th Best Route Through a Network, Oper. Res., 9, pp.578-580, 1961.

[Pr71] W.L. Price, Graphs and Networks - an Introduction, Butterworth & Co., London, 1971.

[PrDe74] M. Prabhaker and N. Deo, On Algorithms for Enumerating all Circuits of a Graph, Tech. Rep. UIUCDCS-R-73-585, Department of Computer Science, University of Illinois, Illinois, 1973 (Revised Mar. 1974).

[ReTa73] R.C. Read and R.E. Tarjan, Bounds on Backtrack Algorithms for Listing Cycles, Paths and Spanning Trees, Mem. ERL-M433, Electronics Research Laboratory, University of Berkeley, Berkeley, Calif., 1973.

[Ro73] N.D. Roussopoulos, A max {m,n} Algorithm for Determining the Graph H from its Line Graph G, Inf. Process. Lett., 2, pp., 108-112, 1973.

[RoFl66] S.M. Roberts and B. Flores, Systematic Generation of Hamiltonian Circuits, Comm. ACM, 9, pp. 690-694, 1964.

[Si71] R.L. Sites, ALGOL W Reference Manual, Tech. Rep. STAN-CS-71-230, Computer Science Department, Stanford University, Stanford, Calif.,1971.

[Sp73] P.M. Spira, A New Algorithm for Finding All Shortest Paths in a Graph of Positive Arcs in Average Time $O(n^2\log^2 n)$, SIAM J. Comput., 2, pp.28-32, 1973.

[Sy73] M.M. Syslo, The Elementary Circuits of a Graph, Comm. ACM, 16, pp.632-633, 1973.

[Sy75] M.M. Syslo, Remark on Algorithm 459: The Elementary Circuits of a Graph, Comm. ACM, 18, pp. 119, 1975.

[SzLa75] J.L. Szwarcfiter and P.E. Lauer, A New Backtracking Strategy for the Enumeration of the Elementary Cycles of a Directed Graph, Tech. Rep. 69, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, 1975.

[SzLa75] J.L. Szwarcfiter and P.E. Lauer, A New Backtracking Strategy of a Directed Graph, Tech. Rep. 69, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, 1975.

[Ta72] R. Tarjan, Depth-First Search and Linear Graph Algorithms, SIAM J. Comput., 1, pp.146-160, 1972.

[Ta73] R. Tarjan, Enumeration of the Elementary Circuits of a Directed Graph, SIAM J. Comput., 2, pp.211-216, 1973.

[Ta74] R. Tarjan, Finding Dominators in Directed Graphs, SIAM J. Comput., 3, pp.62-89, 1974.

[Ta74a] R.E. Tarjan, Edge-Disjoint Spanning Trees, Dominators and Depth-First Search, Tech. Rep. STAN-CS-74-455. Computer Science Department, Stanford University, Stanford, Calif., 1974.

[Ta74b] R.E. Tarjan, A New Algorithm for Finding Weak Components, Inf. Process. Lett., 3, pp. 13-15, 1974.

[Ti70] J.C. Tiernan, An Efficient Search Algorithm to Find the Elementary Circuits of a Graph, Comm. ACM, 13, pp.722-726, 1970.

[Tv67] H. Tverberg, On Dilworth's Decomposition Theorem for Partially Ordered Sets, J. Comb. Theory, 3, pp.305-306, 1967.

[Wa70] H.M. Wagner, Principles of Management Science with Application to Executive Decisions, Prentice Hall, Englewood Cliffs, N.J., 1970.

[We66] J.T. Welch Jr., A Mechanical Analysis of the Cyclic Structure of Undirected Linear Graphs, J. ACM, 13, pp.205-210, 1966.

[We71] M.B. Wells, Elements of Combinatorial Computing, Pergamon Press, Oxford, 1971.

[We72] H. Weinblatt, A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph, J. ACM, 19, pp.43-56, 1972.

[WiWh73] T.A. Williams and G.P. White, A Note on Yen's Algorithm for Finding the Lengths of All Shortest Paths in N-Node Nonnegative Distance Networks, J. ACM, 20, pp.389-390, 1973.

[Ye71] J.Y. Yen, Finding the k Shortest Loopless Paths in a Network, Manage. Sci., 17, pp.712-716, 1971.

[Ye72] J.Y. Yen, Finding the Lengths of All Shortest Paths in N-Node Nonnegative Distance Complete Networks Using $\frac{1}{2}N^3$ Additions and N^3 Comparisons, J. ACM, 19, pp.423-424, 1972.

[**72] ALGOL W Programming Manual, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, 1972.