Durham E-Theses

An investigation of shortest paths algorithms

Tabatabai, Bijan Oni

How to cite:

Tabatabai, Bijan Oni (1987) *An investigation of shortest paths algorithms*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/6685/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

T004C

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

AN INVESTIGATION OF SHORTEST PATHS ALGORITHMS

A Thesis presented
to the
Department of
Computer Science
University of Durham
for the
Degree of
Doctor of Philosophy

by

BIJAN ONI TABATABAI

MAY 1987

13 JAN 1988

ABSTRACT

In this work, we classify the shortest path problems, review all source algorithms and analyse the different implementations of single source algorithms using various list structures and labelling techniques.

Furthermore, we study the Sensitivity Analysis of one-to-all problems and present an algorithm, Senet, for their Post Optimality Analysis. Senet determines all the critical values for the weight of an arc (which could be optimal, non-optimal or non-existant) at which the optimal solution changes. Senet also provides the updated optimal solution for every range formed by two successive critical values.

ACKNOWLEDGEMENTS

I am grateful to Mr A J Slade, my supervisor, for his guidance and encouragement throughout the research leading to this thesis. I would also like to thank J Nellist for typing and preparing the thesis.

This work was done with the financial support of SERC.

CONTENTS

Page

PART I: FOUNDATIONS

1. Introduction 2
2. Graphs and Networks 5
3. Computational Complexity 15
4. Data Structure 19
5. Network and Tree Representations 46
6. Problem Classification 52

PART II: SINGLE SOURCE ALGORITHMS

7. Single Source Algorithms 58
8. Label Correcting Algorithms 69
9. Label Setting Algorithms 91
10. An Empirical Study 120

PART III: ALL SOURCE ALGORITHMS

11. Matrix Multiplication Methods 130
12. Triple Algorithms 133
13. Modified Label Setting Algorithms 137

PART IV: SENSITIVITY ANALYSIS AND POST OPTIMALITY ANALYSIS

14. Sensitivity Analysis 142
15. Post Optimality Analysis 164

PART V: SUMMARY, CONCLUSIONS AND REFERENCES

16. Summary and Conclusions 194
17. References 199

APPENDICES

Appendix A 205
Appendix B 215
Appendix C 226
Appendix D 236
Appendix E 250
Appendix F 263

PART I

FOUNDATIONS

1 INTRODUCTION

Shortest path problems are the most fundamental and the most commonly encountered problems in the study of transportation and communication networks. Many other important network problems involve shortest path computations in their solution methods.

Various shortest path algorithms have been developed since the latter half of the 1950's. The purpose of this work is to evolve a classification of the "efficient" sequential algorithms for a particular class of unconstrained deterministic shortest path problems, and to study their computational efficiency and sensitivity. The work is divided into 5 parts.

In Part I, the introduction is followed by necessary definitions and theorems of graphs and networks in section 2, and computational complexity and data structure in sections 3 and 4. In section 5 the network and tree representations used in this work are presented and analysed.

A classification of sequential algorithms for "THE SHORTEST PATH" is introduced in section 6.

In Part II, single source algorithms are classified and studied in section 7. In section 8 and 9 various label setting and label correcting methods are analysed. In section 10 an empirical study of the most efficient labelling algorithms on small networks, ie. networks with upto 200 nodes is carried out.

In Part III, all source algorithms, matrix multiplication methods, triple algorithms and modified label setting algorithms are reviewed in sections 11, 12 and 13.

In Part IV, various algorithms for sensitivity analysis on "THE SHORTEST PATH PROBLEMS" are studied in section 14, and in section 15 we introduce an algorithm, Senet, for post optimality analysis of "ONE-TO-ALL SHORTEST PATH PROBLEMS". Senet determines every non-negative critical value of an arc weights at which the optimal solution changes and also provides the updated solution. Senet is applicable to basic, non-basic and non-existant arcs in a non-negative network.

Part V, consists of a summary of the work together with conclusions in section 16, and the references in section 17.

The complete Pascal codes of the more complicated and also the most efficient algorithms are presented in the appendices.

2 GRAPHS AND NETWORKS

A Graph $G = (N, A)$ is a structure which consists of a non-empty and finite set of Nodes N of cardinality n, and a set of unordered pairs of Nodes A, called arcs, of cardinality m, the arcs are not necessarily distinct.

ie. $A = \{\langle u, v\rangle : u, v \in N\}$

A digraph is a graph in which all the arcs are directed, ie. the set of arcs is a set of ordered pairs of nodes. A graph can be converted to a digraph by simply replacing every undirected arc by two directed arcs in opposite directions, ie. replacing every unordered pair of nodes by its eqivalent two ordered pairs of nodes. If (u, v) is a directed arc then u is its initial node and v is its terminal node.

A loop is an arc (u, v) with $u = v$. Two arcs (u_1, v_1) and (u_2, v_2) are parallel arcs if $u_1 = u_2$ and $v_1 = v_2$. A graph is called simple if it contains neither loops, nor parallel arcs.

A network is a simple digraph together with a real valued function w defined for every $(u, v) \in A$.

The real number w_{uv} *is the weight of the arc* (u, v).

Node u is said to be isolated if neither an arc (u, v) *nor an arc* (v, u) *exists with* $v \in N - \{u\}$. *A path* q_{uv} *from node u to node v, in G, is an alternating sequence of nodes and arcs, with* $q_{uv} = (u = u_{i1}, x_{j1}, u_{i2}, x_{j2}, \ldots\ldots, x_{jk}, u_{i(k+1)} = v)$, *where* $x_{jr} = (u_{ir}, u_{i(r+1)})$ *for* $1 \leqslant r \leqslant k$. q_{uv} *can also be represented by the node sequence,* $(u = u_{i1}, u_{i2}, \ldots, u_{i(k+1)} = v)$.

A path in which all nodes (except possibly the first and the last, called source and sink of the path) are distinct is an elementary path. We will denote an elementary path from node u to node v by P_{uv}, *and the set of all elementary paths from u to v by* R_{uv}, *ie.* $R_{uv} = \{P^1_{uv}, P^2_{uv}, \ldots\ldots\}$. *The length or total weight of a path is given by,* $d_{uv} = w_{ij}$. *A cycle is a path for which the source and the sink are the same node, ie* q_{uu}. *Node u is said to be directly connected to node v if arc* $(u, v) \in A$. *If there exists a path from node u to node v, then v is reachable from u, disconnected otherwise.*

Define uRv if there exists path q_{uv} and q_{vu}, R is an equivalence relationship. A network in which all uRv is defined for all $u, v \in N$ is strongly connected. Furthermore, the subnetworks $G_i = (N_i, \{(u,v) \mid (u,v) \in A \text{ and } u,v \in N_i\})$, where N_i is an equivalence class under R, are the strongly connected components of G.

A network is complete if every node $u \in N$ is directly connected to every other node $v \in N - \{u\}$.

A network, G, is acyclic if no path in G is a directed cycle, ie. G has no strongly connected component. A graph with n nodes and m arcs is dense if m is "large" compared to n and sparse otherwise. The value of "large" depends on the context, we shall assume that m and n are positive and $(m + n) = O(m)$ for dense graphs and $(m + n) = O(n)$ for sparse graphs. If $m < (n-1)$ then clearly G is disconnected.

A connected network without cycles is called a tree, equivalently a network is a tree if there exists a unique path from any node $u \in N$ to any node $v \in N - \{u\}$. We denote a tree by T. A tree

T is a spanning tree of network G if T is a subnetwork of G containing all nodes of G.

A shortest path from node u to node v is a path q_{uv} *such that* d_{uv} *is a minimum over all paths from u to v. Note that the number of arcs is immaterial. Let* $|q_{uv}|$ *denote the number of arcs in path* q_{uv}*. A path with the minimum number of arcs is arc shortest.*

Theorem 1: *If G is a complete network with n nodes and m arcs then* $m = n(n-1)$*.*

Proof: *By definition, there are n nodes each of which is directly connected to all the other (n-1) nodes, thus there are n(n-1) arcs.* †

Corollary 1.1: *If G is a simple graph with n nodes and m arcs and is undirected then* $m \leq n(n-1)/2$*, and if G is a digraph then* $m \leq n(n-1)$*.*

Theorem 2: ***There exists an elementary path*** P_{uv} ***from node u to node v if and only if there exists a path*** q_{uv}***.***

Proof: By definition, if p_{uv} exists then q_{uv} exists. Now suppose q_{uv} is given, if q_{uv} is not elementary, then for every repeated node in q_{uv} delete all nodes between the two instances of the repeated node and one of the instances of the repeated node, leaving a new path q_{uv}. Continue the process until some q_{uv} is elementary. The path p_{uv} obtained from q_{uv} by the above process is a reduction of q_{uv}. A reduction is not necessarily unique. †

Theorem 3: The set of elementary paths R_{uv} from any node u to any other node v in a complete network G, is of cardinality $|R_{uv}|$, where

$$|R_{uv}| = (n-2)! \sum_{i=0}^{n-2} 1/(n-2-i)!$$

Proof: By definition, an elementary path in a complete network utilises at most (n-1) arcs or has a maximum number of (n-2) intermediate nodes. Furthermore the total number of paths in R_{uv} is the grand total of total numbers of paths with i intermediate nodes, where

$i = 0, 1, \ldots\ldots, (n-2)$.

Now, the total number of paths with exactly i intermediate nodes is given by,

$(n-2)P_i = (n-2)!\ /(n-2-i)!$

Thus we have,

$$|R_{uv}| = \sum_{i=0}^{n-2} (n-2)P_i = (n-2)! \sum_{i=0}^{n-2} 1/(n-2-i)!$$

Theorem 4: There exists a shortest path from node u to node v in network G if and only if there exists at least a path q_{uv}, and furthermore all such paths must not contain a directed cycle of total weight of less than zero.

Proof: Let P_{uv} be the shortest path from u to v in G, thus there is a path $q_{uv} = P_{uv}$. Now suppose there exists a path q'_{uv} which contains a cycle of negative total weight, then a new path q''_{uv} can be constructed in which this cycle is repeated a number of times sufficient for $d(q''_{uv}) < d(P_{uv})$ contrary to assumption. Let q_{uv} be a path from u to v and suppose no path from u to v contains a cycle of total negative weight. Now if P_{uv} is a reduction of q_{uv} then $d(P_{uv}) \leq d(q_{uv})$. Thus the total weights of a number of elementary paths bound from below all total path weights, and since there are a finite number of elementary paths, then among them is a path P_{uv} such that $d(P_{uv}) \leq d(q_{uv})$ for all paths q_{uv}. By definition,

P_{uv} is a shortest path from node u to node v.

†

Corollary 4.1: There exists an elementary shortest path P_{uv}, if there exists a shortest path q_{uv}.

Corollary 4.2: There exists a shortest path from node u to node v in an acyclic network for every node v reachable from node u.

Theorem 5: For any shortest path
$P_{uv} = (u = u_1, u_2, \ldots\ldots, u_k = v)$ each subpath
$P'_{uv} = (u_j, u_{j+1}, \ldots\ldots, u_{j+r})$ where,
$1 \leqslant j \leqslant (j + r) \leqslant k$ is a shortest path from node u_j to u_{j+r}. Furthermore if P_{uv} is arc shortest then so are all its subpaths.

Proof: Suppose that there exists such a subpath which is not the shortest path (arc shortest) from node u_j to node u_{j+r}. But this contradicts the assumption that P_{uv} is the shortest path (arc shortest) from node u to node v. †

Let $\Pi(G, ¥)$ denote a shortest path problem, where G is a network and ¥ is a set of ordered pairs of nodes between which shortest paths are to be

found. The definitions and the notations for a variety of shortest path problems will be discussed in section 6.

A solution to Π (G, ¥) is an assignment

σ : ¥(u, v) $\Leftrightarrow$ (P_{uv}, d_{uv})

Of an elementary path together with its total weight to each element of ¥. If for some ¥(u, v), P_{uv} is not defined, then

σ : ¥(u, v) $\Leftrightarrow$ (o, ∞).

σ will be detailed in sections 5 and 6.

An arc is optimal if it is utilised by a path in a solution, non optimal otherwise. An arc (u, v) is non-existant if u, v $\in$ N and (u, v) $\notin$ A.

The set of all arcs emanating from a given node u is the set of forward star arcs of node u, denoted by FS(u), ie. FS(u) = {(u, i) | (u, i) $\in$ A}. The set of all arcs proceeding from a given node u is the set of backward star arcs of node u, denoted by BS(u), ie. BS(u) = {(i, u) | (i, u) $\in$ A}.

The set of successor nodes of u is defined as

N^{su} = {v | (u, v) $\in$ A and u $\neq$ v}.

The set of predecessor nodes of u is defined as

$^{P}N = \{v \mid (v, u) \in A \text{ and } u \neq v\}$.

The set of adjacent nodes of a given node u is defined as $^{P}N \cup N^{S}$.

The indegree of a given node u is defined as

$E^{-}(u) = |^{P}N(u)|$, *and its outdegree is defined as*

$E^{+}(u) = |N^{S}(u)|$.

By definition, a network is a simple digraph, ie. it contains neither loops nor parallel arcs. A network containing such features can be converted to a standard network, as defined above, by simple preprocessing. Consider the network in figure 1 in which there are parallel arcs between nodes 2 and 3, and also arc (4, 4) is a loop.

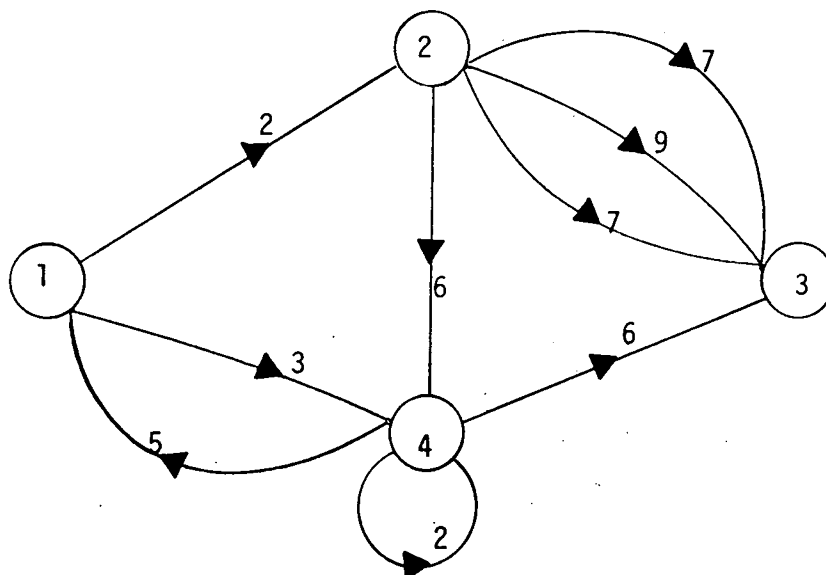


Figure 1: The numbers corresponding to the arcs represent the weights of the arcs.

To convert this network to a standard network, firstly all parallel arcs except one with the smallest weight have to be eliminated, secondly the loop on node 4 has to be eliminated, by converting it to an arc connecting a dummy node 5 to the modified version of node 4 which contains no loop.

All the arcs going into the original node 4 will now go into node 5. The newly created, arc (5, 4) has a weight equal to the weight of the eliminated loop and all the arcs going out of the original node 4 will go out of the modified node 4. Figure 2 shows the drived standard version of the example network in figure 1.

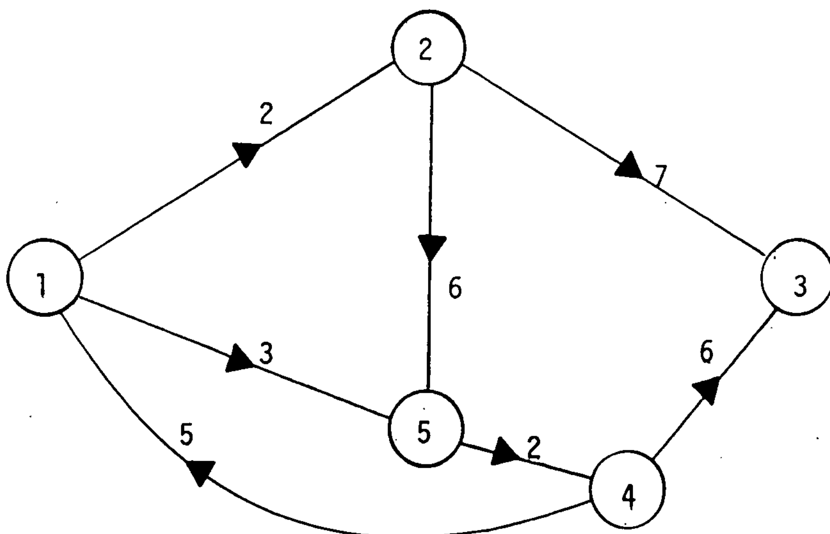


Figure 2: The standard version of the network in figure 1.

3 COMPUTATIONAL COMPLEXITY

For the generality, in this work a random-access machine (RAM) model as suggested in, [AHHU 74], is used for worst case analysis of the algorithms to study their efficiency. A random-access machine consists of a finite program, and a memory in the form of an array of (MAXLENGTH) words, each of which has a unique address between 1 to (MAXLENGTH) and can store an integer (or a real) number. It also contains a finite number of registers, each of which can store an integer (or a real) number. In a random-access machine a single arithmetic, logical, fetch or store operation is performed in one step. For simplicity, the algorithms are expressed in a pascal/english based language; and are introduced throughout the work in order to consider their developments. However, the sophisticated algorithms and also the most efficient algorithms are implemented using a pascal-run compiler on either a VAX 11/750 with UNIX operating system, or IBM 4341 with MTS operating system or prime computers with primus operating system. The corresponding codes are listed in the appendices.

In general, there are two methods of measuring the running time of a shortest path algorithm.

(1) Analysis of average running time:

To evaluate an algorithm in this method, first the algorithm is applied to a diverse set of randomly generated networks, where a random network is one in which two nodes of a network are selected randomly to form a new arc which is to be added to the network. Then the average of the running times is reported.

(2) Worst-case analysis:

In worst case analysis the running time of an algorithm as an upper bound which depends on the problem size is reported.

In this work we shall use the worst case analysis for the evaluation of every single source algorithm, mainly due to the following two reasons:

(a) Worst case analysis guarantees that no problem of a given size will take longer to run than the bound given.

(b) Analysis of average running time is difficult and the concept itself is elusive, because it is not clear what a random distribution of networks with negative arc weights is.

However, in section 10 an analysis of average running time for some of the best single source algorithms is used.

Now consider a shortest path problem π *(G, ¥). The size of this problem can be defined in terms of* $n = |N|$, $m = |A|$ *and* $|¥|$. *But* $|¥|$ *is a function of n, thus we can seek time bounds* $T(n,m)$ *depending on n and m such that* $T(n,m)$ *is the time taken by a certain algorithm to solve a problem of size (n,m) and no problem of this size takes longer. These bounds can be expressed in terms of n only, ie.* $T(n)$, *since* $m = n\ (n-1)$, *[maximum number of arcs in a network with n nodes]. But according to a random-access machine definition each operation, of the types mentioned above, takes one step then we can translate* $T(n)$ *as the number of repetition of an operation with the highest frequency in the algorithm when*

solving a problem of size n, or T(n,m) if the problem size is expressed in terms of (n,m).

4 DATA STRUCTURE

It is obvious that if the arc weights in a given network are all integers, then the total weight of a path is also an integer, since the only operation required in total weight finding is addition, and the sum of integer numbers is an integer. In real life problems arc weights are usually integers and if not, then by multiplying all the arc weights of the given network by an appropriate number they can all be converted to integers. In this work we will only consider the networks with integer arc weights. For simplicity we will also present the nodes by integers, ie. N = {i / i = 1, 2, ..., h}. Beside integer type, we will also consider Boolean type or bit, which can either have the value of ***true*** *or* ***false****. We will also consider more complicated types like arrays, lists, queues, etc. For futher discussions on these types see [KNUT 73a], [KNUT 73b], [AHHU 74] and [FOXB 78].*

(a) *Create A[ ]* produces the empty array A;

(b) *Retrieve (A, Index)* takes as input the array A and an index;

(c) *Store (A, index, value)* is used to enter new index-value pair in array A.

An ordered, or a sequence, or a linear, list is one of the most commonly found data objects. It is either empty or can be written as $(a_1, a_2, \ldots\ldots, a_n)$.

The permitted operations on ordered lists that we are concerned with are as follows:

(i) Find the length of the list, n;

(ii) Read the list from left to right (or right to left);

(iii) Retrieve the ith element, $1 \leq i \leq n$;

(iv) Store new value in i^{th} position, $1 \leq i \leq n$;

(v) Insert a new element at position i, $1 \leq i \leq n + 1$ causing elements numbered $i, i + 1, \ldots\ldots, n$ to become numbered $i + 1, i + 2, \ldots\ldots, n + 1$;

(vi) Delete the element at position i, $1 \leq i \leq n$ causing the elements numbered $i + 1, i + 2, \ldots\ldots, n$ to become numbered $i, i + 1, \ldots\ldots, n-1$.

In the study of data structure we are interested in ways of representing ordered lists so that these operations can be carried out efficiently. The most common way of representing an ordered list is by an array where we associate the list element a_i with the array index i. This can be viewed as a sequential mapping, since using the array representation we are storing a_i and $a_{(i+1)}$ into consecutive locations i and (i + 1) of the array. We can also have access to the list values in either directions by changing the index values in a controlled way. Thus the above operations can be carried out in a list, in a constant amount of time.

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. Given a stack $S = (a_1, a_2, \ldots\ldots, a_n)$ then a_1 is said to be the bottom element and a_i is said to be on top of element $a_{(i-1)}$, $1 \leq i \leq n$. The restrictions on a stack imply that the first element to be removed or deleted from a stack must be the last element inserted in the stack. For this reason stacks are also called Last-In-First-Out, LIFO-lists. In figure 3(a) the value a_1 was the last element inserted into the stack and thus

will be the first to be removed. The value a_n was the first element inserted into the stack and will be the last to be removed. The permitted operations on stacks that we are concerned with are as follows:

- *(i)* *Create (S)* *produces the empty stack S;*
- *(ii)* *Add (i, S)* *inserts the element i into the stack S, at the top position, and returns the new stack S;*
- *(iii)* *Delete (S)* *removes the top element of stack S and returns the new stack S;*
- *(iv)* *Top (S)* *returns the top element of the stack S;*
- *(v)* *EmptyS (S)* *returns the value true if stack S is empty, else false.*

The simplest way to represent a stack is by using a one-dimensional array of size n, denoted by stack (n) where n is the maximum number of allowable entries. The first or the bottom element in the stack will be stored at stack (1), the second at stack (2) and the ith at stack (i). Associated with the array will be a variable, top, which points to the top element in the stack.

A queue is an ordered list in which all insertions take place at one end, the back, and all deletions take place at the other end, the front. Given a queue $Q = (a_1, a_2, \ldots, a_n)$ *then* a_n *is the back element and* a_1 *is the front element. The element* $a_{(i+1)}$ *is said to be behind* a_i*,* $1 \leq i \leq n$*.*

A queue is also called First-In-First-Out, FIFO-list. The permitted operations on queues that we are concerned with are as follows:

(i) *Create (Q)* produces the empty queue Q;

(ii) *AddQ (i,Q)* adds the element i to the back of the queue Q and returns the resulting queue Q;

(iii) *DeleteQ (Q)* removes the front element from the queue Q and returns the resulting queue Q;

(iv) *Front (Q)* returns the front element of the queue Q;

(v) *EmptyQ (Q)* returns the value true if the queue Q is empty, else false.

A double ended queue (dequeue) is a queue in which insertions and deletions can take place at both end points, front and back. In a dequeue

operations (ii) and (iii) above can be extended to the following:

(ii)'   *AddQ (i, L, DQ)* *which adds the element i to the back of the DQ if L = back, and to the front of DQ if L = front;*

(iii)'   *DeleteDQ (L, DQ)* *which deletes the front element of DQ if L = front and its back element if L = back;*

Operation (iv) may also be extended to the following:

(iv)'   *EndDQ (L, DQ)* *which returns the front element of DQ if L = front and its back element if L = back;*

If on a given queue all operations except (iii), deleteQ (Q), can be extended to those on a dequeue, then the queue is called an output restricted dequeue, RDQ. The permitted operations on a RDQ are (i), (ii)', (iii), (iv)', (v). For simplicity we will sometimes refer to RDQ as dequeue or double ended queue, since this is the only form of double ended queue used in this work. Figure 3 illustrates different types of lists.

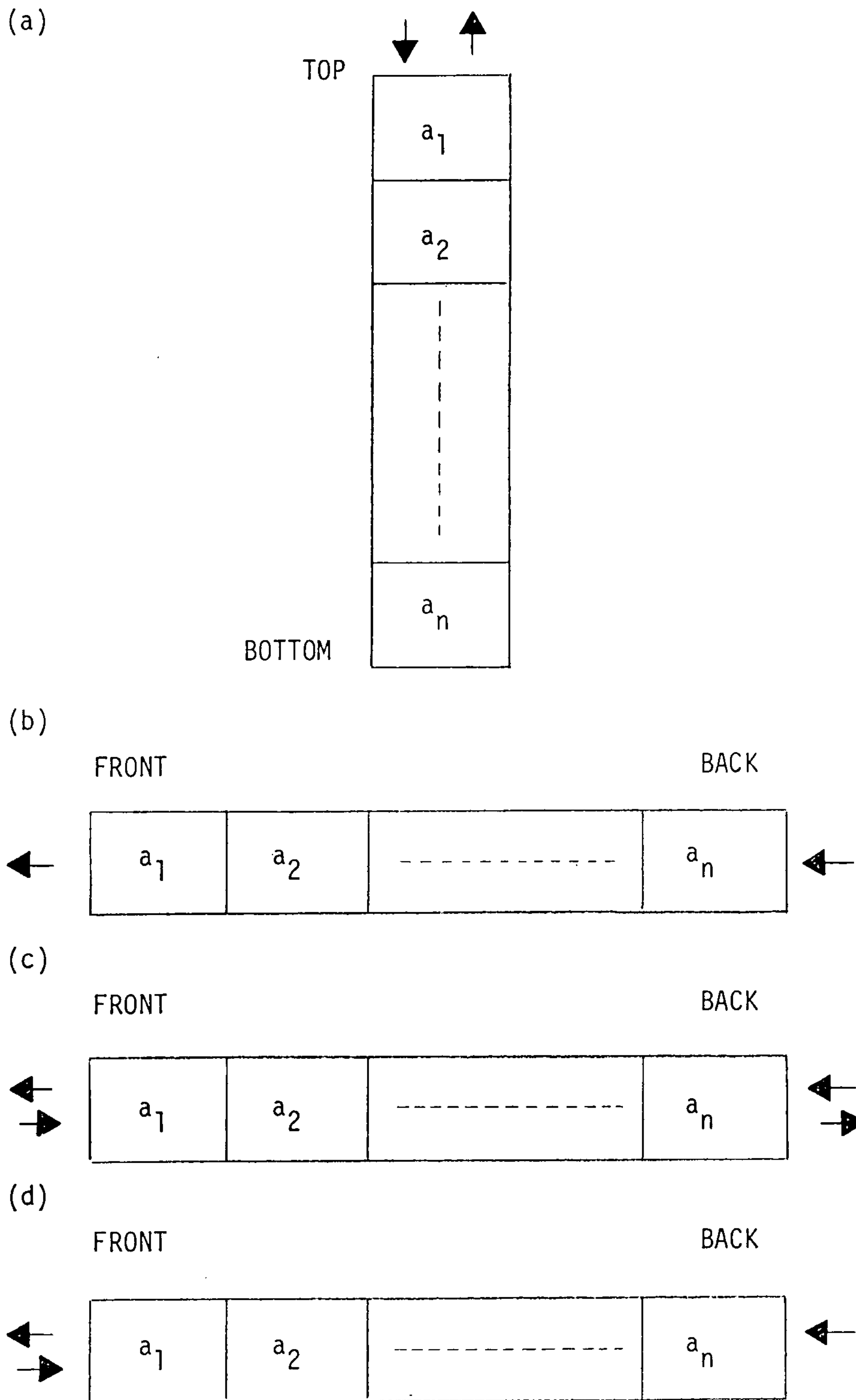


Figure 3: *Types of lists, (a) stack, (b) queue, (c) dequeue, (d) output restricted dequeue*

A node is a collection of data, a_1, a_2, ..., a_n, and pointers or links, L_1, L_2, ..., L_n.

A linked structure is a collection of nodes interconnected by links. In a linked structure node i contains data a_i and an address j in link L_i where j is the address of the next node in the structure. A list can be represented by a linked structure as well as sequential mapping. Figure 4 shows some types of linked lists, pointers are used to show the links. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory, ie. in a sequential representation the order of the elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same.

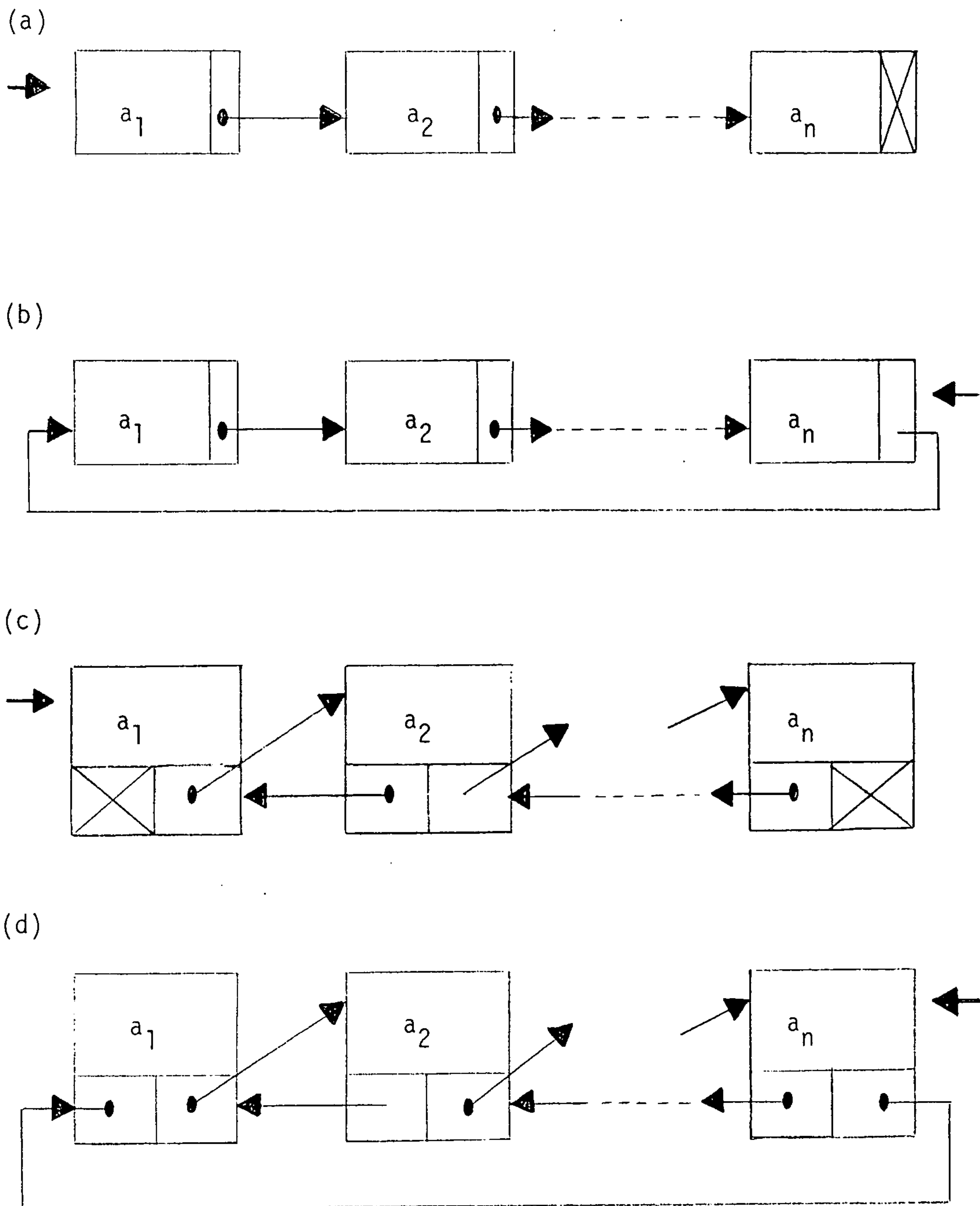


Figure 4: *linked representation of lists, missing pointers are null (a) single linear, (b) single circular, (c) double linear, (d) double circular.*

In a single linear linked list, each node has a pointer to its successor node in the list. In a double linear linked list each node has two links, one pointing to its successor node and one to its predecessor node in the list. In a linear linked list the successor of the last node and the predecessor of the first node are null. In a circular linked list the successor of the last node is the first node and the predecessor of the first node is the last node. A linear linked list is accessed by means of a pointer to its front and a circular linked list is accessed by means of a pointer to its back.

A stack can be represented by a single linear linked list. An output restricted dequeue can be represented by a single circular linked list. A dequeue can be represented by a double circular linked list. In this manner the operations on stacks and queues can be carried out more efficiently. Clearly this efficiency is at the cost of additional memory space for the links, which can be the dominating factor in some situations.

A binary tree, BT, is a type of tree in which every node has at most 2 branches or subtrees, ie. $E^{+}(i) \leq 2$, for all $i \in BT$ and also there is a distinction between the subtrees on the left and on the right of a node. The successor of a node is either null or is a LSUB-NODE if it is on the left and RSUB-NODE if it is on the right. We define the level of a node by initially letting the root be at level 1, then if a node is at level i, then the roots of its subtrees are at level i + 1. The depth of a tree is defined to be the maximum level of any node in the tree.

Theorem 6: *The maximum number of nodes on level i of a binary tree is $2^{(i-1)}$, for $i \geq 1$.*

Proof: *The proof is by induction. The root is the only node on level 1, hence maximum number of nodes on level $i = 1$ is $2^{0} = 1$. Now suppose for a general value j where $1 \leq j \leq i$, the maximum number of nodes on level j is 2^{j-1}. Then by assumption, the maximum number of nodes on level i-1 is 2^{i-2}. Since each binary tree has a maximum outdegree of 2, then the maximum number of nodes on level i is 2 times the maximum number of level i-1 or 2^{i-1}.*

The maximum number of nodes in a binary tree of depth k is given by, $\sum_{i=1}^{k} 2^{(i-1)} = 2^k - 1$ *(geometric progression).* †

Theorem 7: *let* n_0 *and* n_2 *be the number of the nodes with* $E^+ = 0$ *and* $E^+ = 2$ *in a binary tree BT, then* $n_0 = n_2 + 1$.

Proof: *let* n_1, n, *and* b *be the number of nodes with* $E^+ = 1$, *all the nodes and the number of branches in BT. We have,*

$$n = n_0 + n_1 + n_2 \qquad (I)$$

since all nodes in BT have $E^+ \leq 2$.

Clearly $n = b + 1$ (II)

since all the nodes, except the root, in BT have $E^- = 1$. *All branches in BT emanate from a node with either*

$E^+ = 1$ *or* $E^+ = 2$, *thus* $b = n_1 + 2n_2$ (III)

from (II) and (III) we get

$$n = 1 + n_1 + 2n_2 \qquad (IV)$$

and from (I) and (IV) we get

$n_0 = 1 + n_2$. †

A sequential representation of a binary tree is numbering the nodes in the following manner, number the root by 1 then number those nodes on

level 2 and so on. Nodes on any level are numbered from left to right. Now the nodes can be stored in a one dimensional array, BTREE, with the node numbered i being stored in BTREE (i). The following theorem enables us to easily determine the locations of the predecessor, LSUB and RSUB nodes of a given node.

__Theorem 8__: If a complete binary tree with n nodes (ie. depth = $\lfloor Log_2 n \rfloor + 1$) is represented sequentially then for any node with index i, $1 \leq i \leq n$ we have:

(i) predecessor of node i is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, then i is the root and has no predecessor.

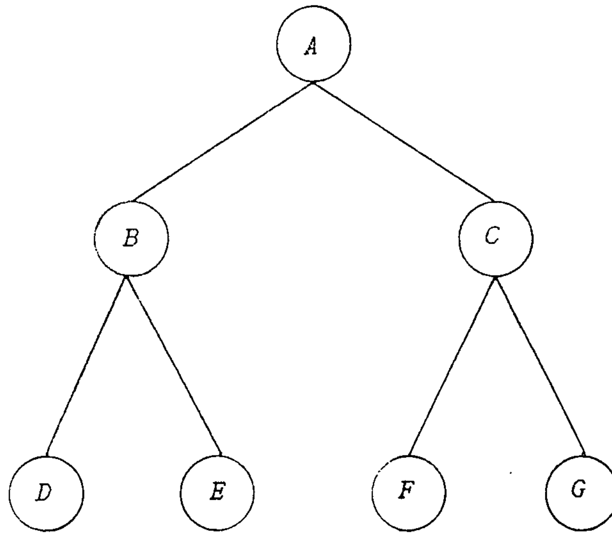
(ii) LSUB-NODE of node i is at 2i if $2i \leq n$. If $2i > n$, then i has no LSUB-NODE.

(iii) RSUB-NODE of node i is at 2i + 1 if $(2i + 1) \leq n$. If $(2i + 1) > n$, then i has no RSUB-NODE.

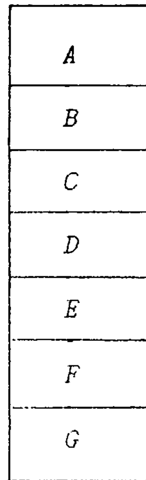
__Proof__: First we prove (ii) by induction, for $i = 1$ clearly LSUB-NODE is at level 2 unless $n < 2$ in which case 1 has no LSUB-NODE. Now assume that for all j, $1 \leq j \leq i$, LSUB-NODE of j is at 2j.

Then the two nodes immediately preceeding LSUB-NODE $(i + 1)$ in the representation are the RSUB-NODE and the LSUB-NODE of i. The LSUB-NODE of i is at $2i$, hence the LSUB-NODE of $(i + 1)$ is at $(2i + 2) = 2(i + 1)$ unless $2(i + 1) > n$ in which case $(i + 1)$ has no LSUB-NODE. (iii) is the immediate consequence of (ii) and the number of nodes on the same level from left to right. (i) follows from (ii) and (iii). †

In this work we sometimes, without loss of generality, assume that the root node is at level zero. Figure 5 illustrates the computer representation of a binary tree.



a full binary tree of depth 3



sequential representation

Figure 5: A binary tree with its sequential representations.

A heap is an abstract data structure consisting of a collection of items, $\{a_1, a_2, \ldots\ldots, a_n\}$, each of which is associated with a real valued data. First we will consider a heap in terms of a binary tree and then expand the definition for other types of heaps. The items are stored at the nodes of a special kind of binary tree. For every node, the value of the item is less than or equal to the values of the items stored at the immediate successor nodes (if such exist) in the tree. Thus, numbering the nodes in the usual way for a binary tree and assuming, for simplicity, that n (number of the items or nodes), is odd, ie. $a_i \leq a_{2i}, a_{2i+1}$ for $1 \leq i \leq n/2$, then this defines a heap. No ordering is implied between the items associated with two nodes if one is not the predecessor of the other, indirectly or directly. Each subtree of heap is also a heap. Node 1 is the root of the heap which is at the top of the tree and its corresponding item is of minimum value. We can represent a heap sequentially as a one dimensional array, see figure 6 below. The operations on heaps that we are concerned with are as follows:

(i) *Makeh (h)* which constructs the empty heap h;

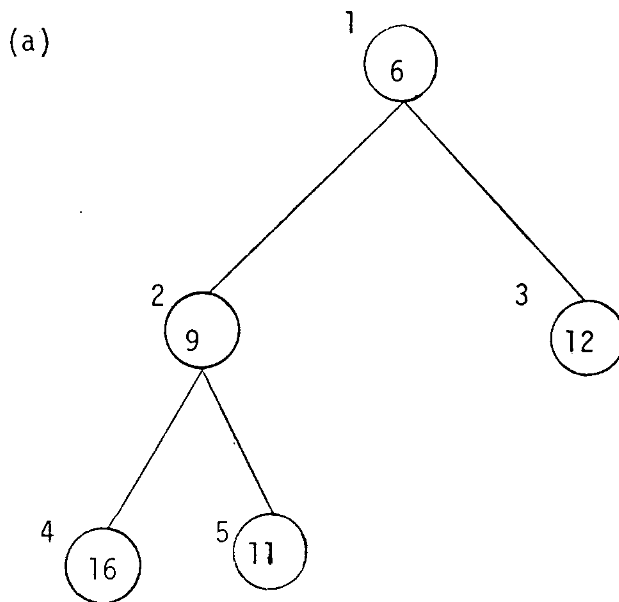
(ii) *Geth (S,h)* which takes the elements of set S as input to heap h;

(iii) *Addh (i, h)* which inserts the new data i to heap h;

(iv) *Delete (i, h)* which deletes the data i from heap h;

(v) *Getmin (h)* finds and returns the data of minimum value from heap h, and returns null if h is empty;

(vi) *Mergeh (h_1, h_2)* which returns the heap formed by combining disjoint heaps h_1 and h_2 and destroying h_1 and h_2. The new heap will have root with a value equal to that of h_1 if the value of the root of h_1 is smaller than that of h_2, otherwise to that of h_2.



(b)

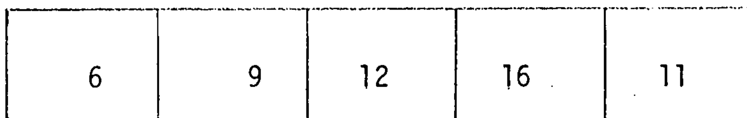


Figure 6: (a) Tree representation of a heap, (b) The computer representation of a heap.

Combining operations (i) and (ii) and calling it heap-former, then the following procedure, coded in standard pascal, will construct a heap out of a given binary tree. In the procedure below n is a global integer representing the number of the elements in the tree, and BINTRE is a one dimensional array type.

```
1     Procedure heapformer (VAR  BT : BINTRE);
2     VAR
3       s, j, nn       : integer;
4       dum            : integer;
5     Begin
6       s := 0;
7       nn := ((n + 1)/2) - 1;
8       j := nn;
9       while (n <> 0) do
10      Begin
11        if (BT(2*j) > BT((2*j) + 1))
12        then
13            s := 2*j + 1
14        else
15            s := 2*j;
16          if ( BT(j) > BT(s))
17        then Begin
18               dum := BT(j);
19               BT(j) := BT(s);
20               BT(s) := dum
21        end;
22        if ((2*s) > n)
23        then Begin
24               nn := nn - 1;
25               j := n
26        end
27      end         {while}
28    end;          {heapformer}
```

In steps 11 to 21 the data of two successors of a node i, ie. LSUB-node(i) and RSUB-node(i) are compared and if the smaller data is less than that of the node i then the nodes are swapped.

In this procedure the initial root of the binary tree is sifted down until it finds its proper place. If a node of a heap were removed, we could make the former last element the new initial root of the corresponding subtree, reducing n by 1, and sift the just move element up or down as appropriate. Sorting the elements of a heap can be done by successively removing the root, replacing it by ∞, and then sifting it down to restore the heap. This sorting scheme is called heapsort. In a heapsort, the depth of the heap is $O(\log n)$ and n elements must be removed, then the total time to reform the heap is $O(n\log n)$. The procedure can be streamlined by eliminating superfluous comparisons.

Theorem 9: The procedure heapformer forms a heap in linear time.

Proof: Let $f(k)$ be the maximum number of swappings necessary to form a heap out of (2^{n-1}) elements. Clearly $f(1) = 0$. Before dealing with node 1, subheaps are formed from the subtrees having nodes 2 and 3 as their roots. By definition forming each of these subheaps takes at

most $f(k-1)$ *swappings. When the two subheaps are merged, all swappings take place along a single path from node 1 to some terminal node with* $E^{+} = 0$. *Since the number of nodes on this path is* k, *at most* $k-1$ *swappings are required for the final merge (normally only a few swappings are required). Thus removing a node from a heap and then restoring the heap structure is an* $O(\log n)$ *process, at worst. Therefore to form a heap,* $f(k) = 2\,f(k-1) + (k-1),\ k \geq 2$ *where* $f(1) = 0$ *and* $f(k) = 2^{k}-1-k$, *we require fewer than one swapping per element. If the number of elements is between* $2^{k}-1$ *and* $2^{(k+1)}-1$, *then the number of swappings to form the heap is at most,*

$f(k+1) = 2\,f(k) + k = O(f(k))$.

And this proves the linear time claim in general.

†

Suppose in a given heap r values change. For the data whose new values are less then the heap's last element, put the new values in their respective former position and for the others put their values at the bottom of the heap and implicitly insert ∞ *in their respective former position. Finally after all the above operations are done, reform the heap. Reforming a heap after*

r elements change takes O[min (n, r logn)] time at worst.

Defining a d-tree to be a tree in which each node has at most d successors, then a d-heap is a d-tree containing one item per node arranged in heap order, see figure 7 below:

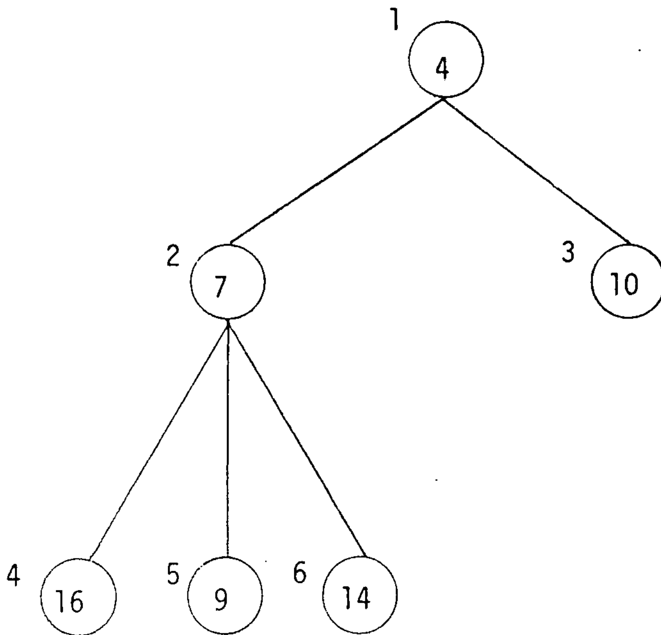
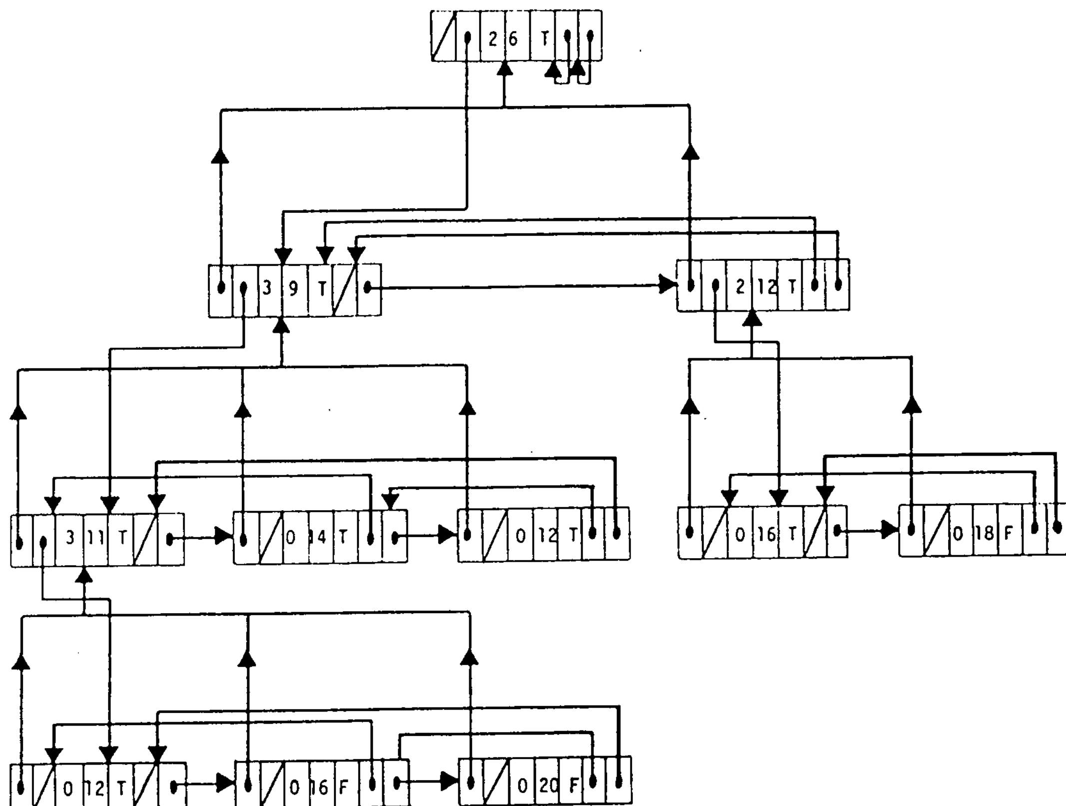


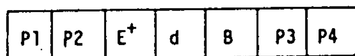
Figure 7: a 3-heap with nodes numbered as in binary tree, ie. top to bottom, left to right

Clearly the operation (v) has a running time of O(1). Operations (iii) and (iv) have a running time of $O(d\log_d n)$, where n is the number of nodes in the tree, since the depth of a d-heap is $\log_d n$. In d-heaps parameter d allows us to choose the data structure to fit the relative frequency of the operations, as the proportion of deletions decrease, we can increase the value of d, saving time on insertion. Due to regular structure of a d-heap we do not require explicit links to represent it. If the nodes are numbered in the manner explained above then the predecessor of node x is $\lceil (x-1)/d \rceil$ and the successors of x are the integers in the interval, $[d(x-1) + 2 \,..\, \min\{dx + 1, n\}]$. To implement a d-heap we use an array of positions from 1 to the maximum size of a heap. We also store an integer giving the size of the heap. We also associate an index h(i) to each item in the heap to give its position in the heap. Operation (vi) is rather difficult and time consuming on d-heaps. The operation d-heapformer, for forming a d-heap, analogous to heapformer, for forming a 2-heap, runs in linear time for $2 \leq d \leq n-1$.

A fibonacci heap or f-heap is a collection of item-disjoint heap-ordered trees. Fredman and Tarjan, [FRTA 85], used the following representation of f-heaps,

Each node has a pointer to its predecessor node or a special node null if it has no predecessor and a pointer to one of its successor nodes. The successors of each node are doubly linked in a circular list. Furthermore an integer is associated with each node indicating its number of successors, E^+, and a bit indicating whether the node is marked or not. The roots of all the trees in the heap are doubly linked in a circular list. A heap is accessed by a pointer to a root containing an item of minimum value, called minimum node of the heap. A minimum node of null denotes an empty heap. Each node has space for its data, four pointers, an integer indicating number of its successor and a bit. Figure 8 shows a f-heap represented in this manner.





P1: Pointer to predecessor in the tree;
P2: Pointer to one successor in the tree;
P3: Pointer to predecessor in the doubly circular linked list;
P4: Pointer to successor in the doubly circular linked list;
E^+: Number of the successors;
d : The value associated with a node;
B : Bit = T if the node is labelled
F otherwise.

Figure 8: f-heap representation.

The double linking of the lists of roots and the successors of a node makes deletion from such a list possible in O(1) time and the circular linking makes the merging possible in O(1) time.

A bucket is a list of nodes whose data fall within a given range, ie. a bucket ρ is a list of nodes i whose data $a(i)$ fall within the half open interval $[\rho z, (\rho + 1)z)$,
ie. $\rho z \leq a(i) < (\rho + 1)z$.

In this work we will represent a bucket by double linear linked lists. Associated with each node k in bucket ρ there is a data $a(k)$, two pointers and other information which we will explore later in section 9. Each data k, except the last, in bucket ρ has a pointer $p1(k)$ to its successor in the bucket. Each data k, except the first, has also a pointer $p2(k)$ to its predecessor in the bucket. To access the buckets we store the heads, address of their respective first elements, of the buckets in a master list, then the master list contains a pointer to the memory location of the first element of each bucket. The computer representation of heaps and buckets will be

explained in more detail in section 9, when required.

5 NETWORK AND TREE REPRESENTATIONS

There are several ways of representing a network $G = (N, A)$ in a computer, and the manner of representation directly effects the performance of algorithms applied to the network. Here we will give two such methods:

(a) Adjacency Matrix:

The adjacency matrix representing a network G is a 2-dimensional $n * n$ array W such that, the element (i, j) of the array, ie. $W(i, j)$, has the value w_{ij}, the weight of the arc (i, j), if $(i, j) \in A$, and ∞ otherwise.

Any algorithm applied to an adjacency matrix would require at least $O(n^2)$ as there are $n(n-1)$ elements to be examined. Storing such a matrix will also require $O(n^2)$ space. Therefore such a representation is excessive for sparse networks in which a large fraction of the elements of W are ∞, but may be considered as a good representation,

because of its simple structure, for dense networks.

(b) *Adjacency Lists:*

The most popular way of representing a network G in a computer is to use linked list structure. In this method, all the forward star arcs of a node are stored together and each arc is represented by recording only its terminal node and weight. A pointer is then kept for each node which indicates the block of computer memory locations for the forward star arc of that node.

In this manner of representation, we need $(n + 2m)$ *space or units of memory and* $O(n + m)$ *time for examining all arcs. The advantages of this method over adjacency matrix specially for sparse networks are obvious. This method of representation is also known as forward star representation, and if the forward star arcs of each node are ordered by ascending length, then the method is*

called sorted forward star representation form.

In this work we will adopt both these methods for network representations. Figure 10 illustrates the storage of the network shown in figure 9, in an adjacency matrix and also in a sorted forward star form.

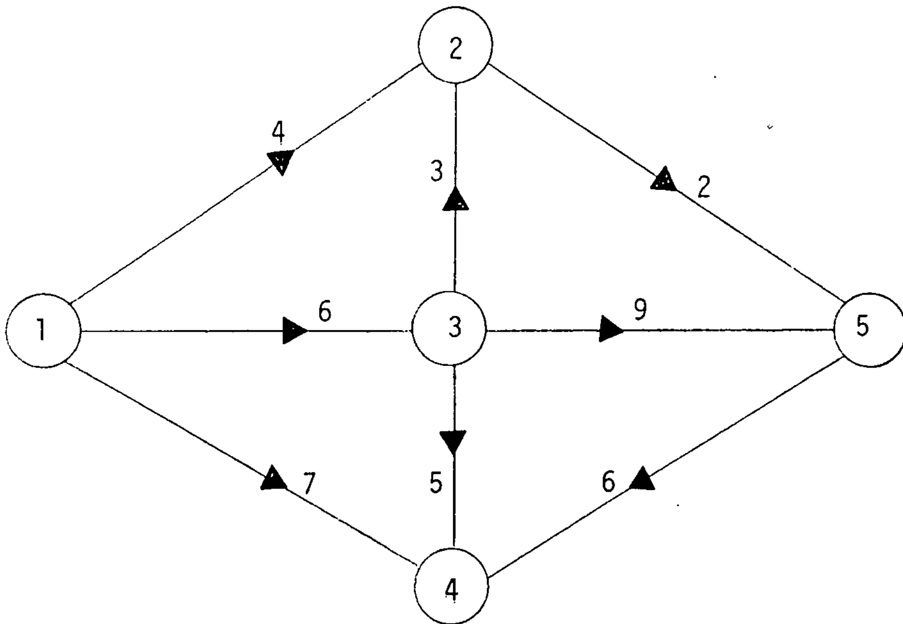


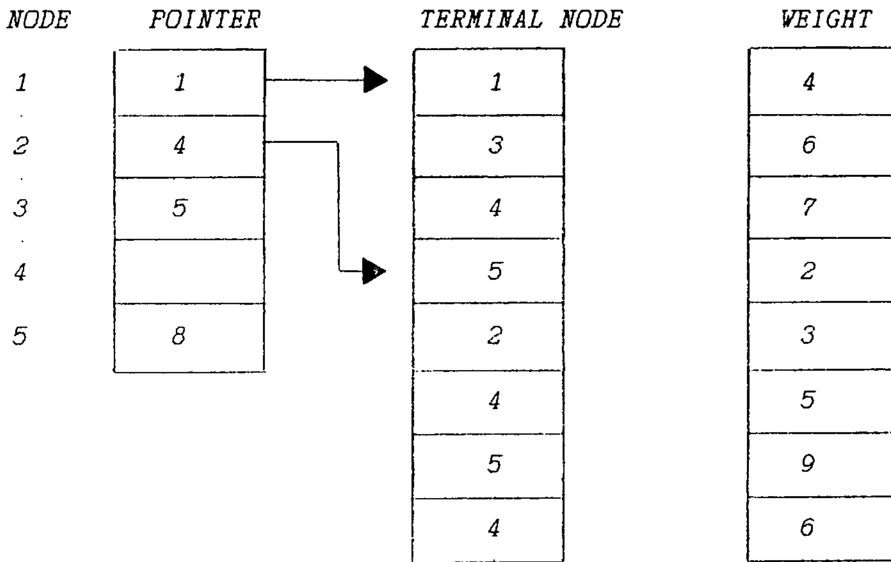
Figure 9: numbers associated with the arcs represent the weights of the arcs

(a)

TERMINAL NODE

INITIAL NODE	1	2	3	4	5
1	∞	4	6	7	∞
2	∞	∞	∞	∞	2
3	∞	3	∞	5	9
4	∞	∞	∞	∞	∞
5	∞	∞	∞	6	∞

(b) null pointer means no forward arcs



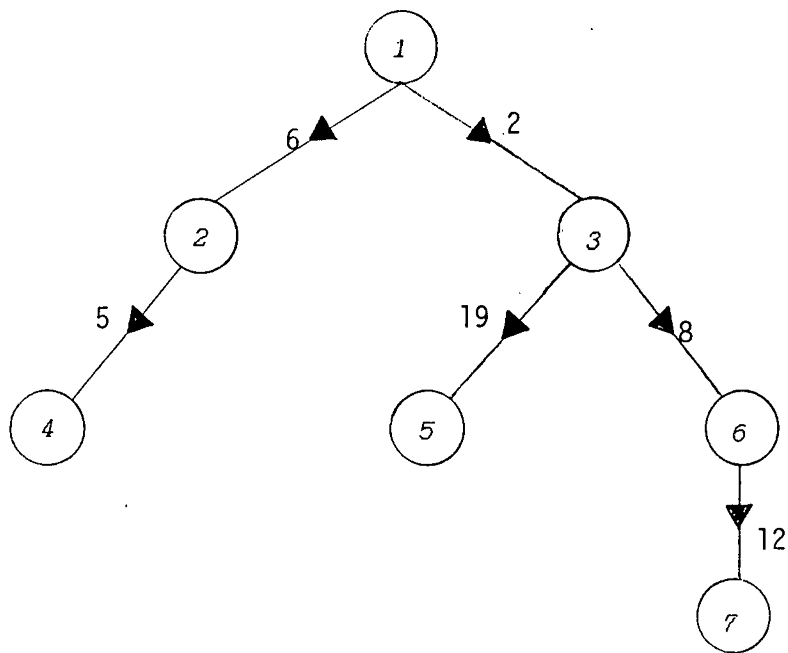
null pointer means no forward arc

Figure 10. network representation, (a) Adjacency Matrix, (b) sorted forward star

One of the most common ways of representing a tree in a computer is to think of the root, s, as the highest node in the tree and all the other nodes hanging below the root. The tree is then represented by keeping an upward pointer list containing the predecessor node of every node in the tree, except the root. We will assume that $PN(s) = s$. Associated with a tree we will also define a list, indexed by the node numbers, containing a label, $d(v)$, for each node v in the tree, whose value is the length or total weight of the unique path from s to v in the tree. In some implementations $d(v)$ is not necessarily the correct length but an over estimate that will eventually converge to the correct length.

If a node, i, does not belong to the tree, then its label is set to ∞, ie. $d(i) = \infty$, and this indicates that node i is not yet reached. We will also assume that $d(s) = 0$.

Figure 11, below, illustrated the computer representation of a tree using two linear lists, both indexed by the nodes.



NODE	PN	d
1	1	0
2	1	6
3	1	2
4	2	11
5	3	21
6	3	10
7	6	22

Figure 11: Computer representation of a tree

6. PROBLEM CLASSIFICATION

In 1957 MINTY, [MINT 57], made the following suggestion for finding a shortest path between a pair of nodes, source and sink, in a given network:

Construct a copy of the network using pieces of strings with lengths proportional to the weights of the arcs. Then place the source node in one hand and the sink node in the other, to stretch and determine the shortest path as the path with tense strings.

Since then there has been considerable development in solution methods for a variety of shortest paths problems. In general the shortest path problems can be divided into four groups, see figure 11.1 below:

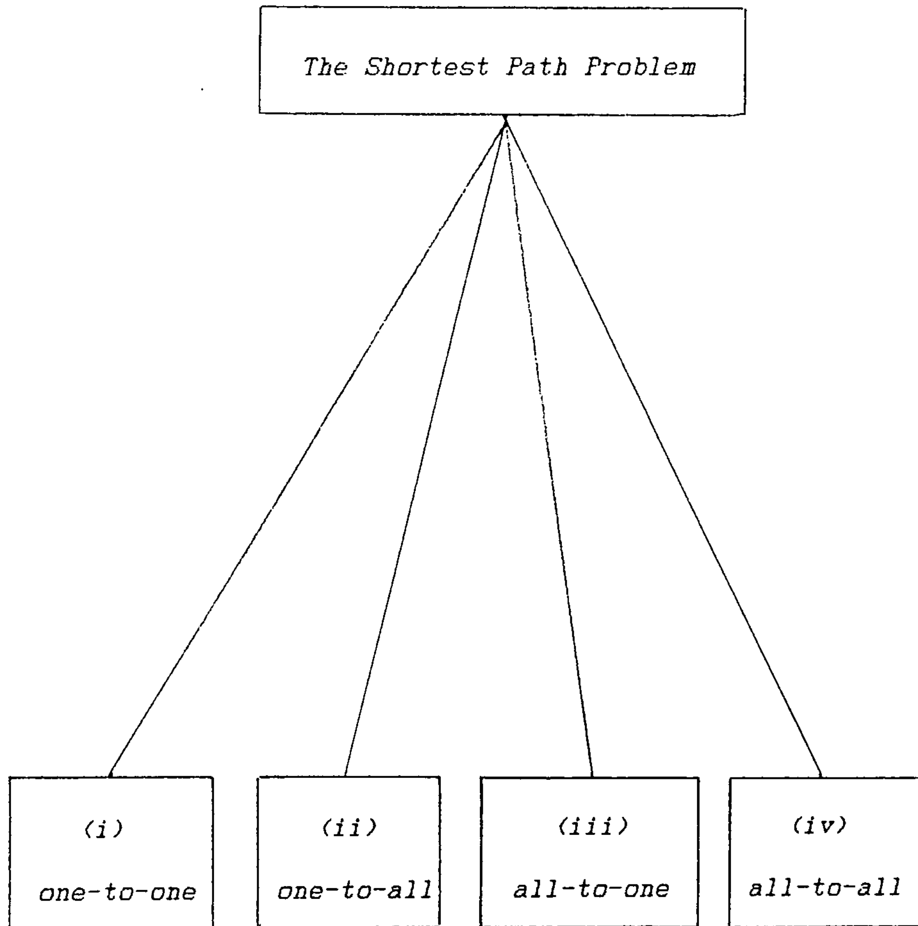


Figure II.1: Problem Classification

Each of these problems for a given network is defined as follows:

(i) *one-to-one problem is to find a shortest path from a given source to a given sink;*

(ii) *one-to-all problem is to find the shortest path from a given source to every other node;*

(iii) *all-to-one problem is to find a shortest path from every other node to a given sink;*

(iv) *all-to-all problem is to find a shortest path between every pair of nodes.*

Up to date, there is no efficient algorithm for solving one-to-one problem in a given network without having to find the shortest paths from the source to at least some of the other nodes, if not all. All-to-one problems and one-to-all problems are directional duals of each other, reversing the directions of the arcs in G converts one to the other. Therefore we will consider the solution methods for (ii) which will include (i) and (iii). We will refer to these solution methods as the single source algorithms. Furthermore, for

solving an all-to-all problem we can adapt an efficient single source algorithm and apply it to every node in the network, ie. apply the algorithm n times to the given network, each time having a different source node. We will refer to the specific algorithms designed for solving all-to-all problems as all source algorithms. As we will see some of the single source algorithms used to solve all-to-all problems, as explained above, are more efficient than most of and as efficient as the best of all source algorithms. Therefore in this work more emphasis is put on single source algorithms.

Extending our shortest paths notations for one-to-all and all-to-all problems,

In one-to-all problem the source node, S, is distinguished, then

$¥_{otm} = \{(S, v) \mid v \in (N - \{S\})\}$

and this can be abbreviated to $¥_{otm} = N - \{S\}$ *since s is distinguished. In all-to-all problems all node pairs, except nodes paired with themselves, are considered, then*

$$¥_{atm} = \{(u, v) \mid u, v \in N, u \neq v\}.$$

Thus a shortest path problem can be stated as $\pi(G,s)$ if it is an one-to-all problem and $\pi(G)$ if it is an all-to-all problem, since the source is understood.

Furthermore we will denote the weight of a shortest path from a source node to a given node v by d_v in a one-to-all problem, since the source is distinguished and d_{uv} in a all-to-all problem when the source node is u.

PART II

SINGLE SOURCE ALGORITHMS

7 SINGLE SOURCE ALGORITHMS

The best algorithms known for the one-to-all problems concatenate arcs to subpaths in order to find new paths. After obtaining a new path its total weight is compared to that of the current shortest path and if it is smaller, then the new path becomes the current shortest path. When the current shortest path cannot be improved any more then it becomes the shortest path.

Consider a network $G = (N, A)$ with no negative cycles, in a one-to-all problem with a source node s, clearly $d_{ss} = 0$. For each node v, $v \neq s$, there must be some final arc (u, v) in the shortest path from s to v. Whatever the identity of u, it is certain that $d_v = d_u + W_{uv}$. As a result of theorem 5, section 2, d_u is the weight of the shortest path from s to u. This is called the principle of optimality. But there are only $(n-1)$ number of choices for u. Clearly u must be a node for which $(d_u + W_{uv})$ is the minimum. Therefore the weights of the shortest paths must satisfy the following system of equations:

$$d_{ss} = 0$$

$$d_v = \min_{u \neq v} \{d_u + W_{uv}\} \quad (v \in N, \ u \neq s)$$

this system of equations was first formulated by Bellman, [BELL 58], and we will refer to them as Bellman's equations.

As a result of theorem 4 and theorem 5, section 2, we can conclude the following:

Suppose d_1, d_2, ..., d_n satisfy Bellman's equations in a network G = (N, A) with no negative cycle, then there exists a tree in G, rooted at the source with exactly (n-1) arcs, such that the path in the tree from the root to each node is the shortest path. We will refer to such a tree as the minimum tree or the shortest path tree.

Now let us consider the uniqueness of a finite solution to Bellman's equations.

Theorem 10: *If a network G = (N, A) contains no nonpositive cycle and there is a path from the source to every other node, then there is a unique finite solution to Bellman's equations.*

Proof: *let d_1, d_2, ..., d_n be the shortest path from the source to all the other nodes in G, and let d'_1, d'_2, ..., d'_n be any other finite*

solution to Bellman's equations, such that $d'_i \neq d_i$ for some i.

$d'_1, d'_2, \ldots\ldots, d'_n$ represent the weights of some paths, not necessarily the shortest paths in G. Accordingly, if $d_i \neq d'_i$ it must be the case that $d'_i > d_i$. Now choosing a node j such that $d'_j > d_j$, but $d'_k = d_k$, where (k, j) is an arc in the minimum tree of G (there must be at least one such arc since $d'_s = d_s$). Then $d'_j > d'_k + W_{kj}$, contrary to the assumption that $d'_1, d'_2, \ldots\ldots, d'_n$ satisfy Bellman's equations. Therefore there is a unique finite solution to Bellman's equations. †

Therefore solving a one-to-all problem in a given network $G = (N, A)$ is equivalent to finding a minimum tree of G rooted at the source. We will denote such a tree by:

$$T_G = (N_T, A_T).$$

To formulate a one-to-all problem as a linear programming model consider each of the Bellman's equations,

$$d_v = \min_{u \neq v} \{d_u + W_{uv}\} \qquad (I)$$

This gives a system of (n-1) inequalities, that is for a fixed v,

$$d_v \leqslant d_u + W_{uv} \qquad (II)$$

for $u = 1, 2, \ldots, (v - 1), (v + 1), \ldots, n$

Conversely, if $d_1, d_2 \ldots, d_{v-1}, d_{v+1}, \ldots, d_n$ *are given fixed values and* d_v *is maximised subject to (II), then (I) is satisfied. This suggests the following linear programming problem,*

maximise $d_2 + d_3 + \ldots + d_n$

subject to

$d_1 = 0$

and $d_v - d_u \leqslant W_{uv}$

for $u = 1, 2, \ldots, n$

$v = 2, 3, \ldots, n$

and $u \neq v$

However, Bellman's equations imply implicit functional relationships, that is each shortest path weight is expressed as a non linear function

of the other shortest path weights. Due to this reason Bellman's equations are not solvable as they stand, but there are methods for overcoming such difficulties which will be considered in the remainder of this part. Furthermore in theorem 10 we required that the network must not have nonpositive cycles, in order to have a unique finite solution to Bellman's equations, but the computational procedures that we consider here are actually effective for networks which contain no negative cycles. That is, although the solution to Bellman's equations is not unique, the computation will terminate with the correct solution.

We now develop a basic algorithm for solving one-to-all problems to which all known algorithms can be related.

Let d and PN be two n-element arrays defining in some algorithm. The ith element of d, d(i), contains the weight of some path from the source to the node, $i \in N$, and the corresponding element of PN, PN(i). contains the predecessor node of i on that path. If at the termination of algorithm d(i), for all $i \in N$, are the shortest paths then the solution is correct. Then the pointer chain

in PN will trace back a shortest path from every node i to the source node.

Now let [IMPROVE (A)] be a property such that,

[IMPROVE (A)]

$\equiv \exists\ (i,j) \in A$, such that $d(j) > d(i) + W_{i,j}$.

[IMPROVE (A)] is true if there is an arc in A which can be used to reduce some element of d.

Theorem 11: Suppose $d(i)$ is defined for all $i \in N$, such that $d(i) = d(P_i)$, where P_i is some finite elementary path from source to node i, then [IMPROVE (A)] is false if and only if $d(i)$ is a shortest path to i for all $i \in N$.

Proof: Suppose [IMPROVE (A)] is false and assume that there exists some node u with a shortest path of $d'(u)$ such that $d(u) \neq d'(u)$.

Clearly $d(u) < d'(u)$ cannot be true, since it implies that there exists a path to u with a weight less than the weight of the shortest path to u. Then $d(u) > d'(u)$, and this implies that $d'(u)$ is defined, ie. $d'(u) > \infty$, and hence there must be a path,

P_u $(s, i_1, i_2, \ldots, i_k, u)$ such that $d'(u) = d(P_u)$. Now let i_j be the first node in P_u such that $d(i_j) > d'(i_j)$, where $d'(i_j)$ is the weight of a shortest path to node i_j. Clearly $i_j \neq s$. Thus, $d(i_j) > d(i_{j-1}) + W(i_{j-1}, i_j)$. [$W(A, B) = W_{AB}$] but this contradicts the assumption that [IMPROVE (N)] is false.

Now suppose $d(i)$ is a shortest path to node i, for all $i \in N$. Then if [IMPROVE (A)] is true, then there is an arc (i, j) such that $d(j) > d(i) + W_{ij}$, implying a path P_j from s with $d(P_j)$ less than the weight of the shortest path from s to node j, which cannot be true.

Therefore [IMPROVE (A)] is false if and only if $d(i)$ is a shortest path to node i for all $i \in N$.
†

As a result of theorem 11 we can write a basic algorithm which may be considered as the underlying structure in all labelling algorithms. We will refer to this algorithm as labelling algorithm.

```
Algorithm labelling;
Step 1    {initialised}
            for i := 1 to n do
            begin
              d(i) := ∞;
              PN(i) := 0
            end;
              d(s) := 0;
              PN(s) := s;

Step 2  {search and update}
while [IMPROVE (A)] do
begin
  for some arc (i, j) satisfying [IMPROVE (N)] do
  begin

    d(j) := d(i) + Wij;
    PN(j) := i;
    end;
  end;
end.
```

d(i) is the weight of some path from s to node i, for all i $\in$ N when d(i) is the weight of a shortest path then this path is elementary. The algorithm enumerates elementary paths in some sequence of sufficient length to guarantee that shortest paths have been found for every node. A search for an arc (i, j) for reducing d(j) will always succeed until d(j) defines the weight of the shortest path to j for all j $\in$ N. In Step 2 of the labelling algorithm d(i) is the weight of some finite path from s whose last arc is (PN(i), i).

Theorem 12: *Labelling algorithms terminates if and only if array d contains the weights of the shortest paths from s to every other node.*

Proof: *The algorithm terminates if [IMPROVE (A)] is false, which in turn implies that d contains the weights of the shortest paths from to every other node. Now if the shortest paths to every node is defined in d, then it is clear that d(i) is the weight of some elementary path. But there is finite number of such paths in any finite network, and each iteration reduced some d(i), then termination must occur.* †

Clearly if a network contains a negative cycle, then the property [IMPROVE (A)] will always be true and hence the loop in Step 2 will never halt. Therefore the algorithm will never terminate.

Although this algorithm is fundamental, but it is not very useful. Firstly the algorithm will not terminate if the network contains a negative cycle and secondly and more importantly it does not outline how [IMPROVE (A)] is evaluated.

Operations required for evaluating [IMPROVE (A)] can be divided into two categories, scanning arcs

and searching nodes. Scanning an arc $(i, j) \in A$ is checking whether or not the inequality $d(j) > d(i) + W_{ij}$ holds and if it holds modifying the labels of node j by setting:

$$d(j) := d(i) + W_{ij};$$
$$PN(j) := i;.$$

Searching node $i \in N$ is scanning every forward star arc of node i.

The algorithms which are based on the labelling algorithm developed above are called labelling algorithms.

According to the manner of searching the labelling algorithms can be classified into two:

1. *label correcting algorithms*
2. *label setting algorithms.*

Both these methods start with a tree $T_0 = (N_T, A_T)$, such that $N_T = \{s\}$ and $A_T = \emptyset$. A label correcting method always updates arcs in A_T in a manner that replaces or shortens the weight of the paths from s to every other node in T, but

does not guarantee that the new path is a shortest path, until the algorithm terminates. A label setting method augments N_T and A_T respectively by one node $i \in N$ and one arc $(i, j) \in A$ at each iteration in such a manner that $i \in N^T$ and $j \in N-N_T$, and the unique path from s to i is a shortest path in G. A label setting method terminates when all arcs in A have their initial nodes and terminal nodes in N_T. We will consider these two general classes of labelling algorithms separately in the next two sections.

8. LABEL CORRECTING ALGORITHMS

An obvious way of evaluating [IMPROVE (A)] of labelling algorithms, section 7, is to use exhaustive searching. Algorithms that use such searching are called label correcting algorithms. This method was first suggested by Ford, [FORD 56], and subsequently details were worked out by others including Bellman, [BELL 58], and similar results were published by Moore, [MOOR 59].

Ford's algorithm is probably the earliest shortest path algorithm to be published.

In Ford's algorithm, each arc (i, j) is scanned in turn or examined for the property $d(j) > d(i) + W_{ij}$. *If no such arc is found then this implies that [IMPROVE (A)] is not true and hence the algorithm halts. Otherwise any arc for which the property holds may be remembered for use in updating the paths.*

```
Algorithm Ford;
begin
Step 1 {initialise}
       for i := 1 to n do
       begin
           d(i) := ∞;
           PN(i) := 0
       and;
       d(s) := 0;
       PN(s) := s;

Step 2 {search and update}
       repeat
          search for an arc (i, j) satisfying [IMPROVE (A)].
          if (the search succeeds) then
          begin
            d(j) := d(i) + Wij;
            PN(j) := i
          end;
       until the search fails;
end.
```

The proof of correction and termination of Ford's algorithm is the direct result of theorems 11 and 12, section 7.

With a sensible search strategy for examining arcs $(i, j) \in A$ to evaluate [IMPROVE (A)], Ford's algorithm has a time bound of $O(n^3)$, see [DERY 69] and [YENJ 70]. However the algorithms can be exponential under very simple search strategies as shown by D B Johnson in, [JOHN 77].

But using a search strategy which retains some information from previous searchs, like remembering the point at which the last search left of is sufficient to yield an $O(n^3)$ algorithm.

To develop algorithms with good bounds we first consider search strategies which are potentially exhaustive.

Let found ⇔ [IMPROVE (A)], then it will hold on termination of the following search:

```
found := false;
repeat
     select (i, j) ∈ A;
     if (d(j) < d(i) + Wij)
     then
        found := true;
     until ((found) or all arcs in A have been selected);
```

Now we can use this searching scheme directly in Ford's algorithm, since testing on found can determine if the search succeeded. The updating is carried out only if and immediately after found becomes true. Now by letting A' denote the set of arcs which have been examined for [IMPROVE (A)] and moving the updating operations into the search loop we get:

```
Step 2 {search and update}
       A' := { };
repeat
  found := false;
  while not (found) and (A - A' ≠ { }) do
  begin
    select (i, j) ∈ A;
    if (d(j) > d(i) + Wij) then
    begin
      found := true;
      d(j) := d(i) + Wij;
      PN(j) := i
    end
  end;
until not (found);
```

The correctness and termination of this algorithm is the direct result of theorems 11 and 12, section 7, if choosing $(i, j) \in A$ is a finite process which, when repeated, eventually chooses every arc in A.

Now consider a sufficient bound B for some rule of choice so that every arc will be chosen within B choices. Again with B defined as above, theorems 11 and 12 will hold for Ford's algorithm with the following refinement:

```
Step 2 {search and update}
repeat
  found := false;
  count := 1;
  while (count < B) do
  begin
    choose (i, j) ∈ A;
    if (d(j) > d(i) + W_ij) then
    begin
      found := true;
      d(j) := d(i) + W_ij,
      PN(j) := i
    end;
    count := count + 1
  end;
until not (found);
```

To find a sufficient value for B, let the rule for choosing $(i, j) \in A$ *be, choose* a_{count}*, where* $a = (i, j) \in A$ *and, in some order* $A = (a_1, a_2, \ldots\ldots, a_m)$*. The first m choices will be exhaustive, so B = m is sufficient under this rule of choice. Let us rewrite the inner loop once more using these ideas. In addition we introduce a variable pweight which counts the number of entries to the inner loop, initially setting pweight := 0 then the inner loop becomes:*

```
          pweight := pweight + 1;
          found := false;
          count := 1;
          while (count < m) do
          begin
(inn)       (i, j) := a_count;
            if(d (j) < d(i) + W_ij) then
            begin
              found := true;
              d(j) := d(i) + W_ij;
              PN(j) := i
            end;
            count := count + 1
          end;
```

it is clear that theorems 11 and 12 hold for Ford's algorithm in which step 2, is replaced by the following:

```
Step 2 (search and update)
       repeat
            inn;
       until not (found);
```

and the variable pweight is ignored. To bound the outer loop define the property,

R ≡ (d(i) is the shortest path length from two to i, for all i for which there exists a shortest path P_i such that $|P_i| \leqslant$ pweight).

Theorems 13: If d(i) defines the shortest paths for all $i \in N$, then $|P_i| \leqslant$ pweight.

Proof: We only need to consider nodes i such that the arc shortest path P_i has exactly (pweight + 1) arcs. By assumption, for some such path $P_i = (s = i_1, i_2, \ldots\ldots, i_k, i)$, it is true that $d(i_k)$ is the weight of the shortest path to i_k, so the inner loop, inn, will set (d_i) to the weight of the shortest path to i and $^{P}N(i)$ to i_k, since it tests every arc. †

Theorem 13 and the preceeding discussion suggest a good exhaustive search in Ford's algorithm as follows:

```
Algorithm Ford with refinements;
begin
Step 1  {initialise);

for i:= 1 to n do
begin
  d(i) := ∞;
  PN(i) := 0
end;
  d(s) := 0;
  PN(s) := s;
  Pweight := 0;

Step 2  {search and update)
  repeat
    inn;
  until not (found) or Pweight > (n-1)
end.
```

In this algorithm two tree functions predecessor and length are only used and, it runs in time proportional to the depth t of a shortest path tree of least depth.

Theorem 14: *The algorithm terminates in O(tm) if ψ is defined for nodes in G and in O(nm) if ψ is not defined for some node in G.*

Proof: *The proof of this theorem is a direct result of theorems 11 and 12 and also the fact that the maximum number of arcs in a path is (n-1).* †

This is one of the best results known under an exhaustive search strategy. Deleting the variable (found) so that the outer loop terminates when pweight ≥ (n-1), then the resulting algorithm leads to Bellman's algorithm, [BELL 58], which is a derivation of Ford's algorithm, [FORD 56], with explicit iteration indices.

```
Bellman's algorithm;
Begin
  for i := 1 to n do
  begin
    d(i) := ∞;
    PN(i) := 0
  end;
  d(s) := 0;
  PN(s) := s;
  for K := 1 to (n-1) do
  begin
    for i := 1 to n do
    for j := 1 to n do
    if (d(j) > d(i) + Wij) then
    begin
      d(j) := d(i) + Wij;                    {search and
      PN(j) := i                              replace}
    end
  end
end.
```

In search and replace step of Bellman's algorithm every possible correction, ie. $i, j \in N$ and $(i, j) \in A$ or $(i, j) \in A$, is examined and this step is repeated $(n-1)$ times. Thus the algorithm always runs in $O(n^3)$ since there are $n(n-1)$ such possible corrections. $\psi_m(i)$, for some node i, undefined can only be detected if a negative cycle on a path to node i includes s and this can be detected by testing $d(s)$ against zero after termination.

An obvious improvement in this algorithm is that the forward star arcs of a node i with $d(i) = \infty$ are not required to be scanned in the search and replace step. This improvement can be made by replacing the search and replace step by the following:

```
for i := 1 to n do
   if (d (i) ≠ ∞) then
   for j := 1 to n do
     if (d(j) > d(i) + W_ij) then
     begin
          d(j) := d(i) + W_ij;
          PN(j) := i
     end;
```

This improvement also indicates that the order in which forward star arcs of nodes are examined is a major factor in the efficiency of the algorithm.

As a result of this observation it can be concluded that if each arc $(i, j) \in FS(i)$ has been scanned and found to satisfy the condition $d(i) + W_{ij} \geq d(j)$ then it is not necessary to scan these arcs until $d(i)$ decreases. Based upon this observation the algorithm can further be improved by only examining the forward star arcs of the nodes which have not been scanned since their label were last changed. This can be accomplished through the use of a boolean set, f, corresponding to set N. Initially the boolean element of each node i, f(i), is set to false until its label is changed. The boolean element of the source node, s, is set to true. Then when the label of a node is changed, its boolean element is set to true until all its forward star arcs are examined and then set to false again. The algorithm terminates when no more flag is set. Bellman's algorithm with this refinement is as follows:

```
Bellman's algorithm with boolean list;
begin
  for i := 1 to n do
  begin
    d(i) := ∞;
    PN(i) := 0;
    f(i) := false
  end;
  count := 1;
  d(s) := 0;
  PN(s) := s;
  f(s) := true;
  while (count > 1) do
  begin
    for i:= 1 to n do
    begin
      count := 0;
      if (f (i) = true) then
      begin
        for j := 1 to n do
        if (d_j > d_i + W_ij) then
        begin
          d(j) := d_i + W_ij;
          PN(j) := i;
          f(j) := true;
          count := count + 1
        end;
        f(i) := false
      end
    end
  end.
```

In this algorithm count is used to check whether a solution if found. Clearly theorems 11 and 12 hold for this algorithm and it runs in $O(nm)$ or $O(n^3)$ in case of complete networks.

Based on the preceding observation it can be seen that the forward star arcs of nodes need not be scanned in numerical order as above, they may instead be scanned in the order in which the nodes

were labelled. That is if node i was labelled before node j, then the forward star arcs of i are scanned before that of node j, regardless of the node numbers i and j. This observation can be implemented efficiently by using a queue structure or a one way linked list as defined in Section 4. This is because all the permissible operations, as stated in Section 4, are in O(1), except the operation CREATE(Q) which is of O(n). In this implementation nodes are placed on the queue as their labels are altered, and removed from the queue as their forward star arcs are scanned. In this form the forward star arcs of nodes are examined in the order in which they are placed on the queue, the queue is said to be managed in FIFO manner.

There is one problem in using a queue and that is if a node is placed on the queue whenever its label is changed, the same node may appear in more than one position on the queue. This means that the size of the queue may be longer than n. One way to avoid this is to use a boolean list of size n corresponding to N. Then initially the elements of this list, flag, are set to false and when a node appears on the queue, its flag is set to true until it leaves the queue when it is set to false

again. The following is Bellman's algorithms with this refinement.

```
Bellman's algorithm with queue;
begin
  for i := 1 to n do
  begin
    d(i) := ∞;
    PN(i) := 0;
    flag(i) := false
  end;
  CREATE (Q);
  ADDQ (s, Q);
  d(s) := 0;
  PN(s) := s;
  flag(s) := true;
  repeat
    u := FRONT(Q);
    flag(u) := false;
    DELETEQ(Q);
    for j := point (u) to (point (u+1) - 1) do
    if (d (term(j)) > d(u) + W_u term(j)) then
    begin
      d(term (j) := d(u) + W_u term(j);
      PN (term (j)) := u;
      if not (flag (term (j)) then
      begin
        flag (term (j)) := true;
        ADDQ (term (j) , Q)
      end
    end;
    until (EMPTYQ(Q))
end.
```

In this algorithm the function FRONT and the procedures CREATE, ADDQ, DELETEQ and EMPTYQ are as explained in section 4, the forward star representation of a network is considered in which variable point (i) is the pointer associated with node i and contains the address of the terminal node of the first forward star arc of node i in

list term. It is clear that theorems 11 and 12 hold for this algorithm and that it has an upper time bound of O(nm) since each node is removed from the queue no more than n times. For algorithms based on this refinement see [GIWI 73], [PAPE 74], [STEE 74], [VLIE 78], [DEFO 79a] and [DGKK 79]. In this implementation if the forward star arcs of the latest node added to the queue is examined before that of a node placed on the queue previously, it is said to be managed in LIFO (last in first out) manner. In general examining the list in a FIFO manner is much more efficient than LIFO alternative, since nodes in some sense closest to the root are scanned before those further out in the tree, that is if a path in the tree is extended from its end node before the labels of nodes closer to the root have been lowered, the extension will have to be relabelled later on.

The preceding observation can also be implemented as outlined by Pape, [PAPE 74], by using an output restricted dequeue, RDQ or simply a dequeue, as explained in section 4. In this implementation the nodes not in the queue are split into two classes.

(i) *the "unlabelled nodes", ie. those that have never entered the queue (ie. whose distance from s are still ∞);*

(ii) *the "labelled and unscanned nodes", ie. those that have passed through the queue at least once, and whose current distance from s has already been used.*

Then the unlabelled nodes are inserted at the end of the queue, while the nodes have been labelled and scanned are inserted at the beginning of the queue. An easy approach to this implementation consists of using a code to distinguish between the two classes of nodes and a node size array with two pointers to indicate the two ends of the queue, see section 4. In addition a node size array, sit, is used to indicate the situation that a node is in. The situation of a node i is one of the following three.

(i) *sit(i) = 1, if node i is currently in the queue;*

(ii) *sit(i) = 0, if node i is not in the queue and has not ever been on the queue, ie. i is unlabelled;*

(iii)   sit(i) = -1, if node i is not currently on the queue, but it had been before, ie. i is labelled and unscanned.

Bellman's algorithm with this refinement is as follows:

```
Bellman's algorithm with RDQ;
begin
  for i := 1 to n do
  begin
    d(i) := ∞;
    PN(i) := 0;
    sit(i) := 0
  end;
  CREATE (RDQ);
  ADDDQ (s, F, RDQ);
  d(s) := 0;
  PN(s) := s;
  sit(i) := 1;
  repeat
    u := FRONT (RDQ);
    sit(u) := -1;
    DELETEDQ (F, RDQ);
    for j := point(u) to (point (u+1) -1) do
    if (d(term(j) > d(u) + W_u term(j)) then
    begin
      d(term(j)) := d(u) + W_u term(j);
      PN (term(j)) := u;
      if (sit (term(j)) = -1) then
      ADDQ (term(j), F, RDQ)
      else
        if (sit (term(j)) = 0) then
        ADDDQ (term(j), B, RDQ)
    end;
    until (EMPTYQ (RDQ))
end.
```

In this algorithm all queue functions and procedures are as defined in section 4, and all, except CREATE which is of O(n), are of O(1). The variables B and F used in some of the queue

operations indicate the front and the back ends of the queue.

In the refinement with RDQ the forward star arcs of the nodes are examined in DEPTH-FIRST-SEARCH manner, that is the forward star arcs of the node which was most recently visited are examined. However, in the refinement with FIFO management they are examined in BREADTH-FIRST-SEARCH manner, that is the forward star arcs of the node which was last recently visited are examined. To examine the efficiency of Depth-First-Search over Breadth-First-Search consider the version of the algorithm with RDQ and let h be the amount by which the label of a node i is decreased, then the labels of all the nodes in the subtree of i must ultimately be decreased by h, unless the subtree later becomes restructured in which case some node labels will decrease by an even greater amount. In the implementation with a queue managed in FIFO manner updating these node labels are postponed, since node i is added to the back of the queue. In contrast, in the RDQ implementation node i is added to the front of the queue, if it is not already in the queue. Thus loosely speaking, nodes in the subtree of i tend to be updated before other nodes are searched. Thus

updating sequence helps to eliminate unnecessary node label corrections that are dominated by the h correction that should be transmitted through the subtree. That is, an arc (i, j) may satisfy the condition $d(i) + W_{ij} < d(j)$ only because d(j) has not been reduced by h.

As a result of this discussion clearly theorem 11 and 12 hold for this algorithm which has an upper time bound of O(nm). Algorithms based on this implementation have also appeared in [MAGO 78], [VLIE 78], [DGKK 79], [DEFO 79] and [PALL 81].

Theoretically, as a result of the above discussions this latest implementation of label correcting algorithms is the most efficient one, however practically this is not always true, see section 10.

All different implementations of the general label correcting algorithms stated in this section can be considered as specialised variants of the primal simplex algorithm where the optimal arcs, ie. arcs in A_T, are the basic variables augmented by nonexistent arcs which could join s to each node $i \in N-N_T$, ie. all arcs (s, i) with $W_{si} = \infty$. The interpretation is specially direct for the

algorithm with the latest refinement which ensures that the node labels always satisfy complementary slackness, ie. $d(j) - d(i) = W_{ij}$ for $(i, j) \in A_T$ and $d(r) - d(s) = W_{sr}$ for $r \in N - N_T$. Then the process of selecting an improving arc (i, j) corresponds to searching for an arc which violates dual feasibility, ie. a non basic with a negative reduced cost. The process of adding such an arc (i, j) to A_T and deleting an arc $(PN(j), j)$ from A_T is equivalent to simplex basis change. The update of node labels after this basic exchange clearly maintains complementary slackness. The pivoting strategy however is different for the algorithm with a FIFO management or the other refinements. In these variants of the algorithm the updating version of the primal simplex algorithm is different from the version of the algorithm with RDQ in the sense that a basis exchange is performed each time an arc is added to A_T, but the full set of updated node labels in a subtree arc not immediately determined. In particular these variants differ from the latest refinement, ie. with RDQ, by requiring the complementary slackness be maintained only locally rather than globally. The result of Dial, Glover, Kannig and Klingman, [DGKK 70], emprical study of Bellman's algorithm with FIFO management and also

with RDQ may support the theory that it is not necessarily beneficial to maintain complementary slackness after each iteration. The version with FIFO management postphones the updating of the dual variables (node labels) and this appears to balance the distortion caused by using locally updating dual variables with the work required to maintain globally updated dual variables.

Although most of the improved versions of the general label correcting algorithm stated in this section, are bounded from above by O(nm), these efficiency changes from algorithm to algorithm. The results of worst case analysis and computer memory requirement of these implementations are tables below:

ALGORITHM	UPPER TIME BOUND				MEMORY	
	SPARSE	RANK	DENSE	RANK	NODE SIZE ARRAY BOOLEAN	NODE SIZE ARRAY NUMERICAL
Ford	$O(n^3)$	4	$O(n^3)$	6		2
Ford with Counter Refinement	$O(nm)$	3	$O(n^3)$	4		3
Bellman	$O(n^3)$	4	$O(n^3)$	5		2
Bellman with flag refinement	$O(nm)$	3	$O(n^3)$	3	1	2
Bellman with FIFO management	$O(nm)$	2	$O(n^3)$	1	1	3
Bellman with RDQ refinement	$O(nm)$	1	$O(n^3)$	2		4

In the above table the codes for the algorithm which are used in this work are considered for worst case analysis and also memory requirement. The structure of the input data is not considered in memory requirement. The "rank" columns indicate the order of performance of the algorithms. This latter conclusion is based on the discussions through out this section about the algorithms; our empirical study (stated in section 10), and also the comparison of many publications on practical and emprical studies of these algorithms such as [DEFO 79a], [DGKK 79], [VLIE 78], [IMAI 84] and [PAPE 74].

9 LABEL SETTING ALGORITHMS

Classifying the nodes either as permanently or temporarily labelled, where a permanently labelled node is one with a label which is the shortest path length. Then if step (2) of general labelling algorithm, in section 7, is modified such that it finds a node r with the minimum temporarily label defined by,

$d(r) \equiv \min \{d(i) + W_{i,j} \mid$ *for all permanently labelled nodes* i *and unlabelled nodes* $j\}$

and makes the label of node r permanent, then the resulting algorithm is the general label setting algorithm. This algorithm was first proposed by Dijkstra, [DIJK 59], also a similar result was obtained independently by Dantzig, [DANT 60].

Now, let set N_T *represent the set of permanently labelled nodes, complemented by set* $(N-N_T)$ *which contains the temporarily labelled nodes. Define,*

$A^{*} (\subset A) = \{(i, j) \mid i \in N_T \text{ and } j \in (N-N_T)\}$

then the general label setting algorithm, named after Dijkstra, is as follows:

```
Dijkstra's algorithm (in general form);
begin
step 1 {initialise}
   for i:= 1 to n do
     begin
        d(i) := ∞;
        PN(i) := 0
     end;
     d(s) := 0;
     PN(s) := s;
     NT := {s};

step 2 {search and replace}
   while (A* = 0) do
      begin
        choose v ∈ (N-NT) such that d(u)+
        Wuv = minimum
         {d(i) + Wij | (i, j) ∈ A*};
         NT := NTU {v};
         A* := A* - {(i, v) | i ∈ N}
      end
   end.
```

If this algorithm, in the process of finding an arc in A^* which yields the shortest path tree extension, in step 2, many possible labels are calculated and discarded. The following implementation of this algorithm retains this information and thus avoids recalculations. This implementation of Dijkstra's algorithm will be referred to as Dijkstra's algorithm.

```
Dijkstra's algorithm;
begin
step 1 {initialise}
for i := 1 to n do
   begin
      d(i) := W_si;
      if (d(i) ≠ ∞) then
      PN(i) := s
   end;
   d(s) := 0;
   PN(s) := s;
   min := ∞;
   dum := 0;
   N_T := {s};

step 2 {search and replace}
while (N - N_T ≠ 0) do
begin
step 2' {update N_T}
  for i := 1 to n do
    if (i not in N_T) and (min > d(i)) then
      begin
         min := d(i);
         dum := i
      end;
  N_T := N_T U dum;
step 2" {update (N - N_T)}
   for i := 1 to n do
      if (i not in N_T) and (d(i) > (d(dum) + W_dum i)) then
      begin
         PN(i) := dum;
         d(i) := d(dum) + W_dum i
      end
   end
end.
```

In the above procedure variables dum and min are used to find the node which will become permanently labelled next.

Theorem 15: *Dijkstra's algorithm terminates in* $O(n^2)$ *time and* $d(i)$ *defines the shortest path*

length from the source to each node i if the network contains no arc with negative weight.

Proof: The proof of termination is by inspection. At each stage of the algorithm the nodes are divided into 2 sets, N_T and $(N-N_T)$. At each repetition of step 2, one more node becomes permanently labelled in step 2' and joins the set N_T. Thus after (n-1) repetition of step 2, $(N - N_T) = 0$ and algorithm terminates. In step 2', each operation is repeated at most n times and so is each operation in step 2". Thus the algorithm runs in $O(n^2)$ time. The proof of validity is inductive. Consider step 2, (search and replace) after k^{th} repetition and suppose that each node in N_T is labelled correctly, that is for each node $i \in N_T$, $d(i)$ defines the length of the shortest path. This is clearly true when $k=1$, since $N_T = \{s\}$ and s is labelled correctly. Now suppose that node $v \in (N - N_T)$ is chosen to be labelled next and let ${}^{P}N(v) = U$, then

$$d(v) = d(U) + W_{uv}$$

clearly if $U \in N_T$ then min $= d(v)$. Now suppose $U \in (N - N_T)$, in fact let node x be the first

node on the path from s to v which is not in N_T *and let* $^{P}N(x) = Z$.

ie.

Then, if all arc weights are non-negative,

$d(v) \geq d(x) + W_{zx}$

but $d(x) + W_{zx} \geq min$, *otherwise x would have been labelled, then,* $d(v) \geq min$.
But if v is chosen to be labelled next, then clearly there is a path from s through z to v with $d(v) \leq min$.

Therefore, $d(v) = min$, *and hence v is going to be labelled with* $^{P}N(v) = u$ *where* $u \in N_T$. *Thus v is labelled correctly and d(v) is the length of the shortest path from the source to node v.*

Note that the proof of validity of the algorithm breaks down if the network contains an arc with a negative weight, since we could not show that $d(v) \geq min$.

Sequencing techniques and lists are also used to improve Dijkstra's algorithm. Yen, [YENJ 72], implemented the general form of Dijkstra's algorithm with a refinement similar to the one above, except that he stored $(N - N_T)$ as a linked list and then in <u>step 2'</u>, {update N_T}, instead of obtaining dum, the node at the top of the list was used and then the upward pointer moves to point to the old pointer's successor. This implementation will still run in $O(n^2)$ time.

The manner in which set $(N-N_T)$ is searched and updated effects the computational timing directly. However having $(N-N_T)$ partially sorted rather than fully sorted as in, [YENJ 72], is more efficient since, firstly some nodes $i \in (N-N_T)$ have $d(i) = \infty$ and secondly set $(N-N_T)$ will usually change slightly from one iteration to the next (these statements will be justified in the remaining of this section).

Before considering further improved implementation of label setting algorithm, let us consider its relationship with simplex method. Let the set of arcs in A_T be the set of basic variables, complemented by artificial arcs which start at the

source, s, and at node i for each $i \in N-N_T$ such that $W_{si} = \infty$. Then the label setting algorithm may be viewed as a special purpose primal simplex method. Clearly, $d(i)$ satisfy complementary slackness at each iteration,

ie. $-d(i) + d(j) = W_{ij}$ for $(i, j) \in A_T$ and $-d(s) + d(i) = W_{si}$, for $i \in N - N_T$.

Furthermore, the process of selecting an improving arc (i, j) to enter the basis corresponds to searching, in some manner, for an arc which violates dual feasibility
(ie. $-d(i) + d(j) > W_{(i, j)}$) by the largest amount. Then the process of adding such an arc to A_T and deleting the artificial and corresponding to the terminal node of this arc, t, from this basis is equivalent to simplex basis exchange. The setting of $d(t)$ after performing this basis exchange simply maintains complementary slackness. Therefore, like label correcting algorithms, label setting algorithms are special purpose primal simplex methods which use different pivot strategies.

To have set $(N - N_T)$ partially sorted, $(N - N_T)$ can be maintained as a heap, as explained in section 4. The use of a heap was evidently first reported for this application by Murchland [MURC 60], however he failed to note that his treatment yields a worst case bound on complete networks of $O(n^2 \log n)$ time, not as good as the original algorithm which runs in $O(n^2)$ time. This was first noted by E Johnson, [JOHN 72].

To consider implementation of the general label setting algorithm with a heap, first let us define two more operations on heaps, these two operations sift up and sift down are parts of the procedure heapformer given in section 4. Furthermore in our implementation as was first suggested by D Johnson, [JOHN 77], each non-empty key of the heap will possess some node i in a non-negative network, and the value of the key will be the value d(i). The two operations sift up and sift down are concerned with a heap in which a single key had its value changed. If the value decreases (this case includes the case where a new node is added at the leftmost empty key on the lowest level), the heap is restored if the path from the root to the key of decreased value is reordered.

This may be done by comparing the value of the changed key with the key above (its predecessor in the tree). If the changed key has a lesser value then the values of the keys are interchanged, and the process is repeated on the key with the original change until no more interchange is required or the root is reached. The cost of this process is proportional to distance the changed value moves in the heap. This cost is bounded by the order of the depth of the heap, $O(\log_k n)$ *where* n *is the number of keys in the heap and value of* k *depends on the tree type, ie.* $k = 2$ *in a binary heap,* $k = d$ *in a d-heap. The procedure for restoring a heap, h, following a reduction in some* $d(v)$ *is as follows:*

```
Procedure siftup (v);
begin
   q := key (v)
   repeat
      if (q not the root) then
        if (d (v) < d(h(PN(q))) then
          begin
             h (q) := h (PN(q));
             q := PN (q)
          end;
      until (no key is moved);
      h (q) := v
end;
```

If the value of a key increases, the ordering of the entire subtree rooted at the key with changed value is affected. Clearly in this case it is

sufficient to reorder the path from the changed key toward the levels which is of the least value at each level. Hence the cost is proportional to $k\log_k n$, *since one of the k choices must be made at k each key of the path except the last. The algorithm for restoring a heap, h, following an increase in some d(v) is as follows:*

```
Procedure siftdown (v);
begin
   q := key (v)
   repeat
       if (q not in last level) then
         begin
            P := key of node u of min d(u) on the
                 subheap rooted at key (q);
            if (d(v) > d(h (p))) then
            begin
               h (q) := u;
               q := P
            end
        end;
      until (no key is moved);
      node (q) := v
end;
```

The proof of termination of these two operations within the time bounds stated are direct results of theorem 9, and more detailed versions of the procedures can be seen in procedure heapformer, given in section 4.

In the implementation of Dijkstra's algorithm, we will change values associated with nodes (creating new keys when necessary on the bottom of the heap)

and also identifying and removing the least element of the heap. This identification is in $O(1)$, since the least element of key value is always at the root of the heap. These operations are explained in section 4, DELETE (i, h) and GETMIN (h). Then the least value which is removed is replaced with the value from the rightmost key on the lowest level of the tree. This preserves the heap. Restoring order is then of $O\ (k\log_k n)$, since the removed in a heap of size n + 1 is equivalent to an increase of the root value in a heap of size n, the following implementation of Dijkstra's algorithm with a heap differs from that of D Johnson, [JOHN 77], mainly in the definition of keys, here are suggested by Tarjan, [TARJ 84], the key of a node v, has a value d(v) which is the length of the shortest path from s to v.

```
Dijkstra's algorithm with a heap;
begin
   for i := 1 to n do
      begin
         d(i) := ∞;
         ᴾN(i) := 0
      end;
        d(s) := 0;
        ᴾN(s) := s;
        heapformer (h);
        v := s;
        while (v ≠ 0) do
           begin
              for i := point (v) to (point (v + 1) - 1) do
              if (d (i) > d(v) + W_vi) then
              begin
                 d(i) := d(v) + W_vi;
                 ᴾN(i) := v;
                 if (i not in h) then
                 begin
                    ADDH (i, h);
                    siftup (i)
                 end
             end;
             v := GETMIN (h);
             DELETE (v, h)
      end
end.
```

By inspection, in this implementation there are one heapformer, n, DELETE operations, n ADDH operations and at most m decrease or label updating operations. Therefore if we use a binary heap, the algorithm runs in $O(m \log n)$, and if a d-heap with $d + 2 + m/n$, then the running time is in $O(m \log_{(2 + m/n)} n)$. The proof of validity and termination of these algorithms in the stated time bounds is the direct result of the above discussions and theorems 9, 11, 12 and 15. The

result of this implementation is clearly superior to that of Dijkstra's for spouse networks.

Fredman and Tarjan, [FRET 85], suggest the use of a heap called, FIBONACCI heap, which is an extension of binomial queues, see section 4, instead of a d-heap to implement Dijkstra's algorithm. The resulting algorithm is then bounded from above by $O(n\log(n+m))$ which gives the best result in implementing the algorithm with a heap. This implementation is the same as the one described above however, we have not analysed it in this work.

Another method which provides a more direct access to a temporary labelled node with the minimum total weight is called "address calculation sort". This method was orginally developed by Dial, [DIAL 65], and is based on the following observations.

If a node v not yet in the minimum tree, ie. $v \in N-N_T$, has a finite total weight, then it has been labelled, ie. a path to node v has been determined. Since any node can only be labelled from a permanently labelled node, then v must have been labelled by a node $u \in N_T$.

Upon being relabelled by node u, v's total weight will have become equal to d(u), total weight of a permanent node u, plus the weight of the arc (u, v). Therefore, for any labelled node $v \in N-N_T$ we have d(v) = d(u) + (the weight of some arc) where $u \in N_T$. Now suppose that node v is a temporary labelled node with the minimum total weight, d(v), then d(v) bounds from above all the permanently labelled nodes, ie. if $u \in N_T$ then $d(u) \leq d(v)$, since a node $u \in N_T$ has entered the tree before $v \in N-N_T$. It also bounds from below the weights of all the temporary labelled nodes, ie. if $t \in N-N_T$ then $d(t) \geq d(v)$. Furthermore, the weight of any temporary labelled node $t \in N-N_T - \{v\}$ is bounded from above by d(v) plus the maximum arc weight in the network, since the total weight of t equals the total weight of some permanently labelled node plus the weight of some arc, and d(v) bounds from above all the permanently labelled nodes. Therefore, denoting the maximum arc weight of a network by WMAX, then

$$d(v) \leq d(t) \leq d(v) + WMAX$$

ie. at any stage in the execution of the algorithm, if node v is a temporary labelled node

with the minimum total weight, then the total weights of all the temporary labelled nodes are bracketed on the lower side by $d(v)$ and on the upper side by $d(v)$ + WMAX.

Using this property, at any stage in the execution of the algorithm, the total weights of all the temporary labelled nodes can be represented modulo WMAX + 1. The best way to illustrate this is by loosely defining an array, NODEARRAY, with (WMAX + 1) locations where:

NODEARRAY(i) stores any labelled node, $u \in N-N_T$, for which $d(u)$ mod (WMAX + 1) = i.

<u>Theorem 16</u>: At any stage in the algorithm, NODEARRAY, can store temporary labelled nodes with every possible total weight, and no location of NODEARRAY will contain nodes with different total weights.

<u>Proof</u>: Suppose that, at some stage in the algorithm a temporary labelled node v has the minimum total weight among such nodes, and let $d(v)$ mod (WMAX + 1) = i. Furthermore let node v be any other temporary labelled node. Node r will

be stored in location i of NODEARRAY. The minimum value of d(r) is d(v) and at this value node r will also be stored in the same location, ie. NODEARRAY(i), since

$d(r) \bmod (WMAX + 1) = i.$

As d(r) increases by one unit at a time, then d(r) mod (WMAX + 1) = i+1, i+2, ..., consequently node r will be stored in locations i+1, i+2, ... When d(r) reaches (WMAX + 1), then d(r) mod (WMAX + 1) = 0, and node r will be stored in location 0, ie. NODEARRAY(0). As d(r) increases from (WMAX + 1), then d(r) mod (WMAX + 1) = 1, 2, ..., and node r will be stored in locations 1, 2, ..., in NODEARRAY. Eventually d(r) reaches the maximum possible value that it can have, ie. d(v) + WMAX, but (d(v) + WMAX) mod (WMAX + 1) = (d(v) - 1) mod (WMAX + 1) and since, d(v) mod (WMAX + 1) = i, then (d(v) - 1) mod (WMAX + 1) = (i-1). Therefore temporary labelled nodes with any possible total weight can be stored in NODEARRAY, and no location of NODEARRAY will contain nodes with different total weights. †

As a result of the theorem above, NODEARRAY achieves an "automatic sort" of the labelled nodes not yet in the tree relative to their total

weights. That is, starting from any location i in NODEARRAY, locations $i+1$, $i+2$, ..., will contain nodes of increasing total weight values. Upon reaching the end of the array, nodes in location 0 will have a higher total weight than those in location (WMAX + 1).

To complement NODEARRAY for computational purposes, it is arranged as follows:

$$NODEARRAY(i) = \begin{cases} NIL & \text{if } i \neq d(v) \bmod (WMAX + 1) \text{ for any } v \in N-N_r; \\ P & \text{where } P \text{ is a pointer to the first node in a linked list of nodes } q \in N-N_r, \text{ such that } d(q) \bmod (WMAX + 1) = i. \end{cases}$$

The current minimum total weight is then found by sequencially examining the elements of NODEARRAY in a "wrap-around" fashion (ie. when the end of the array is reached, go back to the beginning). Each time a pointer is encountered, the current minimum total weight is that of the nodes in the

linked list associated with that pointer. Each node u in this linked list can then be searched and removed from the linked list. A relabelled node v will have its location in NODEARRAY calculated, ie. d(v) mod (WMAX + 1) and added to the appropriate linked list. This may involve removing node v from its original linked list. The examination of NODEARRAY always assumes where the last examination ended so nodes with increasing total weights are encountered each time. The algorithm terminates when NODEARRAY is empty, implying that all the labelled nodes, or reachable nodes from the source, are in the tree.

Here, we explain, rather than give an implementation of this algorithm because of the complexity and the length of it. However, the complete Pascal code of this implementation is in appendix D.

```
Algorithm Address Calculation;
begin
   step 1
       {initialise};
   step 2
       while (NODEARRAY is not empty) do
       begin
          search through NODEARRAY to find the next
          pointer to a linked list;
          if (a pointer to a linked list is found) then
          begin
             repeat
                find the next node u, in the linked list;
                add node u to the tree nodes;
                for each forward star arc of node u,
                (u, v) where v ∈ N-N_T, do
                    if (d(u) + W_uv < d(v)) then
                    begin
                       if (node v is already in a linked
                       list in NODEARRAY) then
                       begin
                          compute node v's current address
                          (location) in NODEARRAY;
                          remove node v from its current
                          linked list pointed to from this
                          address;
                       end;
                       d(v) := d(u) + W_uv;
                       PN(v) := u;
                       calculate node v's new address;
                       add node v to the linked list
                       pointed to, from this address;
                    end;
                    remove node u from the linked list;
                    until (every node, u, in the linked
                    list has been examined);
             end;
         end;  {while}
end.
```

The proof that this algorithm is correct is the direct result of theorems 15 and 16. By inspection, we can also observe that this algorithm runs in O(n(WMAX + 1)) time and requires O(WMAX + 1) memory space. Clearly, it is not

possible to theoretically compare this algorithm with the other labelling algorithms, but almost all empirical studies of such algorithms have identified this implementation as the fastest single source algorithm for both sparse and dense networks in which WMAX is small compared with n and m, ie. $(WMAX) = O(n)$ or at most $(WMAX) = O(m)$.

However, in case of small networks with WMAX rather large, this implementation will be much slower than the other labelling algorithms.

This implementation can be improved by reducing the effort of inserting and removing nodes on the linked lists by postponing adding nodes to the list. This can be done by observing that it is unnecessary to scan the entire forward star of a permanently labelled node v. In particular, only the endpoint of a minimum weight arc in such a forward star needs to be considered for addition to NODEARRAY. This follows from the fact that the total weights of the temporary labelled nodes determined from node v will be bounded from below by the total weight of such an arc with the minimum weight. This refinement was first suggested by Dial, Glover, Karney and Klingman,

[DGKK 79], however it requires that the network to be stored in a sorted forward star form which requires some preprocessing in $O(n^2)$ time and this, clearly, makes the use of such implementation inefficient.

Another method of storing the temporary labelled nodes relative to their total weights is by means of buckets, see section 4. A precursor to this method is given by Loubal, [HITC 68], Dial, [DIAL 65], and also Gilson and Witzgall, [GIWI 73]. In this method, temporary labelled nodes whose total weights fall within a specified range are stored together. The collection of nodes is called a bucket. To sort several temporary labelled nodes of differing total weights, several buckets may be used. Each bucket will contain nodes of a different total weight range. For instance suppose that nodes A, B and C have total weights of 1, 3 and 7, repsectively. Then, if bucket 1 stores nodes v, such that

$$0 \leq d(v) < 4$$

and bucket 2 stores nodes v, such that

$$4 \leqslant d(v) < 8$$

then bucket 1 will contain nodes A and B, and bucket 2 will contain node C.

For any bucket holding nodes v, with total weights within (a, b], ie. $a \leqslant d(v) < b$, $(b - a)$ is its width. For example buckets 1 and 2 above have a width of 4. When several buckets are used to store temporary labelled nodes with different total weights, the set of buckets are arranged in a bucket list. The bucket list is a collection of buckets 0, 1, 2, ..., where bucket i contains nodes v, such that

$$a \leqslant d(v) < b$$

and bucket (i + 1) contains nodes, v such that

$$b \leqslant d(v) < c \text{ etc.}$$

All the buckets in the bucket list have the same width. In general if Z is the bucket width, then bucket i stores nodes v, such that

$$i * Z \leqslant d(v) < (i + 1) Z.$$

The bucket list achieves an automatic sort of the temporary labelled nodes, relative to their total weights. To access the nodes whose total weights are currently the minimum, the lowest non-empty bucket is found. Nodes in this bucket are then searched, ie. their forward star arcs are scanned. Any relabelled node is put into the appropriate bucket. This may require removing the node from its original bucket. Note that only nodes with forward star arcs are placed into the bucket list. This prevents unnecessary searching of a node that can not relabel any other node. The nodes in the lowest numbered non-empty bucket i, can be searched in any order, and this is achieved by setting Z equal to the weight of the lowest weighted arc in the network.

Theorem 17: if $Z = \text{minimum } \{W_{ij} \mid (i, j) \in A\}$, *then no node can relabel another node in the same bucket.*

Proof: Let $WMIN = \text{minimum } \{W_{ij} \mid (i, j) \in A\}$ *and suppose that bucket i contains two nodes u and v, both with temporary labels, and that node u is being searched. If node u relabels node v, then the new total weight of node v will be given by*

$d(v) = d(u) + \text{(the length of some arc)}$.

The lowest possible value that $d(v)$ could have is $(d(u) + WMIN)$ and for node v to be relabelled, its original total weight must have been greater than this. Now bucket i holds node u, such that

$$i * WMIN \leq d(u) < (i+1) * WMIN,$$

therefore the lowest possible value of $d(u)$ is $(i * WMIN)$. Thus the lowest possible value the new total weight of v could have is given by

$$\begin{aligned} d(v) &= (i * WMIN) + WMIN \\ &= (i+1) * WMIN \end{aligned}$$

and the original value of $d(v)$ must have been greater than $(i+1) * WMIN$. But this is contrary to the assumption that bucket i holds node v, since $d(v) < (i+1)\ WMIN$. †

<u>Corollary 17.1</u>: Any relabelled node will always be put into a higher numbered bucket in the bucket list.

Using this property, the search for the next lowest numbered bucket can always resume when the last one stopped.

The algorithm terminates when there are no more non-empty buckets left in the bucket list, implying that every node has been permanently labelled.

To implement the general label setting algorithm with this refinement, let us define the bucket list, BUCKLIST, a linear list, as follows:

BOOKLIST(i) =

- *NIL if bucket i contains no node;*
- *P if bucket i contains one or more nodes, then P is a pointer to the first node in a linked list of nodes in bucket i.*

Bucket i in BUCKLIST will contain node v such that,

$$i * Z \leq d(v) < (i+1) * Z$$

where Z is the bucket width and is set to WMIN. The minimum weight of the weighted arcs. The following is an outline of this implementation, and the complete Pascal code of it is in appendix E.

```
Algorithm bucketsort;
begin
   step 1
      {initialise}
   step 2
      while (there is still a non-empty bucket do
      begin
         search through BUCKLIST to find the next pointer
         indicating the next non-empty bucket;
         if (a pointer is found) then
         repeat
            find the next node, R, in the bucket;
            add node R to N_T;
            for every node C such that (R, C) ∈ A-A_T do
            begin
               if ((d(R) + W_rc) < d(C) then
               begin
                  if (node C is already in bucket) then
                  begin
                     calculate which bucket node C is in;
                     remove node C from its current bucket;
                  end;
               d(C) := d(R) + W_RC;
               FN(C) := R;
               if (node C has a forward star arc) then
               begin
                  calculate which bucket node C is to be
                  put in;
                  put node C into the appropriate bucket;
               end;
            end;
      end;
      remove node R from its bucket;
    until (every node in bucket has been searched);
  end;  {while}
end.
```

The proof of correctness of this algorithm is a direct result of the theorems 15 and 17, and the proof of its termination in $O(m + (n * Z))$ is by inspection and clear, note that the number of buckets necessary for the computation is at most $Z(n-1)$.

The efficiency of the above method, known as 1-level bucket depends highly on the parameter Z. Based on this observation, Denardo and Fox, [DEFO 79a], introduced the 2-level and k-level bucket techniques which have better computation times than the 1-level bucket technique. In 2-level bucket technique the temporary labelled nodes are maintained by a 2-level bucket system. That is on the first level the nodes are distributed into Z buckets of width $Z * WMIN$ and on the second level, the nodes which are contained in the smallest numbered bucket that is non-empty on the first level are distributed into Z buckets of width $WMIN$ of the second level. By doing so, the computation of the method will be reduced to $O(m + n \quad Z)$ time. The k-level bucket technique is similar to 2-level bucket and reduced the computation time to $O(m + KnZ^{1/k})$. However, we have not considered this refinement in this work.

All label setting algorithms run approximately in $O(n^2)$ time in worst case. However, as a result of the above discussions and theorems concerning the label setting algorithms, the study of many practical and empirical surveys such as those used for comparing label correcting algorithms and also our own empirical study of the best of these algorithms which is introduced in section 10, we can draw the following conclusions about the label setting algorithms. In this conclusion, it is assumed that the maximum weight of the weighted arcs in a network is small compared with n^2.

ALGORITHM	UPPER TIME BOUND				MEMORY		
	SPARSE	RANK	DENSE	RANK	NODE SIZE ARRAY BOOLEAN	NUMERICAL	OTHERS
Dijkstra	$O(n^2)$	5	$O(n^2)$	1	1	2	
with Binary heap	$O(m \log n)$	4	$O(n^2 \log n)$	5	1	3	
with d-heap	$O(m \log_{(2+m/n}n)$	3	$O(n^2)$	2	1	3	
with address calculation	$O(n(WMAX+1))$	1	$O(n(WMAX+1))$	3		4	$1(WMAX+1)$
with 1-level bucket	$O(m + nz)$	2	$O(m + nz)$	4		4	$1(m+z)$

10 AN EMPIRICAL STUDY

In this section, five different implementations of labelling algorithms are evaluated by solving the one-to-all problem on a diverse set of randomly generated networks using the same computer (PRIME 750), the same compiler (PASCAL RUN COMPILER) and executing the codes during a time period with a constant demand on CPU time. The implementations studied here are:

1. Dijkstra's label setting, S1;
2. general label setting with address calculation, S2;
3. general label setting with 1-level buckets, S3;
4. general label correcting with a queue, using FIFO management, C1;
5. general label correcting with a output restricted double ended queue, C2.

Each algorithm is used to solve the same set of "small" randomly generated networks, and its performance behaviour is observed as:

(a) the number of nodes in the networks grows;

(b) the number of arcs in the networks grows.

The number of nodes, n, in the networks are 10, 20, 40, 60, ..., 200 and for each node size there are 9 networks which vary with respect to random variation in their number of arcs, m, which is bounded from above by k, where k takes the values,

$$\frac{n(n-1)}{10}, \quad \frac{2n(n-1)}{10}, \quad \ldots\ldots, \quad \frac{8n(n-1)}{10}.$$

In other words we consider a complete network, ie. $m = n(n-1)$, and generate random networks with n nodes which are $(100-k)\%$ arc free, for k90, 80, 70, ..., 20 and we repeat the process for different values of n which are stated above. In all the networks the arc weights are three digit random numbers, regardless of the node size or the arc size. In the following algorithm, used for generating a random network with n nodes and $\frac{k\, n(n-1)}{100}$ arcs for a given n and a given k where $\frac{100}{n} \leq k \leq 100$, the procedures RAND2 and RAND3 produce 2 and 3 digit random numbers.

```
Algorithm Random Network;
begin
   for i := 1 to n do
   for j := 1 to n do
   if (i ≠ j) then
   begin
      RAND2 (num);
      if (num < (100-k-1))
      then
         RAND3 (Wij)
      else
         Wij := ∞
   end
end;
```

Note that we require $m \geq (n-1)$ in order to have a connected network, thus $k \geq \frac{100}{n}$.

The following table illustrates the computational times of the implementations tested.

n NODES	k DENSITY (%)	CPU TIME IN MILLISECONDS C1	C2	S1	S2	S3
10	10	0	0	3	24	79
20	10	6	3	9	61	91
40	10	21	21	63	79	116
60	10	58	58	137	109	145
80	10	97	103	239	130	179
100	10	151	124	366	172	230
120	10	209	206	515	233	287
140	10	352	339	700	273	336
160	10	412	388	903	342	394
180	10	648	842	1152	397	458
200	10	651	730	1400	458	503
10	20	3	3	6	40	82
20	20	12	12	18	48	100
40	20	58	61	73	78	131
60	20	91	94	158	118	170
80	20	164	187	272	173	239
100	20	309	306	421	239	300
120	20	476	500	603	324	376
140	20	530	663	803	397	440
160	20	778	985	1027	485	521
180	20	864	903	1300	615	648
200	20	1409	1576	1591	725	755
10	30	3	6	9	40	88
20	30	19	15	24	49	103
40	30	49	46	82	85	140
60	30	139	154	176	143	203
80	30	228	233	300	218	279
100	30	481	660	467	306	357
120	30	694	745	661	412	454
140	30	921	788	888	527	557
160	30	1045	1222	1167	672	700
180	30	1521	1639	1464	788	809
200	30	1700	2397	1785	943	955
10	40	6	7	9	34	91
20	40	21	15	25	58	106
40	40	97	82	90	97	155
60	40	233	200	200	176	243
80	40	360	418	331	252	306
100	40	676	788	512	367	415
120	40	866	1048	734	521	548
140	40	1321	1618	976	633	667
160	40	1376	1712	1276	788	812
180	40	1967	2328	1594	970	978
200	40	2445	3081	1957	1154	1146

n NODES	k DENSITY (%)	C1	C2	S1	S2	S3
10	50	6	10	10	33	91
20	50	33	34	27	51	112
40	50	97	110	97	112	173
60	50	260	303	212	185	257
80	50	575	591	372	309	363
100	50	585	673	563	425	464
120	50	963	1149	813	594	618
140	50	1609	1600	1079	725	749
160	50	1924	2379	1394	939	954
180	50	2206	3203	1749	1136	1131
200	50	3227	3933	2158	1370	1360
10	60	7	9	9	46	91
20	60	36	36	30	54	109
40	60	109	122	110	115	179
60	60	297	324	234	218	276
80	60	676	952	403	336	385
100	60	852	1012	616	491	524
120	60	1215	1533	869	679	697
140	60	1821	2537	1167	870	885
160	60	2118	2654	1509	1085	1070
180	60	2961	3534	1888	1297	1294
200	60	3836	5815	2330	1554	1545
10	70	9	9	6	36	94
20	70	31	27	33	55	112
40	70	163	185	115	131	200
60	70	333	448	251	225	288
80	70	706	912	430	370	412
100	70	894	1224	661	527	558
120	70	1475	1758	934	739	755
140	70	1882	1970	1251	958	972
160	70	2537	3048	1627	1203	1194
180	70	3188	4309	2037	1476	1451
200	70	4734	7691	2500	1764	1764
10	80	9	9	13	36	94
20	80	30	33	33	60	112
40	80	142	160	122	140	209
60	80	379	524	263	254	316
80	80	621	667	463	403	445
100	80	1100	1354	730	600	639
120	80	1493	1718	1012	797	818
140	80	2657	4566	1343	1057	1054
160	80	3127	3597	1739	1324	1312
180	80	3715	5097	2194	1615	1600
200	80	4591	5897	2691	1933	1927

n NODES	k DENSITY (%)	C1	C2	S1	S2	S3
10	90	12	12	13	40	94
20	90	39	40	36	63	118
40	90	228	249	127	149	212
60	90	445	673	281	276	333
80	90	894	1073	500	448	500
100	90	1139	1737	761	637	667
120	90	2013	2748	1070	876	897
140	90	2591	4263	1436	1136	1139
160	90	3470	4636	1869	1440	1419
180	90	4430	6582	2321	1754	1739
200	90	6143	8967	2867	2161	2131

The following conclusions based on the above table can be drawn about the tested algorithms.

1. *The general label setting implemented with address calculation sort is the most efficient. However, in this study only small networks (ie. $n \leq 200$) are considered and the arc weights are small compared with n^2.*

2. *The general label setting with bucket sort is almost as efficient as the one with address calculation, especially in case of dense networks.*

3. *The general label correcting with a output restricted double ended queue is more efficient than that with a single queue for sparse networks ($K \leq 20\%$) and also for small networks ($n \leq 100$).*

4. *Dijkstra's algorithm becomes more efficient as the number of nodes grows and also as the network becomes more dense, especially for $k \geq 30\%$, Dijkstra's algorithm becomes the third best.*

5. *The general label correcting with a single queue managed with FIFO, becomes the fourth best with $n \geq 120$, and the general label correcting with output restricted double ended queue is the third best with $k \leq 20\%$, the fourth best with $n \leq 100$ and the fifth best otherwise.*

Figure 12, illustrated the graph of the average CPU times of the algorithms against different densities in the same set of diverse randomly generated networks with upto 200 nodes.

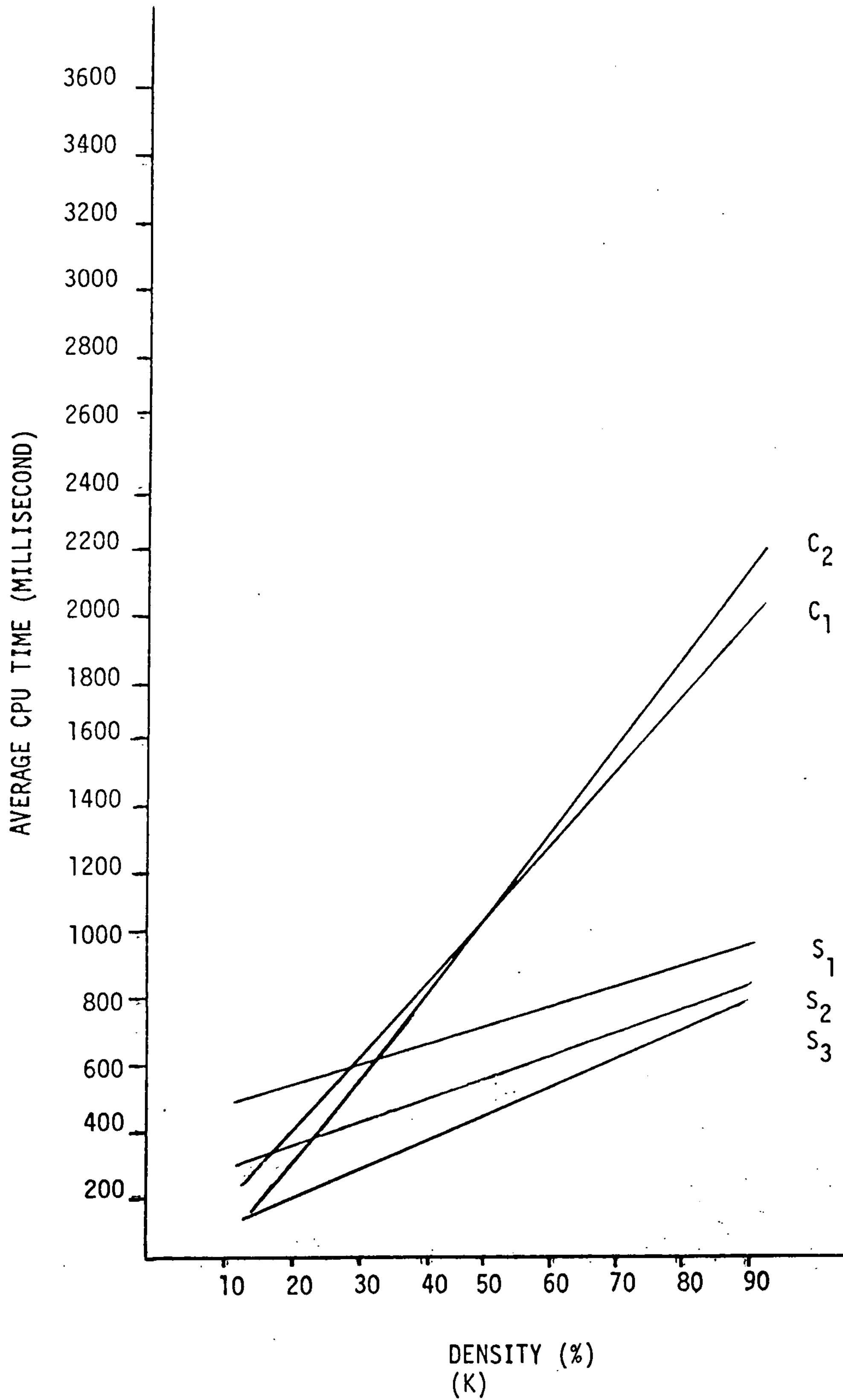


Figure 12, the graph of average CPU times for networks with up to 200 nodes.

PART III

ALL SOURCE ALGORITHMS

11 MATRIX MULTIPLICATION ALGORITHMS

To study all source algorithms, as defined in problem classification,

let $d_{ij}^{(m)}$ = *the length of a shortest path from i to j subject to the condition that the path contains no more than m arcs.*

then if $w_{ii} = 0$, *for all i,*

$$d_{ii}^{(0)} = 0 \tag{11.1}$$

$$d_{ij}^{(0)} = \infty$$

$$d_{ij}^{(m+1)} = \min \{d_{ik}^{(m)} + w_{kj}\}$$

Clearly the computation of (11.1) will converge at the $(n-1)^{st}$ *operation, ie.* $d_{ij}^{(n-1)} = d_{ij}$. *The overall computation is in* $O(n^4)$ *time, since it is the n repetition of Bellman's algorithm which runs in* $O(n^3)$ *time. However, these equations have a property that their computation is equivalent to the "plus-min" inner product,*

ie. let $c = [C_{ij}] = AB$

where $$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

and suppose that the matrix multiplication is redefined as *, where

$$C = [C_{ij}] = A * B$$

and

$$C_{ij} = \min \{a_{ik} + b_{ik}\},$$

ie. let addition take the place of multiplication and minimisation take the place of addition.

Now let $D = [d_{ij}]$ and consider $W = [w_{ij}]$, ie. represent d_{ij}'s in a n*n array and consider the adjacency matrix representation of the arc weights, then:

$$D^{(0)} = [d^{(0)}_{ij}], \text{ where } d^{(0)}_{ij} = \begin{cases} 0 \text{ if } i = j \\ \infty \text{ otherwise} \end{cases}$$

$$D^{(1)} = D^{(0)} * W$$

$$D^{(2)} = D^{(1)} * W = (D^{(0)} * W) * W$$

. . . .

$$D^{(n-1)} = D^{(n-2)} * W = (((D^{(0)} * W) * W) \ldots\ldots * W)$$

Theorem 18: Plus-min inner product method solves equations of (11.1) in $O(n^3 \log_2 n)$.

Proof: For this type of matrix multiplication, clearly, $D^{(0)}$ is the identity matrix.
ie. $D^{(0)} * W = W$, furthermore the multiplication is associative, thus $D^{(n-1)} = W^{(n-1)}$, where $W^{(n-1)}$ is the $(n-1)^{st}$ power of W.

Now since $W^{2k} = W^{(n-1)}$ for any $2^k > (n-1)$, then it is appropriate to square W until a sufficiently high power of W is obtained, ie. $W^2 = W * W$, then, $W^4 = W^2 * W^2$, ..., W^{2k}, for $2^k > (n-1)$. Now clearly this method requires $\log_2 n$ multiplications, each of which is an $O(n^3)$. Thus the method solves the equations in $O(n^3 \log_2 n)$. †

This approach to "all-to-all" shortest path problems was first made by Farley, Land and Murchland, [FALM 67], and the algorithm was called, by them, "cascade algorithm". Hu, [HUTC 67], also gives an extensive discussion on this type of approach to all-to-all problems.

12 TRIPLE ALGORITHMS

The earliest work on this type of algorithm was by FLOYD, [FLOY 62], on a paper by Warshall, [WARS 62], on transitive closure which is equivalent to a shortest path problem in which all arc weights are zero. This method runs in $O(n^3 \log_2 n)$ time. Before considering triple algorithms, let $d_{ij}^{(m)}$ be redefined as:

$d_{ij}^{(m)}$ = the length of a shortest path from i to j subject to the condition that the path does not pass through nodes m, m+1, ..., n (except i and j).

Now, a shortest path from i to j which does not pass through nodes m+1, m+2, ..., n either

(1) does not pass through node m in which case $d_{ij}^{(m+1)} = d_{ij}^{(m)}$;

or

(2) does pass through node m in which case $d_{ij}^{(m)} = d_{im}^{(m)} + d_{mj}^{(m)}$.

Thus we have,

(12.1) $d_{ij}^{(1)} = W_{ij}.$

$$d_{ij}^{(m+1)} = \min \{d_{ij}^{(m)},\ d_{im}^{(m)} + d_{mj}^{(m)}\}$$

and clearly, $d_{ij}^{(n+1)} = d_{ij}$, *the length of a shortest path from i to j. This algorithm is named after Warshall-Floyd and has the following general form:*

```
Algorithm Warshall-Floyd;

begin
  {initialise}
  for i:=  1 to n do
  begin
    for j:=  1 to n do
        dij:=  Wij;
    dii  = 0
  end
  {search and replace}
  k:=  0;
  while (k < n) do
  begin
    k:=  k+1;                       )
    for i:=  1 to n do              |
    for j:=  1 to n do              }            (12.2)
        (dij, (dik + dkj))          |
  end;                              )
  {check}
  for i:=  1 to n do
  if (dii < 0)
  then
      report failure
  else
      report success
end.
```

Theorem 19: Algorithm Warshall-Floyd terminates in $O(n^3)$ reports,

(i) success and defines a shortest path between every pair of nodes if there is no negative cycle;

(ii) failure otherwise.

Proof: The time bound is obvious from inspection of the program, for correction let

$T \equiv d_{ij} = \min \{d_{ij}, (d_{ik} + d_{kj})\}.\ k<n$

Clearly T is satisfied before the start of the minimisation process, ie. after the initialisation steps in the algorithm. Now let $k' = k+1$ for some k under which T is satisfied initially. Clearly (12.2) examines every triple $< i, K', j >$, replacing d_{ij} if and only if there is a shorter path via $\{1, 2, \ldots\ldots, k'\}$ than via $\{1, 2, \ldots\ldots, (k'-1)\}$. But this satisfies T for $K = 0$ to $k \leq n$, due to the fact that there can only be a maximum of $(n-1)$ arcs in a path and also the results of theorems 11 and 12, if there is no negative cycle, ie. the algorithm will halt with a solution if there is no negative cycle. Otherwise

for some $i \in N$, $d_{ii} \leq 0$ which indicates that there is a negative cycle in the network. †

Dantzig, [DANT 67], proposed a variant of Warshall-Floyd's algorithm which requires the same computation time and memory space. Both algorithms are the same except in Dantzig's algorithms the iteration step, ie. (search and replace) is divided into parts. If the following, (12.3), replaces (12.2) of Warshall-Floyd's algorithm, then the resulting algorithm will be that of Dantzig.

```
(12.3)    for i := 1 to k do
             for j := 1 to k do
                dkj:= min (Wki + dij);
             for i:=  1 to k do
               for j:=  1 to k do
                  dik:=  min (dij + Wjk);
            for i:=  1 to k do
               for j:=  1 to k do
                  dij := min (dik + dkj , dij)
```

The proof of correctness and termination of Dantzig's algorithm is the same as that of Warshall-Floyd's algorithm.

Iri and Nakamoni, [IRNA 72], exhibited a set of triple algorithms which run in $O(n^3)$ time. Most of these algorithms are similar to and are based on Warshall-Floyd algorithm.

13 MODIFIED LABEL SETTING ALGORITHMS

The all-to-all problem on a network which contains no arc with negative weight, can be solved by n iterations of a label correcting algorithm, one for each possible source. Then, clearly this solution method will run in $O(nm\log_{(2+m/n)}n)$. If the label setting method is implemented with a d-heap as stated in section 9. This implementation runs faster than $O(n^3)$ time for sparse networks, and in $O(n^3)$ time for dense networks. However, the result can be further improved by implementing the label setting algorithms with address calculation sort, see section 9, or with a f-heap. Then as claimed by Tarjan, n repetition of the algorithm solves an all-to-all problem in a non-negative network in $O(n^2\log(n+m))$ time. Even if the network contains arcs with negative weights, the same time bound can still be obtained by making all the arc weights non-negative in a preprocessing step. Edmonds and Karp, [EDKA 72], defined the appropriate transformation which is as follows:

First we add to network G a new node (n+1) and a ZERO WEIGHT ARC ((n + 1), i) for every node i in G. Then $d_{(n+1)i}$ is calculated for every node i.

Using a label correcting algorithm will take O(nm) time. Finally a new weight for each arc (i, j) can be defined by $W_{ij} = W_{ij} + d_{(n+1)i} - d_{(n-1)j}$. Clearly, $W_{ij} > 0$ for every $(i, j) \in A$. This is due to $d_{(n+1)j}$ being the length of a shortest path from (n+1) to j which gives
$d_{(n+1)i} + W_{ij} > d(n+1)j$ and thus $W_{ij} > 0$. This transformation makes all arc weights non-negative and preserves shortest paths, since it transforms the lengths of all paths from a given node i to a given node j by the same amount, $d_i - d_j$.

Thus this solution method is correct for negative networks as well as non-negative networks and runs in $O(n^3)$ time. Then it may be concluded that the modified label setting algorithms are faster than triple algorithms which are in turn faster than matrix multiplication algorithms. Although this statement is true in case of worst case analysis, the empirical studies of these algorithms do not quite support it. However implementation of a label setting algorithm with a F-heap or address

caculation sort has not yet been considered for all-to-all problems in any empirical study, to the best of the author's knowledge, and that unlike the empirical studies of single source algorithms which mostly report consistant results, in the case of all source algorithms most results are not consistant. For example Dreyfus, [DREY 69], reported that Dijkstra's algorithm requires 50% more time than that of Floyd and that of Dantzig. Yen, [YENJ 70], reported that his implementation of Dijkstra's algorithm is 25% faster than Floyd's algorithm, Kelton and Law, [KELA 78], claimed that the matrix multiplication methods are most efficient on Dense networks, Floyd reported that his algorithm is the fastest, Glover and Klingman, [GLKL 82], have results that shows Dijkstra's algorithm is faster than that of Floyd. However, most of these studies agree that for small networks with up to 400 nodes, modified label setting algorithms are faster, especially in case of sparse networks.

Another reason which makes the use of label setting algorithms in solving all-to-all problems more popular is that in most practical situations

the shortest paths from every node of a subset of N to every other node in N are required, rather than from every node to every other node in N. Supposing K ($< n$) *nodes are to act as source nodes in a given network, then k repetitions of a label setting algorithm will solve the problem rather than n repetitions.*

PART IV

SENSITIVITY ANALYSIS
AND
POST OPTIMALITY ANALYSIS
OF ONE-TO-ALL PROBLEMS

14 SENSITIVITY ANALYSIS

In this section the sensitivity of an optimal solution to a one-to-all problem is studied. More precisely, the methods of characterising the maximum increase and decrease in the weight of an existing arc, optimal or non-optimal, that can be tolerated without changing the optimality of the current solution are analysed. However, before discussing these algorithms, consider the following expansions of definitions and notations of section 2.

Consider a connected and undirected network $G = (N, A)$ *and its minimum spanning tree,* $T_G = (N_T, A_T)$ *rooted at node s, source, where*

$N_T = N$

and $A = \{\langle i, j \rangle \mid i, j \in N$, *and i and j are connected*$\}$

$[\langle\ \rangle$ *denote an unordered pair*$]$.

Furthermore, let P'_v *denote the shortest path from s to v, also* P'_{uv} *denote the subpath from u to v on the shortest path* P'_v*, then* $P'_{uv} \subset P'_v \subset T_G$.

T_G defines a partial ordering of nodes $i \in N$, with respect to their paths from s, ie. if $i \in P'_j$ (ie. node i is on the tree path from s to j) then $d(i) \leq d(j)$ and we write $i \leq j$.

Each arc $\langle i, j \rangle \in A_T$ divides set N into two subsets N_{ij} and N'_{ij}, where

$N_{ij} = \{k \mid k \in N$ and $\langle i, j \rangle \in P'_{kj}\}$
and $N'_{ij} = \{k \mid k \in N$ and $\langle i, j \rangle \in P'_{ik}\}$.

N_{ij} and N'_{ij} are the node sets of the two trees in which T_G transforms after $\langle i, j \rangle \in A_T$ has been deleted. Note that $i \in N_{ij}$ and $j \in N'_{ij}$.

Each arc $\langle i, j \rangle \in A_T$ together with its partition of node set N into N_{ij} and N'_{ij} defines the two following cutsets of G,

$C^+(i, j) = \{\langle u, v \rangle \mid u \in N_{ij}$ and $v \in N'_{ij}\}$
and $C^-(i, j) = \{\langle u, v \rangle \mid u \in N'_{ij}$ and $v \in N_{ij}\}$
note that $\langle i, j \rangle \in C^+(i, j)$.

Each arc $\langle i, j \rangle \in A-A_T$ defines the particular cycle,

$k(i, j) = \{i, \langle i, j\rangle, P''_{ji}, i\}$

where P''_{ji} is the unique tree path connecting node j to node i in T_G.

Theorem 20: Let $T_G = (N_T, A_T)$ be a spanning tree of $G = (N, A)$ and suppose that $\langle u, v\rangle \in A_T$ and $\langle u', v'\rangle \in A-A_T$. Then $\langle u, v\rangle \in k(u', v')$ precisely when, $\langle u', v'\rangle \in C^{+}(u, v)$ or $\langle u', v'\rangle \in C^{-}(u, v)$.

Proof: consider $k(u', v') = \{u', \langle u', v'\rangle, P''_{v'u'}, u'\}$ and,

(i) let $\langle u', v'\rangle \in C^{+}(u, v)$,

then $\langle v, u\rangle \in P''_{v'u'}$

since $u' \in N_{uv}$ and $v' \in N'_{uv}$

thus $\langle u', v'\rangle \in C^{+}(u, v) \Leftrightarrow \langle v, u\rangle \in k(u', v')$

more precisely, $\langle u, v\rangle$ is counterdirected in $k(u', v')$.

(ii) let $\langle u', v'\rangle \in C^{-}(u, v)$

than $\langle u, v\rangle \in P''_{v'u'}$

since $u' \in N'_{uv}$ and $v' \in N_{uv}$

thus $\langle u', v' \rangle \in C^{-}(u, v) \Leftrightarrow \langle u, v \rangle \in k(u', v')$

more precisely, $\langle u, v \rangle$ in codirected in $k(u', v')$.

(iii) let $\langle u', v' \rangle \in C^{+}(u, v)$ and $\langle u', v' \rangle \in C^{-}\langle u, v \rangle$ then clearly $\langle u, v \rangle \in P''_{v', u'}$.

t

[u, v] retraction of G is a reduced network, G', obtained by identifying the two distinct nodes u and v of G and deleting any possible loops that result from this process.

ie. $G' = (N', A') \subset G$

where

$N' = N - \{u\}$

and

$A' = A - A^{c}$ where

$A^{c} = (FS(u)\ \Omega\ BS(v))\ U\ (FS(v)\ \Omega\ BS(u))$

(Ω means intersection)

Note that A^{c} is the set of those arcs in A which would become loop arcs upon identification of u with v. Node u is called a "deal-end" node of T_G

if it is incident with exactly one arc $\langle u, r\rangle$, furthermore arc $\langle u, r\rangle \in T_G$ is called a "dead-end" arc. Clearly, if in $[u, v]$ retraction of G, u is a dead-end node and $\langle u, v\rangle$ is the corresponding dead-end arc then $[u, v]$ retraction T_G' of T_G is again a tree. More precisely, it is the tree which results from T_G by deleting arc $\langle u, v\rangle$ and node u.

The retractions can be used for successively determining the cutsets $C^+(i, j)$ and $C^-(i, j)$ of the tree arc $\langle i, j\rangle \in A_T$, and in the case that $\langle i, j\rangle$ is a dead-end arc, then clearly these cutsets are the forward and backward star arc sets of node i,

ie. $C^+(i, j) = FS(i)$, $C^-(i, j) = BS(i)$.

For a directed network $G = (N, A)$, clearly, if parallel arcs are not allowed, then

$$FS(i)\ \Omega\ BS(j) = \begin{cases} (i, j) & \text{if } (i, j) \in A \\ \varnothing & \text{otherwise} \end{cases}$$

and also,

$$FS(j)\ \Omega\ BS(i) = \begin{cases} (j, i) & \text{if } (j, i) \in A \\ 0 & \text{otherwise} \end{cases}$$

thus,

$$A^{\circ} = \begin{cases} \{(i,\ j),\ (j,i)\} & \text{if} \quad \langle i,\ j \rangle \in A \\ \{(i,\ j)\} & \text{if} \quad (i,\ j) \in A,\ (j,\ i) \in A \\ \{(j,\ i)\} & \text{if} \quad (j,\ i) \in A,\ (i,\ j) \in A \\ \varnothing & \text{otherwise} \end{cases}$$

ie. $\subset^{+}(i,\ j) = FS(i),\ \subset^{-}(i,\ j) = BS(i).$

The following example clarifies the above definitions and theorem. Consider the network given in the following diagram together with its minimum spanning tree rooted at node 1.

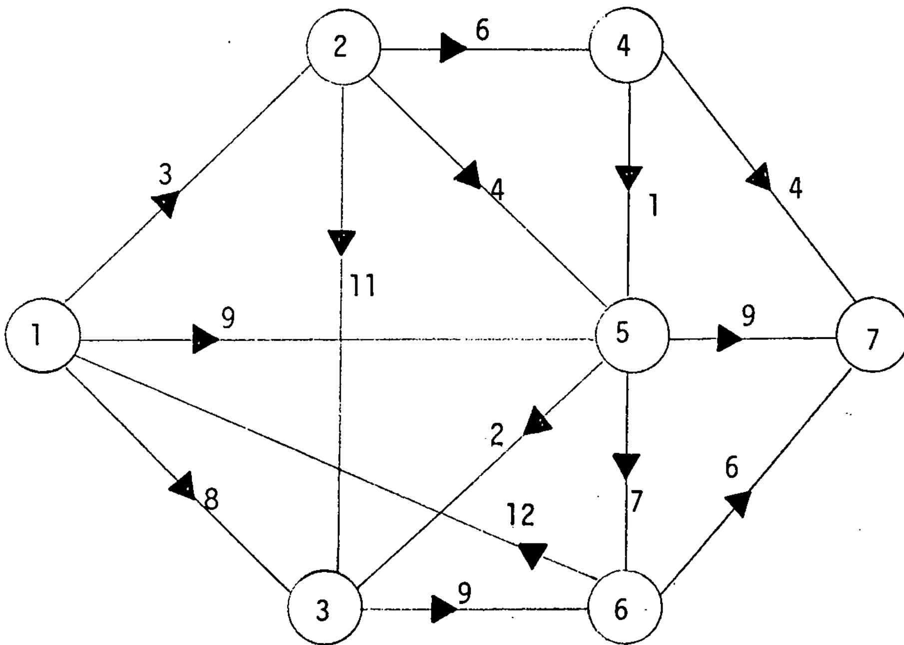
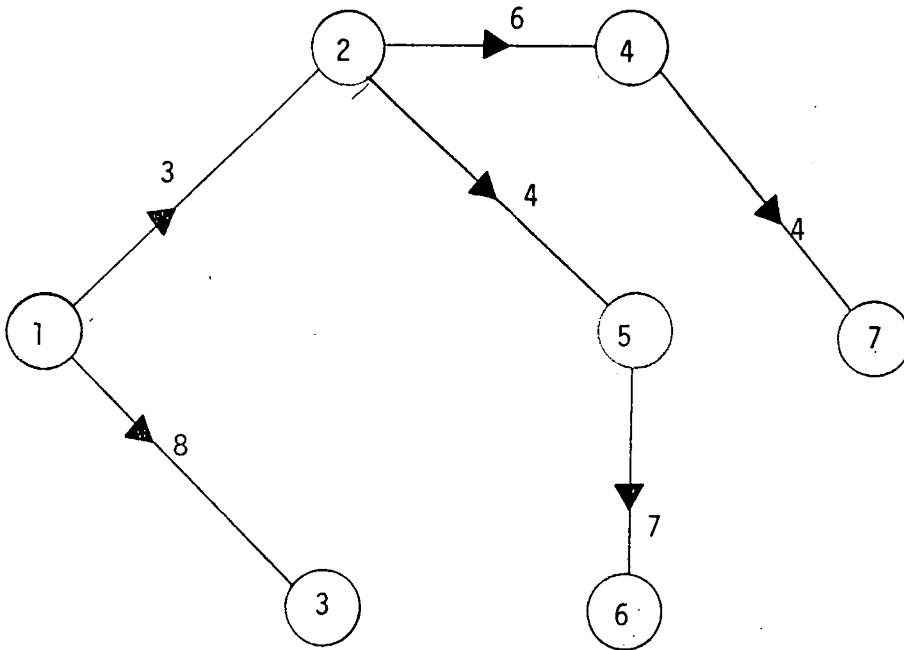


Figure 13: Example network, numbers associated with the arcs are the arc weights.



d	0	3	8	9	7	14	13
PN	1	1	1	2	2	5	4

Figure 14: The shortest path tree of the example network in figure 13.

In particular consider the arcs $(2, 5) \in A_T$ and $(2, 3)$, $(6, 1)$, $(1, 5) \in A-A_T$, then for

$(2, 5) \in A_T$: the two node sets are,

$N_{25} = \{1, 2\}$ and $N_{25}' = \{5, 6\}$;

the two cutsets are,

$c^+(2, 5) = \{(2,5)\}$ and $c^-(2, 5) = \{(6, 1)\}$;

$(2, 3) \in A-A_T$: $k(2, 3) = \{2, (2,3), 3, (1,3), 1, (1,2), 2\}$, then

$(2, 5) \in k(2, 3)$, since $3 \notin N_{25'}$;

$(6, 1) \in A-A_T$: $k(6, 1) = \{6, (6, 1), 1, (1, 2), 2, (2, 5), 5, (5, 6), 6\}$

$(2, 5) \in k(6, 1)$ and is codirected, since $6 \in N_{25}'$ and $1 \in N_{25}$.

$(1, 5) \in A-A_T$: $k(1, 5) = \{1, (1, 5), 5, (2, 5), 2, (1, 2), 1\}$

$(2, 5) \in k(1, 5)$ and is counterdirected, since $1 \in N_{25}$ and $5 \in N_{25}'$.

Now consider [2, 5] retraction of G.

$FS(2) = \{(2, 3), (2, 4), (2, 5)\}$

$BS(2) = \{(1, 2)\}$

$FS(5) = \{(5, 3), (5, 6), (5, 7)\}$

$BS(5) = \{(1, 5), (2, 5), (4, 5)\}$

$N' = \{1, 3, 4, 5, 6, 7\}$

$A^{c} = \{FS(2) \cap BS(5)\} \cup \{GS(5) \cap BS(2)\}$

$= \{(2, 5)\} \cup \emptyset = \{(2, 5)\}$

[Note that A^{c} is the set of loops caused by the retraction].

$A' = A - A^{c} = A - \{(2, 5)\}$

$G' = (N', A')$, the [2, 5] retraction of $G = (N, A)$ is shown in figure 15.

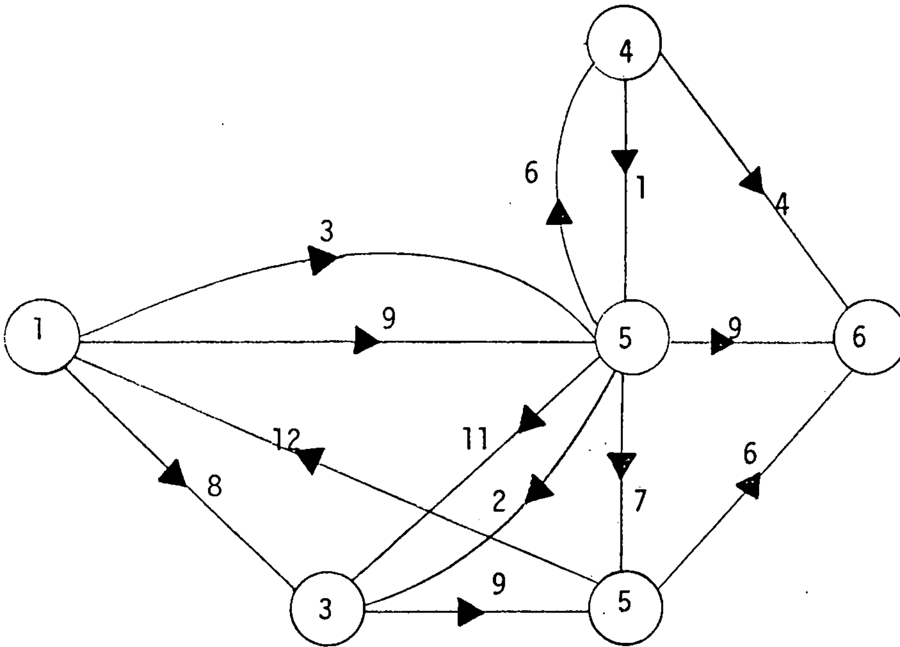


Figure 15: [2.5] retraction of the example network in figure 13.

The retraction has created parallel arcs, which are not allowed. Without loss of generality all parallel arcs except the one with the least weight from a node i to a node j in the resulting network are eliminated. In figure 16, the simplified [2, 5] retraction of the example network is shown.

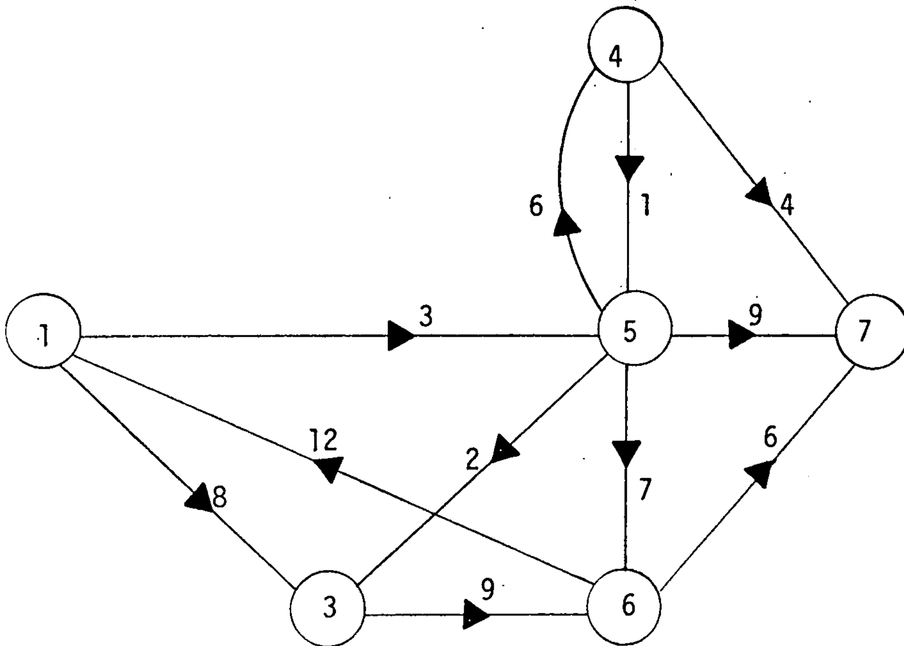


Figure 16: Simplified [2, 5] retraction of the example network in figure 13.

Note that there are always two parallel arcs and the one with the larger weight is eliminated. Consider the $[i, j]$ retraction of a network $G = (N, A)$, then for a node $u \neq i \neq j$ with (i, u) and $(j, u) \in A$, G' will contain (i, u) if $W_{iu} < W_{ju}$, or (j, u) otherwise. Similarly for a node $u \neq i \neq j$ with (v, i) and $(v, j) \in A$, G' will contain (v, i) if $W_{vi} < W_{vj}$, or (v, j) otherwise.

Now consider a network $G = (N, A)$ and its minimum spanning tree $T_G = (N_T, A_T)$ where $N_T = N$ and let

$\Delta(i, j) = W_{ij} + d(i) - d(j)$

clearly T_G is the minimum spanning tree of G if and only if $\Delta(i, j) \geq 0$, for all $(i, j) \in A$ in particular for every $(i, j) \in A-A_T$, since node j would have been labelled from node i, this is the well known optimality criterion.

Let the weight of an arc $(i, j) \in A$ changes by δ, from W_{ij} to $W_{ij} + \delta$, then the problem in this section is to determine $\delta^{+}(i, j) \geq 0$ and $\delta^{-}(i, j) \leq 0$, such that T_G remains optimal as W_{ij} varies by δ, where

$$\delta^{-}(i, j) \leq \delta \leq \delta^{+}(i, j).$$

Furthermore $W_{ij} + \delta^-(i, j)$ is called the lower limit of W_{ij} and $W_{ij} + \delta^+(i, j)$ is called the upper limit of W_{ij}.

Clearly, if (i, j) is a non optimal arc, ie. $(i, j) \in A - A_T$, then

$\delta^+(i, j) = \infty$

$\delta^-(i, j) = -\Delta(i, j)$.

However, in case of an optimal arc (i, j), ie. $(i, j) \in A_T$, the determination procedures of $\delta^+(i, j)$ and $\delta^-(i, j)$ are rather more complicated, and are based on the following theorem.

<u>Theorem 21</u>: let $(u, v) \in A_T$, then

$\delta^+(u, v) = \min\{\Delta(i, j) \mid (i, j) \in C^+(u, v), (i, j) \neq (u, v)\}$

and,

$\delta^-(u, v) = \max\{-\Delta(i, j) \mid (i, j) \in C^-(u, v)\}$.

<u>Proof</u>: If $(u, v) \in A_T$ and $W_{uv} \rightarrow W_{uv} + \delta$, then for a node $k \in N$ either $d(k) \rightarrow d(k)$ if $k \in N_{uv}$, or $d(k) \rightarrow d(k) + \delta$ if $k \in N'_{uv}$. The changes in $d(k)$, for $k = 1$ to n, affect the quantities

$\Delta(i, j)$ for $(i, j) \in A-A_T$, which enter the optimality criterion,

$$\Delta(i, j) \to \begin{cases} \Delta(i, j) & \text{if } (i, j) \in C^-(u, v) \cup C^+(u, v) \\ \Delta(i, j) + \delta & \text{if } (i, j) \in C^-(u, v) \\ \Delta(i, j) - \delta & \text{if } (i, j) \in C^+(u, v). \end{cases}$$

Clearly, $\delta^+(u, v)$ and $d^-(u, v)$ describe the range for δ such that $\Delta(i, j) \geq 0$ for all $(i, j) \in A-A_T$.

The algorithm cutset, stated below is a direct result of the above theorem and determines the lower and upper limits of an arc $(u, v) \in A_T$.

```
Algorithm cutset;
begin     {for the arc (u, v) ε A_T do}
   obtain N_uv and N_uv';
   obtain C⁻(u, v) and C⁺(u, v);
   for all (i, j) ∈ C⁺(u, v) do
     δ⁺(u, v) := minimum {Δ(i, j)};
     for all (i, j) ∈ C⁻(u, v) do
       δ⁻(u,v) := maximum {Δ (i,j)};
       upper := W_uv + δ⁺(u, v);
       lower := W_uv + δ⁻(u, v)
end;
```

In this algorithm N_{uv} and N_{uv}' are obtained by simply checking N_T and clearly this is done in $O(n)$;

$C^+(u, v)$ and $C^-(u, v)$ are obtained by checking $FS(i)$ and $BS(i)$ for every $i \in N_{uv} \cup N'_{uv}$. This procedure in worst case requires examining every arc $(i, j) \in A$ and hence runs in $O(m)$ time or $O(n^2)$ in case of a complete network.

Therefore the algorithm runs in $O(m)$ or $O(n^2)$ time and requires $O(m)$ or $O(n^2)$ additional space. If the lower and upper limits of every arc $(i, j) \in A_T$ is to be obtained then clearly the algorithm has to be repeated (n-1) time, thus resulting in $O(nm)$ or $O(n^3)$ time and $O(m)$ or $O(n^2)$ additional space [notice that the limits of every arc is determined independently of that of the other arcs]. Sheir and Witzgall, [SHWI 80], have proposed three algorithms for obtaining the cutsets. These algorithms are not more efficient than the cutset algorithm, if the aim is to obtain the cutsets of a particular arc $(u, v) \in A_T$, but if the cutsets of all the optimal or tree arcs are to be obtained, then these algorithms prevent the duplication of some of the calculations and hence are more efficient than repeating the algorithm cutset (n-1) times. All these algorithms run in $O(n^2)$ time, require $O(n^2)$ additional space and are based on the following theorem.

Theorem 22: Let $T_G = (N, A_T)$ be a shortest path tree of $G = (N, A)$, and suppose $(u, v) \in A_T$ is a dead-end arc. Let $G' = (N', A')$ and $T' = (N', A_T')$ arise from G and T by $[u, v]$ retraction, then T' is a shortest path tree of G' if for every $(i, j) \in A'$,

$$w_{ij}' \begin{cases} W_{ij} + W_{uv} & \text{if} \quad i = u \\ W_{ij} - W_{uv} & \text{if} \quad j = u \\ W_{ij} & \text{otherwise} \end{cases}$$

Furthermore, $\delta^+(i, j)' = \delta^+(i, j)$ and $\delta^-(i, j)' = \delta^-(i, j)$, for $(i, j) \in A'$.

Proof: Clearly $W_{uv}' = W_{uv}$ for $(u, v) \in T'$,

thus $d'(u) = d(u)$ for $u \in T'$

Now, if $(i, j) \in A' - A_T'$, then

$\Delta(i, j) = \Delta(i, j)$,

and hence, $\Delta'(i,j) \geqslant 0$ which establishes the optimality of the tree T', as well as the equality of lower and upper limits for non optimal arcs.

Also for $(u, v) \in A'$,

$\subset^+(u, v)' = \subset^+(u, v)$ and $\subset^-(u, v)' = \subset^-(u, v)$.

†

In this work, we will consider the implementation and analysis of one of these algorithms, called dead-end retraction, within a more complete algorithm, called sensitivity analysis, which calculates the lower and upper limits of every arc $(u, v) \in A$. The algorithm dead-end retraction, in our opinion, is the most efficient and the simplest to program, among the three algorithms proposed by Shier and Witzgall.

In dead-end retraction algorithm the cutsets of a dead-end arc $(u, v) \in A_T$ are first obtained and then the network is retracted using arc (u, v), and then the process is repeated to the resulting network and tree until all the optimal arcs are considered. This manner of consideration of the optimal arcs, clearly, makes the determination procedure of cutsets more efficient, since the determination of cutsets of a dead-end arc $(u, v) \in A_T$ only involves the examination of $FS(u)$ and $BS(u)$, and after determination of the cutsets of an arc the network is reduced by eliminating the trivial arcs. In algorithm sensitivity analysis, given below, it is assumed that the shortest path tree was obtained by using a label setting algorithm and the order in which

the nodes were labelled is recorded. Then a dead-end arc is obtained by considering the unique backward star optimal arc of the node which was labelled later than the other nodes in a network, the initial network or any retracted network, ie. if the nodes of a network are labelled in the order v_1, v_2, ..., v_n, then consider $(u, v_n) \in A_T$ first, and then after (u, v_n) retraction of the network consider $(u, v_{n-1}) \in A_T$, and so on. In our implementation, given below, a shortest path tree is represented by three node size lists, one called order, initially contains all the nodes of N in the order in which they were labelled in a label setting method, and ^PN and d, as defined before, are ordered accordingly. Furthermore, it is assumed that the network is represented by an adjacency matrix, mat, in order to eliminate the parallel arcs resulted after a retraction more efficiently. The following algorithm calculates the lower and upper limits of every arc $(u, v) \in A$, and uses the dead-end retraction method of Shier and Witzgall to determine these limits for the optimal arcs.

```
 1   Algorithm Sensitivity analysis;
 2   begin
 3      for i := 1 to n do
 4         for j := 1 to n do
 5            Δ(i, j) := ∞;
 6      for i := 1 to n do
 7         for j := 1 to n do
 8            if (W_order(i)j < ∞) then
 9            begin
10               if (^FN (order(i)) ≠ j) then
11               begin
12                  Δ(order(i), j) := W_order(i)j +
                    d(order(i) - d(j);
13                  upper := ∞;
14                  lower := W_order(i)j - Δ(order(i), j)
15               end
16            end;
17         nn := n;
18         min := + ∞;
19         max := - ∞;
20         while (nn > 1) do
21         begin
22            for i := 1 to (nn-1) do
23               if ((W_i order(nn) < ∞) and
                 (^FN(order(nn) ≠ i))) then
24               begin
25                  if (min > Δ (i, order(nn)) then
26                     min := Δ(i, order(nn))
27               end;
28            for i := 1 to (nn-1) do
29               if (W_order(nn)i < ∞) then
30               begin
31                  if (max < (Δ(order(nn), i) * (-1))) then
32                     max := Δ(order(nn), i) * (-1)
33               end;
34            upper := W_[^FN(order(nn))] order(nn) + min;
35            lower := W_[^FN(order(nn))] order(nn) + max;
36            for i := 1 to (nn-2) do
37               if (Δ(i, order(nn)) > Δ(i, order(nn-1)))
                 then
38               begin
39                  Δ(i, order(nn)) := Δ(i, order(nn-1));
40                  W_i order(nn) := W_i order(nn-1)
41               end;
42            for i := 1 to (nn-2) do
43               if (Δ(order(nn), i) > Δ(^FN(order(nn)),
                 i))) then
44               begin
45                  Δ(order(nn), i) := Δ(^FN(order(nn)), i);
46                  W_order(nn)i := W ^FN(order(nn))
47               end;
48            for i := 1 to n do
49            begin
```

```
50                Δ(FN(order(nn)), i) := ∞;
51                Δ(i, FN(order(nn))) := ∞;
52                W[FN(order(nn))]i := ∞;
53                Wi[FN(order(nn))] := ∞
54            end;
55            Worder(nn) order(nn) := ∞;
56            order (FN(order(nn))) := order(nn);
57            nn := nn-1
          end {while}
58    end;
```

In the above algorithm initially all Δ*'s are set to* ∞*, steps 3 to 5, and then for every arc* $(u, v) \in A-A_T$*,* $\Delta(u, v)$ *is calculated in steps 6 to 16. The lower and upper limits of every such arc is then obtained in steps 13 and 14. The variable nn indicates the number of nodes in a retracted network, and initially is set to n. In steps 20 to 58 the lower and upper limits of dead-end arcs in the reverse order of being labelled in a label setting algorithm, are calculated. In steps 22 to 27 the backward star arcs of a node u, the initial node of a dead-end arc, are considered and min or* $\delta^+(u, v)$ *is calculated. In steps 28 to 33 the forward star arcs of such a node are considered and max or* $\delta^-(u, v)$ *is calculated. Then in steps 34 to 35 the lower and upper limits of the dead-end arc* $(u, v) \in A_T$ *are obtained. In steps 36 to 57, the [u, v] retraction of the current shortest path tree is updated accordingly. Clearly this algorithm runs in* $O(n^2)$ *time, since*

the while loop is executed (n-1) times and every other loop in the while loop is executed at most n times, it also requires $O(n^2)$ additional space. The proof of correction of this algorithm is a direct result of the theorems 21 and 22.

Applying the sensitivity analysis algorithm to the example network of figure 13 and its shortest path tree in figure 14, gives the following results:

ARC			LIMITS	
Identity	Weight	Activity	lower	upper
1 ---> 5	9	NOP	7	INF
2 ---> 3	11	NOP	5	INF
2 ---> 4	6	OPT	3	9
2 ---> 5	4	OPT	3	6
3 ---> 6	9	NOP	6	INF
4 ---> 5	1	NOP	0	INF
4 ---> 7	4	OPT	0	7
5 ---> 3	2	NOP	1	INF
5 ---> 6	7	OPT	0	10
5 ---> 7	9	NOP	6	INF
6 ---> 7	6	NOP	0	INF

In the above results, the activity of an arc is OPT, if the arc is a tree arc, and is NOP, if the arc is a non-tree arc. The lower and upper limits of an arc, regardless of its activity or type, gives the range within which the weight of that arc can vary without affecting the optimal solution or changing the paths in the shortest path tree.

Another method for obtaining the cutsets which was also proposed by Shier and Witzgall is called cycle tracing algorithm and is based on an algorithm for transportation problems which was first proposed by Muller-Menback, [MULL, 68]. This algorithm is based on theorem 20. In this algorithm for each non-optimal arc $(u, v) \in A-A_T$, *which contain* (u, v) *are obtained. Then the quantity,* $\Delta(u, v) = W_{uv} + d(u) - d(v)$ *is entered into the optimisation process for calculating* $\delta^{+}(i, j)$ *and* $\delta^{-}(i, j)$, *as shown in theorem 21, for updating tentative minima and maxima which are initially set to* $+\infty$ *and* $-\infty$ *respectively. To obtain all the cutsets which contain* $(u, v) \in A-A_T$, $k(u, v)$ *is first obtained, as described before, and then theorem 20 is used.*

Finally, the third algorithm proposed by Shier and Witzgall is called the tree building. In this algorithm the quantities $\Delta(u, v)$ are calculated in the process of building the shortest spanning tree. This algorithm seems to be the most complicated and is definitely the most inefficient one among the them.

15 POSTOPTIMALITY ANALYSIS

All the labelling algorithms, in fact all known solution methods for one-to-all problems, are applicable to networks with known constant arc weights. The algorithms described in section 14, for sensitivity analysis of shortest path problems give a range within which the weight of a specific arc can vary without affecting the shortest path tree. However, what these algorithms fail to show is the effect on the shortest path tree if an arc weight falls outside of its given range.

Spira and Pan, [SPPA 78], have shown that to update a shortest path tree after a constant increase or decrease in the weight of an existant arc takes $O(n^2)$ time. It may be as efficient, in case of a non-negative dense network at least, to modify the network, ie. setting the weight of the varied arc to its new value, and resolve the problem by a label setting algorithm which will take $O(n^2)$ time. In this section we present an $O(n^2)$ algorithm, Senet, which post optimises the one-to-all problems on non-negative networks whose arc weights are subject to variation. More precisely, algorithm Senet determines all the non-

negative critical values (at each of which the shortest path tree changes further) for the weight of a varying arc. Furthermore, Senet also reports the updated shortest path tree for every range formed by two successive critical values of the varying arc weight. Senet is applicable to the optimal, non-optimal and non-existant arcs and analysis the variations in the arc weights independently.

Let us extend the network terminology, before introducing Senet.

By an optimal solution or simply a solution to a network, we mean a shortest path tree of the network rooted at a distinguished node (source).

Let R_i be the set of all the paths from source to node i, where no arc is traversed more than once in each path. Let P_{ki} be the path number k to node i with a total weight of d_{ki} (ie. $R_i = \{P_{1i}, P_{2i}, \ldots, P_{ki}, \ldots\}$). Now P_{mi} is the optimum of R_i if and only if $d_{mi} = \min \{d_{ji} \mid P_{ji} \in R_i\}$. Node i is said to be labelled if the shortest path from the source to i

is determined. Then the label of node i consists of two parts:

(i) *a node which is immediately before i on the path from source to i, $^{F}N(i)$;*

(ii) *an integer representing the total weight of the path $d(i)$.*

Node i is said to be totally relabelled if the ordered set of nodes in its path from the source is changed.

Node i is disconnected if there exists no path from the source to i. A network is disconnected if it contains at least one disconnected node.

Assume that there exists an optimal solution, solution one, to a given network G. Then the set of arcs A can be divided into two parts,
$A = A_1 + A_2$, where A_1 is the set of optimal arcs (ie. those utilized by the original solution) and A_2 is the complement of A_1, the set of non-existant arcs, A_3, is also considered, where

$$A_3 = \{(i, j) : i, j \in N, (i, j) \notin A\}$$

Now suppose that the effect on the optimal solution caused by variation in the arc (p, q) is to be analysed, $(p, q \in N)$, then the solution can be analysed by considering A_1 and $(A_2 + A_3)$ separately. In the following cases w'_{pq} represents the original weight of (p, q).

(i) <u>Optimal Case</u>

Set w_{pq} to infinity and solve the resulting network G' (ie. find the shortest path from source to each of the other nodes).

If there exists no optimal solution to G', then (p, q) is optimal for all values of w_{pq}. otherwise the solution found becomes solution

two. Solution two would contain a set of nodes which are either totally relabelled or disconnected. These are the nodes whose shortest paths in solution one contained (p, q). Arc (p, q) is always optimal for disconnected nodes. Let N' be the set of totally relabelled nodes and suppose that $K = |N'|$, $(1 \leqslant k \leqslant n)$. For each totally relabelled node, N'_i, obtain the quantity $\Delta(N'(i))$, where

$$\Delta(N'(i)) = w'_{pq} + d2(N'(i)) - d1(N'(i)),$$

for $i = 1 \ldots\ldots k$.

where, $d1(N'(i))$ is the total weight of the shortest path to node $N'(i)$ in solution one and $d2(N'(i))$ is that of node $N'(i)$ in solution two.

Now set w_{pq} to zero and solve the resulting network, G'', obtaining solution three. Suppose that Γ nodes are totally relabelled, excluding the nodes whose total weights are changed only, then for each of these nodes, $N'(k+i)$, calculate $\Delta(N'(k+i))$,

for $i = 1 \ldots\ldots \Gamma$, where

$$\Delta(N'(k+i)) = d1(N'(k+i)) - d3(N'(k+i)),$$

for $i = 1 \ldots \Gamma$.

where, $d3(N'(k+1))$ is the total weight of the shortest path to node $N'(k+1)$ in solution three.

Now rearrange Δ and N', for $i = 1 \ldots k+\Gamma$, in descending order of Δ. In this order, the first k elements of Δ and N' are the ones obtained by solving G' and the rest are those obtained by solving G''. We also have, $\Delta(N'(k+1)) \leqslant w'_{pq} \leqslant \Delta(N'(k))$, optimality range. Now the following conclusions about the values of w_{pq} can be drawn.

OPTIMAL ARCS	
Range	Change in original solution (path = shortest path)
Non-Optimality $\Delta(1) \leq W_{pq}$	Solution two becomes optimal to G
Increase $\Delta(m) \leq W_{pq} \leq \Delta(1)$ for $m \leq k$	(i) The paths to N'(i), for i = m ... k, are as in solution two (ii) Total weights of all the paths to N'(i) for i = 1 ... m, increase by the same amount as w_{pq} increases (iii) The paths to the rest of the nodes are as in solution one
Optimality $\Delta(k+1) \leq w_{pq} \leq \Delta(k)$	No change
Decrease $\Delta(k+1+m) \leq w_{pq} \leq \Delta(k+1)$ $m \leq \Gamma$	(i) Total weight of every path containing (p, q), decreases by $(w'_{pq} - w_{pq})$ (ii) The paths to N'(i), for i = 1 ... m, are as in solution three (iii) The paths to the rest of the nodes are as in solution one
Alternative optimal solutions exist for those values of w_{pq}, which fall on the boundary of a range	

(ii) *Non-Optimal and Non-Existant Case*

Set w_{pq} to zero and solve the resulting network, G'. Let solution two be the optimal one obtained for G'.

Let N' be the set of totally relabelled nodes and suppose that $k = |N'|$, $(0 \leq k \leq n)$. For each totally relabelled node $N'(i)$, $(i = 1 \ldots k)$, calculate $\Delta(N'(i))$ where

$\Delta(N'(i)) = d1(N'(i)) - d3(N'(i))$.

Now rearranging Δ and N' in descending order of Δ, the following conclusions about the values of w_{pq} can be drawn

NON-OPTIMAL AND NON-EXISTANT ARCS	
Range	Change in original solution (path = shortest path)
Non-optimality $\Delta(1) \leqslant w_{pq}$	No change
General $\Delta(m) \leqslant w_{pq} \leqslant \Delta(1)$ for $m \leqslant k$	(i) The paths to N'(i), for i = 1 ... m, are as in solution two (ii) The paths to the rest of the nodes are as in solution one
Final $0 \leqslant w_{pq} \leqslant \Delta(k)$	Solution two becomes optimal for G
Alternative optimal solutions exist for those values of w_{pq}, which fall on the boundary of a range	

Supposing that arc (p, q) in a network G with an optimal solution ($^{P}N1$, d1), ie. $^{P}N1$ contains the predecessor nodes and d1 the shortest path weights, is to be analysed, then Senet can be structured in the following manner.

```
 1   algorithm Senet;
 2   begin
 3      K := 0;
 4      kk := 1;
 5      get (p, q);
 6      act := activity (P, q);
 7      W'pq := Wpq;
 8      if (act = OPT) then
 9      begin
10         data (P, q) := ∞;
11         shortest-path (data, d2, PN2);
12         compare (PN2, N', k);
13         for i := 1 to k do
14         Δ(i) := W'pq + d2(N'(i)) - d1(N'(i));
15         kk := k
16     end;  {if}
17     data (P, q) := 0;
18     shortest-path (data, d3, PN3)
19     compare (PN3, N', k);
20     if (k > 0) then
21     begin
22        for i := kk to k do
23        Δ(i) := d1(N'(i)) - d3(N'(i));
24        descend (N', Δ, k)
25     end
26   end.
```

In the above implementation, analysis of an optimal arc requires the execution of all 26 steps and analysis of a non-optimal or a non-existant arc require the execution of the steps from 1 to 7 and from 17 to 26, inclusive. The function activity determines the type of the arc (p, q) which may be:

OPT = optimal,

NOP = non-optimal,

NEX = non-existant.

This function can be implemented as follows:

```
function activity (P, q);
begin
   if (W_pq = ∞)
   then
      activity := NEX
   else
      if (^pN1(q) = P)
      then
         activity := OPT
      else
         activity := NOP
end;
```

Procedure shortest-path is a label setting algorithm which solves a one-to-all problem in a network represented in data. Procedure descend rearranges N' and Δ in descending order of Δ.

Procedure compare, obtains the totally relabelled nodes after a change in data and stores them in N', a node-size linear list. This procedure is used twice if arc (p, q) is optimal and once otherwise. Here we give two different implementations of this procedure. In each implementation a node-size linear list of boolean type, L, is used to prevent a node entering N' more than once. In the first implementation we have used a queue with FIFO management, Q, to identify the totally relabelled nodes.

```
 1   (1) procedure compare (PN, N', k);
 2   begin
 3      for i := 1 to n do
 4         L(i) := false;
 5      for i := 1 to n do
 6      if (PN(i) ≠ PN1(i)) then
 7      begin
 8         L(i) := true;
 9         ADDQ(i, Q)
10      end;
11      while not (EMPTYQ (Q)) do
12      begin
13         u := FRONT(Q);
14         DELETEQ(Q);
15         K := k+1;
16         N'(k) := i;
17         for i := 1 to n do
18         if ((U = PN1(i)) and (L(i) = false)) then
19         begin
20            ADDQ(i, Q);
21            L(i) := true
22         end
23   end;
```

In the second implementation we have used N', a node-size linear list, to directly identify and store the totally relabelled nodes. Associated with N' there are two pointers, one (K') indicates the location of the next totally relabelled node in N' which is to be searched and the other, K, indicates the location in N' for inserting a new totally relabelled node. N' is in a way treated like a queue with FIFO management, except that no deletion takes place.

```
 1    (2) Procedure compare (^PN, N', k);
 2    begin
 3       for i := 1 to n do
 4         L(i) := false;
 5         k' := 0;
 6         k := 0;
 7         for i := 1 to n do
 8         if (^PN(i) ≠ ^PN1(i)) then
 9         begin
10            L(i) := true;
11            k := k+1;
12            N'(k) := i
13         end;
14         repeat
15         if (k > 0) then
16         begin
17            k' := k'+1;
18            for i := 1 to n do
19            if ((N'(k') = ^PN1(i)) and (L(i) = false)) then
20            begin
21               L(i) := true;
22               K := k+1;
23               N'(k) := i
24            end
25         end;
26         until (k'=k)
27    end;
```

Clearly, both implementations run in $O(n^2)$ time, however, the second one is more space efficient. In both implementations $^{P}N1$ represents the predecessor node set of solution one, and ^{P}N represents that of a new solution, either solution two or solution three.

Theorem 23: Senet determines all the critical values for the weight of an arc and reports the correct effects on the optimal solution at each critical value. Furthermore, $|N'| = k \leq n$.

Proof: Consider R_i, as shown in theorem 3, this set is finite and has a size of k_i, where

$$\max(k_i) = (n-2)! \sum_{r=0}^{n-2} 1/(n-2-r)! \qquad \text{for } n \geq 2,$$

R_i can be divided into two parts. $R_i = R'_i + R''_i$, where R'_i is the set of paths containing a particular possible connection (ie. (p, q), where $p, q \in N$) and R''_i is its complement. Now let w'_{pq} be the original weight of (p, q) in G, and also P_{1i} and P_{2i} be the optimums of R'_i and R''_i respectively then,

(i) if w_{pq} is set to infinity, then
P_{2i} = optimum (R_i)

(ii) if w_{pq} is set to zero and

(a) P_{1i} = optimum (R_i), then
$d_{1i} \leq d_{2i} - w'_{pq}$

(b) P_{2i} = optimum (R_i), then
$d_{2i} \leq d_{1i} - w'_{pq}$

for $i = 1 \ldots n$

The optimal case and the non-optimal case, which includes the non-existant case, are considered separately:

(i) Optimal Case

Let P_{1i} and P_{2i} be the shortest paths to a node i in G and G' respectively, where G' is determined from G by letting $w_{pq} \to \infty$. The following are now true.

(1.1) if P_{2i} does not exist for some i, then P_{1i} is optimal for all values of w_{pq}.

(1.2) if P_{1i} does not include (p, q), then $P_{1i} = P_{2i}$.

(1.3) if $P_{2i} \neq P_{1i}$, then $d_{1i} \leq d_{2i}$, now let $d'_i = d_{2i} - d_{1i}$, then for a general value of w'_{pq} in G we have

(1.3.1) P_{1i} is optimal if $w_{pq} < w'_{pq} + d'_i$

(1.3.2) P_{2i} is optimal if $w_{pq} > w'_{pq} + d'_i$

(1.3.3) P_{1i} and P_{2i} are, alternative optimal paths (ie. $d_{1i} = d_{2i}$) if $w_{pq} = w'_{pq} + d'_i$.

Therefore, (p, q) is in the optimal path to i if $w_{pq} \leqslant w'_{pq} + d'_i$ and clearly, this is true for every $i \in N$.

Now let P_{2i} be the optimal path to node i in G'', where $G'' = G$, but $w_{pq} = 0$, then the following are true,

(2.1) if P_{1i} exist, then P_{2i} exists.

(2.2) if P_{2i} does not include (p, q), then $P_{2i} = P_{1i}$.

(2.3) if P_{2i} includes (p, q) and,

(2.3.1) P_{1i} includes (p, q), then

$$d_{2i} = d_{1i} - w'_{pq}$$

(2.3.2) P_{1i} does not include (p, q) then $d_{2i} \leqslant d_{1i}$.

Now let $d'_i = d_{1i} - d_{2i}$, $d'_i \geqslant 0$, then

(a) P_{2i} is optimal if

$$0 \leqslant w_{pq} \leqslant d'_i$$

(b) P_{1i} is optimal if

$$d'_i \leqslant w_{pq} \leqslant w'_{pq}$$

Therefore (p, q) is in the optimal path to i if $0 \leqslant w_{pq} \leqslant d'_i$ and this is true for every $i \in N$. However (1.3.1) and (2.3.2.b) above, together imply that for the range $d'_i \leqslant w_{pq} \leqslant w'_{pq} + d'_i$,

the original path, P_{1i}, is optimal. Now assume that the shortest paths to k nodes in solution one include (p, q), where ($k \leq n$). Then clearly as a result of (1.2) above, only k nodes are totally relabelled in solution two. In solution three the set of nodes N can be divided into three parts:

(a) *the set of nodes whose labels are unchanged*

(b) *the set of nodes whose labels are totally changed*

(c) *the set of nodes whose total weights are decreased only.*

Now let Γ', Γ, and Γ'' be the sizes of the above three subsets of N respectively, then

(1) $\Gamma + \Gamma' + \Gamma'' = n$

(2) $\Gamma'' = k$, *as a result of (2.2) and (2.3) above.*

Therefore $k + \Gamma \leq n$ as $\Gamma' \geq 0$, (ie. maximum number of relabelled nodes, when analysing an optimal arc is n), ie. $|N'| = k \leq n$.

(ii) Non-Optimal and Non-Existant Case

In this case let P_{1i} and P_{2i} be the optimal paths to i in G and G' respectively, where $G' = G$, but $w_{pq} = 0$. Then the following are true,

(a) if P_{1i} exists, then P_{2i} exists

(b) $d_{2i} \leq d_{1i}$, for all $i \in N$

(c) if $d_{1i} = d_{2i}$ then $P_{1i} = P_{2i}$

(d) if $d_{2i} < d_{1i}$, then P_{2i} includes (p, q) and if $d'_i = d_{1i} - d_{2i}$, then for a general value of $w_{pq} \geq 0$,

we have:

(d.1) P_{2i} is optimal for $0 \leq w_{pq} \leq w'_{pq} - d'_i$

(d.2) P_{1i} is optimal for $w_{pq} \geq w'_{pq} - d'_i$

Therefore, (p, q) is in the optimal path to i if $0 \leq w_{pq} \leq w'_{pq} - d'_i$, and clearly this is true for every $i \in N$. Furthermore, it is clear that $|N'| = k \leq n$. †

Theorem 24: Senet terminates in $O(n^2)$ time and requires $O(n)$ additional memory space.

Proof: The termination of the algorithm depends on the number of critical values for the weight of an arc. The set of critical values of the weight of an arc in a network of size n is finite and has a maximum size of n, since:

At each successive critical value at least one more node becomes totally relabelled, and a node is totally relabelled at most once in the process of analysing an arc. Furthermore, if no node is totally relabelled, then the algorithm terminates.

The proof that Senet terminates in $O(n^2)$ time in worst case is by inspection. A label setting algorithm runs in $O(n^2)$ time, procedure compare runs in $O(n^2)$ time and rearranging the totally relabelled nodes in procedure descend takes $O(n^2)$ time. Therefore, Senet runs in $O(n^2)$.

In case of analysing an optimal arc, there are seven additional node-size linear lists, four to represent solutions two and three, two for N' and Δ and one, L, for identification of totally relabelled nodes. In case of analysing a non-optimal or a non-existant arc, there are five additional node size linear lists, all similar to

the case of analysing an optimal arc with the exception that only two such lists are required to represent one new solution only. Therefore the maximum number of additional memory units required for analysing an arc is 7n. †

To compare Senet with the algorithms of chapter 14, consider the example network of figure 13 and its solution in figure 14. Furthermore, suppose that arcs (2, 5), (1, 5), (2, 6) and (3, 2) are to be analysed, where arcs (2, 6) and (3, 2) are non-existant. Analysing the arcs separately:

(i) *arc (2, 5) is optimal,*

act = OPT;

$W'_{25} = 4;$

$W_{25} \leftarrow \infty;$

solution 2:

d2	0	3	8	9	9	16	13
$^{P}N2$	1	1	1	2	1	5	4

totally relabelled nodes:

N'	5	6

Δ	6	6

$W_{25} \leftarrow 0$;

solution 3:

d3	0	3	5	9	3	10	12
ᴾN3	1	1	5	2	2	5	5

N'	5	6	3	7

Δ	6	6	3	1

rearranging:

N'	5	6	3	7
Δ	6	6	3	1

(ii) *arc (1, 5) is non-optimal,*

act := NOP;

$W_{15}' := 9;$

$W_{25} \leftarrow 0;$

solution 2:

d3	0	3	2	9	0	7	13
N3	1	1	5	2	1	5	4

N'	3	5

Δ	6	7

rearranging:

N'	5	3
Δ	7	6

(iii) *arc (2, 6) is non-existant;*

act := NEX;

$W_{26}' := \infty;$

$W_{26} \leftarrow 0;$

solution 2:

d3	0	3	8	9	7	3	9
PN3	1	1	1	2	2	2	6

N'	6	7

Δ	11	4

rearranging:

N'	6	7
Δ	11	4

(iv) *arc (3, 2) is non-existant,*

act := NEX;

$W_{32}' := \infty;$

$W_{32}' \leftarrow 0;$

solution 2:

d3	0	3	8	9	7	14	13
PN3	1	1	1	2	2	5	4

N'	0

Now the following conclusions about the arc weights can be made.

***** POST-OPTIMALITY ANALYSIS

++ THE "EFFECT" OF EACH RANGE, EXCEPT THE OPTIMAL AND NON-OPTIMAL, IS AN ACCUMULATION OF THE "EFFECTS" OF THE OTHER RANGES FROM THE SIGN "+" OR "+" TO "-" OF THE "ACCUMULATION" COLUMN, FOR EACH ARC ++

ARC			RANGE			EFFECT		
identity	weight	activity	upper	lower	accumulation	node	t-weight	route <--------
2---> 5	4	OPT	INF	6	"-"	5	9	5 1
					"+"	6	16	6 5 1
			6	3		****	OPTIMAL-RANGE	
			3	1	"+"	3	5+w(2, 5)	3 5 2 1
			1	0	"-"	7	12+w(2, 5)	7 5 2 1
1---> 5	9	NOP	INF	7		****	NON-OPTIMAL RANGE	
			7	6	"+"	5	0+w(1, 5)	5 1
						6	7+w(1, 5)	6 5 1
			6	0	"-"	3	2+w(1, 5)	3 5 1
2---> 6	INF	NEX	INF	11			NON-OPTIMAL RANGE	
			11	4	"+"	6	3+w(2, 6)	6 2 1
			4	0	"-"	7	9+w(2, 6)	7 6 2 1
3---> 2	INF	NEX	***	***		****	NON-EFFECTIVE	

In the above output:

The weight of the optimal arc (2, 5) can vary from 6 to infinity and this will change the routes to nodes 5 and 6 in solution one to (1 → 5) and (1 → 5 → 6) with total weights of 9 and 16, respectively, only. The weight of this arc can vary from 3 to 6 without affecting the structure of the shortest path tree of solution one. If this weight varies between 1 and 3, then the route to node 3 will change to (1 → 2 → 5 → 3) with a total weight of (5 + W_{25}*). If it varies between 0 and 1, then beside the change in the route to node 3, the route to node 7 will change to (1 → 2 → 5 → 7) with a total weight of (12* W_{25}*);*

The weight of the non-optimal arc (1, 5) can vary from 7 to infinity without effecting the optimal solution (ie. solution one). If it varies from 6 to 7 the routes to nodes 5 and 6 will change to (1 → 5) and (1 → 5 → 6) with total weights of (0 + W_{15}*) and (7 +* W_{15}*) respectively. If it varies between 0 and 6 however, beside the changes in the routes to the nodes 5 and 6 the route to node 3 will also change to (1 → 5 → 3) with a total weight of (2 +* W_{15}*);*

If the non-existant arc (2, 6) is to be created and its weight is between 11 and infinity, then the optimal solution will not be effected. However if it has a weight between 4 and 11, then it will become an optimal arc and will change the route to node 6 to (1 → 2 → 6) with a total weight of (3 + W_{26}), and if it has a weight between 0 and 4, then the route to node 7 will also be changed to (1 → 2 → 6 → 7) with a total weight of (9 + W_{26});

The creation of the arc (3, 2) with a total weight between 0 to infinity will not effect the optimal solution.

The complete pascal code of the algorithm Senet together with a sample run is given in appendix F.

The algorithms of section 14, for sensitivity analysis, determine only two of the critical values, maximum increase and decrease, within which the weight of a given arc can vary, independently, without changing the structure of the shortest path tree. Furthermore, they do not report the updated weights of the shortest path tree within the given range and do not indicate the structural changes of the shortest path tree when an arc weight falls outside of its determined range. Senet provides all the critical values for the weight of an arc together with the updated weights of the shortest paths and the structural changes between every two successive critical values. This is because, in analysing a non-optimal arc, sensitivity analysis algorithms only consider the affect on the terminal node of the arc, when the weight of the arc is reduced. This node is obviously the very first one which may be affected as a result of the reduction. Senet, however considers every other node which could be affected after the terminal node of the arc is affected. In case of analysing an optimal arc, sensitivity analysis algorithms consider all the nodes that Senet considers, but they do no use all the information that they obtain. Sensitivity

analysis algorithms do not consider the non-existant arcs, although with a simple modification, some of them could become capable of analysing such arcs. Analysis of an arc for sensitivity or post-optimality takes $O(n^2)$ time, however, in Senet the additional memory space required is $O(n)$ and in the sensitivity analysis algorithms is (n^2). Some of the sensitivity analysis algorithms analyse all the m arcs in $O(n^2)$ time and Senet analyses them in $O(n^2m)$ time, but still Senet will require $O(n)$ additional memory space.

Senet can be modified to analyse negative networks as well as non-negative networks. In case of negative networks which do not contain negative cycles, the lowest critical value for an arc (u, v) will be t rather than zero, where t is the minimum weight of a cycle containing arc (u, v). thus the modified version must be capable of determining such cycles.

PART V

SUMMARY, CONCLUSION AND REFERENCES

16 SUMMARY AND CONCLUSION

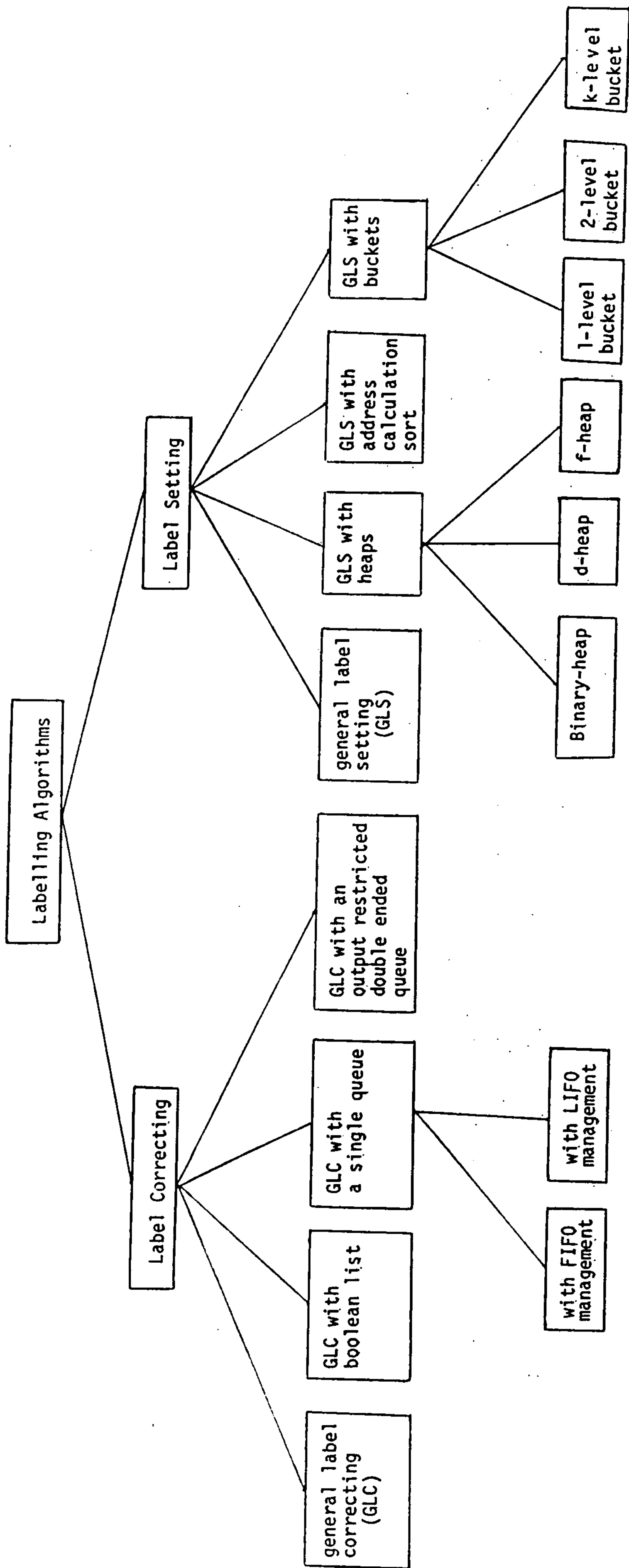
Section 1, in a way, could be considered as a summary, futhermore at the end of each section the corresponding conclusions are drawn. However, in this section we present a brief summary coupled with an outline of the conclusions made throughout the work.

In section 6, we classified the deterministic unconstrained shortest path problems in order to outline the importance of one-to-all problems.

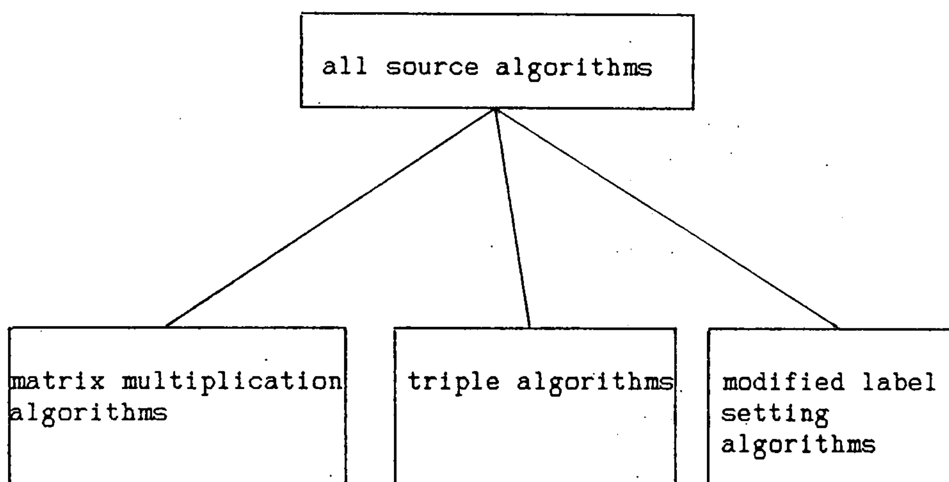
In section 7, we developed an algorithm, labelling, which is the underlying structure of all the labelling algorithms. We then used this algorithm and its properties, directly or indirectly, to study, classify, analyse and compare the different labelling algorithms.

In sections 8 and 9 we considered all different implementations of labelling algorithms using various data structures and sorting techniques, and analysed and compared most of such implementations. All the analysed algorithms in these two sections were evaluated by using worst

case analysis and their memory space requirements. In section 10, the most efficient labelling algorithms were compared using their average computation times on a set of diverse randomly generated networks. Results of the classifications of the labelling algorithms can be generalised as follows:


Labelling Algorithms
Label Correcting
Label Setting
general label correcting (GLC)
GLC with boolean list
GLC with a single queue
GLC with an output restricted double ended queue
with FIFO management
with LIFO management
general label setting (GLS)
GLS with heaps
GLS with address calculation sort
GLS with buckets
Binary-heap
d-heap
f-heap
1-level bucket
2-level bucket
k-level bucket

In section 11 to 13, the all source algorithms were reviewed, classified and compared. The classification of these algorithms can be generalised as follows:



In section 14, the sensitivity analysis of one-to-all problems was considered and the best of such methods was implemented and analysed.

In section 15, we introduced an algorithm, SENET, for the post optimality analysis of one-to-all problems. In this section we also considered the advantages of this new approach to such problems over the existing sensitivity analysis, probably the closest class of algorithms to SENET.

All the theory behind the shortest path problems, one-to-all in particular, were developed throughout the work in terms of definitions, algorithms and theorems. However, the emphasis in this work is on sections 6 to 10 and in particular on section 15.

17 REFERENCES

[AHHU 74] Aho, Hopcroft and Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, MA, 1974.

[BELL 58] Bellman, "On a Routing Problem", Quart. Appl. Math., Vol 16, 1958.

[DANT 60], Dantzig, "On The Shortest Route Through a Network", Management Sci, Vol 6, 1960.

[DANT 67] Dantzig, "All Shortest Routes in a Graph", in Theory of Graphs, P Rosenstiehl, Ed. Gordon and Breach, New York, 1967.

[DEFO 79a] Denardo and Fox, "Shortest-Route Methods: 1 Reaching, Pruning and Buckets", Operational Research, Vol 27, 1979.

[DEFO 79b] Denardo and Fox, "Shortest-Route Methods: 2 Group Knapsacks, Expanded Networks and Branch-and-Bound", Operational Research, Vol 27, 1979.

[DERY 69] Deryfus, "An Appraisal of some Shortest-Path Algorithms", Operations Research, Vol 17, 1969.

[DGKK 79] Dial, Glover, Karney and Klingman, "A Computational Analysis of Alternative Algorithms and Labelling Techniques for Finding Shortest Path Tree", Networks 9, 1979.

[DIAL 65] Dial, "Algorithm 360 Shortest Path Forest with Topological Ordering", Communications of the ACM, Vol 12, 1965.

[DIAL 69] Dial, "Algorithm 360 Shortest Path Forest with Toplogical Ordering", Comm. ACM, Vol 12, 1969.

[DIJK 59] Dijkstra, "A Note on Two Problems in Connexion with Graphs", Numer Math, Vol 1, 1959.

[EDKA 72] Edmonds and Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", J of ACM, Vol 19, 1972.

[FALM 67] Farbey, Land and Murchland, "The Cascade Algorithm for Finding All Shortest Distances in a Directed Graph", Management Sci, Vol 14, 1967.

[FLOY 62] Floyd, "Algorithm 97: Shortest Path", Comm. ACM, Vol 5, 1962.

[FORD 56] Ford, Jr, "Network Flow Theory", Report p-923, Rand Corp, Santa Monica, CA, 1956.

[FOXB 78] Fox, "Data Structures and Computer Science Techniques in Operations Research", Operations Research, Vol 26, No 5, 1978.

[FRET 85] Fredman and Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimisation Algorithms", to be published in IEEE, 1985 onwards.

[GIWI 73] Gilsinn and Witzgall, "A Performance Comparison of Labelling Algorithms for Calculating Shortest Path Trees", NBS Technical Note 772, US Department of Commerce, 1973.

[GLKL 82] Glover F, Klingman D, "A Computer Study of Efficient Algorithms for Shortest Path Trees and Assignment Problems", University of Texas, Austin, 1982.

[HITC 68] Hitchner L E, "A Comparative Investigation of the Computational Efficiency of Shortest Path Algorithms", Report ORC 68-25, Operations Research Centre, University of California, Berkeley, 1968.

[HUTC 67] Hu, Tc, "Revised Matrix Algorithms for Shortest Paths", Siam J. Appl. Math., Vol 15, 1967.

[IMAI 84] Imai and Iri, "Practical Efficiencies of Existing Shortest-Path Algorithms and a New Bucket Algorithm", Journal of the Operations Research Society of Japan, Vol 27, No 1, 1984.

[JOHN 72] Johnson E L, "On Shortest Path and Sorting", in Proceedings, 25th Conference of the Association for Computing Machinary", Boston, 1972.

[JOHN 77] Johnson D B, "Efficient Algorithms for Shortest Path in Sparse Networks", J Assoc. Comput. Mach, Vol 24, 1977.

[KELA 78] Kelton W D, Law A M, "A Mean Time Comparison of Algorithms for the All Pairs Shortest Path Problem with Arbitary Arc Length", Networks, Vol 8, 1978.

[KNUT 73a] Knuth D, "The Art of Computer Programming, Vol 1: Fundamental Algorithms", Addison-Wesley, Reading, Mass, 1973.

[KNUTH 73b] Knuth D, "The Art of Computer Programming, Vol 3: Sorting and Searching", Addison-Wesley, Reading, Mass, 1973.

[MAGO 78] Magnanti T L, and Golden B L, "Transportation Planning: Network Models and Their Implementation", Technical Report 143, Operations Research Centre, Mass. Institute of Technology, Cambridge, MA, 1978.

[MINT 57] Minty, "A Comment on Shortest Route Problems", Operations Research, Vol 5, 1957.

[MOOR 59] Moore E, "The Shortest Path Through a Maze", Proceeding of the International Symposium on the Theory of Switching, 1959.

[MULL 68] Muller-Merbach, "Sensibilitatsanalyse Von Transportproblemen Der Linearen Planunescrechnung (Mit Algol-Programm)", Elektron. Datenverarbeitung, 10, 1968.

[MURC 69] Murchland J D, "The 'Once Through' Method of Finding All Shortest Distance in a Graph from a Single Origin", Tranport Network Theory Unit, London Graduate School of Business Studies, Report LBS_TNT_56.1, 1969.

[PALL 81] Pallottino, "Shortest Path Methods: Complexity, Interrelations and New Propositions", Center De Research Sur Les Transports - Publication # 233, 1981.

[PAPE 74a] Pape U, "Changes in Networks and Adjustment of the Lengths of Shortest Paths", Computing, Vol 12, 1974.

[PAPE 74b] Pape U, "Implementation and Efficiency of Moore_Algorithms for the Shortest Route Problems", Math. Programming, Vol. 7, 1974.

[SHWI 80] Shier and Witzgall, "Arc Tolerances in Shortest Path and Network Flow Problems", Networks, Vol 10, 1980.

[SPPA 78] Spira and Pan, "On Finding and Updating Spanning Trees and Shortest Paths", Siam J Comput, Vol 4, No 3, 1978.

[STEE 74] Steenbrink, "Optimisation of Transport Networks", Printed by J W Arrowsmith Ltd, Bristol, England, 1974.

[TARJ 83] Tarjan, "Sensitivity Analysis of Minimum Spanning Trees and Shortest Path Trees", Information Processing Letters (North-Holland), 1982.

[TARJ 85] Tarjan, "Data Structure and Network Algorithms [44]", CBMS-NSF Regional Conference Series in Applied Mathematics, 1984.

[VLIE 78] Vliet, "Improved Shortest Path Algorithms for Transport Networks", Transpn. Res, Vol 12, 1978.

[WARS 62] Warshall, "A Theorem on Boolean Matrices", J. ACM 9, 1962.

[WILL 64] Willaims, "Algorithm 232: Heapsort", Comm. ACM 7, 1964.

[YENJ 70] Yen, "A Shortest Path Algorith", PhD Thesis, University of California, Berkeley, 1970.

[YENJ 72] Yen, "Finding the Lengths of All Shortest Paths in N-Node Non-Negative-Distance Complete Networks Using ½(N^3) Additions and (N^3) Compares", ACM 19, 3, 1972.

APPENDICES

The following appendices contain the complete Pascal codes of:

(a) reading and writing a network in both adjacency matrix and forward star forms;

(b) label correcting algorithm with a single queue managed in FIFO manner;

(c) label correcting algorithm with double ended queue (or actually output restricted double ended queue);

(d) label setting algorithm with address calculation sort;

(e) label setting algorithm with one level bucket sort.

In all the codes the variable names are chosen in a manner that makes their functions self explanatory.

APPENDIX A

This appendix describes the user input text file, INFILE. It also contains the Pascal coding for the procedures CHARTOINT, READADJMATRIX, READFORSTAR and PRINTADJMATRIX. Procedures READADJMATRIX or READFORSTAR read the adjacency matrix representation of a network stored in INFILE and represent it in the form of an adjacency matrix or adjacency lists (ie. forward star form), respectively. Both these procedures read the arc weights as characters and use the procedures CHARTOINT to convert them back to their integer values, there are two versions of this procedure, the one which excludes negative numbers is used for label setting algorithms. Procedure PRINTADJMATRIX outputs the adjacency matrix representation of the network.

INFILE is a text file that the user must create prior to running any of the programs in this study. INFILE contains an adjacency matrix representation of the network the user wishes the program to operate on. The adjacency matrix must be formatted in the following manner:

(i) *Each row of the adjacency matrix must be on one line, starting in the first column of the file.*

(ii) *Each number in the adjacency matrix must be in a field width of 4 characters. For example, if X represents a space, then the number 3 would be written:*

3 X X X X

(iii) *One clear line must be between the rows of the adjacency matrix.*

(iv) *To mark the end of each row of numbers in INFILE, an asterisk, *, must follow the last character in the row.*

(v) *The end of all the rows to be input is identified by an asterisk in the first column of a row.*

To illustrate these requirements, consider the adacency matrix:

	1	2	3	4
1	ø	1	3	ø
2	2	-5	ø	ø
3	ø	ø	ø	6
4	4	-33	224	ø

The correct INFILE format for this adjacency matrix is

1st Column in INFILE

↓

øxxx1xxx3xxxøxxx*

x

2xxx-5xxøxxxøxxx*

x

øxxxøxxxøxxx6xxx*

x

4xxx-33x22xøxxx*

x

*

(where x represents a space)

```
PROCEDURE CHARTOINT(CHARARRAY:WORD5;VAR VALUE: INTEGER);
{This procedure converts a number held in character form,}
(in CHARARRAY, to its integer value, VALUE )

VAR
  I,MULTFAC: INTEGER;    {MULTFAC stores the multiplication}
                         {factor}

BEGIN
  VALUE:= 0;
  I:= 5;
  WHILE (CHARARRAY[I] = ' ') DO
    BEGIN
      I:= I-1;
    END;    {Find the last digit of the number}
  MULTFAC:= 1;
  REPEAT
    IF (CHARARRAY[I] <> '-') THEN
      BEGIN  {Convert the digit to its integer value}
        VALUE:= VALUE+ (MULTFAC*((ORD(CHARARRAY[I]))-
        ORD('0'))));
        MULTFAC:= MULTFAC*10;
      END;
    I:= I-1;
  UNTIL (I=0);
  IF (CHARARRAY[I] = '-') THEN
    VALUE:= (VALUE * (-1));  {Convert a -ve number to its}
                             {correct value}
END; {CHARTOINT}
```

```
Procedure CHARTOINT(CHARARRAY : WORD5;VAR VALUE : INTERGER);
{This procedure converts a number held in character form,}
{in CHARARRAY to its integer value, VALUE.  This version}
{of CHARTOINT terminates processing on encountering a}
{negative number}


VAR
  I,MULTFAC:INTEGER;  {MULTFAC stores the multiplication}
                      {factor}

BEGIN
  IF (CHARARRAY[I] = '-') THEN
    BEGIN
      WRITELN('NEGATIVE WEIGHT ARC ENCOUNTERED - ILLEGAL');
      GOTO 99;
    END;
  VALUE:= 0;
  I:= 5;
  WHILE CHARARRAY[I] = ' ' DO
    BEGIN
      I:= I-1;
    END; {Locate the last digit of the number}
  MULTFAC:= 1;
  REPEAT {Convert the digit to its integer value}
    VALUE:= VALUE+ (MULTFAC*((ORD(CHARARRAY[I]))-
    (ORD('0'))));
    MULTFAC:=  MULTFAC*10;
    I:= I-1;
  UNTIL (I=0);
END;{CHARTOINT}
```

```
PROCEDURE READADJMATRIX;
{This procedure reads the adjacency matrix representation}
{of the network from INFILE into ADJMATRIX}


VAR
    ROW,COL,I,J,VALUE: INTEGER;
    NUMBER:WORD5;
    {NUMBER holds the number read from INFILE, in}
    {character form}
    ENDROW,ENDCOLS:BOOLEAN;
    {ENDROW = TRUE if end of row is reached i.e. a * is}
    {detected}
    {ENDCOLS =TRUE when all rows in adjacency matrix have}
    {been read}
    CH:CHAR;


BEGIN
  RESET(INFILE);
  ENDCOLS:= FALSE;
  FOR I:=  1 TO 100 DO
    BEGIN
      FOR J:=  1 TO 100 DO
        BEGIN
          ADJMATRIX[I,J]:= 0;
        END;
    END;  {Initialize ADJMATRIX}
  ROW:= 0;
  WHILE NOT(ENDCOLS) DO
    BEGIN
      ENDROW:= FALSE;
      COL:= 1;
      ROW:= ROW+1;
      WHILE NOT(ENDROW) DO
        BEGIN
          FOR I:= 1 TO 5 DO
            NUMBER[I]:= ' ';
          I:= 1;
          REPEAT  {Read the next number from INFILE}
            READ(INFILE,CH);
            NUMBER[I]:= CH;
            I:= I+1;
          UNTIL ((I = 5) OR (NUMBER[1] = '*'));
          IF (NUMBER[1] = '*') THEN
            BEGIN  {End of row detected}
              ENDROW :=TRUE;
              IF (COL=1) THEN
                ENDCOLS:= TRUE;  {End of Adjacency matrix}
            END
          ELSE
            BEGIN
              IF (NUMBER[1] <> '0') THEN
```

```
                  BEGIN  {Insert the number into ADJMATRIX}
                    CHARTOINT(NUMBER,VALUE);
                    ADJMATRIX[ROW,COL]:= VALUE;
                  END;
                COL:= COL+1;  {Increment column reference}
              END;
       END;
     IF NOT(ENDCOLS) THEN
       BEGIN
         READLN(INFILE);
         READLN(INFILE);
       END;  {Move to next row of the adjacency matrix}
  END;
NUMNODES:= ROW-1;  {Record the number of nodes in the}
                   {network}
END;  {READADJMATRIX}
```

```
PROCEDURE READFORSTAR;
{This procedure reads the adjacency matrix representation }
{of the network from INFILE to the 3 forward star arrays, }
{POINTERARRAY, STARARRAY and WEIGHTARRAY                 }


VAR
  ROW,COL,I,EDGEPOINTER,EDGEPOINTSTORE,VALUE:INTEGER;
  {EDGEPOINTER stores the next free location number in   }
  {STARARRAY}
  {EDGEPOINTSTORE stores the first location number in  }
  {STARARRAY used to store the current nodes forward star}
  NUMBER:WORD5;
  {NUMBER holds the number read from INFILE , in character}
  {form}
  ENDROW,ENDCOLS:BOOLEAN;
  {ENDROW = TRUE if end of row is reached i.e. a * is}
  {detected ENDCOLS = TRUE when all rows in adjacency}
  {matrix have been read}
  CH:CHAR;


BEGIN
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:=  0;
      WEIGHTARRAY[I]:= 0;
    END;  {Initialise forward star arrays}
  ENDCOLS:= FLASE;
  ROW:= 0;
  EDGEPOINTER:= 1;
  WHILE NOT(ENDCOLS) DO
    BEGIN
      ENDROW:= FALSE;
      COL:= 1;
      {EDGEPOINTER currently contains the first location}
      {number in STARARRAY that will be used to store the}
      {forward star of the next node}
      EDGEPOINTSTORE:= EDGEPOINTER;
      ROW:= ROW+1;
      WHILE NOT(ENDROW) DO
        BEGIN
          FOR I:= 1 TO 5 DO
             NUMBER[I]:= ' ';
          I:= 1;
          REPEAT   {Read the next number from INFILE}
            READ(INFILE, CH);
            NUMBER[I]:= CH;
            I:= I+1;
          UNTIL ((I = 5) OR (NUMBER[1] = '*'));
          IF (NUMBER[1] = '*') THEN
```

```
            BEGIN    {End of row detected}
              ENDROW:= TRUE;
              IF (COL=1) THEN
                ENDCOLS:= TRUE;     {End of adjacency matrix}
            END
          ELSE
            BEGIN
              IF (NUMBER[I] <> '0') THEN
                BEGIN {Insert information into the 3 arrays}
                  POINTERARRAY[ROW]:= EDGEPOINTSTORE;
                  CHARTOINT(NUMBER,VALUE):
                  STARARRAY[EDGEPOINTER]:= COL;
                  WEIGHTARRAY[EDGEPOINTER]:= VALUE;
                  EDGEPOINTER:= EDGEPOINTER+1;
                  {set pointer to next free location in}
                  {STARARRAY}
                END;
              COL:= COL+1;   {Increment column reference}
            END;
          END;
        IF NOT(ENDCOLS) THEN
          BEGIN
            READLN(INFILE);
            READLN(INFILE);
          END;   {Move to the next row of the adjacency }
                 {matrix}
      END;
      NUMNODES:= ROW-1;   {Record the number of nodes in}
                          {the network}
      POINTERARRAY[NUMNODES+1]:=  EDGEPOINTER;
      {Insert dummy pointer in POINTERARRAY}
END;  {READFORSTAR}
```

```
PROCEDURE PRINTADJMATRIX;
{This procedure displays the adjacency matrix}
{representation of the network to the screen}

VAR
  I: INTEGER;
  CH: CHAR;


BEGIN
  WRITELN('    ADJACENCY MATRIX    ');
  WRITELN('    ----------------    ');
  WRITELN;
  RESET(INFILE);
  WRITE('     ');
  FOR I:= 1 TO NUMNODES DO
     BEGIN
       WRITE(CHR(ORD((ORD('0'))+I)));
       IF (I>9) THEN
        WRITE('   ')
       ELSE
        WRITE('    ');
     END;
   WRITELN;
   WRITE('     ');
   FOR I:= 1 TO NUMNODES DO
     BEGIN
       WRITE('----');
     END;
   WRITELN;
   FOR I:= 1 TO NUMNODES DO
      BEGIN
        WRITE(CHR(ORD((ORD('0'))+I)));
        IF (I>9) THEN
          WRITE(' |')
        ELSE
          WRITE(' |');
        REPEAT
          READ(INFILE, CH);
          IF (CH<>'*') THEN
          WRITE(CH);
        UNTIL (CH = '*');
        READLN(INFILE);
        READLN(INFILE);
        CH:= ' ';
        WRITELN;
        WRITELN('  |');
      END;
   WRITELN;
   WRITELN;
END;  {PRINTADJMATRIX}
```

APPENDIX B

This appendix contains the PASCAL code for the program - FIFOSEQULST and the procedures PUTINLIST and PRINTFIFO. FIFOSEQLIST is the label correcting algorithm with a single queue using FIFO management and procedure PUTINLIST adds a node to the end of the queue, ie. ADDQ. both are discussed in section 8. PRINTFIFO displays the contents of the sequence list upon being called. Prior to running FIFOSEQULST, a correctly formatted version of INFILE must be available.

Some sample runs of this program are also shown in this appendix.

```
PROGRAM FIFOSEQULST ( INPUT,OUTPUT,INFILE);
{This program finds the shortest paths from a node, START  }
{to every other node in a network using the label          }
correcting algorithm.  This program implements a FIFO      }
sequence list and uses forward star representation of the  }
{network}


LABEL 99,88;

CONST
  INFINITY = 99999;

TYPE
  WORD5 = ARRAY[1..5] OF CHAR;
  ARRAY2 = ARRAY[1..2] OF INTEGER;
  ARRAY100 = ARRAY[1..100] OF INTEGER;
  LISTINFOTYPE = ARRAY[1..100] OF ARRAY2;


VAR
  POINTERARRAY,STARARRAY,WEIGHTARRAY,P,SEQULIST,d:ARRAY100;
  LISTINFO: LISTINFOTYPE;
  NUMNODES,R,FIRST,LAST,N,C,I,J:INTEGER;
  START,NEXT,ENTRYPOINTER,LEAVEPOINTER:INTEGER;
  INFILE: TEXT;


BEGIN  {MAIN}
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:= 0;
      WEIGHTARRAY[I]:= 0;
      P[I]:= 0;
      d[I]:= INFINITY;
      SEQULIST[I]:= 0;
      LISTINFO[I, 1]:= 0;
      LISTINFO[I, 2]:= 0;
   END;  {Initialise the arrays}
  READFORSTAR;  {Read in the network}
  PRINTADJMATRIX;  {Display the network}
  WRITELN('THIS IS THE GRAPH REPRESENTED IN FORWARD STAR
         FORM');
  WRITELN('----------------------------------------------')
  WRITELN;
  WRITELN('    POINTERARRAY   STARARRAY  WEIGHTARRAY');
  WRITELN('    ------------   ---------  -----------');
  WRITELN;
  FOR I:= 1 TO POINTERARRAY[NUMNODES + 1] DO
    BEGIN
      WRITE('     ' POINTERARRAY[I],'     ',STARARRAY[I]);
```

```
      WRITELN('     ',WEIGHTARRAY[I]);
    END;
  WRITELN;
  WRITELN('WHICH IS THE START NODE ?');
  READLN(START);
  WRITELN;
  d[START]:= 0;
  P[START]:= START;
  IF (POINTERARRAY[START] = 0) THEN
    GOTO 80;  {There are no paths from the starting node}
  LEAVEPOINTER := 1;
  SEQULIST[LEAVEPOINTER] := START;
  ENTRYPOINTER := 2; {Insert starting node in the sequence}
                     {list}
  WHILE (SEQULIST[LEAVPOINTER] <> 0) do
  BEGIN
    R := SEQULIST[LEAVEPOINTER];
    SEQULIST[LEAVEPOINTER] := 0;
    {Remove the next node from the sequence list}
    LEAVEPOINTER := LEAVEPOINTER + 1;
    IF (LEAVEPOINTER > 100) THEN
      LEAVEPOINTER := 1;  {Implement circular property of }
                          {queue}
    LISTINFO[R,1] := 0; {Node R is no longer in the}
                        {sequence list}
    IF (SEQULIST[LEAVEPOINTER] <> 0) THEN
      PRINTFIFO;  {Display the sequence list}
    FIRST := POINTERARRAY[R];
    N:= R;
    REPEAT
      N:= N+1;
      LAST:= POINTERARRAY[N];
    UNTIL (LAST <> 0);
    LAST:=  LAST - 1;
    FOR J := FIRST TO LAST DO
      BEGIN
        C := STARARRAY[J];
        IF (d[C] > (d[R] + WEIGHTARRAY[J]) THEN
          BEGIN  {Relabel node C}
            d[C] := (d[R] + WEIGHTARRAY[J]);
            P[C]:= R;
            IF (POINTERARRAY[C] <> 0) THEN
              PUTINLIST(C);  {Add node C to the back of the}
                             {queue}
          END;
      END;  {FOR loop}
  END;  {WHILE loop}
  {Trace the shortest paths through the tree}
88: FOR I:=  1 TO NUMNODES DO
      BEGIN
        IF (I <> START) THEN
          BEGIN
            IF (d[I] = INFINITY) THEN
```

```
          BEGIN
            WRITELN;
            WRITELN ('THERE IS NO ROUTE FROM ', START,'
            TO', I)
          END
        ELSE
          BEGIN
            WRITELN;
            WRITELN('DISTANCE FROM',START,'TO',I,'IS',
                    d[I]);
            WRITELN;
            WRITELN('ROUTE IS:');
            WRITELN;
            WRITE(I);
            NEXT := P[I];
            WHILE (NEXT <> START) DO
              BEGIN
                WRITE(NEXT);
                NEXT := P[NEXT];
              END;
            WRITELN(START);
          END;
      END;
  END;
99: END.
```

```
PROCEDURE PUTINLIST(NODE: INTEGER);
{This procedure adds a node, NODE, to the end of the}
{queue formed by the sequence list}


BEGIN
  IF  (POINTERARRAY[NODE] <> 0) AND (LISTINFO[NODE,2] <> 1)
  THEN
  {Check that NODE has a forward star and is not already}
  {in the queue}
  BEGIN
    SEQULIST[ENTRYPOINTER]:= NODE;   {Insert NODE in queue}
    ENTRYPOINTER:= ENTRYPOINTER+1;
    {Set ENTRYPOINTER to refer to the new 'end' of the}
    {queue}
    IF ENTRYPOINTER > 100 THEN
      ENTRYPOINTER:= 1;   {Implement circular property of}
                          {queue}
    PRINTFIFO;  {Display the contents of the queue}
    LISTINFO[NODE,2] := LISTINFO[NODE,2] + 1;
    {Increment no. of timed NODE has been in the queue}
    LISTINFO[NODE, 1]:= 1;   {Indicate that NODE in the}
    {queue}
    IF (LISTINFO[NODE, 2] = (NUMNODES + 1)) THEN
      BEGIN
        WRITELN('THIS GRAPH CONTAINS A NEGATIVE CIRCUIT -
        ILLEGAL');
      END;
  END;
END; {PUTINLIST}
```

```
PROCEDURE PRINTFIFO;
{This procedure displays the contents of the queue formed}
{by the sequence list}


VAR I: INTEGER;

  BEGIN
     WRITELN('STATE OF THE SEQUENCE LIST');
     WRITELN('-------------------------');
     WRITELN;
     WRITELN('NEXT NODE OUT');
     WRITELN('             |');
     WRITE('         ');
     FOR I:= LEAVEPOINTER TO (ENTRYPOINTER - 1) DO
       BEGIN
         WRITE(SEQULIST[I]: 4);
       END;
     WRITELN;
     WRITE('         ');
     FOR I:=  LEAVEPOINTER TO (ENTRYPOINTER - 2) DO
       BEGIN
         WRITE('       ');
       END;
     WRITELN('  |');
     WRITE('           ');
     FOR I:=  LEAVEPOINTER TO (ENTRYPOINTER - 2) DO
       BEGIN
         WRITE('       ');
       END;
     WRITELN('LAST NODE IN');
     WRITELN;
     WRITELN;
  END; {PRINTFIFO}
```

```
OK, PASCALG P408U>FIFOSEQULST.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing FIFOSEQULST

    ADJACENCY MATRIX
    ----------------

      1    2    3    4
     -----------------
1 |  0    5    0    0
  |
2 |  0    0    0    2
  |
3 |  0    1    0    0
  |
4 |  0    0   -3    0
  |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

    POINTERARRAY    STARARRAY    WEIGHTARRAY
    ------------    ---------    -----------

          1             2             5
          2             4             2
          3             2             1
          4             3            -3
          5             0             0

WHICH IS THE START NODE ?
1

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
            |
            2
            |
            LAST NODE IN


STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
            |
            4
            |
            LAST NODE IN
```

```
STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
         |
         3
         |
         LAST NODE IN

DISTANCE FROM             1 TO      2 IS        5

ROUTE IS:

     2         1

DISTANCE FROM             1 TO      3 IS        4

ROUTE IS:

     3         4         2         1

DISTANCE FROM             1 TO      4 IS        7

ROUTE IS:

     4         2         1
```

```
OK, PASCALG P408U>FIFOSEQULST.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing FIFOSEQULST

    ADJACENCY MATRIX
    ----------------

    1    2    3
    ------------
1  |0    2    0
   |
2  |0    0    8
   |
3  |-12  0    0
   |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

      POINTERARRAY     STARARRAY   WEIGHTARRAY
      ------------     ---------   -----------

           1               2             2
           2               3             8
           3               1           -12
           4               0             0

WHICH IS THE START NODE ?
1

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
           |
           2
           |
           LAST NODE IN


STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
           |
           3
           |
           LAST NODE IN
```

```
STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
             |
             1
             |
             LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
             |
             2
             |
             LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
             |
             3
             |
             LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
             |
             1
             |
             LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
             |
             2
             |
             LAST NODE IN
```

```
STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
            |
            3
            |
            LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
            |
            1
            |
            LAST NODE IN

STATE OF THE SEQUENCE LIST
--------------------------

NEXT NODE OUT
            |
            2
            |
            LAST NODE IN

THIS GRAPH CONTAINS A NEGATIVE CIRCUIT - ILLEGAL
```

APPENDIX C

This appendix contains the PASCAL code for the program - DBENDQUEUE and the procedure PUTINDEQUEUE and PRINTDEQUEUE. DBENQUEUE is the label correcting algorithm with output restricted double ended queue and procedure. PUTINDEQUEUE adds a node to the top or bottom of a queue, ADDDQ, both are discussed in sections 4 and 8. PRINTDEQUEUE, upon call, displays the contents of the output restricted double ended queue. Prior to running DBENDQUEUE, a correctly formatted version of INFILE must be available.

Some sample runs of this program are also shown in this appendix.

```
PROGRAM DBENDQUEUE(INPUT,OUTPUT,INFILE);
{This program finds the shortest paths from a node, START  }
{to every other node in a network using the label          }
{correcting algorithm.  This program implements an output  }
{restricted double ended queue and uses forward star       }
{representation of the network                             }


LABEL 99,88;

CONST
  INFINITY = 99999;

TYPE
WORD5 = ARRAY[1..5] OF CHAR;
ARRAY2 = ARRAY[1..2] OF INTEGER;
ARRAY100 = ARRAY[1..100] OF INTEGER;


VAR
POINTERARRAY,STARARRAY,WEIGHTARRAY,P,d,DEQUEUE : ARRAY100;
ENTRYCOUNT: ARRAY100;
NUMNODES,R,N,C,I,J,START,NEXT,FRONTQUEUE,BACKQUEUE,FIRST,
LAST:INTEGER;
ENTRY,TOP:BOOLEAN;
INFILE:TEXT;


BEGIN  {MAIN}
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:= 0;
      WEIGHTARRAY[I]:= 0;
      P[I]:= 0;
      d[I]:= INFINITY;
      DEQUEUE[I]:= 0;
      ENTRYCOUNT[I]:= 0;
    END;  {Initialise the arrays}
  READFORSTAR;  {Read in the network}
  PRINTADJMATRIX;  {Display the network}
  WRITELN('THIS IS THE GRAPH REPRESENTED IN FORWARD STAR
  FORM');
  WRITELN('----------------------------------------------');
  WRITELN;
  WRITELN('    POINTERARRAY    STARARRAY    WEIGHTARRAY');
  WRITELN('    ------------    ---------    -----------');
  WRITELN;
  FOR I:= 1 TO POINTERARRAY[NUMNODES + 1] DO
    WRITELN('      ',POINTERARRAY[I], '      ',STARARRAY[I],
    '       ',WEIGHTARRAY[I]);
  WRITELN;
```

```
WRITELN('WHICH IS THE START NODE ?');
READLN(START);
WRITELN;
d[START]:= 0;
DEQUEUE[START]:= INFINITY;
P[START]:= START;
IF (POINTERARRAY[START] = 0) THEN
  GOTO 88;
FRONTQUEUE:= START;   {Insert the starting node in the}
                      {dequeue}
BACKQUEUE:= START;
WHILE  (FRONTQUEUE <> INFINITY) DO
  BEGIN
    R := FRONTQUEUE:
    {Remove the next node from the dequeue}
    ENTRY :=FALSE;
    PRINTDEQUEUE;  {Display the contents of the dequeue}
    FRONTQUEUE:=  DEQUEUE[FRONTQUEUE]; {Reset queue}
                                       {pointer}
    IF (FRONTQUEUE = INFINITY) THEN
      BACKQUEUE:=  INFINITY;  {Empty queue condition}
    DEQUEUE[R]:=  -1;
    FIRST := POINTERARRAY[R];
    N:= R;
    REPEAT
      N:= N+1;
      LAST:= POINTERARRAY[N];
    UNTIL  (LAST <> 0);
    LAST := LAST - 1;
    FOR J:=  FIRST TO LAST DO
       BEGIN
         C:= STARARRAY[J];
         IF (d[C] > (d[R] + WEIGHTARRAY[J])) THEN
           BEGIN  {Relabel node C}
             d[C] := (d[R] + WIEIGHTARRAY[J]);
             P[C]:= R;
             IF (POINTERARRAY[C] <> 0) THEN
               PUTINDEQUEUE(C);  {Add node C to the}
                                 {dequeue}
           END;
        END;
    END;   {WHILE loop}
    {Trace the shortest paths through the tree}
88:   FOR I:=  1 TO NUMVERT DO
        BEGIN
          IF (I <> START) THEN
            BEGIN
              IF (d[I] = INFINITY) THEN
                BEGIN
                  WRITELN;
                  WRITELN('THERE IS NO ROUTE FROM', START,
                  'TO',d[I]);
                  WRITELN;
```

```
                    WRITELN('ROUTE IS:');
                    WRITELN;
                    WRITE(I);
                    NEXT:= P[I];
                    WHILE (NEXT <> START) DO
                      BEGIN
                        WRITE(NEXT);
                        NEXT:= P[NEXT];
                      END;
                    WRITELN(START);
                  END;
               END;
           END;
99:     END.
```

```
PROCEDURE PUTINDEQUEUE(NODE: INTEGER);
{This procedure adds a node, NODE, to the front or the}
{back of the double ended queue, as required}


BEGIN
  ENTRYCOUNT[NODE]:= ENTRYCOUNT[NODE] + 1;
  {Increment no. of times NODE has been in the dequeue}
  IF (ENTRYCOUNT[NODE] = (NUMNODES + 1)) THEN
    BEGIN
      WRITELN('NEGATIVE LENGTH CIRCUIT ENCOUNTERED -
      ILLEGAL');
      GOTO 99;
    END;
  IF (DEQUEUE[NODE] = -1) THEN
    BEGIN  {Insert NODE at the front of the dequeue}
      TOP:= TRUE;
      DEQUEUE[NODE]:=  FRONTQUEUE;
      FRONTQUEUE:=  NODE;
      IF (BACKQUEUE = INFINITY) THEN
        BACKQUEUE:=  NODE;
      ENTRY:= TRUE;
      PRINTDEQUEUE;   {Display the contents of the dequeue}
    END
  ELSE
    BEGIN  {Insert NODE at the back of the dequeue}
      IF (DEQUEUE[NODE] = 0) THEN
      BEGIN
        TOP:= FALSE;
        DEQUEUE[NODE]:=  INFINITY;
        IF (BACKQUEUE <> INFINITY) THEN
          DEQUEUE[BACKQUEUE]:=  NODE;
        BACKQUEUE:=  NODE;
        IF (FRONTQUEUE = INFINITY) THEN
          FRONTQUEUE:=  NODE;
        ENTRY:= TRUE;
        PRINTDEQUEUE;   {Display the contents of the dequeue}
      END;
    END;
END;  {PUTINDEQUEUE}
```

```
PROCEDURE PRINTDEQUEUE;
{This procedure displays the contents of the double}
{ended queue formed by the sequence list}


VAR
  I,NUMPRINTED: INTEGER;


BEGIN
  WRITELN('STATE OF THE DOUBLE ENDED QUEUE');
  WRITELN('-------------------------------');
  WRITELN;
  IF NOT (ENTRY) THEN
    BEGIN
      WRITELN('NODE ABOUT TO LEAVE');
      WRITELN('                  |');
    END
  ELSE
    BEGIN
      IF (TOP) THEN
        BEGIN
          WRITELN('   NODE JUST ENTERED');
          WRITELN('                  |');
        END;
    END;
  I:=  FRONTQUEUE;
  NUMPRINTED:= 0;
  WRITE('                 ');
  REPEAT
    WRITE(I:4);
    NUMPRINTED:= NUMPRINTED+1;
    I:= DEQUEUE[I];
  UNTIL  (I = INFINITY);
  WRITELN;
  IF (ENTRY = TRUE) AND (NOT TOP) THEN
    BEGIN
      WRITE('              |');
      FOR I:= 1 TO (NUMPRINTED - 1) DO
         BEGIN
           WRITE('     ');
         END;
      WRITELN('      ');
      FOR I:=  1 TO (NUMPRINTED - 1) DO
         BEGIN
           WRITE('      ');
         END;
      WRITELN('  NODE JUST ENTERED');
    END;
  WRITELN:
  WRITELN;
END;  {PRINTDEQUEUE}
```

```
OK, PASCALG P408U>DEQUEUE.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing DBENDQUEUE

   ADJACENCY MATRIX
   ----------------

   1   2   3
   -------------
1 |0   2   0
  |
2 |0   0   8
  |
3 |0   0   5
  |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

    POINTERARRAY     STARARRAY    WEIGHTARRAY
    ------------     ---------    -----------

         1               2             2
         2               3             8
         3               3             5
         4               0             0

WHICH IS THE START NODE ?
1

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 1
                 |


STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

                 2
                 |
  NODE JUST ENTERED
```

```
STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 2

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

                 3
                 |
  NODE JUST ENTERED

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 3

DISTANCE FROM        1 TO          2 IS          2

ROUTE IS:

       2         1

DISTANCE FROM        1 TO          3 IS         10

ROUTE IS:

       3         2          1
```

```
OK, PASCALG P408U>DEQUEUE.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing DBENDQUEUE

    ADJACENCY MATRIX
    ----------------

    1   2   3
    ------------
1  |0   2   0
   |
2  |0   0   8
   |
3  |0   0   -12
   |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

     POINTERARRAY    STARARRAY    WEIGHTARRAY
     ------------    ---------    -----------

          1              2             2
          2              3             8
          3              3           -12
          4              0             0

WHICH IS THE START NODE ?
2

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                  |
                  2


STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

                  3
                  |
  NODE JUST ENTERED
```

```
STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 3

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

  NODE JUST ENTERED
                 |
                 3

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 3

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

  NODE JUST ENTERED
                 |
                 3

STATE OF THE DOUBLE ENDED QUEUE
-------------------------------

NODE ABOUT TO LEAVE
                 |
                 3

NEGATIVE LENGTH CIRCUIT ENCOUNTERED - ILLEGAL
```

APPENDIX D

This appendix contains the PASCAL code for the program, ADCALC, the label setting algorithm with address calculation, the procedure ADDNODE and REMOVENODE, and the modified PASCAL code for the procedure READFORSTAR. The PASCAL code for the procedure PRINTNODEARRAY is also included. This procedure, upon call, displays the contents of the non-empty locations of NODEARRAY. Prior to running ADCALC, a correctly formatted version of INFILE must be available.

Some sample runs of this program are also shown in this appendix.

```
PROGRAM ADCALC(INPUT,OUTPUT,INFILE);
{This program finds the shortest paths from a node,  START}
{to every other node in a network using the label setting }
{algorithm.  This program implements an address           }
{calculation sort and uses forward star representation of }
{the network                                               }


LABEL  99;


CONST
   INFINITY = 99999;


TYPE
  WORD5 = ARRAY[1..5] OF CHAR;
  ARRAY100 = ARRAY[1..100] OF INTEGER;
  POINTER = ^NODE;
  PTRARRAY = ARRAY[0..1000] OF POINTER;

  NODE = RECORD
           NAME: INTEGER;
           NEXT: POINTER;
         END;
  BOARRAY: ARRAY[1..100] OF BOOLEAN;


VAR
  NODEARRAY: PTRARRAY;
  PTR: POINTER;
  POINTERARRAY,STARARRAY,WEIGHTARRAY,P,d : ARRAY100;
  NUMNODES,N,C,I,J,MODULUS,START,NEXT,R : INTEGER;
  ARRAYREF,STARTREF,CURRENTLOC,NEWLOC,FIRST,LAST:INTEGER;
  TERMINATE: BOOLEAN;
  INFILE: TEXT;
  INTREE: BOARRAY;


BEGIN  {MAIN}
  RESET (INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:= 0;
      WEIGHTARRAY[I]:= 0;
      P[I]:= 0;
      d[I]:= INFINITY;
      INTREE[I]:= FALSE;
    END; {Initialise the arrays}
  READFORSTAR;  {Read in the network}
  PRINTADJMATRIX;  {Display the network}
  WRITELN('THIS IS THE GRAPH REPRESENTED IN FORWARD STAR
```

```
FORM');
WRITELN('-------------------------------------------------');
WRITELN;
WRITELN('     POINTERARRAY    STARARRAY    WEIGHTARRAY');
WRITELN('     ------------    ---------    -----------');
WRITELN;
FOR I:= 1 TO POINTERARRAY[NUMNODES + 1] DO
  WRITELN('      ',POINTERARRAY[I],'     ',STARARRAY[I],
  '      ',WEIGHTARRAY[I]);
WRITELN;
WRITELN('WHICH IS THE START NODE ?');
READLN(START);
WRITELN;
d[START]:= 0;
P[START]:=  START;
FOR I =0 TO MODULUS DO
  NODEARRAY[I]:= NIL;
ARRAYREF:=  -1;
NEW(PTR);
PTR^.NAME:= START;
PTR^.NEXT:= NIL;
NODEARRAY[0]:= PTR; {Insert starting node in NODEARRAY}
IF (POINTERARRAY[START] <> 0) THEN
  TERMINATE:= FALSE  {No paths from start node}
ELSE
  TERMINATE:= TRUE;
WHILE (TERMINATE = FALSE) DO
  BEGIN
    STARTREF:= ARRAYREF;
    REPEAT
      ARRAYREF:= ARRAYREF + 1;
      IF (ARRAYREF > MODULUS) THEN
        ARRAYREF:= 0;
    UNTIL (NODEARRAY[ARRAYREF] <> NIL) OR
    (ARRAYREF = STARTREF);
    {Search for next non NIL entry in NODEARRAY}
    IF ARRAYREF = STARTREF THEN
      TERMINATE:= TRUE  {NODEARRAY is empty}
    ELSE
      BEGIN
        PTR:= NODEARRAY[ARRAYREF];
        REPEAT {For each node in the linked list located};
          R:=  PTR^.NAME;
          INTREE[I] := TRUE;
          WRITELN('EXAMINING NODE ',R:3);
          WRITELN;
          FIRST:= POINTERARRAY[R];
          N; =R;
          REPEAT
            N:= N+1;
            LAST:= POINTERARRAY[N];
          UNTIL (LAST <> 0);
          LAST:= LAST - 1;
```

```
            FOR J:= FIRST TO LAST DO
              BEGIN
                C:= STARARRAY[J];
                IF ((d[R] + WEIGHTARRAY[J]) < (d[C]))
                    AND (INTREE[C] = FALSE) THEN
                  BEGIN  {Relabel node C}
                    IF (d[C] <> INFINITY) AND
                    (POINTERARRAY[C] <> 0) THEN
                    BEGIN {Remove C from its current pos.}
                    {in NODEARRAY}
                      CURRENTLOC:=  (d[C] MOD MODULUS);
                      {* Calculate C's current address in}
                      {NODEARRAY *}
                      REMOVENODE(CURRENTLOC,C);
                      PRINTNODEARRAY(FALSE,C);
                      {Display contents of NODEARRAY}
                    END;
                 d[C]:=  d[R] + WEIGHTARRAY[J];
                 P[C]:= R;
                 IF (POINTERARRAY[C] <> 0) THEN
                   BEGIN
                     NEWLOC:=  (d[C] MOD MODULUS);
                     {* Calculate C's new address in}
                     {NODEARRAY *}
                     ADDNODE(NEWLOC,C);
                     PRINTNODEARRAY(TRUE,C);
                     {Display contents of NODEARRAY}
                   END;
              END;
          END;
          PTR:= PTR^.NEXT;
          {Set pointer to refer to the next node in the}
          {linked list}
          REMOVENODE(ARRAYREF,R); {Remove R from the}
                                  {linked list}
          PRINTNODEARRAY(FALSE,R);
          {Display contents of NODEARRAY}
        UNTIL (PTR = NIL);)
        {End of linked list has been reached}
      END;  {IF ARRAYREF = STARTREF}
   END;  {WHILE TERMINATE <> FALSE}
{Trace the shortest paths through the tree}
FOR I:=  1 TO NUMNODES DO
  BEGIN
    IF (I <> START) THEN
      BEGIN
        IF (d[I] = INFINITY) THEN
          BEGIN
            WRITELN;
            WRITELN('THERE IS NO ROUTE FROM',START,'TO',I)
          END
        ELSE
          BEGIN
```

```
            WRITELN;
            WRITELN('DISTANCE FROM',START,'TO',I,'IS',d[I]);
            WRITELN;
            WRITELN('ROUTE IS:');
            WRITELN;
            WRITE(I);
            NEXT:= P[I];
            WHILE (NEXT <> START) DO
              BEGIN
                WRITE(NEXT);
                NEXT:= P[NEXT];
              END;
            WRITELN(START);
          END;
      END;
    END;
99: END.
```

```
PROCEDURE ADDNODE(LOC,NODE:INTEGER);
{This procedure adds a node, NODE, to the end of the}
{linked list pointed to from location LOC in NODEARRAY}


VAR PTR,NEWPTR:POINTER;

BEGIN
  PTR:=NODEARRAY[LOC];
  IF (PTR <> NIL) THEN
    BEGIN {There is already a linked list pointed to from}
          {location LOC}
      WHILE (PTR^.NEXT <> NIL) DO
        BEGIN
          PTR:=PTR^.NEXT;
        END;  {Find the end of the linked list}
      NEW(NEWPTR);
      NEWPTR^.NAME:=NODE;
      PTR^.NEXT:=NEWPTR;
      NEWPTR^.NEXT:=NIL; {Add NODE to the end of the linked}
                         {list}
    END
  ELSE {There is currently no linked list pointed to from}
       {location LOC}
    BEGIN
      NEW(NEWPTR);
      NEWPTR^.NAME:=NEWPTR;
      NEWPTR^.NEXT:=NIL;
   END; {Add NODE as first (and only) node in the linked}
        {list}
END;  {ADDNODE}
```

```
PROCEDURE REMOVENODE(LOC,NODE:INTEGER);
{This procedure removes a node, NODE, from the linked}
{pointed to from location LOC in NODEARRAY}


VAR PTR,OLDPTR:POINTER;

BEGIN
  PTR;=NODEARRAY[LOC];
  IF (PTR^.NAME <> NODE) THEN
  {Check if NODE is the first node in the linked list}
     BEGIN
       REPEAT
         OLDPTR:=PTR;
         {OLDPTR points to the node before NODE in the}
         {linked list}
         PTR:=PTR^.NEXT;
       UNTIL (PTR^.NAME = NODE); {Locate NODE in the linked}
                                 {list}
      OLDPTR^.NEXT:=PTR^.NEXT; {Bypass NODE in the linked}
                               {list}
      DISPOSE(PTR);
    END
  ELSE
    BEGIN {The node after NODE becomes the first in the}
          {linked list}
      NODEARRAY[LOC]:= PTR^.NEXT;
      DISPOSE(PTR);
    END;
END: {REMOVENODE}
```

```
PROCEDURE READFORSTAR;
{This procedure reads the adjacency matrix representation }
{of the network from INFILE to the 3 forward star arrays -}
{POINTERARRAY, STARARRAY and WEIGHTARRAY.  This version of}
{the procedure also obtains the value of MODULUS required }
{by the program}

VAR
  ROW,COL,I,EDGEPOINTER,EDGEPOINTSTORE,VALUE:INTEGER;
  {EDGEPOINTER stores the next free location number in}
  {STARARRAY}
  {EDGEPOINTSTORE stores the first location number in }
  {STARARRAY used to store the current nodes forward  }
  {star}
  NUMBER:WORD5;
  {NUMBER holds the number read from INFILE, in character}
  {form}
  ENDROW,ENDCOLS:BOOLEAN;
  {ENDROW = TRUE if end of row is reached i.e. a * is }
  {detected ENDCOLS = TRUE when all rows in adjacency }
  {matrix have been read}
  CH:CHAR;


BEGIN
  MODULUS:= 0;
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
     BEGIN
       POINTERARRAY[I]:= 0;
       STARARRAY[I]:= 0;
       WEIGHTARRAY[I]:= 0;
     END:  {Initialise forward star arrays}
  ENDCOLS:= FALSE;
  ROW:= 0;
  EDGEPOINTER:= 1;
  WHILE NOT(ENDCOLS) DO
    BEGIN
      ENDROW:= FALSE;
      COL:= 1;
      {EDGEPOINTER currently contains the first location}
      {number in STARARRAY that will be used to store the}
      {forward star of the next node}
      EDGEPOINTSTORE:= EDGEPOINTER;
      ROW:= ROW+1;
      WHILE NOT(ENDROW) DO
        BEGIN
          FOR I:=1 TO 5 DO
             NUMBER[I]:= '   ';
          I:= 1;
          REPEAT  {Read the next number from INFILE}
            READ(INFILE,CH);
            NUMBER[I]:= CH;
```

```
          I:= I+1;
        UNTIL ((I = 5) OR (NUMBER[1] = '*'));
        IF (NUMBER[1] = '*') THEN
          BEGIN  {End of row detected}
            ENDROW:= TRUE;
            IF (COL=1) THEN
              ENDCOLS:= TRUE;  {End of adjacency matrix}
          END
        ELSE
          BEGIN
            IF (NUMBER[1] <> '0') THEN
              BEGIN {Insert information into the 3 arrays}
                POINTERARRAY[ROW]:= EDGEPOINTSTORE;
                CHARTOINT(NUMBER,VALUE);
                IF (VALUE > MODULUS) THEN
                  MODULUS:= VALUE;
                STARARRAY[EDGEPOINTER]:= COL;
                WEIGHTARRAY[EDGEPOINTER]:= VALUE;
                EDGEPOINTER:= EDGEPOINTER+1;
                {Set pointer to next free location in}
                {STARARRAY}
              END;
           COL:= COL+1;  {Increment column reference}
          END;
        END;
      IF NOT(ENDCOLS) THEN
        BEGIN
          READLN(INFILE);
          READLN(INFILE);
        END;  {Move to the next row of the adjacency}
        {matrix}
  END;
  NUMNODES:= ROW-1;  {Record the number of nodes in the}
  {network}
  POINTERARRAY[NUMNODES+1]:=  EDGEPOINTER;
  {Insert dummy pointer in POINTERARRAY}
  MODULUS:= MODULUS+1;  {MODULUS := Lmax + 1}
END;  {READFORSTAR}
```

```
PROCEDURE PRINTNODEARRAY(ADDED:BOOLEAN;NODENUM:INTEGER);
{This procedure displays the contents of the non-empty}
{locations of NODEARRAY.  It also outputs which node has}
{just been added or removed from NODEARRAY}


VAR
  K:INTEGER;
  PTR:POINTER;

BEGIN
  WRITELN('STATE OF NODEARRAY');
  WRITELN('------------------');
  WRITELN;
  IF (ADDED) THEN
    WRITELN('NODE',NODENUM:4,'ADDED')
  ELSE
    WRITELN('NODE',NODENUM:4,'REMOVED');
  WRITELN;
  WRITELN('LOCATION IN NODARRAY     LIST FROM LOCATION');
  WRITELN('--------------------     ------------------');
  WRITELN;
  FOR K:= 0 TO MODULUS DO
    BEGIN
      PTR:= NODEARRAY[K];
      IF (PTR <> NIL) THEN
        BEGIN
          WRITE(K:11);
          WRITE('             ');
          REPEAT
            WRITE(' ------ ');
            WRITE(PTR^.NAME:3);
            PTR:= PTR^.NEXT;
          UNTIL (PTR = NIL);
          WRITELN;
          WRITELN;
        END;
    END;
END;  {PRINTNODEARRAY}
```

```
OK, PASCALG P408U>ADCALC.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing ADCALC


     ADJACENCY MATRIX
     ----------------

     1   2   3
     ------------

1  |0    1   4
   |
2  |0    0   3
   |
3  |0    0   0
   |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

      POINTERARRAY    STARARRAY   WEIGHTARRAY
      ------------    ---------   -----------

            1              2           1
            3              3           4
            0              3           3
            4              0           0

WHICH IS THE START NODE ?
2

STATE OF NODEARRAY
------------------

NODE    2 ADDED

LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------

          0                 -----    2

EXAMINING NODE    2

STATE OF NODEARRAY
------------------

NODE    2 REMOVED

LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------
```

```
THERE IS NO ROUTE FROM        2 TO         1

DISTANCE FROM              2 TO        3 IS        3

ROUTE IS:
      3        2
```

```
OK, PASCALG P408U>ADCALC,PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing ADCALC

     ADJACENCY MATRIX
     ----------------

     1    2    3
     ------------
1   |0    1    4
    |
2   |0    0    3
    |
3   |0    0    0
    |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

     POINTERARRAY   STARARRAY   WEIGHTARRAY
     ------------   ---------   -----------

           1             2            1
           3             3            4
           0             3            3
           4             0            0

WHICH IS THE START NODE ?
1

STATE OF NODEARRAY
------------------

NODE    1 ADDED

LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------

          0                  -----    1

EXAMINING NODE 1

STATE OF NODEARRAY
------------------

NODE    2 ADDED
```

```
LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------

          0                ----- 1

          1                ----- 2

STATE OF NODEARRAY
------------------

NODE    1 REMOVED

LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------

          1                ----- 2

EXAMINING NODE   2

STATE OF NODEARRAY
------------------

NODE    2 REMOVED

LOCATION IN NODEARRAY    LIST FROM LOCATION
---------------------    ------------------


DISTANCE FROM          1 TO         2 IS          1

ROUTE IS:

      2         1

DISTANCE FROM          1 TO         3 IS          4

ROUTE IS:

      3         1
```

APPENDIX E

This appendix contains the PASCAL code for the program, BUCKETSORT, the label setting algorithm with 1-level bucketsort, the procedures ADDNODEBUCK and REMOVENODEBUCK and the modified PASCAL code for READFORSTAR. The program and its associated procedures are discussed in section 9. The PASCAL code for the procedure RINTBUCKETS is also included. This procedure, upon call, displays the contents of the non-empty buckets in the bucket list. Prior to running BUCKETSORT, a correctly formatted version of INFILE must be available.

Some samples runs of this program are also shown in this appendix.

```
PROGRAM BUCKETSORT(INPUT,OUTPUT,INFILE);
{This program finds the shortest paths from a node , START}
{to every other node in a network using the label setting }
{algorithm.  This algorithm implements a bucket sort and  }
{uses forward star representation of the network          }

LABEL 99;

CONST
  INFINITY = 99999;

TYPE
  WORD5 = ARRAY[1..5] OF CHAR;
  ARRAY100 = ARRAY[1..100] OF INTEGER;
  POINTER = ^NODE;
  PTRARRAY = ARRAY[0..1000] OF POINTER;

  NODE = RECORD
           NAME: INTEGER;
           NEXT: POINTER;
         END;
  BOARRAY = ARRAY[1..100] OF INTEGER;

VAR
  BUCKLIST: PTRARRAY;
  PTR: POINTER;
  POINTERARRAY,STARARRAY,WEIGHTARRAY,P,d : ARRAY100;
  NUMNODES,N,C,I,J,LMIN,START,NEXT:INTEGER;
  BUCKREF,CURRENTBUCKET,NEWBUCKET,FIRST,LAST: INTEGER;
  TERMINATE: BOOLEAN;
  INFILE: TEXT;
  INTREE: BOARRAY;

BEGIN  {MAIN}
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:= 0;
      WEIGHTARRAY[I]:= 0;
      P[I]:= 0;
      d[I]:= INFINITY;
      INTREE[I]:= FALSE;
    END;
  READFORSTAR;  {Read in the network}
  PRINTADJMATRIX;  {Display the network}
  WRITELN('THIS IS THE GRAPH REPRESENTED IN FORWARD STAR
  FORM');
```

```
WRITELN('---------------------------------------------');
WRITELN;
WRITELN('     POINTERARRAY    STARARRAY    WEIGHTARRAY');
WRITELN('     ------------    ---------    -----------');
WRITELN;
FOR I:= 1 TO POINTERARRAY[NUMNODES + 1] DO
  WRITELN('     ',POINTERARRAY[I],'        ',STARARRAY[I],
  '      ',WEIGHTARRAY[I]);
WRITELN;
WRITELN('WHICH IS THE START NODE ?');
READLN(START)
WRITELN;
d[START]:= 0;
P[START]:= START;
FOR I := 1 TO 1000 DO
  BUCKLIST[I]:= NIL;
BUCKREF:=  -1;
NEW(PTR);
PTR^.NAME:= START;
PTR^.NEXT:= NIL;
BUCKLIST[0]:= PTR;
{Insert starting node in BUCKET 0 }
IF (POINTERARRAY[START] <> 0) THEN
  TERMINATE:= FALSE  {No paths from start node }
ELSE
  TERMINATE:= TRUE;
WHILE (TERMINATE = FALSE) DO
  BEGIN
    REPEAT
      BUCKREF:= BUCKREF + 1;
    UNTIL (BUCKREF = 1001) OR (BUCKLIST[BUCKREF] <> NIL);
    {Search for the next non-empty bucket}
    IF BUCKREF = 1001 THEN
      TERMINATE:= TRUE
    ELSE
      BEGIN
        PTR:= BUCKLIST[BUCKREF];
        REPEAT  {For each node - R, in the bucket linked}
                {list}
          R:=  PTR^.NAME;
          INTREE[R]:= TRUE;  {Add R to the tree}
          WRITELN('EXAMINING NODE',R:3);
          WRITELN;
          FIRST:= POINTERARRAY[R];
          N:= R;
          REPEAT
            N:= N+1;
            LAST:= POINTERARRAY[N];
          UNTIL (LAST <> 0);
          LAST:= LAST - 1;
          FOR J:=  FIRST TO LAST DO
            BEGIN
              C:=  STARARRAY[J];
```

```
                IF ((d[R] + WEIGHTARRAY[J]) < d[C]) AND
                   (INTREE[C] = FALSE) THEN
                 BEGIN  {Relabel node C}
                   IF (d[C] <> INFINITY) AND (POINTERARRAY
                   [C] <> 0) THEN
                     BEGIN  {If C is already in a bucket}
                       CURRENTBUCKET:=  (d[C] DIV LMIN);
                       {Find C's current bucket}
                       REMOVENODEBUCK(CURRENTBUCKET,C);
                     END;
                   d[C] := (d[R] + WEIGHTARRAY[J]);
                   P[C]:=  R;
                   IF (POINTERARRAY[C] <> 0) THEN
                   (Check if C has a forward star)
                   BEGIN
                     NEWBUCKET:=  (d[C] DIV LMIN);
                     {Calculate C's new bucket}
                     ADDNODEBUCK(NEWBUCKET,C);
                     {Insert C in its new bucket}
                  END;
              END;
          END;  {FOR Loop}
          PTR:= PTR^.NEXT;
          REMOVENODEBUCK(BUCKREF,R);
        UNTIL (PTR = NIL);
      END;
  END;  {WHILE loop}
FOR I:=  1 TO NUMNODES DO
  BEGIN
    IF  (I <> START) THEN
      BEGIN
        IF (d[I] = INFINITY) THEN
          BEGIN
            WRITELN;
            WRITELN('THERE IS NO ROUTE FROM',START,'TO',I)
          END
        ELSE
          BEGIN
            WRITELN;
            WRITELN('DISTANCE FROM',START,'TO',I,
                    'IS',d[I]);
            WRITELN;
            WRITELN('ROUTE IS:');
            WRITELN;
            WRITE(I);
            NEXT:= P[I];
            WHILE (NEXT <> START) DO
              BEGIN
                WRITE(NEXT);
                NEXT:= P[NEXT];
              END;
            WRITELN(START);
          END;
```

```
        END;
    END;
99:END.
```

```
PROCEDURE ADDNODEBUCK(BUCKNUM,NODE:INTEGER);
{This procedure adds a node, NODE, to bucket K in the }
{bucket list}


VAR
  PTR,NEWPRT:POINTER;


BEGIN
  PTR:= BUCKETARRAY[BUCKNUM];
  IF (PTR <> NIL) THEN  {Bucket BUCKNUM is not empty}
    BEGIN  {Find the last node in bucket BUCKNUM  }
      WHILE (PTR^.NEXT <> NIL) DO
        BEGIN
          PTR:= PTR^.NEXT;
        END;
      NEW(NEWPTR);
      NEWPTR^.NAME:= NODE;
      PTR^.NEXT:= NEWPTR;
      NEWPTR^.NEXT:= NIL;
    END
  ELSE
    BEGIN  {NODE is added as the first node in bucket K}
      NEW(NEWPTR);
      NEWPTR^.NAME:= NODE;
      BUCKETARRAY[BUCKNUM]:= NEWPTR;
      NEWPTR^.NEXT:= NIL;
    END.
  PRINTBUCKETS(TRUE,NODE); {Display the non-empty buckets}
END;  {ADDNODEBUCK}
```

```
PROCEDURE REMOVENODEBUCK(BUCKNUM,NODE: INTEGER);
{This procedure removes a node, NODE, from bucket BUCKNUM}
{in the bucket list}


VAR
  PTR,OLDPTR:POINTER;

BEGIN
  PTR:= BUCKETARRAY[BUCKNUM];
  IF (PTR^.NAME <> NODE) THEN
    BEGIN {NODE is not the first node in bucket K}
      REPEAT {Locate NODE in bucket K}
        OLDPTR:= PTR;
        PTR:= PTR^.NEXT;
      UNTIL (PTR^.NAME = NODE);
      OLDPTR^.NEXT:= PTR^.NEXT;
      {Bypass NODE in the linked list representing bucket K}
      DISPOSE(PTR);
    END
  ELSE
    BEGIN {NODE is the first node in bucket K}
      BUCKETARRAY[BUCKNUM]:= PTR^.NEXT;
      DISPOSE(PTR);
    END;
  PRINTBUCKETS(FALSE,NODE);
END;  {REMOVENODEBUCK}
```

```
PROCEDURE READFORSTAR;
{This procedure reads the adjacency matrix representation}
{of the network from INFILE to the 3 forward star arrays }
{- POINTERARRAY, STARARRAY and WEIGHTARRAY.  This version}
{of the procedure also obtains the value of LMIN required}
{by the program}


VAR
  ROW,COL,I,EDGEPOINTER,EDGEPOINTSTORE,VALUE : INTEGER;
  {EDGEPOINTER stores the next free location number in    }
  {STARARRAY EDGEPOINTSTORE stores the first location     }
  {number in STARARRAY used to store the current nodes    }
  {forward star}
  NUMBER : WORD5;
  {NUMBER holds the number read from INFILE , in character}
  {form}
  ENDROW,ENDCOLS : BOOLEAN;
  {ENROW = TRUE if end of row is reached i.e. a* is       }
  {detected ENDCOLS = TRUE when all rows in adjacency     }
  {matrix have been read}
  CH : CHAR;


BEGIN
  LMIN:=  INFINITY;
  RESET(INFILE);
  FOR I:= 1 TO 100 DO
    BEGIN
      POINTERARRAY[I]:= 0;
      STARARRAY[I]:= 0;
      WEIGHTARRAY[I]:= 0;
    END;  {Initialise forward star arrays}
  ENDCOLS:= FALSE;
  ROW:= 0;
  EDGEPOINTER:= 1;
  WHILE NOT(ENDCOLS) DO
    BEGIN
      ENDROW:= FALSE;
      COL:= 1;
      {EDGEPOINTER currently contains the first location }
      {number in STARARRAY that will be used to store the}
      {forward star of the next node}
      EDGEPOINTSTORE:= EDGEPOINTER;
      ROW:= ROW+1;
      WHILE NOT(ENDROW) DO
        BEGIN
          FOR I:= 1 TO 5 DO
            NUMBER[I]:= ' ';
          I:= 1;
          REPEAT  {Read the next number from INFILE}
            READ(INFILE,CH);
            NUMBER[I]:= CH;
```

```
            I:= I+1;
          UNTIL ((I = 5) OR (NUMBER[1] = '*'));
          IF (NUMBER[1] = '*') THEN
            BEGIN  {End of row detected}
              ENDROW:= TRUE;
              IF (COL=1) THEN
                ENDCOLS:= TRUE;  {End of adjacency matrix}
            END
          ELSE
            BEGIN
              IF (NUMBER[1] <> '0') THEN
                BEGIN {Insert information into the 3 arrays}
                  POINTERARRAY[ROW]:= EDGEPOINTSTORE;
                  CHARTOINT(NUMBER,VALUE);
                  IF (VALUE < LMIN) THEN
                    LMIN:= VALUE;
                  STARARRAY[EDGEPOINTER]:= COL;
                  WEIGHTARRAY[EDGEPOINTER]:= VALUE;
                  EDGEPOINTER:= EDGEPOINTER+1;
                  {set pointer to next free location in}
                  {STARARRAY}
                END;
              COL:= COL+1;  {Increment column reference}
            END;
          END;
        IF NOT(ENDCOLS) THEN
          BEGIN
            READLN(INFILE);
            READLN(INFILE);
          END; {Move to the next row of the adjacency matrix}
      END;
      NUMNODES:= ROW-1; {Record the number of nodes in the}
      {network}
      POINTERARRAY[NUMNODES+1]:= EDGEPOINTER;
      {Insert dummy pointer in POINTERARRAY}
END;  {READFORSTAR}
```

```
PROCEDURE PRINTBUCKETS(ADDED,BOOLEAN,NODENUM,INTEGER);
(This procedure displays the contents of the non-empty )
(buckets in the bucket list.  It also outputs which node)
(has just been added or removed from the bucket list)


VAR
  K,LOW,HIGH : INTEGER
  PTR : POINTER;

BEGIN
  IF (ADDED) THEN
    WRITELN('NODE',NODENUM:4,'  ADDED')
  ELSE
    WRITELN('NODE',NODENUM:4,'  REMOVED');
  WRITELN;
  WRITELN('                 NON - EMPTY BUCKETS              ');
  WRITELN('                 -------------------              ');
  WRITELN;
  FOR K:=  0 TO 1000 DO
     BEGIN
       PTR:= BUCKLIST[K];
       IF (PTR <> NIL) THEN
         BEGIN
           WRITE(K:4);
           LOW:=  K*WIDTH;
           HIGH:=  (K+1)*WIDTH;
           WRITE(LOW: 13);
           WRITE(' <= DISTANCE < ');
           WRITE(HIGH:4);
           WRITE('          ');
           REPEAT
             WRITE(PTR^.NAME:3);
             PTR:=  PTR^.NEXT;
             WRITE('    ');
           UNTIL (PTR = NIL);
           WRITELN;
           WRITELN;
         END;
     END;
End;  (PRINTEBUCKETS)
```

```
OK, PASCALG P408U>BUCKETSORT.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing BUCKETSORT

    ADJACENCY MATRIX
    ----------------

    1   2   3
    -------------
1  |0   1   0
   |
2  |0   0   3
   |
3  |4   0   0
   |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

     POINTERARRAY     STARARRAY     WEIGHTARRAY
     ------------     ---------     -----------

          1               2              1
          2               3              3
          3               1              4
          4               0              0

WHICH IS THE START NODE ?
1

EXAMINING NODE   1

NODE   2  ADDED

                    NON - EMPTY BUCKETS
                    -------------------

BUCKET         DISTANCE VALUE RANGE          NODES IN BUCKET
------         --------------------          ---------------

   0              0  <= DISTANCE <    1            1

   1              1  <= DISTANCE <    2            2

NODE   1  REMOVED

                    NON - EMPTY BUCKETS
                    -------------------

BUCKET         DISTANCE VALUE RANGE          NODES IN BUCKET
------         --------------------          ---------------
```

1	1 <= DISTANCE < 2	2

EXAMINING NODE 2

NON - EMPTY BUCKETS

BUCKET	DISTANCE VALUE RANGE	NODES IN BUCKET
1	1 <= DISTANCE < 2	2
4	4 <= DISTANCE < 5	3

NODE 2 REMOVED

NON - EMPTY BUCKETS

BUCKET	DISTANCE VALUE RANGE	NODES IN BUCKET
4	4 <= DISTANCE < 5	3

EXAMINING NODE 3

NODE 3 REMOVED

NON - EMPTY BUCKETS

BUCKET	DISTANCE VALUE RANGE	NODES IN BUCKET

DISTANCE FROM 1 TO 2 IS 1

ROUTE IS:

2 1

DISTANCE FROM 1 TO 3 IS 4

ROUTE IS:

3 2 1

```
OK, PASCALG P408U>BUCKETSORT.PAS
[Sheffield Pascal version 3.3.1b]
No errors reported.

Executing BUCKETSORT

        ADJACENCY MATRIX
        ----------------

        1    2    3
        -------------
1      |0    1    4
       |
2      |0    0    3
       |
3      |0    0    0
       |

THIS IS THE GRAPH REPRESENTED IN FORWARD STAR FORM
--------------------------------------------------

       POINTERARRAY    STARARRAY     WEIGHTARRAY
       ------------    ---------     -----------

            1              2              1
            3              3              4
            0              3              3
            4              0              0

WHICH IS THE START NODE ?
2

EXAMINING NODE  2

NODE   2  REMOVED

                           NON - EMPTY BUCKETS
                           -------------------

BUCKET              DISTANCE VALUE RANGE          NODES IN BUCKET
------              --------------------          ---------------


THERE IS NO ROUTE FROM                2 TO       1

DISTANCE FROM              2 TO              3 IS        3

ROUTE IS:

        3          2
```

APPENDIX F

This appendix contains the PASCAL codes for the program SENET, and all the procedures used in the program. A correctly formatted version of INFILE must be available before the execution of the program.

A sample run of this program is also shown in this appendix.

```
PROGRAM SENET(INPUT,OUTPUT,INFILE);
{This program first finds the shortest path tree rooted at}
{a node START in a network stored in INFILE using}
{DIJKSTRA's algorithm.  It then applies the algorithm}
{SENET, to all the possible arcs, for the purpose of post}
{optimality analysis}


LABEL  99;

CONST
  INFINITY = 99999;


TYPE
  BOARRAY = ARRAY[1..100] OF BOOLEAN;
  WORD5 = ARRAY[1..5] OF CHAR;
  ARRAY100 = ARRAY[1..100] OF INTEGER;
  ADJARRAY = ARRAY[1..100] OF ARRAY100;


VAR
  ACT : WORD5;
  ADJMATRIX : ADJARRAY;
  P1,P2,P3,d1,d2,d3, CHANGEDNODES : ARRAY100;
  NUMNODES,I,J,MINIMUM,START : INTEGER;
  MIDPOS,K,KK : INTEGER;
  DELTA : ARRAY100;
  INFILE : TEXT;


BEGIN  {MAIN}
  RESET(INFILE);
    BEGIN
      P1[I]:= 0;
      P2[I]:= 0;
      P3[I]:= 0;
      d1[I]:= INFINITY;
      d2[I]:= INFINITY;
      d3[I]:= INFINITY;
    END;
    READADJMATRIX;  {Read in the network}
    PRINTADJMATRIX  {Display the network}
    WRITELN('WHICH IS THE START NODE ?');
    READLN(START);
    WRITELN;
    P1[START]:= START;
    P2[START]:= START;
    P3[START]:= START;
    d1[START]:= 0;
    d2[START]:= 0;
    d3[START]:= 0;
    SHORTESTPATH(P1,d1);
```

```
      TRACEPATH;
      K:= 0;
      KK:= 1;
      WRITESHEAD;
      FOR I:=  1 TO NUMNODES DO
        FOR J:=  1 TO NUMNODES DO
          IF (I <> J) THEN
            BEGIN
              WEIGHT:= ADJMATRIX[I,J];
              IF(WEIGHT = INFINITY)
                THEN
                  ACT:= 'NEX  '
                ELSE
                  IF (P1[J] = I)
                    THEN
                      ACT:= 'NOP  '
              IF (ACT = 'OPT  ') THEN
            BEGIN
              ADJMATRIX [I,J]:= INFINITY;
              SHORTESTPATH(P2,d2);
              COMPARE(P2,K);
              MIDPOS:= k;
              IF (k > 0) THEN
                BEGIN
                  FOR IJ:= kk TO k do
                     DELTA[IJ]:= WEIGHT + d2[CHNGEDNODE[IJ]]
                                 -d1[CHANGEDNODE[IJ]];
                  kk:=  k+1
                END;
              ADJMATRIX[I,J]:= 0
            END;
            SHORTESTPATH(P3,d3);
            COMPARE(P3,k);
            IF (k > 0) THEN
              BEGIN
                FOR IJ:=  kk TO k DO
                  DELTA[I]:= d1[CHANGEDNODE[IJ]] -
                             d3[CHANGEDNODE[IJ]]
              END;
            IF (k > 0)
            THEN
              DESCEND;
            WRITELN;
            WRITELN;
            WRITESENET
    END
END. {MAIN}
```

```
PROCEDURE COMPARE(VAR P : ARRAY100; H : INTEGER);
{This procedure determines the nodes whose labels totally}
{change after a reoptimisation}


VAR
  I,HH:INTEGER;
  L:BOARRAY;


BEGIN
  FOR I:= 1 TO NUMNODES DO
    L[I]:= FALSE;
  HH:= 0;
  FOR I:= 1 TO NUMNODES DO
  IF(P[I] <> P1[I]) THEN
    BEGIN
      L[I]:= TRUE;
      H:= H+1;
      CHANGEDNODE[H]:= I
    END
  REPEAT
    IF (H > 0) THEN
      BEGIN
        HH:= HH+1;
        FOR I:= 1 TO NUMNODES DO
          IF ((CHANGEDNODE[HH] = P1[I]) AND
             (L[I] = FALSE)) THEN
            BEGIN
              L[I]:= TRUE;
              H:= H+1;
              CHANGEDNODE[H]:= I
            END
      END
  UNTIL(HH = H)
END;  {COMPARE}
```

```
PROCEDURE SHORTESTPATH(VAR P,d:ARRAY100);
{This procedure finds the shortest path tree rooted at }
{node START in a network stored in ADJMATRIX.  The     }
{procedure is based on Dijkstra's algorithm            }


VAR
  R,NEXT,I,J:INTEGER;
  INTREE : ARRAY100;


BEGIN
  FOR I := 1 TO NUMNODES DO
    BEGIN
      INTREE[I] := 0;
      P[I] := 0;
      d[I]: INFINITY;
    END;
  REPEAT
    MINIMUM := INFINITY;
    FOR I := 1 TO NUMNODES DO
      BEGIN
        IF ((d[I] < MINIMUM) AND (INTREE[I] = 0)) THEN
          BEGIN
            R := I;
            MINIMUM := d[R]
          END
      END;  {Find the node with minimum total weight}
    IF (MINIMUM <> INFINITY) THEN
      BEGIN
        INTREE[I] := 1;
        FOR J := 1 TO NUMNODES DO
          BEGIN
            IF (ADJMATRIX[R, J] <> 0) THEN
               IF (((d[R] + ADJMATRIX[R, J]) < d[J]) AND
                  (INTREE[J] = 0)) THEN
                  BEGIN  {Relabel node J}
                    d[J] := d[R] + ADJMATRIX [R, J];
                    P[J] := R
                  END
          END
      END;
    UNTIL (MINIMUM = INFINITY)
99 : END;  {SHORTESTPATH}
```

```
PROCEDURE TRACEPATH;
{This procedure traces the shortest paths through the tree}

VAR
  NEXT,I : INTEGER;

BEGIN
  FOR I := 1 TO NUMNODES DO
    IF (I <> START) THEN
    IF (d[I] = INFINITY) THEN
    BEGIN
      WRITELN;
      WRITELN('THERE IS NO ROUTE FROM',START,'TO',I)
    END
    ELSE
      BEGIN
        WRITELN;
        WRITELN('DISTANCE FROM',START,'TO',I,'IS',d[I]);
        WRITELN('ROUTE IS:');
        WRITELN;
        TRACKPATH(P1,I)
      END
END;  {TRACEPATH}
```

```
PROCEDURE TRACKPATH (P:ARRAY100;SINK:INTEGER);
{This procedure traces the unique tree path to a node sink}

VAR
  NEXT : INTEGER;

BEGIN
  WRITE(SINK);
  NEXT:= P[SINK];
  WHILE (NEXT <> START) DO
    BEGIN
      WRITE(NEXT);
      NEXT:=P[NEXT]
    END;
  WRITELN(START)
END;  {TRACKPATH}
```

```
PROCEDURE DESCEND;
{This procedure arranges the arrays CHANGEDNODE and DELTA}
{in DESCENDKING ORDER OF DELTA}


VAR
  KI,II,Dumd,DUMP,KJ  :INTEGER;


BEGIN
  FOR KI := 1 TO kk DO
    BEGIN
      DUMd := DELTA[KI];
      DUMP := CHANGEDNODE[KI];
      II := KI+1;
      FOR KJ := II TO kk DO
      IF (DELTA[KI] < DELTA[KJ]) THEN
        BEGIN
          DELTA[KI] := DELTA[KJ];
          CHANGEDNODE[KI] := CHANGEDNODE[KJ];
          DELTA[KJ] := DUMd;
          CHANGEDNODE[KJ] := DUMP
        END
    END
END;  {DESCEND}
```

```
PROCEDURE WRITESHEAD;
{This procedure writes the headings for SENET}


BEGIN
  WRITELN;
  WRITELN;
  WRITELN('   *****  POST-OPTIMALITY ANALYSIS');
  WRITELN;
  WRITELN;
  WRITE(' ':20,'++ THE "EFFECT" OF EACH RANGE, EXCEPT THE
       OPTIMAL AND NON-OPTIMAL, IS AN  ');
  WRITELN('ACCUMULATION OF THE "EFFECTS" OF THE OTHER ');
  WRITE('RANGES FROM THE SIGN "↑" OR "↓" TO "-" OF THE ');
  WRITELN('"ACCUMULATION" COLUMN FOR EACH ARC ++');
  WRITELN;
  WRITELN;
  WRITELN;
  WRITELN(' ':10,'ARC',' ':29,'RANGE',' ',:48,'EFFECT');
  WRITE('  identity   weight   activity',' ':10);
  WRITE('upper   lower     accumulation   node');
  WRITE('    t-weight',' ':6,'route <-------');
  WRITELN;
  WRITELN
END;
```

```
PROCEDURE WRITESENET(DUMI,DUMJ : INTEGER);
{This procedure writes the results of SENET}


VAR
  POSFLAF : BOOLEAN;
  OLDINDEX,NEWINDEX,OLDLIMIT,NEWLIMIT,I,DUM: INTEGER;
  DUMd,DUMP,DUMMY : INTEGER;


  FUNCTION NEW VALUE (IJ : INTEGER) : INTEGER;
  VAR
    DUMMY : INTEGER;
  BEGIN
    DUMMY := IJ;
    WHILE ((DELTA[DUMMY] = OLDLIMIT) AND (DUMMY <= k)) DO
      DUMMY := DUMMY + 1;
    NEWVALUE := DUMMY
  END;  {NEWVALUE}


BEGIN  {WRITESENET}
  OLDINDEX := 1;
  NEWLIMIT := INFINITY;
  POSFLAG := FALSE;
  WRITE(DUMI:3, '--->', DUMJ:3, '    ');
  IF(ACT = 'NEX  ')
    THEN
      WRITE('INF')
    ELSE
      WRITE(WEIGHT:5);
  WRITE(' ':6);
  WRITE(ACT,' ':9);
  IF (k = 0) THEN
  BEGIN
    WRITELN(' ':20,'NON-EFFECTIVE');
    WRITELN
  END
  ELSE
    BEGIN
      NEWLIMIT := INFINITY;
      WRITE('   INF');
      OLDLIMIT := NEWLIMIT;
      WRITE(' ':36);
      NEWINDEX := NEWVALUE(OLDINDEX);
      NEWLIMIT := DELTA[NEWINDEX];
      WRITE(' ':3,NEWLIMIT:5,' ':3);

      IF(OLDLIMIT = INFINITY)
        THEN
          WRITE(' ':4,'NON-OPTIMAL RANGE')
        ELSE
          BEGIN
```

```
WRITELN;
WRITELN(' ':46,'"-"');
IF((ACT = 'OPT  ') AND (OLDLIMIT < = WEIGHT)

AND (NEWLIMIT > = WEIGHT))
THEN
  WRITELN(' ':14,'****    OPTIMAL RANGE')
ELSE
  BEGIN
    IF (ACT = 'OPT  ') THEN
    BEGIN
      IF (OLDINDEX = 1)
      THEN
        WRITE(' ':6,'"↑")
      ELSE
        IF (OLDINDEX < MIDPOS)
        THEN
          WRITE(' ':6,'"4"')
    END
    ELSE
    WRITE(' ':6,'"↓"');
    DUM = NEWINDEX;
    OLDINDEX := OLDINDEX + 1;
    FOR I := OLDINDEX TO DUM DO
      BEGIN
        WRITELN;
        WRITE(' ':52);
        POSFLAG := (I < = MIDPOS) OR
                   NOT (ACT = 'OPT  ');
        DUMP := CHANGEDNODE[I];
        WRITE('  ', DUMP:5,' ':4);
        IF(CHANGEDNODE[I] = 0)
        THEN
          DUMd := 0
        ELSE
          BEGIN
            IF (POSFLAG)
              THEN
                DUMd := d2[CHANGEDNODE[I]]
              ELSE
                DUMd := d3[CHANGEDNODE[I]
          END;
       IF (DUMd = INFINITY)
          THEN
            IF (ACT = 'OPT  ') THEN
              BEGIN
                DUMd := d1[CHANGEDNODE[I]] -
                        WEIGHT;
                WRITE(DUMd:4,'+W(',DUMI:3,',');
                WRITE(DUMJ:3,')')
              END
              ELSE
                WRITE('   INF   NO ROUTE')
```

```
                        ELSE
                          BEGIN
                            IF (POSFLAG)
                              THEN
                                WRITE(DUMd:4,'+W(',DUMI:3,','
                                ,DUMJ:3,')')
                              ELSE
                                WRITE(DUMd:5,' ':11);
                            WRITE(DUMP:4,' ':2);
                            REPEAT
                              IF(DUMP <> 0) THEN
                                BEGIN
                                  DUMMY := DUMP;
                                  IF (POSFLAG)
                                    THEN
                                      DUMP := P2 [DUMP]
                                    ELSE
                                      DUMP := P3 [DUMP]
                                  IF (DUMP <> DUMMY)
                                    THEN
                                      WRITE(DUMP:4,' ':2)
                          END;
                      UNTIL (DUMP = START) OR (DUMP = 0)
                    END
                END
            END;
            WRITELN;
            WRITELN;
            WRITELN
END;  {WRITESENET}
```

```
OK, PASCALG P408U>SENET.PAS
[Sheffield Pascal version 3.3.1.b]
No errors reported.

Executing SENET

    ADJACENCY MATRIX
    ----------------

    1   2   3   4
    -----------------
1  |0   11  21  0
   |
2  |0   0   6   17
   |
3  |0   0   0   4
   |
4  |0   0   0   0
   |

WHICH IS THE STARTING NODE ?
1


DISTANCE FROM          1 TO    2 IS    11

ROUTE IS:

       2       1

DISTANCE FROM          1 TO    3 IS    17

ROUTE IS:

       3       2       1

DISTANCE FROM          1 TO    4 IS    21

ROUTE IS:

       4       3       2       1
```

***** POST-OPTIMALITY ANALYSIS

++ THE "EFFECT" OF EACH RANGE, EXCEPT THE OPTIMAL AND NON-OPTIMAL, IS AN ACCUMULATION OF THE "EFFECTS" OF THE OTHER RANGES FROM THE SIGN "↑" OR "↓" TO "-" OF THE "ACCUMULATION" COLUMN, FOR EACH ARC ++

	ARC		RANGE				EFFECT	
identity	weight	activity	upper	lower	accumulation	node	t-weight	route<-------
1---> 2	11	OPT	INF	15	"-"	3	21	3 1
						4	25	4 3 1
					"↑"	2	0+w(1, 2)	2 1
			15	0		****	OPTIMALITY-RANGE	
1---> 3	21	NOP	INF	17		****	NON-OPTIMAL RANGE	
			17	0	"↓"	3	0+w(1, 3)	3 1
					"-"	4	4+w(1, 3)	4 3 1
1---> 4	INF	NEX	INF	21		****	NON-OPTIMALITY RANGE	
			21	0	"-"	4	0+w(1, 4)	4 1
2---> 1	INF	NEX	***	***		****	NON-EFFECTIVE	

***** POST-OPTIMALITY ANALYSIS

++ THE "EFFECT" OF EACH RANGE, EXCEPT THE OPTIMAL AND NON-OPTIMAL, IS AN ACCUMULATION OF THE "EFFECTS" OF THE OTHER RANGES FROM THE SIGN "+" OR "+" TO "-" OF THE "ACCUMULATION" COLUMN, FOR EACH ARC ++

identity	ARC weight	activity	RANGE upper	RANGE lower	accumulation	node	EFFECT t-weight	route<-------		
2---> 3	6	OPT	INF	17	"-"	4	28	4	2	1
			17	10	"+"	3	21	3	1	
			10	0		****	OPTIMALITY RANGE			
2---> 4	17	NOP	INF	10		****	NON-OPTIMALITY RANGE			
			10	0	"-"	4	11+w(2, 4)	4	2	1
3---> 1	INF	NEX	***			****	NON-EFFECTIVE			
3---> 2	INF	NEX	***			****	NON-EFFECTIVE			
3---> 4	4	OPT	INF	11	"-"	4	28	4	2	1
			11	0		****	OPTIMALITY-RANGE			
4---> 1	INF	NEX	***			****	NON-EFFECTIVE			
4---> 2	INF	NEX	***			****	NON-EFFECTIVE			
4---> 3	INF	NEX	***			****	NON-EFFECTIVE			