

https://dspace.org/

A survey of the most recent state-of-the art shortest path algorithms and their applications to different types of networks.

Comninos, Gerry

1991
http://hdl.handle.net/10539/22935

Downloaded from a DSpace repository. https://dspace.org/

A SURVEY OF THE MOST RECENT STATE-OF-THE-ART SHORTEST PATH ALGORITHMS AND THEIR APPLICATIONS TO DIFFERENT TYPES OF NETWORKS.

By Gerry Comninos

This research report is submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in part fulfilment of the requirements for the degree of Master of Science

Johannesburg 1991

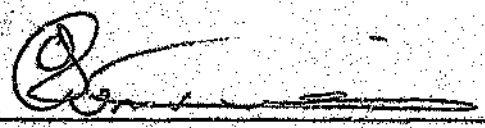
ABSTRACT

The title of this research report is a " A Survey of the most recent State-of-the-Art Shortest path algorithms"

One may ask what is the need and aim of this survey. During the decades of the 1970's and 1980's there has been considerable research and publication in shortest path algorithm. Each new publication is either a new and faster algorithm or survey of recent methods. Even with all the new advances in this field Dijkstra's method is still applied in many respects in industry. The aim of this survey is to isolate the different groups of algorithms and their efficient applications. The algorithms are the evaluated and then compared.

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.



Gerry Comninos

13 day of May, 1991.

DEDICATION

This is dedicated to my parents who have made this all possible

Table of Contents

1.0 Introduction 1

2.0 Historical Background 4

3.0 Mathematical Background of the Shortest Path Problem 6
3.1 Definition of the length of a path 9
3.2 Definition of the Shortest Path Problem 9
3.3 Linear Programming Formulation of the Shortest Path Problem 10
3.3.1 PRIMAL 10
3.3.2 DUAL 11

4.0 The Analysis of algorithms 12
4.1 Assumptions of Analysis 12
4.2 The analysis of the Algorithm 12
4.2.1 Priori Analysis 13
4.2.2 Posteriori Testing 13
4.3 Measurement of performance from the analysis 13
4.4 Priori Analysis 14
4.4.1 Asymptotic Notation 14
4.4.2 Definition 14
4.5 Theorem 16
4.5.1 The Classes of Algorithms 16
4.5.2 The Limitations of the "Big Oh" 19

THE SHORTEST PATH ALGORITHMS 20

5.0 The General Algorithm 21

6.0 The types of algorithm 24
6.1 Label Correcting Methods 24
6.2 Label Setting Methods 26
6.3 Shortest Path Algorithms and their complexity 27

7.0 Selection Criteria for Lists and Priority Queues 29

8.0 The Label Correcting Algorithms (List Search Algorithms) 31
8.1 The Bellman-Ford-Moore Algorithm 35
8.2 The D'Esopo-Pape Algorithm 37
8.3 The Pallottino 2queue Algorithm 40
8.4 The Partitioning Shortest Path Algorithm (Glover, Klingman and Phillips) 43
8.5 The Threshold Algorithm (Glover and Klingman) 46

9.0 The Concept of Sharpness 53
9.1 The Globally Sharp Algorithm (Glover and Klingman 1989) 54
9.2 The Locally Scan-Sharp Desrochers Algorithm 57
9.3 The Sharp Threshold Algorithm For non negative lengths 59

10.0 Label Setting Algorithms (Shortest First Search Algorithms) 66
10.1 Buckets and Heaps 68
10.1.1 Operations performed on Heaps 69
10.2 The Dijkstra Algorithm 70
10.3 The Ordered Priority Queue Algorithm 72
10.4 The Dial (bucket) Algorithm 74
10.5 The Binary Heap Algorithm 77

11.0 One/Two-Level Redistributive Heaps 81
11.1 The One-Level Redistributive Heap Algorithm 81
11.2 The Two-Level Redistributive Heap Algorithm 86
11.3 Further improved Redistributive Heaps 91
11.3.1 Fibonacci Heaps 92
11.3.2 Operations performed on Fibonacci Heaps 93
11.3.3 Mathematical Properties of Fibonacci Heaps 94
11.3.4 Lemma 1 95
11.3.5 Corollary 1 95

12.0 Computational Analysis of the Algorithms 97
12.1 The Algorithms implemented. 98
12.1.1 Label Setting algorithms 98
12.1.2 Label Correcting Algorithms 98
12.2 The evaluation of the algorithms 100
12.2.1 Complete Networks 101
12.2.2 Sparse networks 104
12.2.2.1 Grid Networks 104
12.2.2.2 Sparse Random Networks 107

13.0 Conclusion 128

14.0 Notation Used 130

15.0 References 132

List of Figures

Figure 2.1 The Evolution of Shortest Path Algorithms 5
Figure 3.1 Forward Star 8
Figure 4.1 Growth rate of complexity functions 18
Figure 8.1 Stack 32
Figure 8.2 Queue 32
Figure 8.3 Deque 34
Figure 8.4 2queue 34
Figure 101 Binary Heap 69
Figure 121 Complete Networks 103
Figure 122 Grid Networks 106
Figure 123 Random Sparse Networks (density = 5) 118
Figure 124 Random Sparse Networks (density = 10) 119
Figure 125 Random Sparse Networks (density = 20) 120
Figure 126 Random Sparse Networks (density = 30) 121
Figure 127 Random Sparse Networks (density = 40) 122
Figure 128 Random Sparse Networks (density = 60) 123
Figure 129 Effect of density changes on algorithms - 250 nodes 124
Figure 1210 Effect of density changes on algorithms - 500 nodes 125
Figure 1211 Effect of density changes on algorithms - 750 nodes 126

LIST OF TABLES

Table 4.1 15
Table 12.1 108
Table 12.2 127
Table 12.3 127

1.0 Introduction

The design and analysis of network flow techniques has been fertile ground from the mid 1950's for an astonishing amount of recent research in the field of Applied Mathematics (graph theory and combinatorial analysis as well as linear programming and Optimization), Discrete Mathematics, Operations Research, Business Management and Computer Science. A combinatorial approach to computer science techniques and data structures has contributed a major part to recent research.

For network flow problems finding the shortest path within the network is a fundamental problem in combinatorial optimization. Finding shortest path is often the foundation to solving many real-life large-scale network models. This would explain why there have been so many papers published in this area, resulting in a number of different algorithms solving the same problem. It is interesting to note that although there are many different algorithms there are a handful of general methods for solving the shortest path problem.

The general methods of shortest path algorithms differ in the way that the data structure for obtaining the most scan eligible is exploited. The Algorithms are generally divided into two classes, label correcting and label setting. There is a further breakdown of general methods within these two streams of shortest path algorithms. Of particular interest in the stream of label correcting algorithms is the concept of partitioning ori-

ginally introduced by D'Esopo (1974) and Pape (1980). This concept has been taken extensively further by Glover and Klingman which has lead to the sophisticated Threshold algorithm. From the label correcting type algorithms there has been an emergence of a combination of buckets, originally introduced by Dial (1963), and Heaps by Dernato and Fox (1980) and Tarjan (1988).

In this survey as in other recent surveys (Gallo and Pallottino 1986) specific algorithms representing the various methodologies have been chosen from an historical importance, computational relevance as well as from a practical performance rather than from just a theoretical analysis. Most of the algorithms presented are evaluated from both a computational as well as a theoretical view. It is also noted that the different algorithms perform better or worse for different types of networks. From a computational point of view the methods are all tested on three different classes of networks, grid, complete and, both sparse and dense random networks. These networks are generated and will vary in size up to a practical level of testing.

The most recent survey published has been that of Gallo and Pallottino (1986). The basic structure of that survey has been used to structure this survey. This survey then extends further and considers recent work done by Glover and Klingman as well as that of Tarjan and Orlin. To start with a brief history of key types of shortest path algorithms is given. This is followed by some definitions and the mathematical formulation and foundation to the problem. The analysis of the algorithms are then defined and discussed. A general algorithm which will act as a basis to the

different types of algorithms will then be presented. The different types of algorithms are then grouped. Each group of algorithms is then presented with a brief background to relevant data structures. Each algorithm within the group is presented in pseudo code followed by the worst case analysis of the complexity of the algorithm. Finally a comparison of the algorithms is made using both the theoretical worst case analysis and experimental data of computational implementation of the algorithms. The computational implementation and analysis is done using three types of graphs: complete, grid and sparse random graphs.

The survey is then concluded with a recommendation of the best methods for particular types of graphs based on the computational and theoretical analysis of the algorithms.

2.0 Historical Background

Optimising flow in network problems generally started in the early 1940's. Pioneering works are usually attributed to Hithcock (1941) and Kantorich (1942) and toward the end of the 1940's Koopmans (1947). Originally network flow problems were used to solve the transportation problems.

The 1950's saw a firm foundation for further research in the network flow problem. Algorithmic work was now specifically aimed at solving the assignment , transportation, maximum flow and pure minimum cost problem. Pioneers in this respect were G Danzig (1951) (also well known for his work in the development of Linear Programming), Flood (1953), Charnes and Cooper (1954), Greyzal (1955), Ford and Fulkerson (1956), Munkres (1957), Eisemann (1957) and Dennis (1958).

The need to specifically find the shortest path in the most efficient way became more and more apparent. In 1958 Bellman published a paper "On a routing problem" in which he formulated foundational equations to the shortest path problems. The solution to these equations has also been regarded as the solution to the shortest path problem. From this two trends of algorithm developed. In 1957 the Bellman-Ford-Moore algorithm appeared, this later became a classic model to the label correcting technique type of algorithm. In 1959 Dijkstra developed an algorithm that became a classic model for the label setting technique. This algorithm was however limited to a network with positive arc lengths.

Evolution outline of corner stone Shortest Path Algorithms follows:

The Evolution of Shortest Path Algorithms

(1958 - 1990)

Algorithm Based on Bellman's equations

Two trends of algorithms

LABEL SETTING

Moore and Dijkstra

(LOWEST SEARCH FIRST)

1959 - Dijkstra's Algorithm

- Ordered version

BUCKETS and HEAPS

1963 - Dials Bucket (k-pass) Algorithm (Radix Sorting)

1966 - Dantzig (1-pass) Algorithm (Radix Sorting)

(Buckets are used as one)

(one way linked lists)

1977 - Dernato and Fox (Buckets and pruning)

DYNAMIC BUCKETS and HEAPS

1988 - Tarjan & Orlin Et al

- One-Level Redistributive Heaps
- Two-Level Redistributive Heaps
- Fibonacci Heaps

LABEL CORRECTING

Ford

QUEUES and STACKS

1957 - Bellman-Ford-Moore (BFM) (LQueue)(Gallo 1988)

- Partitioned Queues

1974 - D'Esopo - Pape Algorithm (Pape 1980)

- Pallottino (1984)
- 2Queue (2 Queues)
- dQueue (Queue + Stack)

PARTITIONING and THRESHOLD

1983 - Glover & Klingman

- PSP Algorithm (1985)

1983 - Glover & Klingman (1984)

- THRESHOLD Algorithm (1985)

SHARPNESS BOUNDS

1989 - New Sharpness Bounds with Threshold & Partitioned Algorithm

Figure 2.1 The Evolution of Shortest Path Algorithms

3.0 Mathematical Background of the Shortest Path Problem

Graph : A configuration of a set of two or more different "points". These points may be joined by one or more "lines". The result of this configuration may be regarded as a graph.

The "points" are usually called **nodes** or **vertices** and the "lines" are called **edges** or **branches**. More than one edge can join two nodes. The **degree** of a node i is the number of edges that have node i as an end point. An edge that has a specific direction from a specific node of origin i to a node of destination j is usually called an **arc** denoted as arc(i,j)

The nodes of a graph can be thought of as points in euclidean space, where the edges or arcs are either straight or curved lines joining these points. If however, graphs are considered from an abstract point of view, a node can be considered to be any type of element. The only property that an edge has is that of defining two end points. The theory of graphs can thus be considered as the theory of combinatorial analysis.

For the context of this study, a graph will be regarded as a set of nodes N ith a corresponding set of arcs A that are represented as a configuration d.awn in a plane.

The following definitions are specific to networks in this study

Path : A path that joins node i to node j is an ordered set of edges or arcs such that, with possible exception to the first and last node, each node in the set has exactly two end points belonging to the arcs or edges.

Circuit or Cycle : If for a path from node i to node j, node i is equal to node j then the path is called a Circuit or a cycle.

A Connected Graph : A graph is said to be connected if for any two nodes i, j there exists a path from node i to node j

Tree : A graph that is connected with no cycles is a tree. If a tree has exactly n nodes then it will have exactly $n-1$ edges.

Network : A graph within which a flow can take place from one node to the other along the edges or the arcs is called a network. The direction of the arc between two nodes in a network is taken as the direction of flow. There is a limit to the capacity of flow through each particular arc known as the capacity in the network. The node from which flow is commenced is known as the source node *(s)*. The node where the flow is terminated is known as the sink node *(t)*

Forward Star : A forward star of node j is defined as the set of nodes i adjacent to node j originating from node j. A forward star is one of the most efficient ways of storing a network as a data structure on computer. The following diagram illustrates a network with its corresponding forward star representation.

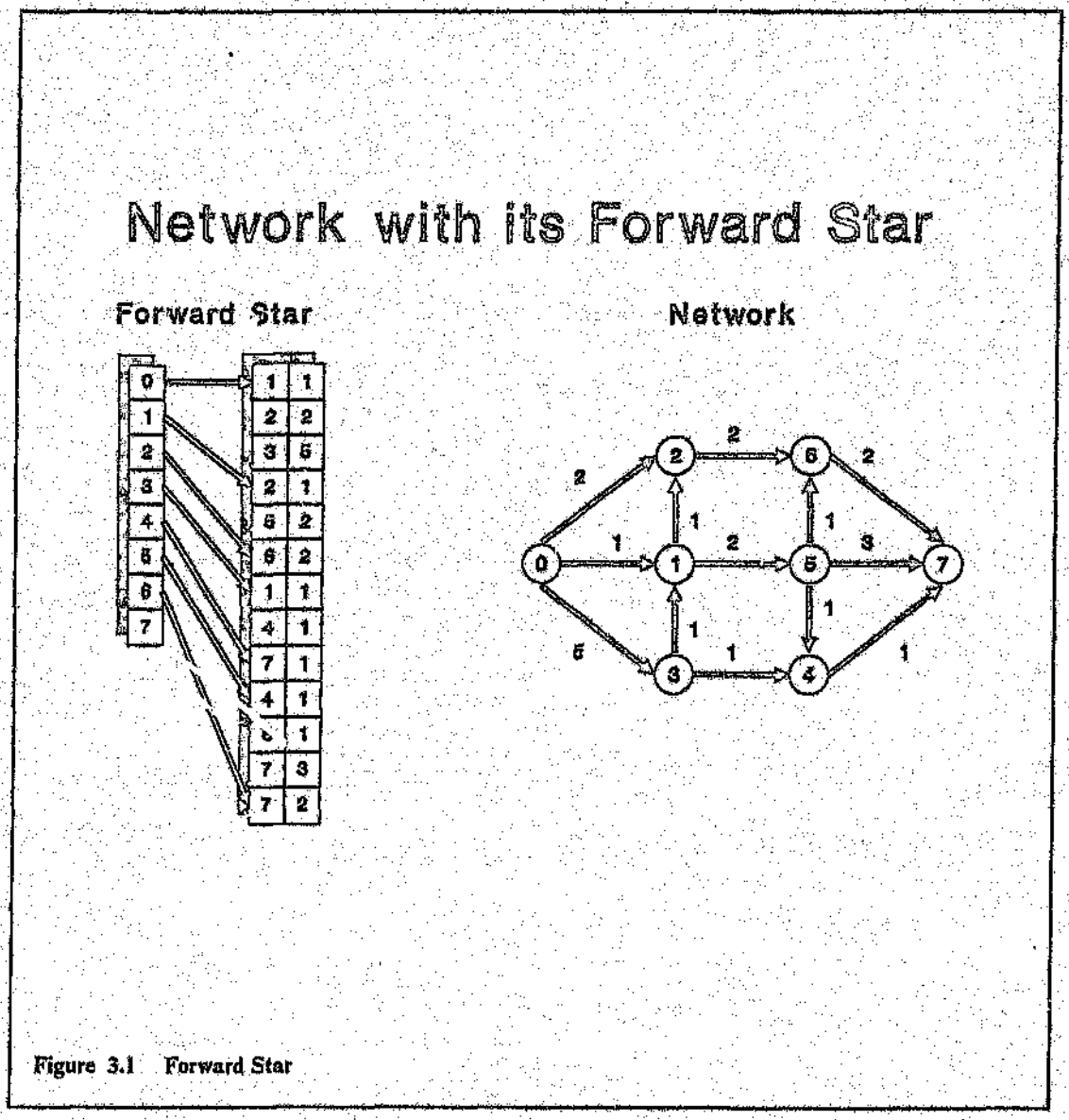


Figure 3.1 Forward Star

From the above definitions, the shortest path problem can now be formulated:

Given :

a network $G(N,A)$ where all the nodes belong to N

and all the arcs have a length or cost associated with it.

3.1 *Definition of the length of a path*

Path length is defined to be the sum of the lengths of all the arcs traced along that path.

3.2 *Definition of the Shortest Path Problem*

The shortest path problem is to find a path whose sum is the smallest from a source node (s) to a sink node (t)

The shortest path problem is also regarded as the shortest tree problem.

In solving the shortest path probem there are three main types of networks that need to be taken into consideration.

1. A network $G(N,A)$ where no arcs are non-negative ie. $A = \{a_i / a_i > 0, \ \forall \ a_i\}$

2. Should some arcs in a network $G(N,A)$ be negative there do not exist any negative cycles.

3. A network $G(N,A)$ does contain negative arcs, which can also be included in a negative cycle.

NOTE : From the above three cases it is only possible to derive algorithms to find the shortest paths for the first two cases. To derive an algorithm to find the shortest path of the third case of algorithm discussed above is impossible. It is intuitively obvious that should there be negative cycles the algorithm would continually iterate within the negative cycle to reduce the length of the path to either zero or a negative value.

3.3 *Linear Programming Formulation of the Shortest Path Problem*

As a network flow problem the shortest path problem can be formulated as a linear programming problem. The general linear programming problem (Florian, Nguyen and Pallottino 1981) is given as follows.

Given : Directed Graph $G(N,A)$

where N - Set of all nodes

A - Set of all arcs

let

$t(a)$ denote tail arc, $h(a)$ denote the head arc.

c_a - length associated with each arc

$F(i) = \{(a)/t(a) = i, a \in A\}$ - Set of arcs that originate at node i

$B(i) = \{(a)/h(a) = i, a \in A\}$ - Set of arcs that terminate at node i

3.3.1 PRIMAL

Minimise $\sum c_{(a)} f_{(a)}$ ... (i)

$$\sum_{a \in F(i)} f_{(a)} - \sum_{a \in B(i)} f_{(a)} = -1 \quad i \in N \text{ ...(ii)}$$

$$\sum_{a \in F(0)} f_{(a)} - \sum_{a \in B(s)} f_{(a)} = n - 1 \quad i \in N \text{ ..(iii)}$$

$f(a) \geq 0 \qquad a \in A$ ..... (iv)

$f(a) \leq n \qquad a \in A$

3.3.2 DUAL

let $-\pi_i$ be dual variable associated with constraint i

Maximize $\sum_j(\pi_j - \pi_r)$ .....(v)

Subject to

$c_{ij} + \pi_i - \pi_j \geq 0 \quad (i,j) \in A$ .....(vi)

Optimality conditions for this dual pair are as follows:

Primal - That which is defined from (i) to (iv). The primal feasible solution is one that satisfies (ii), (iii) and (iv).

ie. (a) Conservation of flow (ii) and (iii)
(b) $f(a) \geq 0$ (iv)

Dual -
The dual feasibility solution is one that satisfies the set of dual variables π_i that satisfy (vi)

ie. Restricted Dual Feasibility
$(c_{ij} + \pi_i - \pi_j \geq 0)$

4.0 The Analysis of algorithms

4.1 Assumptions of Analysis

As a basis to the analysis of algorithms certain assumptions about the type of computer system must be made. For the purposes of this research it is assumed that a "conventional computer" is used. It is also assumed that the following criteria characterise the "conventional computer".

1. Instructions are carried out one at a time (Von Neumann machine)

2. The greatest overhead in the execution of an algorithm is the amount of instructions carried out.

3. Random access memory is available which allows us to retrieve and replace any variable in a fixed time.

4. No parallelism is taken into account.

4.2 The analysis of the Algorithm

The analysis of an algorithm is broken up into two basic steps :

1. Priori Analysis

2. Posteriori Testing

4.2.1 Priori Analysis

The main criteria used to evaluate the algorithm in this type of analysis are the type of operations that are executed. These operations are divided into two types:

1. The four basic operations on integers. ie. addition, multiplication, division and subtraction.

2. Arithmetic on floating point numbers. ie. comparisons, assigning values to a variable, executing procedure calls.

The time taken to execute the type of operations defined above is regarded as fixed. The execution time of each operation is thus regarded as bounded by a constant.

4.2.2 Posteriori Testing

To perform this type of evaluation it is important to understand the workings of the algorithm. Data must be created that will test the best and worst performance of the algorithm.

4.3 Measurement of performance from the analysis

Priori Analysis :
Obtain a function that bounds the algorithms computing time

Posteriori Testing :
Actual statistics of the algorithm is collected. These statistics provide actual time required to run particular data as well as the space required while it was executing.

4.4 *Priori Analysis*

4.4.1 Asymptotic Notation

When evaluating computing time of an algorithm using priori analysis, all the factors that depend on a specific machine or programming language are ignored.

The general format of this type of analysis is to develop a function that represents the amount of operations to be performed, each operation bounded by a fixed time. The most common mathematical notation used is called the **O-Notation ("Big Oh").**

4.4.2 Definition

$$f(n) = O(g(n))$$

f of n equals big oh of g and h iff there exists
two positive constants c and n_0 such that
$|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

For the function $f(n)$ of computing time the variable n is determined by the various operations and iterations thereof. It may thus be the amount of inputs or outputs or the sum of both of them.

Algorithms should thus be compared at the pseudocode stage when using an a priori analysis. At the pseudocode stage, certain factors about the hardware and software are unknown. This makes it impossible to determine the exact resource requirements of a real implementation. To avoid

implementation-dependent factors that are based on the input size n when determining the algorithm complexity, constant factors are eliminated from the analysis of the algorithm.

It is noted that $f(n)$ is machine dependent. This makes it impossible to use an a priori analysis alone to determine it. A function $g(n)$ where $O(g(n)) = f(n)$ may be determined by an a priori analysis. An algorithm with computing time $O(g(n))$ will be machine independent, providing that the data is the same with increasing values of n, less than some constant $|g(n)|$

More specifically $f(n)$ is $O(g(n))$ for $c,i \geq 0$ such that $f(n) < c \times g(n)\ \forall\ n > i$. This means that assuming there is a large enough data input set n the function $f(n)$ will grow no faster than a constant factor faster than $g(n)$ To further explain this consider the following example:

$f(n)$	$O(g(n))$
30	$O(1)$
$0.6 \times n^2 + 7$	$O(n^2)$
$13 \times n^3 - 7$	$O(n^3)$
$n^3 + n^2 - 12$	$O(n^3)$

The exact values of the constants are insignificant when considering $f(n) < c \times g(n)$ Note however this relationship may not hold for specifically small size of data input n

NOTE : To determine the order of the magnitude $f(n)$ the smallest $g(n)$ such that $f(n) = O(g(n))$ will be obtained.

The following theorem states that if the number of executions of a particular statement in an algorithm can be expressed as a polynomial such as $A(n)$, then the computing time of that statement is $O(n^m)$

4.5 Theorem

IF $A(n) = a_m n^m + \ldots\ldots\ldots + a_1 n + a_0$ is a polynomial of degree m then $A(n) = O(n^m)$

The proof of this theorem can be found in Horowitz and Sahni (1987)

4.5.1 The Classes of Algorithms

The following five different classes of algorithms are important with respect to this study. These algorithms are classed by the following time required to execute them

1. Constant time - $O(1)$

2. Logarithmic time - $O(\log(n))$

3. Linear time - $O(n)$

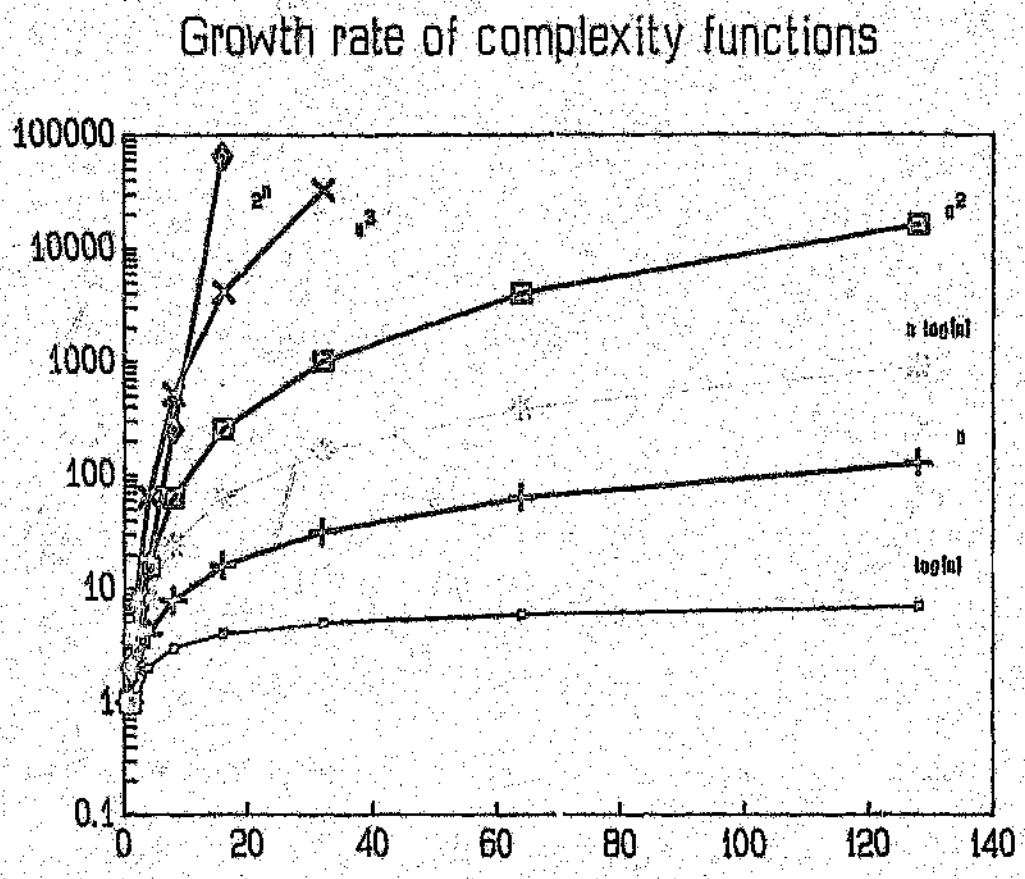
4. Polynomial time - $O(n^k)$ for fixed $k > 1$

5. Exponential time - $O(k^n)$ for fixed $k > 1$

Another common algorithmic time requirement is $O(n \log(n))$. From this type of classification it can been seen that computing times of different algorithms are related as follows

$$O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) < O(2^n)$$

The rate of growth of the above algorithms with respect to the input size is seen in figure 4.1 The type algorithm with the least computational time requirement would thus be that of constant time $O(1)$. This algorithm will be completed within a time regardless of the data input size. On the other end of the scale for polynomial algorithms the resource requirements needed to complete the algorithm will grow exponentially as the size of the input data grows. Exponential time algorithms, such as $O(k^n)$, $O(n^n)$ and $O(n!)$ become more and more worthless since as the size of the input data increases the computing time required to complete the algorithm becomes impractical.


Growth rate of complexity functions
100000
10000
1000
100
10
1
0.1
0
20
40
60
80
100
120
140
2^n
n^3
n^2
n log(n)
n
log(n)

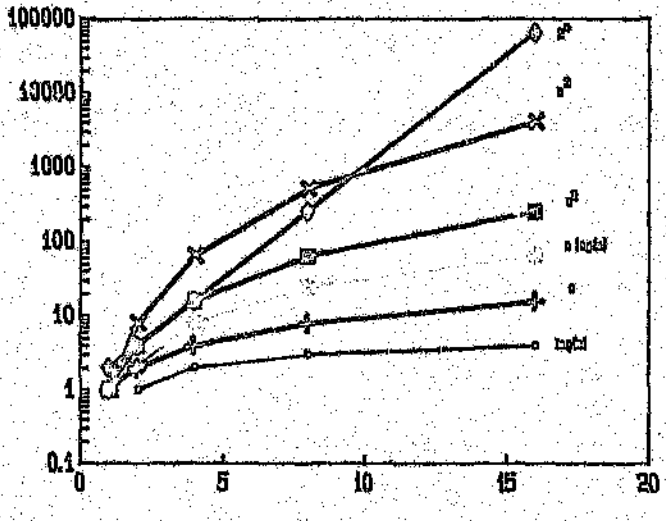

100000
10000
1000
100
10
1
0.1
0
5
10
15
20

Figure 4.1

4.5.2 The Limitations of the "Big Oh"

When analysing an algorithm using the Big Oh it must be noted that the function developed of the upperbound of the time required to complete the algorithm is that of the theoretical worst case during the execution of the algorithm. This theoretical worst case bound may be deceiving though, since a particular algorithm with a high worst case upperbound (ie. a high $O(f(n))$) may on average perform much better than an algorithm with a lower worst case upperbound for specific input data. Certain computer science techniques, such as dynamic data structures may be able to exploit the format or structure of the input data. Never the less the Priori analysis with particular reference to the Big Oh notation can provide generally good evaluation of the computing time required for specific algorithms. The Big Oh may also provide useful and imperative information about the structure of an algorithm.

The Shortest Path Algorithms

5.0 The General Algorithm

The algorithm presented is a general outline of the basic framework on which all shortest path algorithms are based. The algorithm is usually broken down into three steps :

1. The Initialising step

2. The Scanning and Updating of a selected node

3. The Iterating step

The algorithm is given as follows :

Given : Directed graph $G(N,A)$ with root node r
d_{ij} - distance of arc(i,j)

STEP 1 : Initialisation

In general all the nodes are initialised with the following values

$d_i = \infty \quad i \neq r$
$d_r = 0$

For every vertex d_v denotes the length of the path from r to v

5.0 The General Algorithm

The algorithm presented is a general outline of the basic framework on which all shortest path algorithms are based. The algorithm is usually broken down into three steps :

1. The Initialising step

2. The Scanning and Updating of a selected node

3. The Iterating step

The algorithm is given as follows :

Given : Directed graph $G(N,A)$ with root node r
d_{ij} - distance of arc(i,j)

STEP 1 : Initialisation

In general all the nodes are initialised with the following values

$d_i = \infty \quad i \neq r$
$d_r = 0$

For every vertex d_v denotes the length of the path from r to v

Now create a list of Scan Eligible (SE) nodes usually denoted by Q. This list would initially consist of the node r

ie. $Q = \{r\}$

STEP 2 : Select and Update node u

Selection :- Select u from Q the SE list. Should Q be empty the shortest path problem has been solved and for every node i the label d_i is the value of the shortest path from r to i

Once u has been selected from Q the forward star of u is scanned and updated. Most algorithms will scan the whole forward star, however this is not always the case.

Scanning :- For each $(u,v) \in FS(u)$ such that $d_u + d_{uv} < d_v$
DO

$d_v = d_u + d_{uv}$

$Q = Q \cup \{v\}$

STEP 3 : Iterate

IF Q is not empty THEN goto step 2
ELSE STOP.

NOTE from Step 3 a node from the SE nodes Q must be selected for step 2. This is the area in which most optimisation occurs. The majority

of algorithms under consideration today, will evaluate the complete forward star in one iteration, where as some will scan only selected nodes from the forward star.

The crucial point is the selection of a node u from a set of candidate (SE) nodes Q.

6.0 The types of algorithm

The selection criteria of nodes belonging to the forward star $FS(u)$ has given rise to two different classes of shortest path algorithms.

1. Label Correcting

2. Label Setting

6.1 Label Correcting Methods

The label correcting algorithm is attributed to Ford algorithm and is based on the idea of selecting a node from a Queue or a Stack. The distinct characteristic of the label correcting algorithm is that the labels of the nodes are updated at each iteration. Other general characteristics of the label correcting algorithm are as follows

- Label correcting algorithms are more general than label setting in that they are not restricted to non-negative arc lengths and may start with any tree.

- Rather than having to select the node with minimum distance label from the SE set of nodes Q any node may be selected from Q

- Label correcting methods differ in the way nodes are selected from the SE set of nodes Q

The principle of this algorithm is, at each iteration to improve that tree T by reducing the distance label of at least one node in T. This would be done by

1. Adding at least one node or arc to T

2. The replacement of an existing arc which would shorten an existing path to a node already existing in T

3. Updating a non-sharp distance label without adding a new arc or node to T

The change of selection criteria of a node from the SE set of nodes Q is in complete contrast to the fundamental property of the label setting methods. The feature that T is always the shortest path tree currently for all the nodes in T is thus lost in the label correcting methods. This would imply that label correcting methods may not necessarily be polynomially bounded or sharp. Theoretical analysis would thus indicate that label c recting methods may be far inferior to their label setting counterparts. (Glover, Glover and Klingman 1984, 1981, 1985 Dial et al 1979) have shown through empirical studies that for sparse or grid networks, the effort required to select the minimum distance label from the SE set of nodes Q is often outweighed when taking into account the advantage of the structure grid and sparse networks using label correcting methods. The first polynomial bounded label correcting algorithm known as the Bellman-Ford-Moore (B-F-M) is usually attributed to Bellman (1985), Ford (1956) and Moore (1959).

6.2 *Label Setting Methods*

The original label setting algorithm is attributed to Dijkstra (1959) and Whiting and Hillier (1960). The label setting algorithm is based on the assumption that $d_{ij} \geq 0 \; \forall \; (i,j) \in A$. The general algorithm of label setting methods is to start with a tree T consisting of a root node r. The distance of the root node is initially set to zero. All the other nodes are initialised to a distance value of infinity. The principle of the algorithm is to start with the root node. The entire forward star of the root node is then scanned. The arcs are examined to determine if any distance labels can be improved. The node with the shortest arc distance is selected to ensure this node need not be scanned again. This is a fundamental property of all label setting methods that is based on the following proposition (Gallo and Pallottino 1984, 1984)

> **Proposition 1 :** If $d_{ij} \geq 0, \; \forall \; (i,j) \in A$, then each node is removed from (and inserted into) Q exactly once.

The above proposition holds, provided there are no negative arc lengths, since if u is a minimum label element of the SE set of nodes Q then d_u is the shortest distance from the root node r to u.

It follows thus that for the tree T is always the shortest path tree for all nodes currently in T at each iteration. Label setting algorithms only scan nodes with sharp distance labels. The most expensive operation is that of determining the node u which minimises d_j rather than the temporary label set.

6.3 *Shortest Path Algorithms and their complexity*

The complexity of the shortest path algorithms will be derived with reference to the general algorithm. The complexity will be given as a function of the operations performed. The complexity of the different algorithms will be given as a function of n (the number of nodes in the network) and m (the number of arcs in a network). Based on the notation used in Gallo, Pallottino (1986) the complexity of each step in the general algorithm will be denoted as follows:

q_0 - Step 1, Initialisation of Q

q_1 - Select and removal of node u from Q

q_2 - Insertion of node v into Q

c_1 - The number of times step 2 is performed (The number of iterations ie each time a new forward star is examined.)

c_2 - The number of label improvements (The number of times that a label is put into Q)

c_3 - Max number of selection of a single node from Q

The complexity of the general algorithm will thus be $O(q_0 + q_1 \times c_1 + q_2 \times c_2)$

NOTE : $c_3 \geq \max \{h(v): v \in N\}$ where $h(v)$ is the number of subsets that the forward star $FS(v)$ is partitioned into.

$c_3 \geq 1$

$c_1 \leq n \times c_3$ and $c_2 \leq \min \{m_{FS}(U) \times c_1, m \times c_3\}$ where

$m_{FS}(u) = \max \{FS_k(v): 1 \leq k \leq h(v), v \in N\}$

For Label Setting algorithms, from proposition 1, $c_3 = 1$ So in general $c_1 = n$ and $c_2 = m$ for label setting algorithms.

For label correcting algorithms $c_3 \geq 1$ and so for most label correcting algorithms $c_1 = n \times c_3$ and $c_2 = m \times n$ The nature of label correcting algorithms ensures that $q_1 = q_2 = constant$ since there are no specific criteria to chose and remove or insert a node.

7.0 Selection Criteria for Lists and Priority Queues

The main point of interest in the General Shortest Path Algorithm is that of the selection of a node u from the set of Scan Eligible nodes Q. The data structures that are most commonly used to store and manipulate Q are those commonly known as a List and a Priority Queue. A typical list is a sequence of elements. The first element of a list is known as the *head* of the list and the last element is regarded as the *tail* of the list. The elements of a list can be manipulated with the following typical operations:

Addition : The addition of a new element will form a new head to the list. The addition of an element could also form a new tail to the list.

Removal : The removal of an element involves retrieving the element and deleting it either from the head or the tail of the list.

Concatenating : Two lists can be concatenated by linking a pointer from the tail of one list to the head of another.

Insertion : An element can be inserted after an element in a specific location

Deletion : An element can be deleted from a specific location in the list

There are two common search strategies used when manipulating lists with reference to shortest path algorithms:

1. Breadth First Search, commonly known as First-In-First-Out (FIFO)

2. Depth First Search, commonly known as Last-In-First-Out (LIFO)

The Label Correcting Algorithms make use of these two strategies and any other strategies derived from them.

A Priority Queue is a collection of elements that each have an associated numerical value. The basic operations that are performed to a priority queue are that of the addition of a new element, the removal of the maximum or minimum element or replacement or correction of an element in its correct location.

The most common strategy by used shortest path algorithms when Q is implemented as a Priority Queue is that of a Best First Search. Specifically for shortest path algorithms the best first search involves searching for the element with the minimum value. Priority Queues with the Best First Search are used in the implementation of Q by Label Setting Methods.

8.0 The Label Correcting Algorithms (List Search Algorithms)

Before further expansion on particular algorithms, it would be a good idea to pursue the various types of lists used by these algorithms. The two fundamental list structures that are used as defined or as a part of or a combination of each other are the concepts of a *Stack* and a *Queue* and are defined as follows:

Stack : A stack is a list that is characterised by the Depth-First Search (LIFO). The idea is to add and remove elements onto and from the head of the stack. If the elements A,B,C and D are added to the stack in that order, they will be removed in the order D,C,B and A. The terminology used for the addition of elements to the stack is to PUSH an element onto the stack. When removing an element from the stack it is POPPED from the stack. See figure 8.1

Queue : A queue is a list that uses the Breadth-First Search (FIFO) strategy. A new element is added to the tail of the queue while all the elements are removed from the head of the queue. If the elements A,B,C and D were added to the queue in that order, they would be removed in the same order. See figure 8.2

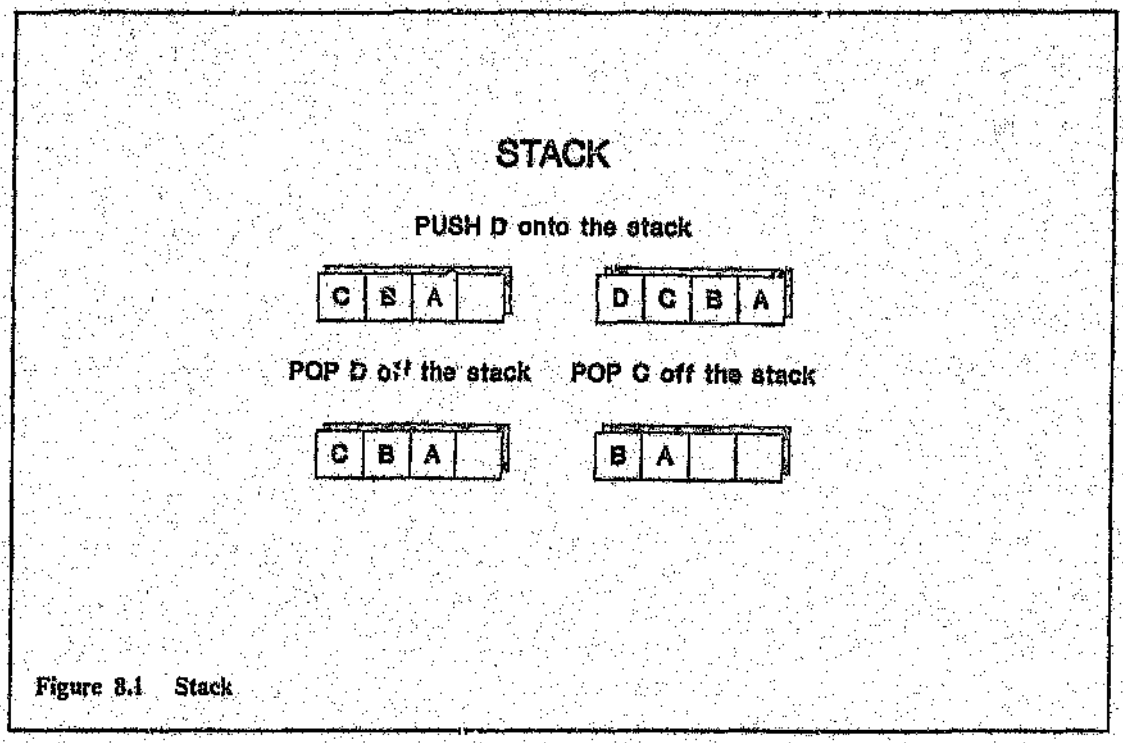


Figure 8.1 Stack

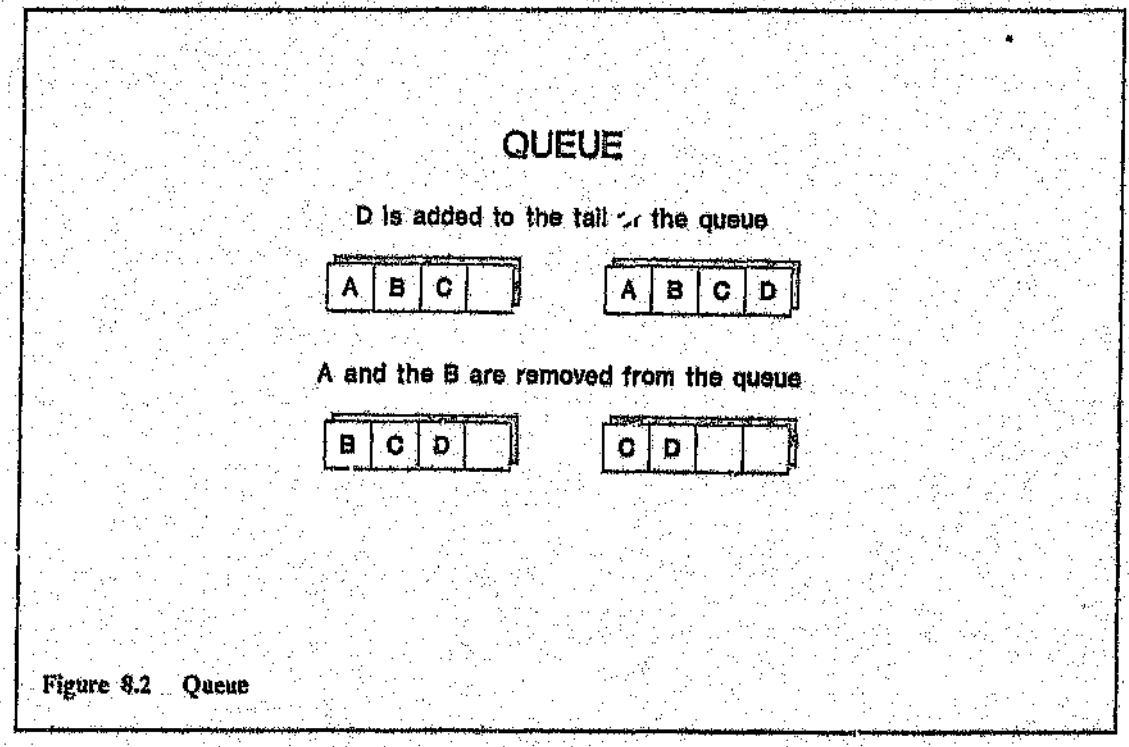


Figure 8.2 Queue

Two other lists that have been derived as a combination of the two types defined above commonly used in shortest path algorithms are *Deque* and *2queue* both defined Knuth vol 1 (1973), Horowitz and Sahni (1987) and Pallottino (1984) and are defined as follows:

Deque : This is a list that acts as double ended queue. Additions and deletions to the queue can be done at either ends. In the context of the shortest path algorithms that implement Q as a deque list, addition are made at both ends of the list while deletions are only made at the head. Deque can be thought of as a stack and queue connected in series. The tail of the stack would point to the head of the queue. See figure 8.3

2queue : 2queue is a list that consists of two queues connect in series. Elements can thus be added to the tail of both queues but remove only from the head of the one queue. See figure 8.4

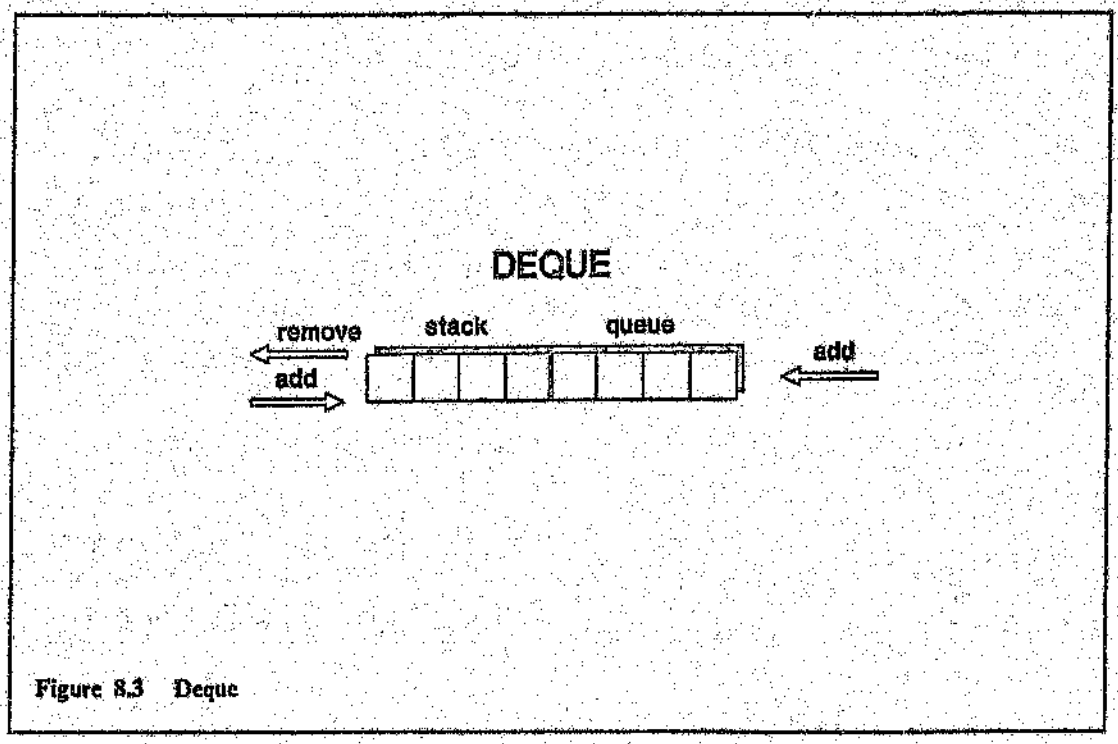


Figure 8.3 Deque

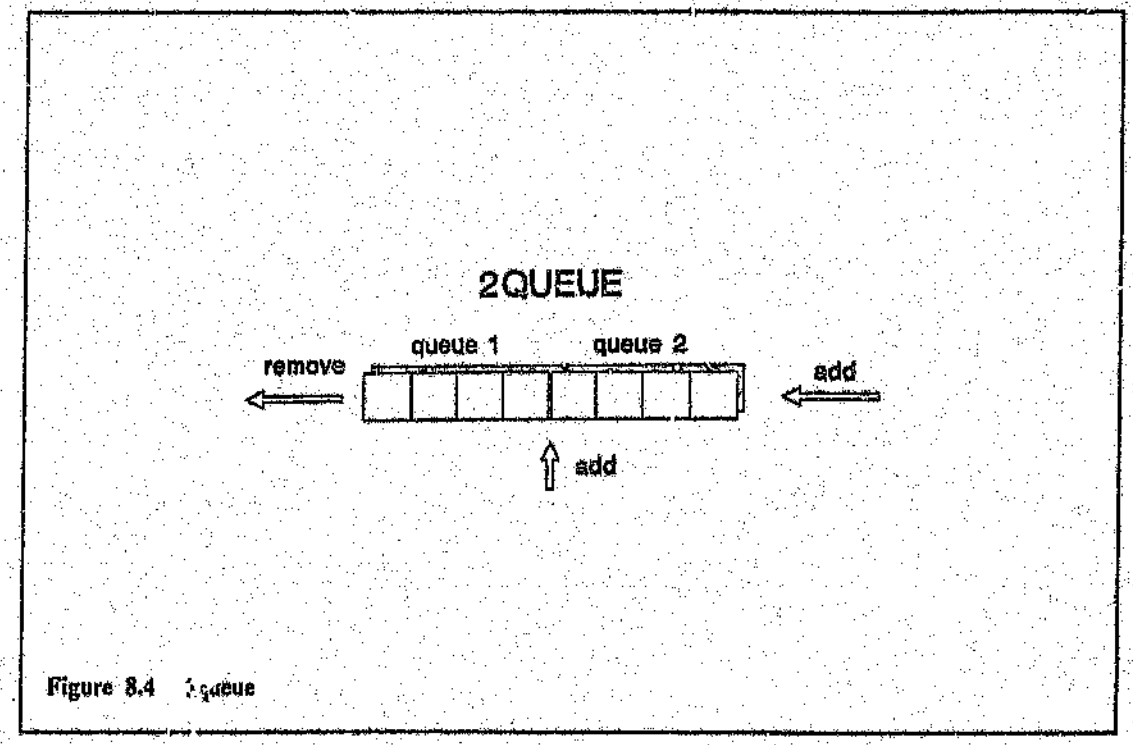


Figure 8.4 2queue

8.1 *The Bellman-Ford-Moore Algorithm*

The first algorithm using a list to implement the set of SE nodes Q to be considered has been generally credited to Bellman, Ford and Moore. This algorithm uses the common definition of a queue as the SE list of nodes Q

The procedure of this algorithm based on the general algorithm is as follows:

Procedure B-F-M (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as the queue of SE nodes

STEP 2 *Select and Update node u*

Node u is removed from the head of Q

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in Q

THEN v is inserted into the tail of Q

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2
ELSE STOP

End

An efficient implementation of the queue in the above algorithm is to use a one way linked list.

Computational Complexity

Theorem 1
The Bellman-Ford-Moore algorithm has a worst case complexity of $O(n \times m)$

Proof Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : Since the selection and removal of the node u is done using a FIFO queue it is done in constant time. So q_1 = constant, say x

Inserting v into Q : The node v is just added to the list Q so q_2 = constant, say y.

The number of label improvements : At most the label improvements during the exploration of a forward star can consist of m arcs, . and since each node can be explored as a forward star at the most n times thus $c_2 = O(m \times n)$

Number of iterations of Step 2 : Each node can be examined as a forward star at the most n times, the algorithm can iterate at most n^2 times. $c_1 = O(n^2)$

The complexity of the B-F-M algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + n^2 \times x + m \times n \times y) = O(n^2 + n \times m) = O(n \times m)$$

8.2 *The D'Esopo-Pape Algorithm*

This algorithm uses the idea of a *double-ended queue, deque* as it is defined above as an implementation of the SE list of nodes Q. The idea originally by D'Esopo was implemented by Pape (1980) with superior results to that of the common queue used in the B-F-M algorithm. The reasoning in using a double-ended queue is to split the Scan Eligible nodes into two classes :

1. The unlabelled nodes ie. Those nodes that have not yet entered Q. Nodes whose distance values still retain the initial value of infinity. These nodes are inserted at the end of Q.

2. The labelled and already scanned nodes ie Those nodes that already have passed through Q. Nodes whose distance values have already been updated at least once. These nodes are inserted at the beginning of Q.

The advantage of this method is that temporary minimal distances that still need correcting are often corrected as soon as they are detected without being allowed to progress further.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure D'Esopo - Pape (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as the queue of SE nodes

STEP 2 *Select and Update node u*

Node u is removed from the head of Q

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in Q THEN

IF

$d_v = +\infty v$ is inserted into the tail of Q

ELSE v is inserted into the head of Q

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE STOP

End

An efficient implementation of the queue in the above algorithm is to use a one way linked list of pointers with two additional pointers to indicate the two ends of Q.

Computational Complexity

Theorem 2

The D'Esopo-Pape algorithm has a worst case complexity of $O(n \times 2^n)$

Proof

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : Since the selection and removal of the node u is done using both a queue and a stack and it is removed only from the head of the stack it is done in constant time. So q_1 = constant, say x

Inserting v into Q . The node v is just added to the list Q so q_2 = constant, say y.

The number of distance labels that will be updated : It is shown Gallo and Pallottino (1988), Kershenbaum (1981) that at most 2^n distance labels can be updated thus $c_2 = O(n \times 2^n)$

Number of iterations of Step 2 : Since it is shown Gallo and Pallottino (1988), Kershenbaum (1981) that at most 2^n distance labels can be updated the algorithm will iterate at most $n \times 2^n$ times. $c_1 = O(n \times 2^n)$

The complexity of the D'Esopo-Pape algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + 2^n \times x + n \times 2^n \times y) = O(n \times 2^n)$$

8.3 *The Pallottino 2queue Algorithm*

An adaptation of the D'Esopo-Pape Algorithm was done by Pallottino (1979) which is similar in performance. The difference between the two algorithms is that the double-ended queue is divided into two queues Q' and Q'' connected in series. The Scan Eligible nodes are again divided as before except that both are inserted into the tail of both queues.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure 2queue (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialized as the queue of SE nodes

STEP 2 *Select and Update node u*

Node u is removed from the head of Q

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in Q THEN

IF

$d_v = +\infty v$ is inserted into the tail of Q

ELSE v is inserted into the tail of Q

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE STOP

End

The most noticeable difference between the D'Esopo-Pape and 2queue Algorithm is that of the worst case complexity. In practice however as shown by Pallottino (1984) both algorithms seem to be as fast.

Computational Complexity

Theorem 3

The Pallottino 2queue algorithm has a worst case complexity of $O(n^2 \times m)$

Proof

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : [illegible] removal of the node u is done using a doubl[illegible] done in constant time. So q_1 = constant, say x

Inserting v into Q : The node v is just added to the list Q so q_2 = constant, say y.

The number of distance labels updated : At most the forward star can consist of m arcs, and since each node can be explored as a forward star at the most n^2 times (Pallottino (1984)) thus $c_2 = O(m \times n^2)$

Number of iterations of Step 2 : Each node can be examined as a forward star at the most n^2 times, the algorithm can iterate at most n^3 times. $c_1 = O(n^3)$

The complexity of the 2queue algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + n^3 \times x + m \times n^2 \times y) = O(n^3 + n^2 \times m) = O(n^2 \times m)$$

8.4 *The Partitioning Shortest Path Algorithm (Glover, Klingman and Phillips)*

As already introduced in the D'Esopo - Pape and 2queue algorithm the idea of partitioning the SE list of nodes Q seems to be a viable technique in improving the computational complexity of label correcting methods. This technique has also shown superior results through empirical studies. Dial, Glover, Karney and Klingman (1979), Glover, Glover and Klingman (1981, 1984)

Glover, Klingman and Phillips (1985) developed the partitioned idea by actually separating the SE list of nodes Q into two separate queues *NOW* and *NEXT*. This algorithm will be regarded as the Partitioning Shortest Path Algorithm (PSP). The purpose of the algorithm is best described by the following quotes from the Glover, Klingman and Phillips (1985) : " The PSP algorithm explicitly identifies the partitioning concept and provides an alternative foundation for computational complexity analysis." The division of the nodes is on the same bases of that the D'Esopo - Pape and 2queue algorithm. " The PSP algorithm also explicitly identifies an efficient node scan procedure; that is, only nodes whose distance labels have been improved are scanned, and improved distance labels are used as soon as they are determined. "

The steps of the PSP algorithm are as follows :

NOTE : An extra step (STEP 4) has been included to the general shortest path algorithm to repartition the SE nodes.

Procedure PSP (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as two mutually exclusive and collectively exhaustive lists of SE nodes called *NOW* and *NEXT*

NOW is initialised with the root node r and

NEXT is initialised as an empty set.

Set $k = 0$

STEP 2 *Select and Update node u*

NOW is now scanned : IF empty goto STEP 4

ELSE select any node u from *NOW*

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in *NEXT* or *NOW*

THEN add v to *NEXT*

End;

STEP 3 *Iterate*

When the whole forward star of u has been explored return to to step 2

STEP 4 *Repartition Scan Eligible Nodes and Iterate*

IF *NEXT* is empty stop

ELSE set $k = k + 1$ and transfer all the nodes from *NEXT* to *NOW* and return to step 2.

End

This is a polynomially bounded algorithm. The key factor in deriving the polynomial complexity of this algorithm is the partitioning of the SE list of nodes denoted by Q in the general shortest path algorithm. Partitioning the SE list of nodes gives this algorithm the fundamental property of an upper bound to the number of times that nodes are transferred from the one set of SE nodes, *NEXT* to the other set *NOW*. This is due to the fact that partitioning guarantees at least one node has its distance label permanently set at each iteration. Although the original Threshold Algorithm by Glover and Klingman (1981) appeared before that of the PSP algorithm, the PSP algorithm servos as a good basis to the concept of the Threshold algorithm.

Computational Complexity

According to a result stated and proved in Glover, Klingman and Phillips (1985) the PSP algorithm has a complexity of $O(m \times n)$.

8.5 *The Threshold Algorithm (Glover and Klingman)*

As introduced in the PSP algorithm the SE list is maintained in two parts, a NOW list and a NEXT list. The Threshold algorithm is an extension of the idea of splitting the SE nodes into two lists. The idea is to separate the SE nodes by means of a threshold value t computed from the existing SE nodes. The basic outline of the Threshold algorithm is given as follows:

To start with all the SE nodes are placed on the NEXT list. From this list a threshold value t is computed. Nodes from the NEXT list with a distance value less than that of the threshold value t are transferred to the NOW list. At this stage the algorithm starts the scanning and updating process, corresponding to the second step of the general Shortest Path Algorithm. The NOW acts as the SE list of nodes defined as Q in the general algorithm. The scanning of the NOW list is done using the FIFO (Breadth-First) search strategy. (ie. NOW is implemented as a common queue). During the iteration of step 2 any nodes that become scan eligible for the first time are placed on the NEXT list. A later variation of the threshold algorithm is to place any new Scan Eligible node that is less than the current threshold value t immediately on the NOW list. Once the NOW list has been completely scanned, all the elements have been removed and updated, a new threshold value is computed. The new threshold value would either be computed directly from the elements in the NEXT list or from update rules that will be defined later. Before the threshold value can be computed from NEXT, certain criteria to be defined later, must be met. Should this criteria not hold the specific updating rules will be used to compute the threshold value t The algorithm continues to iterate by transferring elements from the NEXT

list that are less than the threshold value t to the NOW list and the algorithm is taken back to the start of step 2.

The following parameters and rules were originally defined by Glover, Glover and Klingman (1984) :

MIN = Minimum distance label for the nodes on NEXT

AVG = The average distance of the distance labels for the nodes on NEXT

MINWT = a positive weight applied to MIN

AVGWT = a positive weight applied to AVG

WTCNG = a positive "weight change factor" applied to AVGWT

The general formula to compute the threshold value t is given by

$$t = (\text{MINWT}*\text{MIN} + \text{AVGWT}*\text{AVG})/(\text{MINWT} + \text{AVGWT}) \quad \text{... (i)}$$

To start with the parameters MINWT and AVGWT are related as follows

$$\text{MINWT} + \text{AVGWT} = 100$$

This is only an initial relationship as MINWT is regarded only as an input parameter. The initial value of AVGWT is thus computed as 100 -

AVGWT. As the algorithm proceeds the value of AVGWT changes, while that of MINWT retains its initial input value.

The other value that is applied to AVGWT is that of the "weight change factor," WTCNG. This is a "(very) rough interpretation as a target percentage of nodes on NEXT that one desires to transfer to NOW ".

When nodes are transferred from NEXT to NOW, AVGWT is revised by the rule

$$AVGWT = AVGWT * (WTCNG/100)*(|NEXT|/|NOW|) \quad \text{... (ii)}$$

The following modifications of the value of AVG are also observed :

$$\text{IF } 7 *|NOW| \leq |NEXT| \text{ THEN } AVG = 1.1 * AVG \quad \text{...(iii)}$$

$$\text{IF } 1.3 *|NOW| \geq |NEXT| \text{ THEN } AVG = 0.9 * AVG \quad \text{...(iv)}$$

The steps of the Threshold Algorithm are as follows :

NOTE : An extra step (STEP 4, 5 & 6) has been included to the general shortest path algorithm to repartition the SE nodes and calculate the threshold value.

Procedure Threshold (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as two mutually exclusive and collectively exhaustive lists of SE nodes called *NOW* and *NEXT*

NOW is initialised with the root node r and

NEXT is initialised as an empty set.

Set $k = 0$

STEP 2 *Select and Update node u*

NOW is now scanned : IF empty goto STEP 4

ELSE select any node u from *NOW*

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in *NEXT* or *NOW*

THEN add v to *NEXT*

End;

STEP 3 *Iterate*

When the whole forward star of u has been explored return to to step 2

STEP 4 *Repartition Scan Eligible Nodes and Iterate*

IF *NEXT* is empty stop

ELSE

Begin

the first time that this step is reached, ie after scanning $u = r$ Goto step 5 to compute the initial value of t

IF the scanned node u in question is not equal to r

THEN transfer the nodes in the list NEXT that are less than or equal to the current threshold value t.

IF no nodes have been transferred then goto step 5

ELSE goto step 6

End

STEP 5 *Compute MIN and AVG from NEXT*

MIN and AVG are computed from the elements in NEXT that are less than or equal to the current value of t. This would mean that at the very least nodes that are equal to value of MIN would be transferred from NEXT to NOW.

STEP 6 *Compute the threshold value t and iterate*

Use the relation (ii) to re-evaluate the "weight change factor" AVGWT and use rule (iii) and (iv) to compute the new value of t and return to step 2.

End

Computational Complexity

According to Glover, Glover and Klingman (1984) the computational complexity is $O(n^2 \times m)$

Theorem 4

The Threshold algorithm has a worst case complexity of $O(n^2 \times m)$

Proof

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Select and removal of node u : The NOW queue can be in two states: (a) not empty or (b) empty. Consider the case (a) where NOW is not empty : Since the selection and removal of the node u is done from the head of the queue NOW it is done in constant time. So q_1 = constant, say x

Consider the case (b) where NOW is empty : In order to obtain a new Threshold value t the list NEXT must be scanned. This would mean that $q_1 = O(n)$ during the refreshing of the threshold value.

It is also noted that the threshold value of t can only increase during the execution of the algorithm. This implies that any node that passed the threshold test and has been transferred from NEXT to NOW, will never re-enter NEXT. This would imply that the number of times that NEXT is scanned to refresh the threshold value, in order to transfer new nodes to NOW is bounded n This would mean

that the total cost of the refreshing operation in the worst case is $O(n^2)$

Inserting v into NEXT : The node v is just added to the list NEXT so q_2 = constant, say y.

Number of iterations of Step 2 : When the algorithm is running between successive refreshing operations (ie. NOW is not empty) it behaves similar to the B-F-M (FIFO queue) algorithm for a subgraph of the main graph. This would mean that number of nodes to be explored as forward stars extracted from NOW is bounded by n Hence $c_3 \leq n^2$. The number of iterations would thus be $c_1 = O(n^3)$

The time that the distance labels of nodes are updated : Since $c_3 \leq n^2$ it implies $c_2 = O(c_3 \times m)$
$c_2 = O(n^2 \times m)$

The complexity of the Threshold algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + n^3 \times x + n^2 \times m \times y) = O(n^2 \times m)$$

9.0 The Concept of Sharpness

The sharpness concept introduced by Shier and Witzgall (1981) has been expanded on by Glover, Klingman and Phillips (1985) and Glover and Klingman (1989) to produce better algorithm and complexity bounds on the Partitioning and Threshold Shortest Path Procedures. The concept of sharpness relates the distance labels d_u and the distance of the path to the node u denoted by $d(P_u)$, where P_u is the actual path from the root node r to u not just an upper bound.

NOTE : A label setting algorithm by proposition 1 scans sharp labels only. An algorithm that scans sharp labels only is called a sharp algorithm.

$$d(P_u) = \sum d_{ij} \ni \forall \text{ arcs}(i,j) \text{ belong to the path fro.. } r \text{ to } u$$

Shier and Witzgall (1981) defined sharpness as follows: An algorithm is called sharp if at the time that node u is scanned $d_u = d(P_u)$ (ie.the distance label of node u is equal to that of the path distance from the root r to node u).

There are three levels of sharpness defined in Glover and Klingman (1989)

1. Globally Sharp : $d_i = d(P_i)\ \forall\ i \in N$ each time the tree T is changed.

2. Globally Scan-Sharp : $d_i = d(P_i)\ \forall\ i \in N$ each time a node is chosen to be changed.

3. Locally Scan-Sharp : $d_u = d(P_u)$ such that u is chosen to be scanned at the time of scanning u is initiated.

It is interesting to note that the third level of sharpness defined is equivalent to that defined by Shier and Witzgall (1981). The first two levels of sharpness defined are more restrictive.

9.1 The Globally Sharp Algorithm (Glover and Klingman 1989)

The algorithm that is now presented is a polynomially bound algorithm that only scans nodes with sharp distance labels. The concepts of partitioning the scan eligible nodes (Glover, Klingman and Philips 1985) are expanded on to ensure that the nodes that are to be scanned are sharp.

The algorithm of the PSP-Global-Sharpness is outlined as follows:

Procedure PSP-G-S (r)
Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as two mutually exclusive and collectively exhaustive lists of SE nodes called *NOW* and *NEXT*

NOW is initialised with the root node r and

NEXT is initialised as an empty set.

Set $k = 0$

STEP 2 *Select and Update node u*

NOW is now scanned : IF empty goto STEP 4

ELSE select any node u from *NOW*

The Forward Star of u is scann 1 and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$\delta = d_u + d_{uv} - d_v$

IF $\delta < 0$ THEN

Begin

2.1 $d_v = d_v + \delta$

IF v not in *NEXT* or *NOW*

THEN add v to *NEXT*

2.2 T is changed by setting $p_v = u$

2.3 For each node $i \neq v$ in the subtree of T

rooted at v, let $d_i = d_i + \delta$

IF i not in *NEXT* or *NOW*

THEN add v to *NEXT*

End;

STEP 3 *Iterate*

When the whole forward star of u has been explored return to to step 2

STEP 4 *Repartition Scan Eligible Nodes and Iterate*

IF *NEXT* is empty stop

ELSE set $k = k + 1$ and transfer all the nodes from *NEXT* to *NOW* and return to step 2.

End

Computational Complexity The computational complexity of the above algorithm is analysed and upper bounds are proved by the following Lemmas and Theorems (Glover, Klingman and Phillips 1985), (Glover, Klingman, Phillips and Schneider 1985) and (Glover and Klingman 1989).

Lemma 1 : At the kth iteration of step 3 of the PSP algorithm, each node in the *NEXT* list attains the following two properties:

1. The depth of the node is at least k

2. The shortest path to the node has at least a depth of k

Lemma 2 : At least one of the nodes in the current *NEXT* list will have its shortest path determined when the algorithm iterates at step 3.

Theorem 5

The PSP-GS algorithm has a worst case bound $O(m \times n^2)$ and only scans nodes with sharp distance labels.

Proof

Part 1

The depth of any shortest path will be at most $n - 1$.

From lemma 1 and lemma 2 this algorithm will iterate at most $n - 1$ times.

The most arcs that can be examined from a forward star rooted at say node u is m and there will be at most $n - 1$ distance labels that will be updated for a node u rooted at the forward star.

The complexity of the PSP-GS algorithm will thus be $O(m \times n^2)$

Part 2

The updating rule of the distance labels after each node in the forward star has been examined ensures that only nodes with sharp distance labels are scanned. This completes the proof.

9.2 *The Locally Scan-Sharp Desrochers Algorithm*

This algorithm is locally scan-sharp as it satisfies the locally scan sharp condition that $d_u = d(P_u)$ when the scan of u is undertaken.

Steps 1,3 and 4 are the same as in the previous algorithm, PSP-Global-Sharpness step 2 however is changed as shown.

Procedure PSP-Locally-Scan-Sharp (r)

Begin

STEP 1 *Initialise* As in previous algorithm

STEP 2 *Select and Update node u*

NOW is now scanned ; IF empty goto STEP 4

ELSE select any node u from *NOW*

Before deleting node u from *NOW* the path P_u is generated from predecessor labels.

The path distance $d(P_u)$ is then computed and d_u is set to $d(P_u)$

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$\delta = d_u + d_{uv} - d_v$

IF $\delta < 0$ THEN

Begin

2.1 $d_v = d_v + \delta$

IF v not in *NEXT* or *NOW*

THEN add v to *NEXT*

2.2 T is changed by setting $p_v = u$

End;

STEP 3 *Iterate* As in previous algorithm

STEP 4 *Repartition Scan Eligible Nodes and Iterate*

As in previous algorithm

End

9.3 The Sharp Threshold Algorithm For non negative lengths

The PSP-G-S algorithm is now presented modified to include the threshold concept while retaining the global sharpness property.

The algorithm of the Threshold Global-Sharpness is outlined as follows:

Procedure Sharp Threshold-G-S for non-negative lengths (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as three mutually exclusive and collectively exhaustive lists of SE nodes called *NOW*, *NOW'* and *NEXT*

NOW is initialised with the root node r and

NOW' is initialised as an empty set.

NEXT is initialised as an empty set.

Set $k = 0$ and threshold value $t = 0$

STEP 2 *Select and Update node u*

NOW is now scanned : IF empty goto STEP 4

ELSE select any node u from

NOW which was last placed in

NOW (ie, *NOW* is a LIFO queue)

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

DO

Begin

$\delta = d_u + d_{uv} - d_v$

redefine $d_v = d_u + d_{uv}$

set $p_u = u$

IF $d_v < t$

THEN node v is not a member of *NOW* or

NOW" add node v to the tail of

NOW' and if v is a member of *NEXT*'

delete it from *NEXT*'

ELSE add node v to the tail of the *NEXT* queue

Now add δ to the distance labels of nodes w

rooted at node v belonging to the current tree, and add

w to either *NEXT* or *NOW*" depending on the

criteria used by node v above.

STEP 3 *Iterate*

When the whole forward star of u has been explored return to

to step 2

STEP 4 *Repartition Scan Eligible Nodes and Iterate*

4.1 IF *NOW* is empty go to 4.2

ELSE transfer all the nodes from

NOW' to *NOW* and return to step 2.

4.2 IF *NEXT* is empty stop

ELSE set $k = k + 1$ and recompute the threshold value such that $t \geq \min d_i \; \forall \; i \in NEXT$ and transfer all the nodes i from *NEXT* to *NOW* such that they satisfy $d_i \leq t$ and return to step 2.

End

Choice of Threshold value

One of the advantages of choosing the threshold value t is that information of the arc density can be incorporated into the algorithm. In the original threshold algorithm (Glover, Glover and Klingman 1984) the threshold value t is computed as a weighted combination of true or estimated minimum and average distance labels of the nodes currently in NEXT. This method though successful is rather intricate. (Glover, Klingman, Phillips and Schneider 1985) developed a new method to calculate t based on two observations:

1. The performance of the shortest path procedures seems to be affected by the arc density of the network. Preliminary Testing (Glover, Klingman, Phillips and Schneider 1985) has shown that the arc density affected the choice of a good threshold value. The arc density should thus be included in the computation of the threshold value t.

2. At the kth iteration of the algorithm all the distance labels of the nodes in the NEXT list are less than that of the previous threshold value t_{k-1} It can obviously be seen that if a distance label of a node is greater than that of t_{k-1} it would have been placed in the NOW'

list. t_{k-1} is thus a lower bound on the minimum distance label of the nodes that have currently been left behind in the NEXT list. Since it is time consuming to find the true minimum distance of nodes in the NEXT list a surrogate minimum value $t_{k-1} + 1$ is introduced for the iteration k. The new heuristics value of the threshold value can thus be calculated as follows:

$t_k = AVG_k - (AVG_k - MIN_k) * (0.03 * DENSE - 0.15)$ where

AVG_k = True average distance labels for nodes in NEXT at the end of each iteration $k - 1$

MIN_k = Either (i) the true minimum distance of nodes in NEXT for $k = 0,1$ or (ii) $MIN_k = t_{k-1} + 1$ for $k = 2,3,.....$

DENSE = $\min(35, m/n)$

It may happen that at an iteration k there may be nodes in the list NEXT that pass the threshold criteria. If this were the case it will be necessary to find the true value minimum distance label of the nodes in NEXT.

The threshold t_k value moves towards the AVG_k value for low density networks and the value of t_k moves toward the MIN_k value for higher density networks. This would imply that the algorithm displays label setting characteristics for more dense networks.

Computational Complexity :- The complexity of the algorithm is analysed and proved in the following Lemmas and Theorems found in Glover, Klingman, Phillips and Schneider (1985)

Theorem 6

For negative arc lengths the Threshold Global Sharpness (T-G-S) algorithm has the following properties:

1. After each iteration, all the nodes in NEXT have sharp distance labels "naturally"

2. At the start of each iteration, all the nodes in NOW have sharp distance labels "naturally"

3. The polynomial worst case bound of the algorithm is $O(n^2 \times m)$

4. Only nodes with sharp distance labels are scanned.

In order to prove the above theorem the following lemma and corollary (Glover, Klingman and Phillips 1985) is used.

Lemma 3 Consider NEXT at the beginning of step 3
Assume $d_{ij} \geq 0 \; \forall \; (i,j) \in A$: and
$d_{min} = \min\{d_{ij} / \forall \; (i,j) \in$ NEXT$\}$ at the beginning of step 3.
d_{min} is then nondecreasing at each successive iteration.
Moreover, at this stage a shortest path has been determined to each node i in NEXT for $d_i = d_{min}$ such that nodes i will not appear in any subsequent NEXT set.

Corollary Every time nodes are transferred from NOW′ to NOW and are scanned, a shortest path has been determined to each node i whose distance label is the minimum of those nodes in NOW′. Such nodes will not appear in any subsequent set.

Proof

Parts 1 and 2

Let t_k be the threshold value used at iteration k By the end of the kth iteration, every node in NEXT will have a corresponding predecessor node from a previous NOW set. This would be due to the fact that all nodes that have been placed in the NOW set must have been less than the threshold value t_k. The corresponding predecessor node i will thus have a distance label such that $d_i \leq t_k$. For every node that will subsequently be improved it must trace by a predecessor path to an element of the current NEXT set. By the assumption that all arcs are nonnegative it follows that no node i such that $d_i \leq t_k$ can be improved. This confirms that there is no node i an element of the current NEXT set whose corresponding predecessor will ever be rescanned. This would imply that the distance labels of every node i such that $i \in$ current NEXT set represents the actual in-tree distance, in other words the distance labels of the nodes in the current NEXT set are "naturally" sharp. Part 1 of the theorem is thus proved. It follows thus that the nodes in the NOW set at iteration $k + 1$ are also sharp "naturally". This completes part 2 of the theorem.

Part 3

From the corollary of lemma 3 there can be at most $n-1$ nodes transferred from the NOW' to NOW and from NEXT to NOW. There can be at most m arcs that can be examined after each transfer and at most $n-1$ nodes that will have their distance labels updated in the subtree rooted at v of the current tree. This concludes the proof of part 3.

Part 4

Part 4 is proved in the same way that Theorem 5 is proved.

10.0 Label Setting Algorithms (Shortest First Search Algorithms)

Before the detailed discussion of the label setting algorithms, it is necessary to expand on the different ways of implementing Priority Queues As has been previously mentioned a *priority queue* or *heap* (Ahuja, Melhorn, Orlin and Tarjan, 1988) is a data structure consisting of a set of items each with an associated real value on which the following operations are possible :

Insertion : The insertion of a new element with a predefined value into the priority queue or heap.

Delete Min : The element with the minimum real value is found and deleted from the priority queue or heap.

Decrease : Replace the key of an item x in a heap by say Δ.
NOTE : Δ must be smaller than the old ... of x

The elements of a priority queue, the set of SE nodes Q can either be an ordered priority queue or an unordered priority queue. The difference between these two queues is crucial to the complexity of the algorithm. To maintain a priority queue requires continually updating the whole

priority queue when a individual element has had its value changed or has been inserted or deleted.

For the purpose of this study we assume the operations

1. Is Q empty ?

2. Is v is in Q ?

are done in constant time.

The different general implementation of priority queues with reference to shortest path algorithms are as follows:

Unordered Linked-List This is the simplest implementation of a priority queue Q. When an element is added to this type list it is inserted to the head of the list. The only type of operation that can be performed on Q is that of finding and removing the minimum value of the list. To perform this operation the whole of Q must be scanned.

Ordered Linked-List In this implementation of a priority queue Q a two way linked list can be used. Q is sorted in a increasing order of label values. The operations that are performed on this type of implementation are to insert an element in its correct position in Q or delete the minimum element from the head of Q

10.1 Buckets and Heaps

Buckets A list of nodes that can be arranged using their labels into a specific set of ranges where each range defines a bucket. For the simplest case a bucket i contains those nodes that fall into the interval from iw to $(i+1)w$ where w is the width of the bucket.

Heaps Let S denote a set $\{Y_1, Y_2, ..., Y_n\}$

Where Y_i is the *key* associated with the item k The keys are linked to the items with pointers. The keys are stored at the nodes of tree called a Heap-ordered tree.

Heap-ordered tree : A tree that contains a set of items, one item to each node. The items are arranged in *heap order*

Heap order : The parent of a node x has a key that is no more than the key in x itself. Intuitively the root of the tree contains an item of minimum key.

Heap : A heap is thus defined as follows:

Assume N is odd for convenience

$Y_i \leq Y_{2i}$ and $Y_i \leq Y_{2i+1}$; for $1 \leq i < N/2$

NOTE : Each sub tree of a heap is also a heap. The root of the heap is the first node and has the smallest key.

10.1. Operations performed on Heaps

Insert (i,h) : A new item *i* is inserted with a predefined key into heap *h*

Delete Min : The item of minimum key in heap *h* is found returned and then deleted from *h*.

Decrease Key (*value*, *i*, *h*) : The key of item *i* in heap *h* is decreased by *value*.

NOTE: The position of *i* is known and *value* must be smaller than that of the old key.

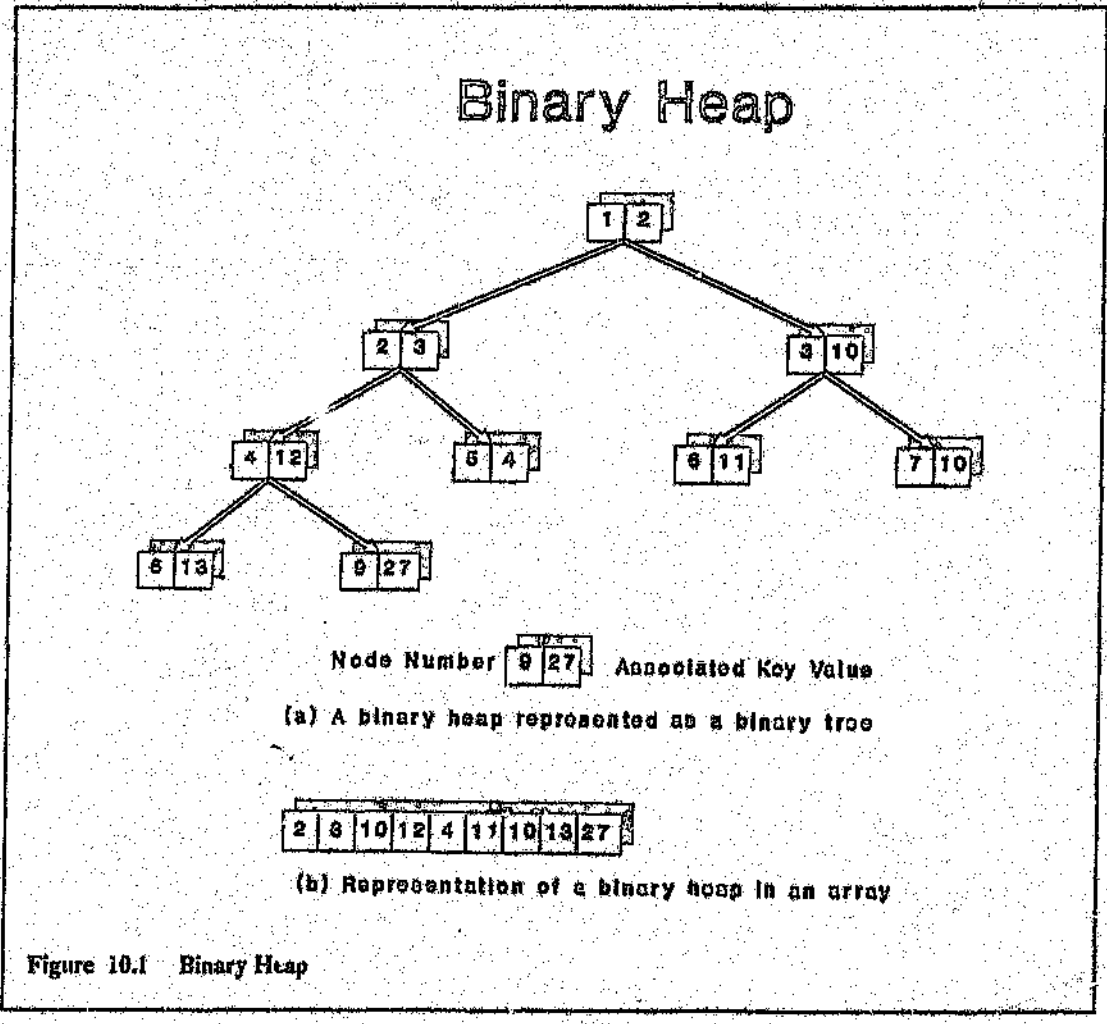


Figure 10.1 Binary Heap

10.2 The Dijkstra Algorithm

The first algorithm presented is the simplest implementation of a priority queue Q It is usually credited to Dijkstra although some references do credit it to Moore. The algorithm is based on the basic Label setting property given in proposition 1 and implements Q as an unordered linked-list.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure Dijkstra (r)
Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values
Q is initialised as the queue of SE nodes

STEP 2 *Select and Update node u*

Q is scanned and the minimum label node u
is removed from Q
The Forward Star of u is scanned and updated
For every $(u,v) \in FS(u)$ such that
$d_u + d_{uv} < d_v$
$d_v = d_u + d_{uv}$
$p_v = u$
IF v not in Q
THEN v is inserted into Q

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2
ELSE STOP

End

An efficient implementation of the queue in the above algorithm is to use a two way linked list.

Computational Complexity

Theorem 7

Dijkstra's algorithm has a worst case complexity of $O(n^2)$ for all $d_{ij} \geq 0$

Proof

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : In order to select node u Q must be searched for the node with the smallest distance label. Since the cardinality of Q is at most $n-1$ the order of this procedure is at most $n_q - 1$ so $q_1 = O(n_q - 1) \leq O(n_q)$

Inserting v into Q : The node v is just added to the list Q so $q_2 =$ constant, say x.

The number of labels that are updated : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times, thus $c_1 = O(n)$

The complexity of the Dijkstra algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + n \times n_q + x \times m) \leq O(n_q \times n + m) \leq O(n^2)$$

10.3 The Ordered Priority Queue Algorithm

This algorithm is an extension of Dijkstra's algorithm where the efficiency of obtaining the minimum distance label from Q is improved by means of implementing Q as a two way ordered linked-list. The minimum label is thus always the first element of Q. To insert and correct an element the priority queue need only be scanned once.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure Ordered-Dijkstra (r)
Begin

STEP 1 *Initialise*
predecessor and distance labels p, d are given initial value.
Q is initialised as the queue of SE nodes

STEP 2 *Select and Update node u*

u = $Q(1)$ and is removed from Q

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

$d_v = d_u + d_{uv}$

$p_v = u$

IF v not in Q

THEN v is inserted into Q to its proper position

ie. In a position where it maintains Q as an increasing ordered queue.

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE STOP

End

An efficient implementation of the queue in the above algorithm is to use a two way linked list.

Computational Complexity

Theorem 8

The ordered version of Dijkstra's algorithm has a worst case complexity of $O(n \times m)$ $\forall$ $d_{ij} \geq 0$

Proof

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : Since Q is implemented as a priority queue Q the selection of node u is constant since the node with the smallest distance label is always in the first position. q_1 = constant say x

Inserting v into Q : The node v must be added to the list Q in its correct position so to maintain the nondecreasing order of Q Since the maximum cardinality of Q is n_q, hence $q_2 = O(n_q)$

The number of labels that are updated : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times, thus $c_1 = O(n)$

The complexity of the Ordered Dijkstra algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + n \times x + n_q \times m) \leq O(n \times x + m \times n_q) \leq O(n \times m)$$

10.4 The Dial (bucket) Algorithm

This algorithm makes use of buckets to implement Q The elements are partitioned into a sequence of buckets that each have a range with a width k. Q is thus partitioned into buckets each with a range such that the

bucket i contains all the elements that are within the set $k(i-1) \leq d_v < ki$
In order to find the range of all the buckets together define

$dmax = max\{d_{ij} \ni (i,j) \in A\}$

Now no distance label can ever be greater than $(n-1)dmax$ so the upper bound of the last bucket must be $(n-1)dmax$ and the lower bound of the first bucket must be 0. In order to access a bucket i a vector of pointers say q is used where the pointer q_i points to bucket i.

In order to remove and scan the minimum node the vector q is scanned and the first non-empty bucket is found. From this bucket the minimum element is removed. To insert a new element it is inserted into the bucket with the range within that of the new element.

Dial proposed a shortest path algorithm using bucket of size $k = 1$, where d_{ij} is an integer $\forall (i,j) \in A$. NOTE : in this case each bucket will contain nodes that have the same label.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure Dial (Bucket) (r)
Begin

STEP 1 *Initialise*
predecessor and distance labels p, d are given initial values
Q is initialised as buckets of SE nodes

STEP 2 *Select and Update node u*

Search for the first non-empty bucket and remove node u from this bucket. ie. $u = Bucket(i)$

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

IF $v \in Q$ then v is removed from its appropriate bucket.

$d_v = d_u + d_{uv}$

$p_v = u$

v is inserted into Q into Bucket i such that

$v \in range(Bucket(i))$

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE STOP

End

An efficient way to implement Dials Buckets is to use a two way linked list for each bucket.

Computational Complexity

Theorem 9

Dials Bucket algorithm has a worst case complexity of $O(m + n \times dmax)\ \forall\ d_{ij} \geq 0$

Proof

Initialisation : All the distance labels of the nodes and the vector q are initialised $q_0 = O(n + dmax)$

Selection and removal of node u : To select a node the vector q is scanned to find the first empty bucket. $q_1 = O(dmax)$

Inserting v into Q : The node v must be added to the list Q in its correct bucket. This operation is done in constant time q_2 = constant, say x

The number of labels that are updated : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times thus $c_1 = O(n)$

The complexity of Dials Bucket algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + dmax + n \times dmax + x \times m) = O(n \times dmax + m)$$

10.5 The Binary Heap Algorithm

An implementation of the priority queue as a binary heap. The elements of a binary heap are stored as nodes of a binary tour. The label of an

element is always smaller or equal to that of its children. The root node is thus the smallest element of the heap. A more detailed discussion of heaps and buckets is presented a little later. The complexity of removing the minimum label element, inserting a new element and correcting a label is of the order $\log n_q$.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure Heap (Binary) (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as the queue of SE nodes

STEP 2 *Select and Update node u*

u = the root node of the binary heap.

Re-order the heap.

The Forward Star of u is scanned and updated

For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

$d_v = d_u + d_{uv}$

$p_v = u$

IF v is not an element of the heap THEN insert v

into the heap and re-order the heap.

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE STOP

End

Computational Complexity

Theorem 10

The binary heap algorithm has a worst case complexity of $O(m \times \log_2(n)) \; \forall \; d_{ij} \geq 0$

Proof

Before the proof is started it is noted that the complexity of reordering a binary heap is $O(\log(n))$ (J Esakov and T Weiss (1989), Aho, Hopcraft and Ullman (1974), Horowitz and Sahni (1987))where log is to the base 2

Initialisation : All the distance labels of the nodes are initialised $q_0 = O(n)$ since there are at most n nodes.

Selection and removal of node u : Since Q is implemented as a heap (priority queue) the selection of node u is constant since the node with the smallest distance label is always in the root position. However, with the removal of this node the heap must be reordered which will take at most $\log(n)$ operations. $q_1 = O(\log(n))$

Inserting v into Q : The node v must be added to the heap into a correct position so to maintain the binary heap. To insert and reorder the heap will take at most $\log(n)$ operations. $q_2 = O(\log(n))$

The amount of labels that are updated : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times, thus $c_1 = O(n)$

The complexity of the Binary Heaped algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$
$$= O(n + n \times \log(n) + m \times \log(n)) = O((n + m)\log(n)) = O(m \times \log(n))$$

11.0 One/Two-Level Redistributive Heaps

(Ahuja, Mehlhorn, Orlin and Tarjan, 1988) Redistributive heaps exploit special properties of the heap operations in Dijkstra's algorithm. A redistributive heap is a collection of buckets, each with a range that is an interval of integers.

The following two properties are essential in the formulation of Redistributive heaps (Ahuja, Mehlhorn, Orlin and Tarjan, 1988)

1. For node $v \in N$, $d(v) \in (0..nC)$ $\forall$ $d(v) \geq d(v)$ is finite.

2. For node $v \in N$, such that $v \neq s$, if vertex v is labelled then $d(v) \geq (d(s\ ,\ 1)..d(x) + C)$ where $d(x)$ is the most recently scanned node (ie. successive delete min operations return nodes in nondecreasing order of cost)

The second property is the most important of the two properties

11.1 The One-Level Redistributive Heap Algorithm

A one-level redistributive heap is a collection of $B = \log_2(C + 1)$ buckets that are sized as follows:

Size of

$size(1) = 1$

$size(i) = 2^{i-2}$ for $2 \leq i \leq B - 1$

$size(B) = nC + 1$

NOTE: The following inequality is satisfied

$$\sum_{i=1}^{j-1} size(i) \geq \min \{size(j), C + 1\} \text{ for } 2 \leq j \leq B \quad \ldots \text{ (i)}$$

To start with node s is stored in bucket 1, As the algorithm progresses all the labelled nodes are stored in the buckets. The node v is stored in the bucket x if $d(v)$ is within the ra ge(i)

To start with the interval $(0..nC + 1)$ is partitioned into the different ranges of the buckets. After the first node has been scanned the bucket ranges partition the interval $(d_{min}..nC + 1)$ where d_{min} is the maximum label of the scanned nodes. The upper bound of each bucket i $u(i)$ is maintained, where the rarge of each bucket is i is $(u(i-1)+1..u(i))$ such that $u(0) = d_{min} - 1$ and the range i = empty set if $u(i-1) \geq u(i)$ Initially the upper bound $u(i) = 2^{i-1} - 1$ for $1 \leq i \leq B - 1$ and the upper bound of B, $u(B)$ is $nC + 1$

NOTE : Although the ranges of the buckets may vary during the execution of the algorithm the sizes remain the same.

With reference to the three heap operations they are performed as follows:

Insert : The values of bucket i are examined starting with $i = B$ and continuing until the largest i is found such that $u(i) < d(v)$ and node v is inserted into bucket $i + 1$

Decrease Key : The key of node v is decreased by removing v from its current bucket. The key v is reduced and v is reinserted using the Insert operation.

Delete Min : This is the most complex of the three operations. The bucket with the smallest index is found. If the first bucket is empty find the non-empty bucket with the smallest index, lets say it was bucket j. Find the node u with smallest distance label by scanning this bucket, j and delete it from the bucket. The remaining nodes are then redistributed amongst buckets of smaller index using the following rules:

$u(0)$ replaced by $d(v) - 1$,
$u(1)$ replaced by $d(v)$,
For i from 1 to $j - 1$, $u(1)$ replaced by min $\{u(i - 1) + size(i), u(i)\}$,

The nodes in bucket j are removed and reinserted using the decease operation.

From the inequality (i) it follows that for $j \geq 2$ by the delete min operation every node in bucket j will move to a bucket of smaller index.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure One-Level Redistributive Heap (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as a heap of buckets as defined.

STEP 2 *Select and Update node u*

Search for the first non-empty bucket of the smallest index and remove node u from this bucket.

IF the first non-empty bucket is *bucket*(1)

return any node from this bucket

ELSE scan the bucket and return the node with the smallest distance value.

The remaining elements in the buckets are redistributed to buckets with a smaller index by the rules defined.

The Forward Star of u is scanned and

updated For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

IF $v \in Q$ then v is removed from its

appropriate bucket.

$d_v = d_u + d_{uv}$

$p_v = u$

v is inserted into Q into Bucket i

such that $v \in range(Bucket(i))$

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE (ie. all buckets are non empty) STOP

End

The one-level redistributive heap is implemented using a two way linked list for each bucket.

Computational Complexity

Theorem 11

The One-Level Redistributive heap algorithm has a worst case complexity of $O(m + n \times \log(C)) \; \forall \; d_{ij} \geq 0$

Proof

Initialisation : All the distance labels of the nodes are initialised with the buckets $q_0 = O(n + B)$ since there are at most n nodes and by definition of a One-Level Redistributive heap at most B buckets.

Selection and removal of node u : The selection of node u is done by finding the first non-empty bucket, and removing the node with the smallest distance label. With the removal of this node the remaining nodes in this bucket will be reshuffled to buckets with a smaller index. The time taken to complete this operation will be of

order $O(B)$ since there are at most B buckets by definition of the data structure of this algorithm so $q_1 = O(\log(C))$

Inserting v into Q : The node v must be inserted into its correct bucket containing in its correct range. q_2 could thus be regarded as constant, say x if a function could map the distance label to the correct bucket range.

Exploration of the forward star FS(u) : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times thus $c_1 = O(n)$

The complexity of the One-Level Redistributive Heap algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + B + n \times \log(C) + m \times x) = O(m + n \times \log(C))$$

11.2 The Two-Level Redistributive Heap Algorithm

One way to further optimise the running time of the algorithm implemented with a one-level redistributive heap is to reduce the numbers of reinsertions of vertices into buckets. The sizes of the bucket are limited by the inequality (i), which makes it impossible to increase the size of the

buckets to reduce the number of insertions of nodes. Another idea is to divide the buckets into equal sized segments. Like the one-level redistributive heap the two-level redistributive heap is define as a collection of bucket but with an added parameter K which indicates the number of segments in a bucket. The number of buckets is given as $B = \lfloor log_K(C + 1) \rfloor + 1$ and the sizes of the buckets are defined as follows:

$$size(i) = K^i \text{ for } 1 \leq i \leq B - 1$$

$$size(B) = nC + 1$$

The range of the buckets of the two-level redistributive heaps is the same as that of the one-level redistributive heaps. Where the range of bucket i is $\lfloor u(i-1) + 1..u(i) \rfloor$ where the initial upperbounds for the buckets are as follows:

$$u(0) = d_{\min} - 1 \text{ and } u(B) = nC + 1$$

$$u(j) = \sum_{i=1}^{j} K^i - 1 \text{ for } 1 \leq j \leq B - 1$$

Now each bucket i is partitioned into K segments the size of K^{i-1} Bucket B consists of a single segment. The range of the segments are defined as follows:

$$\text{range of segment } (i,k) = \lfloor u(i,k-1) + 1..u(i,k) \rfloor \text{ where}$$

$$u(i,k) = \max \{u(i-1), u(i) - (K-k)K^{i-1}\}$$

Where the segments are indexed by ordered pairs, segment k of bucket i is denoted by (i,k)

The ranges of the buckets are maintained, the ranges of the segments change and are computed as follows:

For $x \in$ the range of (i,k) the value of k is computed by the formula

$$k = K - \lfloor (u(i) - x)/K^{i-1} \rfloor$$

With reference to the three heap operations they are performed as follows:

Insert : The values of bucket i are examined starting with $i = B$ until the bucket i is located such that $d(v) \in$ the range of (i). The value of k is then computed and v is inserted into the segment (i,k)

Decrease Key : The key of node v is decreased by removing v from its current bucket. The key v is reduced and v is reinserted using the Insert operation.

Delete Min : This operation is performed in the same way as that in the one-level heap, only that the nodes of a single element are distributed.

The procedure of this algorithm based on the general algorithm is as follows:

Procedure Two-Level Redistributive Heap (r)

Begin

STEP 1 *Initialise*

predecessor and distance labels p, d are given initial values

Q is initialised as a heap of buckets and segments within the buckets as defined.

STEP 2 *Select and Update node u*

Search for the first non-empty bucket of the smallest index, now find the first non-empty segment within the bucket. The segment is scanned and the node u with the smallest distance label is removed from the segment.

The upperbounds of the buckets are redefined as in the one-level heap. The remaining elements in the segment are redistributed to the correct segments within the buckets with the appropriate range.

The Forward Star of u is scanned and

updated For every $(u,v) \in FS(u)$ such that

$d_u + d_{uv} < d_v$

IF $v \in Q$ then v is removed from its segment from its appropriate bucket.

$d_v = d_u + d_{uv}$

$p_v = u$

v is inserted into Q into its correct segment within the Bucket i

such that $v \in range(Bucket(i))$

STEP 3 *Iterate*

IF Q is not empty THEN goto STEP 2

ELSE (ie. all buckets are non empty) STOP

End

The two-level redistributive heap is implemented using an array of doubly- linked lists as segments and a two way linked list for each bucket .

Computational Complexity

Theorem 12

The Two-Level Redistributive heap algorithm has a worst case complexity of $O(m + n \times \log_k(C))\ \forall\ d_{ij} \geq 0$

Proof

Initialisation : All the distance labels of the nodes are initialised with the buckets $q_0 = O(n + B)$ since there are at most n nodes and by definition of a Two-Level Redistributive heap at most B buckets.

Selection and removal of node u : The selection of node u is done by finding the first non-empty bucket. The node with the smallest distance label is removed from its appropriate segment. With the removal of this node the remaining nodes in the segment of this bucket will be reshuffled to segments with a smaller index. The time taken to complete this operation will be of order $O(B)$ since there are at most B buckets by definition of the data structure of this algorithm

so $q_1 = O(\log_K(C))$ If the correct segment was found by scanning the segments in the bucket it would take at most $O(K)$ time.

Inserting v into Q : The node v must be inserted into its correct bucket containing in its correct range and then to its correct segment. q_2 could thus be regarded as constant say x if a function could map the distance label to the correct bucket range and then the correct segment. Ahuja, Mehlhorn, Orlin and Tarjan (1988) set K proportional to $\log(C)/\log\log(C)$ which will make B of the order $O(\log(C)/\log\log(C))$

Exploration of the forward star FS(u) : At most the forward star can consist of m arcs, thus $c_2 = O(m)$

Number of iterations of Step 2 : Since by proposition 1 that each node is examined only once, the algorithm can iterate at most n times, thus $c_1 = O(n)$

The complexity of the Two-Level Redistributive Heap algorithm is thus

$$O(q_0 + q_1 \times c_1 + q_2 \times c_2)$$

$$= O(n + B + n \times \log_K(C) + m \times x) = O(m + n \times \log_K(C))$$

11.3 *Further improved Redistributive Heaps*

The running time of the Two-Level redistributive can be reduced to $O(m + n \times \sqrt{\log}(C))$by using a variant of Fibonacci heaps, Fredman and

Tarjan (1987) to find non-empty segments, Ahuja, Melhorn, Orlin an' Tarjan (1988).

When manipulating rooted trees in the F-heaps the following invariant is maintained. $rank(x) = O(\log size(x))$ for any node x as the root of a subtree where $size(x)$ is the number of node.

The time bound per delete min is $O(\log(n))$ By extending F-Heaps this time bound can be reduced to a time bo nd of $O(\log(\min(n,N)))$ where $N = KB$

11.3.1 Fibonacci Heaps

The following is basic discussion of the Fibonacci heaps

Heap-ordered tree : A tree that contains a set of items, one item to each node. The items are arranged in *heap order*

Heap order . The parent of a node x has a key that is no more than the key in x itself. Intuitively the root of the tree contains an item of minimum key.

Linking This is ndamental operation in heap-ordere' tree Linking combines ... item disjoint trees into one. Two trees are linked when the keys of the items of the two roots are compared. The item with the smaller key is made parent of the other root.

rank(x) The number of children of a node x is regarded as the rank of x The rank of a node does not have an upper bound

A **Fibonacci Heap** *(F-Heap)* is a collection of item-disjoint heap-ordered trees. There are no constraints on the number or structure of the trees. The only constraints that do exist are those in the way the trees are manipulated.

11.3.2 Operations performed on Fibonacci Heaps

Make Heap : This operation just returns a pointer to null.

Find Min (h) : This operation will return the minimum item of h

Insert (i,h) : A new heap that only contains the node i is created and then replaces h with the *meld* of h and the new one node heap containing i

Meld (h_1, h_2) : The root lists of h_1 and h_2 are combined to form a single list. The node with the item of the smaller key is returned as the minimum node.

Delete Min : The minimum node of h is deleted to start with. The list of the children of the deleted node are now concatenated with the following step:

> *Linking Step* : To perform this step find two trees with the same rank. These two trees are then linked. The root of the new tree will have a rank one greater than that of the two original trees.

When there are no longer any trees with roots of the same rank a list of the remaining roots is made to find a root containing an item of minimum key. The node that contains this key now becomes the minimum

node of the modified heap. Deletion min is completed by saving and returning the item from the minimum node which is now destroyed.

Decrease Key (Δ, i, h) : With this operation Δ is deleted from the key of i, The node containing i is found. The edge joining this node to its parent is cut. The node containing i is thus the root of a new tree of h

Delete (i, h) : Here the node containing i is found and the edge joining it to its parent is cut. The list children of this node (containing i) are concatenated with the list of original roots. The node containing i is then destroyed.

Cascading Cut : After the linking step when a root x node has been made a child of another and the node x loses two of its children through cuts. the edge joining x and its parent is also cut. Node x is thus made a new root as done in the decrease key operation. This type of cut is called a cascading cut.

In order to keep track of where to make cascading cuts the nodes are marked. When the linking step is used to make a root node x a child of another node, the node x is unmarked. Now when x is cut from its parent the parent node $p(x)$, the rank of $p(x)$ is decreased and $p(x)$ is checked to see whether it is a root node. Should it be a root node and is unmarked it is marked If it was marked the edge to its parent is cut.

11.3.3 Mathematical Properties of Fibonacci Heaps

The following Lemma and Corollary explain the derivation of and is the source name of the Fibonacci Heaps.

12.2.2 Sparse networks

The second type of networks considered, were sparse networks. This type of network can vary in density. The sparse networks generated can be divided into two further networks:

1. Grid networks

2. Random networks

The first type of sparse network the grid network has a specific structure so the number of arcs m are determined by the number of nodes n. A grid network is strongly connected. There is always a path from any node i to any node j. The second type of sparse network considered was the random networks comprising randomly structured arcs connecting the nodes. This allows one to vary the density of the network. This type of network is structured only in that there exists a Minimal Spanning Tree (MST), although there is not always a path from any node i to any node j

12.2.2.1 Grid Networks

The conceptual structure of a grid network is a rectangular grid of nodes arranged in rows and columns. Each node only being adjacent to the nodes above, below, to the left and to the right of it. The number of arcs are thus determined by the number of nodes in the network. The size of a grid network can be defined in terms of the height H (the amount of nodes in the of the columns of the network) and the width W (the amount of nodes in the rows of the network). The amount of nodes are thus in total $(H + W)$ and the amount of arcs are $H(W-1) + W(H-1)$

11.3.4 Lemma 1

Let x be any node in an F-Heap. Let the children of x. be arranged in the order that they were linked to x (ie. Linked from earliest to latest). The i*th* child of x will have a rank of at least $i-2$

Proof : Let y be the *ith* child of x.

The time just before y was linked, x had at least $i-1$ children.

NOTE: Some children may have been lost just before linking.

Before the linking both x and y both had the same rank. Both had a rank of at least $i-1$

After the linking the rank of y could have decreased by at most one without causing y to be cut as a child of x.

11.3.5 Corollary 1

A node of rank k in an F-Heap has at least $F_{k+2} \geq \phi^k$ descendents, including itself, where F_k is the *kth* Fibonacci number and $\phi = (1+\sqrt{5})/2$ is the golden ratio.

Proof : Let S_k be the minimum possible number of descendants of a node of rank k

It is obvious that $S_0 = 1$ and $S_1 = 2$ It is implied from lemma 1 that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$

$F_{k+2} = \sum_{i=2}^{k} F_i + 2$ is satisfied by the Fibonacci numbers. It follows by induction on k that $S_k \geq F_{k+2}$ for $k \geq 0$. $F_{k+2}\phi^k$ is a well known inequality that is explored in (Knuth Vol 1 1973).

12.0 Computational Analysis of the Algorithms

An extensive computer system was written to compare the speed of the predominant algorithms on various networks. The system was written for an IBM compatible. The programming language used was turbo Pascal and was implemented on a Microsoft DOS operating system. Incorporated into the system was a network generator. It is possible to generate three type of networks

1. Complete Networks

2. Grid Networks

3. Random networks

Once a network has been generated it can then be loaded into memory and the shortest path from any start node to any end node (should one exist of course) with any one of the algorithms proposed can be found. The speed of finding the shortest path is recorded thus making it possible to compare the different methods finding the same shortest path on the same network. The testing was done on a IBM 80386 compatible machine.

12.1 The Algorithms implemented.

Twelve algorithms, including four different versions of the threshold algorithm were implemented. The algorithms were split into the same two catergorise used for the theoretical analysis namely, label correcting and label setting algorithms. The first two algorithms implemented were the Dijkstra and the Bellman-Ford-Moore (queue) algorithms, since they are the foundation of the label setting and label correcting algorithms respectively.

12.1.1 Label Setting algorithms

At least one type of each well known algorithm was implemented. Included in the label setting algorithms was a heap algorithm and the Dial's bucket algorithm. The bucket algorithm known to perform poorly with a large *dmax* included a variation of extending the bucket range from Dial's original range of 1 to a range suited to the *dmax* of a specific network. The following label setting algorithms were implemented.

- The original Dijkstra and the ordered Dijkstra algorithms
- The heap algorithm (binary heap used)
- Dial's bucket algorithm including variation of a variable bucket range.

12.1.2 Label Correcting Algorithms

Most of the known label correcting methods were implemented. The threshold algorithm was implemented in four versions differing in the method of calculating the threshold value *t*. When comparing the threshold algorithm to that of the other algorithms, only the version with

the best performance was used. This usually varied between the increment and percentage versions. The following label correcting algorithms were implemented.

- The Bellman-Ford-Moore algorithm (queue)
- The D'Esopo-Pape algorithm (deque)
- Pallottino's 2queue algorithm (2queue)
- The Partitioning Shortest Path algorithm
- The threshold algorithms (4 versions)

The running time of the 2queue algorithm was found in most cases to be very close to that of deque. In comparison to the running time of the other algorithms the running time difference of the deque and 2queue algorithms was not really noticeable. The difference between the two algorithms is the computational complexity. The computational complexity of 2queue is $O(m \times n^2)$ as opposed the complexity of deque $O(n \times 2^n)$. The difference in complexity did however, not induce any significant difference in performance of the two algorithms. For this reason the running time of the 2queue does not specifically appear on the graphs comparing the running times of all the algorithms. The running time of the deque algorithm represents both the deque and 2queue algorithms. Similar results with respect to these two algorithms were published by Gallo and Pallottino (1988)

12.2 *The evaluation of the algorithms*

The networks used to evaluate the algorithms can be divided into two general types:

1. Complete networks

2. Sparse networks

The significant difference of these networks is that of the network density. The density of a network being defined as the ratio of number of arcs m over the number of nodes n, m/n. The computational performance of the algorithms is evaluated by the speed in which the shortest path is found. The performance of the algorithms were tested for varying sizes and densities of networks. The performance of the algorithms on different structured networks is also evaluated, such as a random sparse network and a sparse grid network. The results of the computational analysis can then be compared to the projected theoretical computational worst case analysis.

It is also important to note what types of network are found in the real world applications. The major characteristics are (Gallo and Pallottino, (1986))

1. The networks are sparse, ie. $m << n^2$ and usually $m = O(n)$

2. The topology of the networks is in most cases planar. If the network is not planar it can more than likely be made planar by removing a few arcs.

3. The size of the networks are thousands of nodes and arcs.

12.2.1 Complete Networks

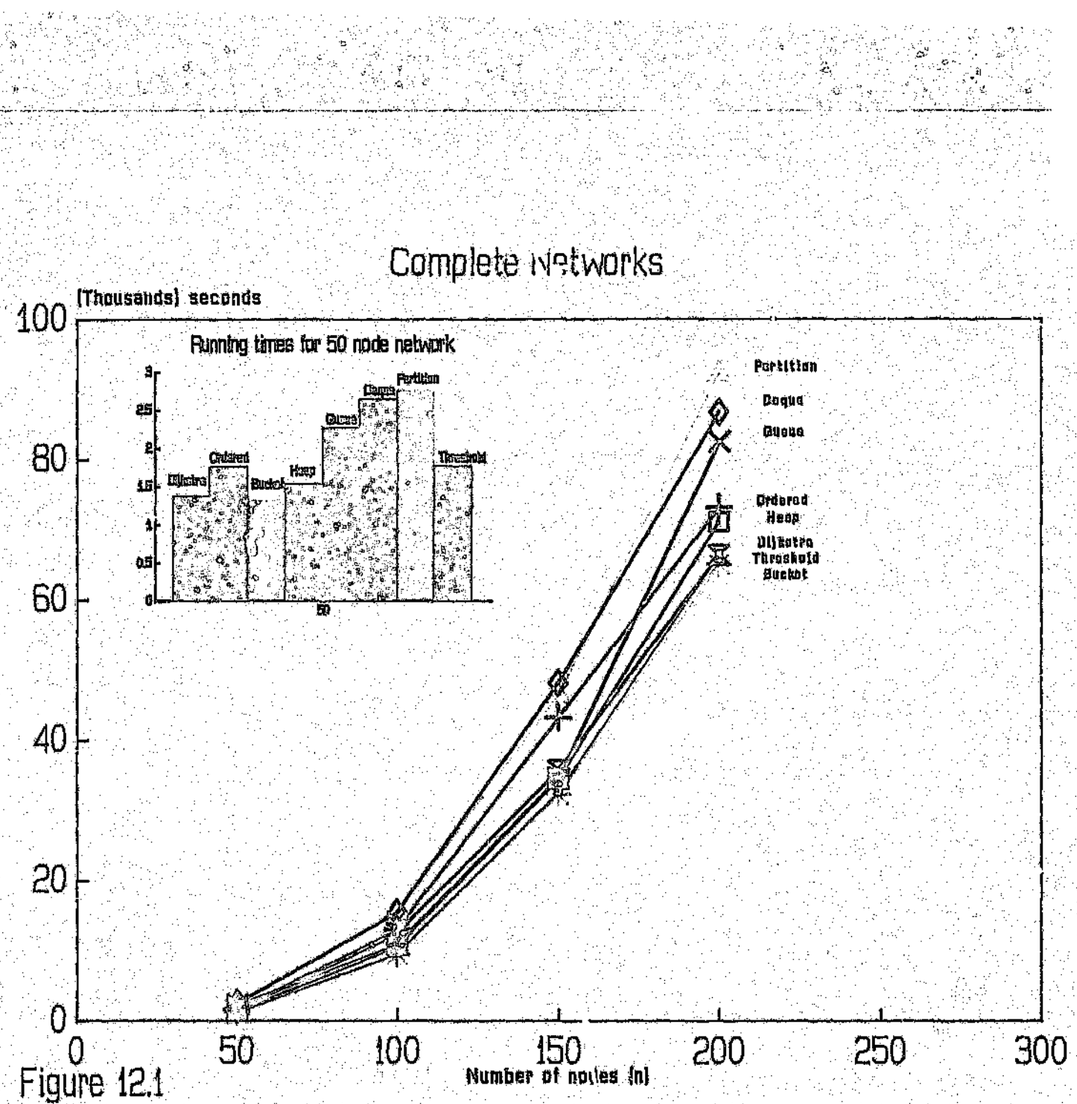
The first type of network to be used in the computational evaluation of the shortest path algorithms are the complete networks. This network, although somewhat unnatural, can be useful in predicting the trend of the algorithm performance toward the more dense networks. Provided that there are no repeat arcs in the networks, the complete network is the most dense a network of a particular size can be. The density of complete networks grows exponentially to the size of the network. (ie. m grows exponentially to the size of n in complete networks)

As expected from the computational results of Gallo and Pallottino (1988) and that of Glover, Klingman and Phillips (1985) the label setting algorithms dominated the label correcting algorithms with exception to the threshold algorithm, see figure 12.1. The algorithm that performed the best in finding the shortest path for increasing sizes of the complete networks was Dial's bucket algorithm. The performance of the bucket algorithm was followed closely by the threshold algorithm which was followed by Dijkstra's algorithm. Next in line was the heap algorithm which, although it performed fairly well with respect to all the algorithms, was below expectations with regard to the projected theoretical worst case computational complexity. Label correcting algorithms with the exception of the threshold algorithm performed poorly as would be expected in complete networks. It is interesting to note that from the nature of the threshold algorithm the more dense the network the more it performs as label setting algorithm. (see chapter 9, page 59)

The maximum number of nodes of complete networks tested is 200. This was due to memory limitations of the hardware used. Similar problems were experienced by Gallo and Pallottino (1988)

Comparison to other published results

The comparison of running times produced by the implementation of the algorithms for complete networks correlate very closely to that of Gallo and Pallottino (1988) for most of the algorithms. My ordered version of the Dijkstra algorithm has however, a far better running time in comparison to the other algorithms, unlike the performance of the ordered algorithm implemented by Gallo and Pallottino.


Complete networks
(Thousands) seconds
Running times for 50 node network
Dijkstra
Ordered
Bucket
Heap
Queue
Partition
Threshold
100
80
60
40
20
0
0
50
100
150
200
250
300
Number of nodes (n)
Partition
Dequa
Queue
Ordered
Heap
Dijkstra
Threshold
Bucket
Figure 12.1

The performance of the label setting algorithms again dominated in the computational experimentation, see figure 12.2 The performance of the threshold algorithm being rather poor although it does appear to improve towards the end. The label setting algorithms seemed to perform equally well. The best performing label correcting algorithm was deque.

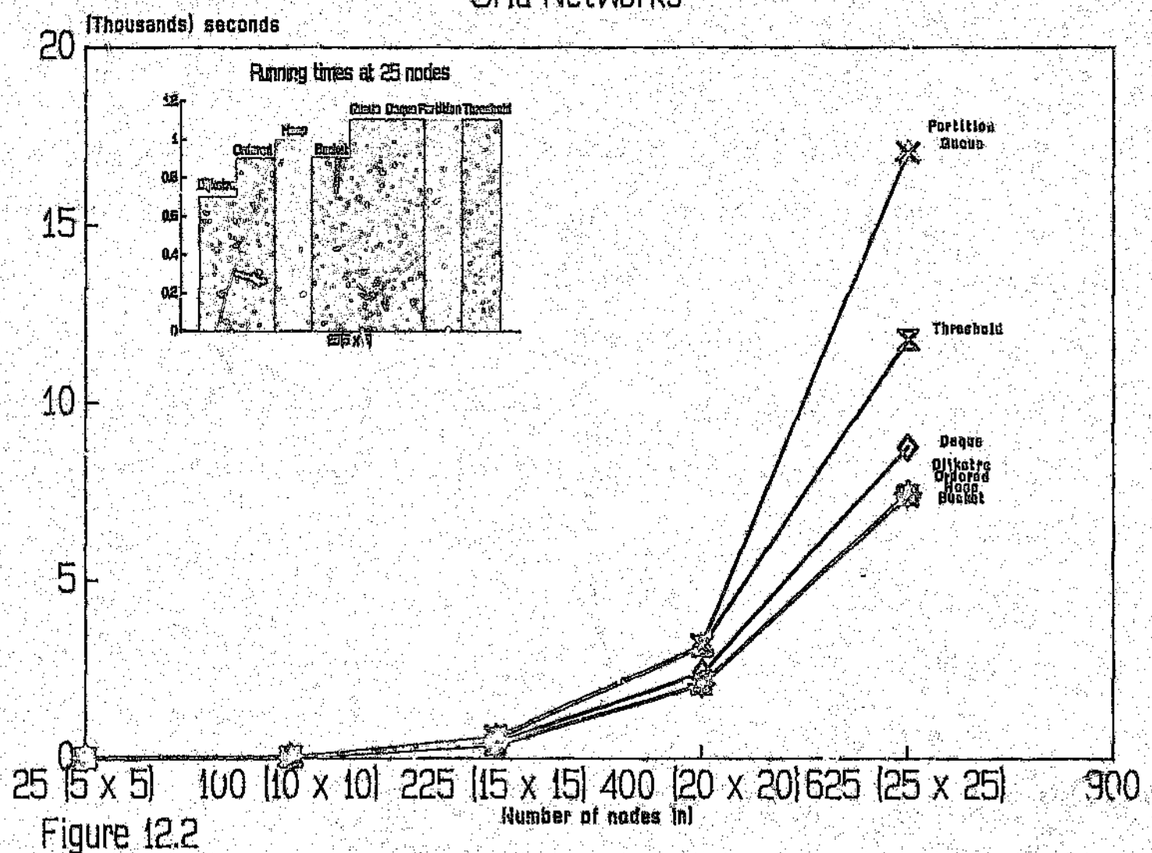

(Thousands) seconds
20
15
10
5
0
Running times at 25 nodes
Dijkstra
Chained
Heap
Bucket
Deque
Partition
Threshold
Partition Queue
Threshold
Deque
Dijkstra
Chained
Heap
Bucket
25 (5 x 5)
100 (10 x 10)
225 (15 x 15)
400 (20 x 20)
625 (25 x 25)
900
Number of nodes (n)

Figure 12.2

12.2.2.2 Sparse Random Networks

This type of network is the most important in the priori testing. The reason is, sparse random networks that can be generated are the closest to that of most real world networks. Real world networks as mentioned previously are large, planar and usually sparse. With sparse random networks one can vary the size and density of the network. Apart from being more realistic it also allows us to compare the performance of different algorithms within these two network variants. A large range of networks with respect to size and density were generated. Although some of the networks generated may not quite be regarded as sparse, (ie $m/n > 30$) it was felt they be included for analytical purposes. Due to memory constraints the larger the density of the network the greater the limit to the size of the network generated. Networks with a density $m/n = 60$ for instance, could only be created to a size of up to 500 nodes. Networks with a density $m/n = 5$, on the other hand could have up to 1500 nodes.

Since the networks are generated randomly it was necessary to take an average of the algorithm performance on several networks of the same size and density.

To compare the performance of the algorithms on networks varying only by increasing size, networks varying in size from $n = 250$ nodes to $n = 1500$ nodes with constant densities varying from $m/n = 5$ to $m/n = 60$ were generated and explored.

The following sets of networks were generated :

Density	Number of nodes					
5	250	500	750	1000	1250	1500
10	250	500	750	1000	1250	
20	250	500	750	1000		
30	250	500	750	1000		
40	250	500	750			
60	250	500				

Table 12.1 Sets of networks generated.

The above sets of networks allowed for the comparison of the algorithms with regard to increasing size whilst keeping the density constant and increasing density whilst keeping the size constant. It was also possible from the above table of networks generated to compare the effect by increasing the density of a standard size network has on the performance of the various algorithms.

General overview of the performance

As theoretically expected the label setting algorithms dominated the label correcting algorithms with the exception of the threshold algorithm, see figures 12.3, 12.4, 12.5, 12.6, 12.7 and 12.8 for the effect of increasing size, as well as figures 12.9, 12.10 and 12.11 for the effect of only increasing the density of a specific sized network. The top three algorithms in all cases were the Glover-Klingman threshold algorithm, Dials bucket algo-

rithm and the heap algorithm. In general the performance of the remaining label setting algorithms, the Dijkstra and the ordered Dijkstra algorithm followed the top three algorithms. The remaining label correcting algorithms, the queue, deque and partition algorithms performed the worst. The performance of these three algorithms continually interchanged. The deque algorithm showed a certain amount of unstability as its performance at times improved and at other times dramatically deteriorated. To further assess and discuss the algorithms they are grouped according to similarity of structure and performance. The performance of the algorithms are assessed with respect to varying size and density of the network. The performance of the algorithm is compared to that of the predicted worst case analysis and to previous published results of the algorithm implementation.

Dijkstra and the ordered Dijkstra Algorithm

Performance versus theoretical worst case analysis

The performance of these two algorithms was very similar, see figures 9 and 10 in all the the sparse random networks explored. From a theoretical point of view the performance of Dijkstra's algorithm should be similar to the ordered version for sparse networks. The worst case complexity of the Dijkstra algorithm is $O(n^2)$ whereas that of the ordered version is $O(n \times m)$. Considering that a connected network would have at least $n - 1$ arcs (ie $m \geq n - 1$) implies that Dijkstra's algorithm has a better worst case complexity with a significant increase in the density of the network than the ordered version. These complexities also imply that

'kstra's algorithm would be affected exponentially only by size where as the Ordered version would be affected exponentially by size and by the number of arcs for a higher density network m/n

Size : The first experiment evaluated the performance of the algorithms by increasing the size from networks of $n = 250$ nodes to $n = 1500$ nodes, while maintaining a low density $m/n = 5$, see figure 12.1 The results showed the ordered algorithm performing slightly better than the original Dijkstra algorithm. The second experiment explored networks varying in size from $n = 250$ nodes to $n = 1250$ nodes but with a constant density $m/n = 10$, see figure 12.2. The results again showed the ordered algorithm performing better than the original Dijkstra algorithm but by not as a large margin as in the networks of a density $m/n = 5$.

Density : To compare the effect of a network varying only by increasing density on the two algorithms with a constant size $n = 250$ and increasing density $m/n = 5$ to $m/n = 60$ were explored. The experimental results showed the ordered algorithm performing better than the Dijkstra algorithm for networks of lower a density. As the density of the networks increased though, the ordered algorithm performed increasingly worse than the Dijkstra algorithm. See figures 12.9, 12.10 and 12.11.

Comparison to other published results

The results obtained from the computational analysis for the Dijkstra and the ordered algorithm with respect to the other algorithms are not consistent with those produced by Gallo and Pallottino (1988) or Dial,

Glover and Klingman (1979). The results show a much better performance by the original Dijkstra and the ordered algorithm with respect to the other algorithms. My results tend to follow the theoretically projected computational worst case complexity analysis.

Dials bucket and the heap algorithm

Dial's bucket algorithm for networks with a low *dmax* was, as anticipated, of the best performing algorithms. As expected though, with a substantial increase of *dmax*, the running time of Dial's algorithm deteriorated drastically. To combat this problem it was proposed to increase the range of each bucket from the value of 1 to a value relative to *dmax*. Implementation of this idea proved to be successful provided, the varying arc lengths were reasonably evenly distributed among the arcs. It was found for instance that for $dmax = 10000$ a bucket range of 300 would produce a competitive running time relative to the faster algorithms.

Performance versus theoretical worst case analysis

These two algorithms were amongst the three best performing algorithm in all the sets of networks explored. Theoretically from their worst case analysis both algorithms should perform better than any of the other algorithms. The experiments seemed to prove the theoretical predictions with the exception of the threshold algorithm which in many cases beat both the bucket and heap algorithms. When comparing the two algorithms theoretically the heap algorithm ($O(m \times \log_2 n)$) should perform very similarly to the bucket algorithm ($O(m + n \times dmax)$) for very sparse graphs (often in sparse graphs m is $O(n)$ Gallo (1986)) and a low *dmax*.

For a very high *dmax* the heap algorithm should perform better than the bucket algorithm.

Size : The bucket and the heap were amongst the three algorithms least effected by increasing the size of a network whilst maintaining a constant density. The bucket algorithm performed better than of the heap algorithm in all the tests. The performance of the bucket algorithm was better than the heap algorithm in every network explored providing *dmax* was kept low. The difference in performance though between the bucket and the heap algorithms was negligible in comparison to the other algorithms with the exception of the threshold algorithm. The performance of the threshold algorithm seemed to fluctuate either between the heap and bucket algorithms or better than that of the bucket algorithm with regard to increasing the network size. The performance of the threshold algorithm was occasionally worse than the heap algorithm when increasing the size of higher density algorithms.

Density : The performance of both the heap and bucket algorithms were not abnormally affected by the increase of density. Both the algorithms seemed to maintain a certain amount of stability when increasing the density of a network.

Comparison to other published results

Good running times of Dial's bucket algorithm have been published by Gallo and Pallottino (1984), (1986) and(1988). In some publications Gallo and Pallottino (1984) Dial's bucket algorithm dominated all other algorithms including the threshold algorithm, where as in more recent publi-

cations, Glover, Glover and Klingman (1984), Glover, Klingman, Phillips and Schneider (1985) and Gallo and Pallottino (1988) the running times of the threshold algorithm was better than the bucket algorithm. Dial, Glover, Karney and Klingman (1979) found some of the label correcting algorithms, dominating Dials algorithm. The results I obtained correlate the best with the results published by Gallo and Pallottino (1988). Reasons for varying results could be explained by the use of different hardwar and software. Through out the publications, Dial's algorithm was always examined for networks with varying arc lengths. This factor played important role in the evaluation of Dial's algorithm. When using a bucket range of 1 in a network with large *dmax* it was discovered that the algorithm performed poorly, as found in all the published results. As mentioned this problem was partly rectified by allowing the user to vary the size of the bucket range relative to *dmax*.

ue, deque and the partition algorithm

Performance versus theoretical worst case analysis

The queue, deque and partition algorithms all performed poorly in comparison to the label setting algorithms. When considering the theoretical worst case complexities of these three algorithms they are not expected to perform as well as most of the label setting algorithms. Previously published results however indicate that the queue and deque algorithms perform relatively well in comparison to the label setting algorithms. The queue and partition algorithms are essentially the same algorithm with slightly different data structures. The worst case complexities of these two algorithms are $O(n \times m)$ which is the same as that of Dijkstra's or-

dered version. The worst case complexity of the deque algorithm is the worst of all the algorithms being $O(n \times 2^n)$. These three algorithms are expected to perform at least as bad as the ordered version of Dijkstra's algorithm. The queue and partition algorithm should be effected by both the size n and the density m/n for high density networks as their worst case complexities are of the order of both m and n.

Size : The queue, deque and partition algorithms performed poorly with respect to increasing the network size. The running times of these three methods seemed to escalate when the size of the networks was increased. The times also seemed to be somewhat unstable. The running time of each algorithm, at times being effected very differently by different network size. With the lower density networks the queue algorithm performed the worst out of the three with respect to increasing size. Deque performed the best in the lower density networks with respect to increasing the network size only. Increasing the size in the higher density networks seemed to effect deque the worst. The queue and partition algorithm had similar performance, the partition performing better in the higher density networks. The deque was the most unstable of the three algorithms producing erratic results at times.

Density : The performance of the queue, deque and partition algorithms was rather unstable with respect to density changes of networks making it difficult to comment on the varying densities. The effect of density change was similar with the queue and partition algorithms, which although somewhat unstable became increasingly worse as the density increased. The performance of deque became increasingly worse and more unstable as the density increased.

Comparison to other published results

Most of the published results showed that the label correcting algorithms, within particular queue and deque performing competitively against label correcting algorithms such as the Dijkstra, ordered, heap and sometimes Dial's bucket algorithm, Gallo and Pallottino (1988), Dial, Glover, Karney and Klingman (1979), Glover, Glover, Klingman (1984). The implementation and experiments of these algorithms did not however reflect this. My results reflected rather poor performances of all the label correcting algorithms with the exception of the threshold algorithm.

Threshold Algorithm

Glover, Klingman, Phillips and Schneider (1985) suggest re-evaluating the threshold value t by merely incrementing the previous threshold value (see section 9.3, page 59) The initial threshold value is still evaluated heuristically from the data in the *NEXT* list as well as the density of the network. This approach was implemented with an additional extension of allowing the user to initially enter the size of the increment. The idea of using an increment of the previous threshold value was further developed, to increment the threshold by a percentage of the previous threshold value. The percentage to be used is initially entered by the user.

At the iteration k the new threshold value t_{k-1} would be $t_{k-1} + (t_{k-1} \times PTGE)$, where $PTGE$ is a percentage initially entered by the user.

The reason for the threshold algorithm extensions is as follows : It became obvious when using an increment of 1 to update the threshold

value, as suggested by Glover, Klingman, Phillips and Schneider (1985) the larger *dmax* was the longer the running time of the algorithm. Included with the prolonged running time was the increase in the number of times that the threshold value was re-evaluated whilst exploring the network. It became evident that the size of the increment should be relative to *dmax* provided the variation of *dmax* is reasonably evenly distributed. The user is thus allowed to enter size of the threshold increment.

It may be intuitive that as the shortest path is found from node to node with a near label setting algorithm such as that of the threshold algorithm, and provided that the arc length is reasonably evenly distributed in the given network, the length of the shortest path grows as an approximate percentage of itself. Taking this idea into account, it was proposed to increment the threshold value as a percentage of itself. This approach was implemented with surprisingly good results for larger networks with a higher density.

In general with particular reference to the more sparse larger networks the algorithm using a fixed size increment appeared to dominate the other threshold algorithms. The running time of this algorithm was used to represent the threshold algorithms in comparison to the other algorithms. It was found using a standard *dmax* of 20 and increment value of 3 gave the best results.

The theoretical worst case complexity of the threshold algorithm ($O(m \times n^2)$) is the second worst of all the algorithms analysed. Published results Glover, Klingman, Phillips and Schneider (1985) and Gallo and Pallottino (1988) of the threshold algorithm indicate that it is one of the fastest algorithms implemented to date. The performance of the thresh-

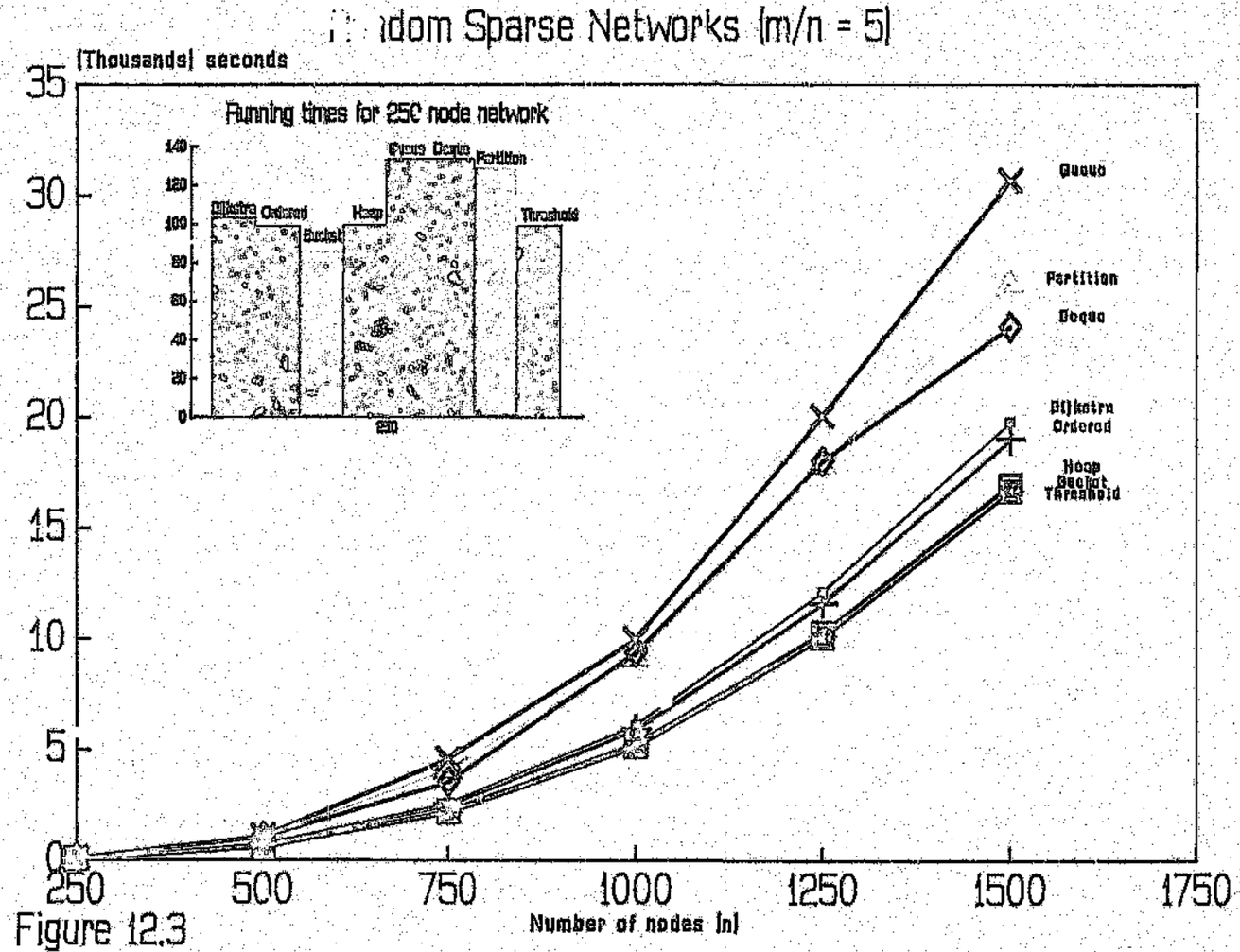
old algorithm was analysed with the most interest of all the algorithms. In most of the random sparse networks explored the threshold algorithm performed better than the other algorithms.

Size : The performance of the threshold algorithm with regard to increasing size was possibly the best of all the algorithms. The running time of the threshold algorithm seemed to occasionally fluctuate with that of the bucket algorithm. In most of the networks explored the threshold algorithm beat the bucket algorithm as the size of the network increased whilst keeping the density constant.

Density : The threshold algorithm seemed relatively unaffected by the increasing density of the networks explored. It did not however perform as well in larger networks with a high density, see figures 12.3 and 12.4.

Comparison to other published results

The only publication that presents poor results of the threshold algorithm is Gallo and Pallottino (1984). Results published by Gallo and Pallottino (1988) seemed to rectify the performance of the threshold algorithm. All publications mentioned agree that the threshold algorithm is one of the best performing algorithms. The results obtained correlate the most with those published by Gallo and Pallottino (1988), which although show a good performance by the threshold algorithm, do not show the threshold algorithm as good a algorithm as shown by Glover, Glover and Klingman (1984) and Glover, Klingman, Phillips and Schneider (1985)


Random Sparse Networks (m/n = 5)
(Thousands) seconds
Running times for 250 node network
Dijkstra
Ordered
Bucket
Heap
Fibonacci
Deque
Partition
Threshold
250
Queue
Partition
Deque
Dijkstra
Ordered
Heap
Bucket
Threshold
35
30
25
20
15
10
5
0
250
500
750
1000
1250
1500
1750
Number of nodes (n)
Figure 12.3

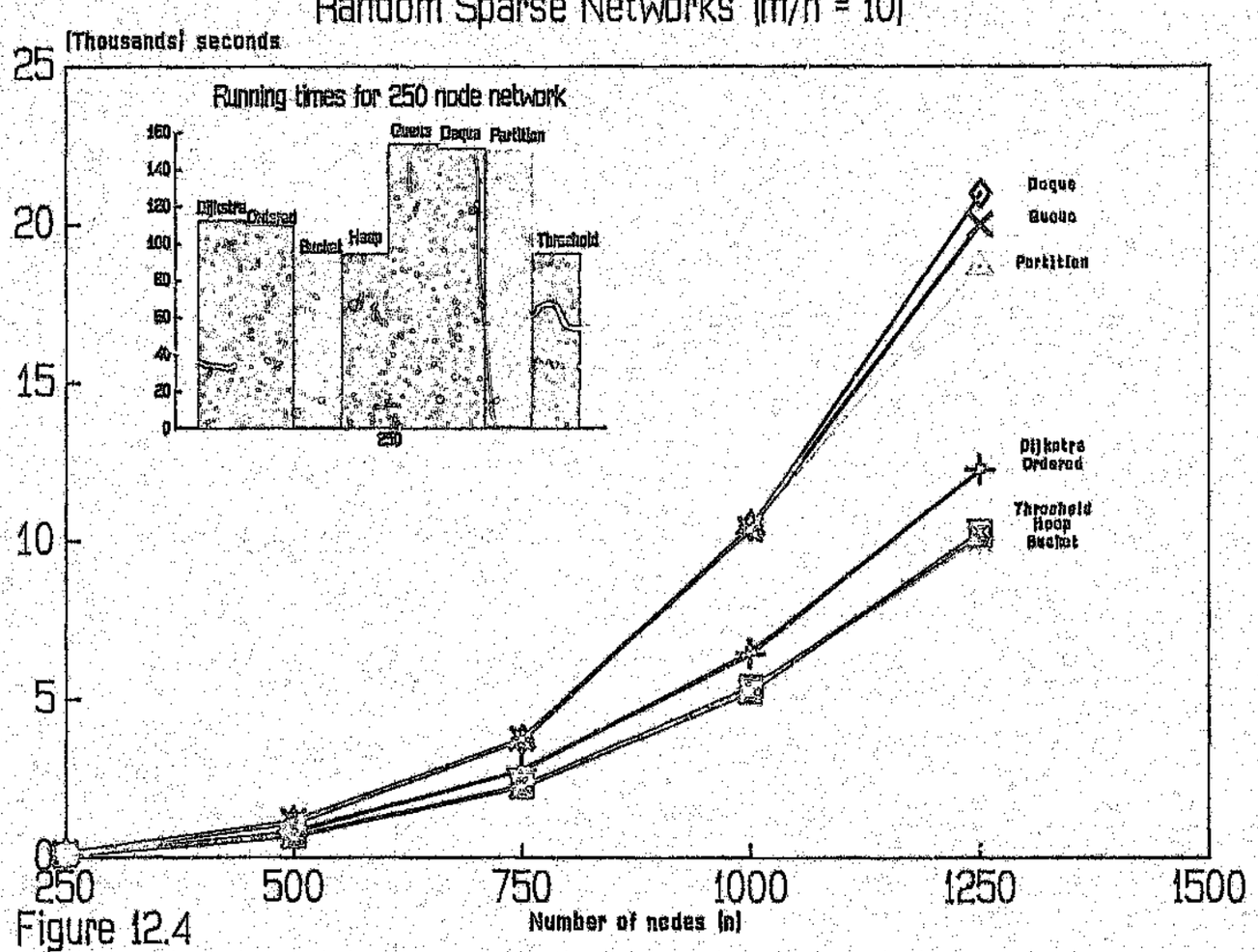

Random Sparse Networks (m/n = 10)
(Thousands) seconds
25
20
15
10
5
0
Running times for 250 node network
160
140
120
100
80
60
40
20
0
Dijkstra
Ordered
Bucket
Heap
Queue
Dequa
Partition
Threshold
250
Dequa
Queue
Partition
Dijkstra
Ordered
Threshold
Heap
Bucket
250
500
750
1000
1250
1500
Number of nodes (n)

Figure 12.4

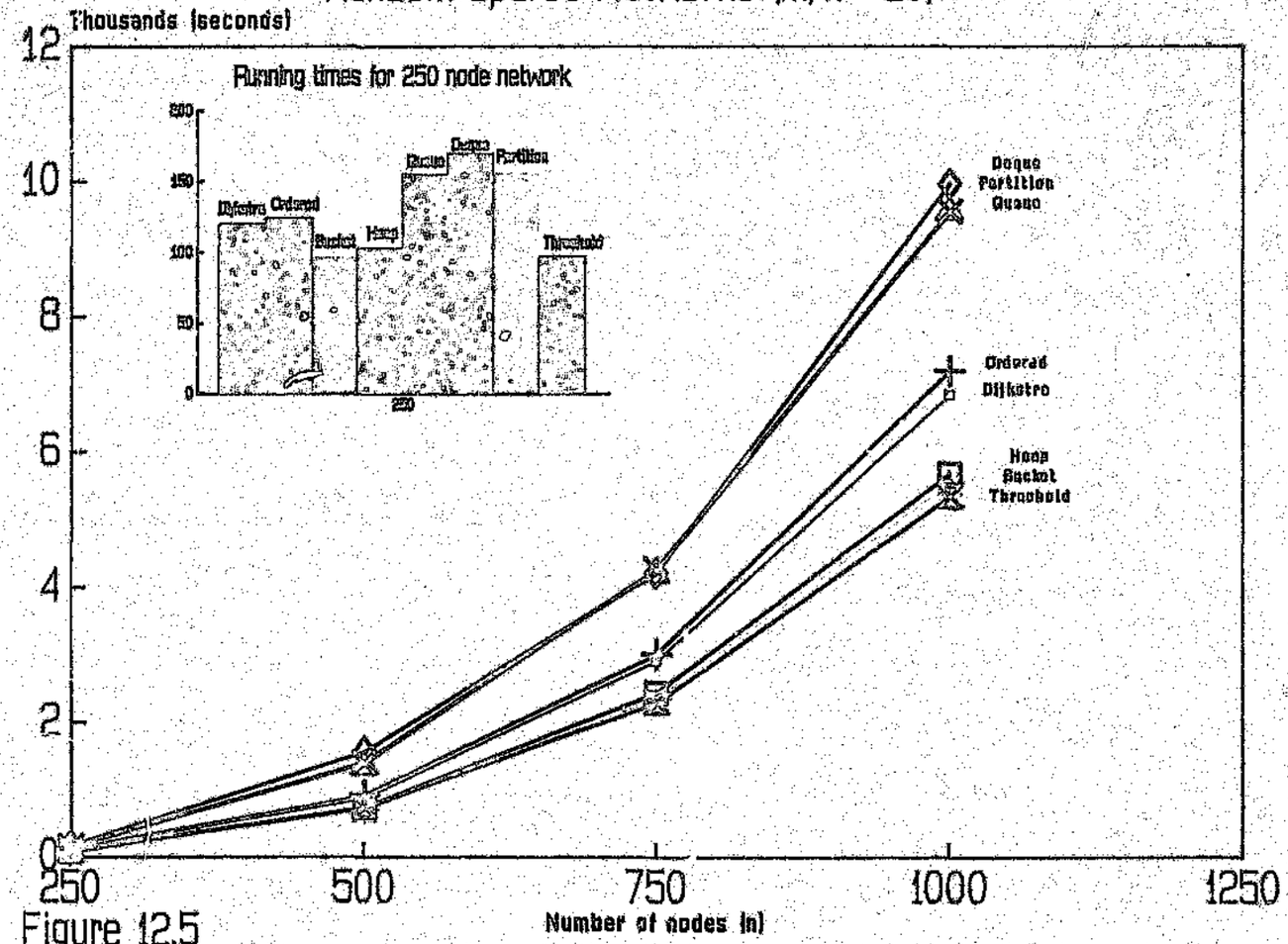

Random Sparse Networks (m/n = 20)
Thousands (seconds)
12
10
8
6
4
2
0
Running times for 250 node network
Dijkstra
Ordered
Bucket
Heap
Deque
Partition
Threshold
250
Deque
Partition
Queue
Ordered
Dijkstra
Heap
Bucket
Threshold
250
500
750
1000
1250
Number of nodes (n)

Figure 12.5

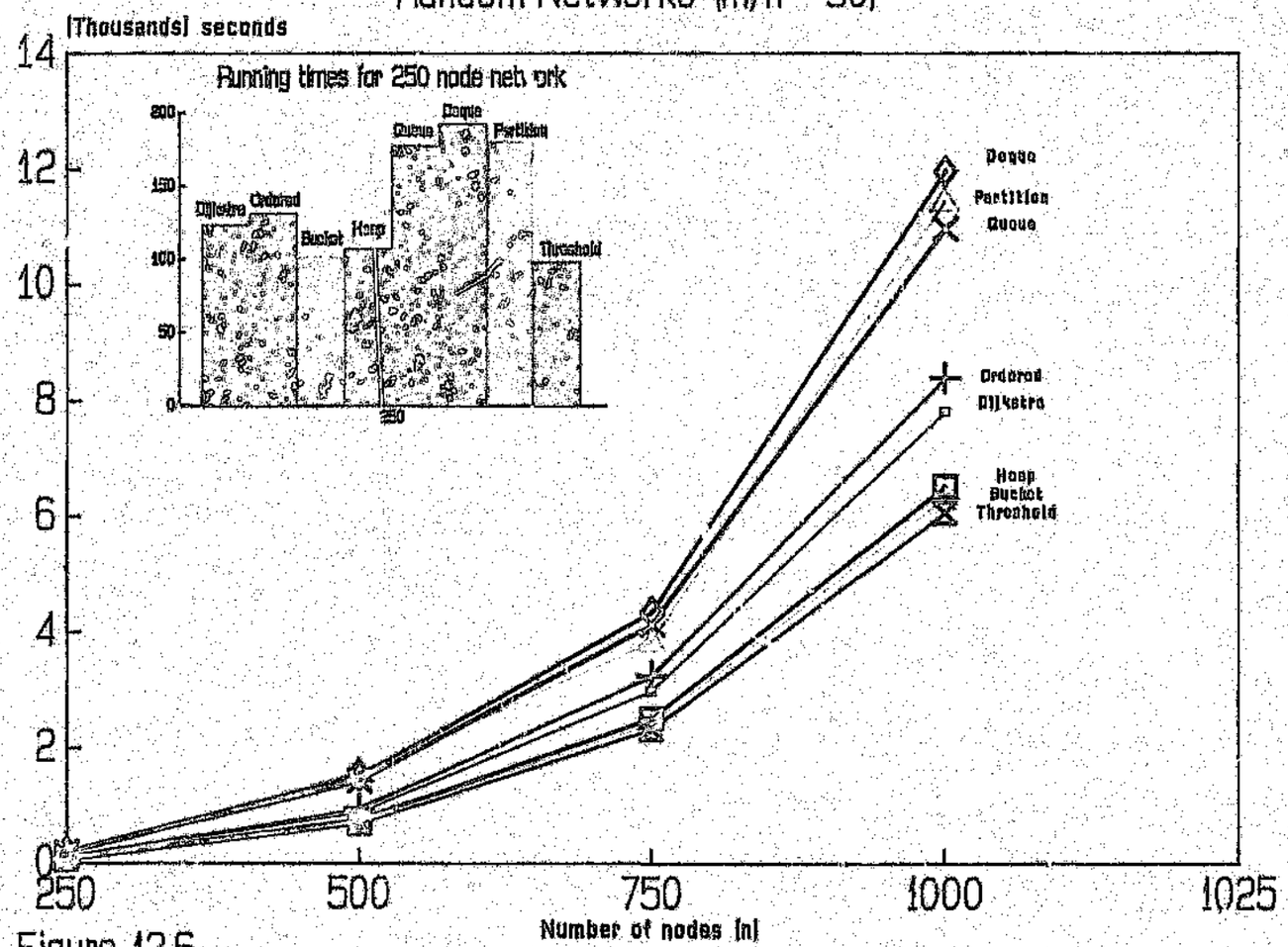

Random Networks (m/n = 30)
(Thousands) seconds
Running times for 250 node network
Dijkstra
Ordered
Bucket
Heap
Queue
Deque
Partition
Threshold
14
12
10
8
6
4
2
0
250
500
750
1000
1025
Number of nodes (n)

Figure 12.6

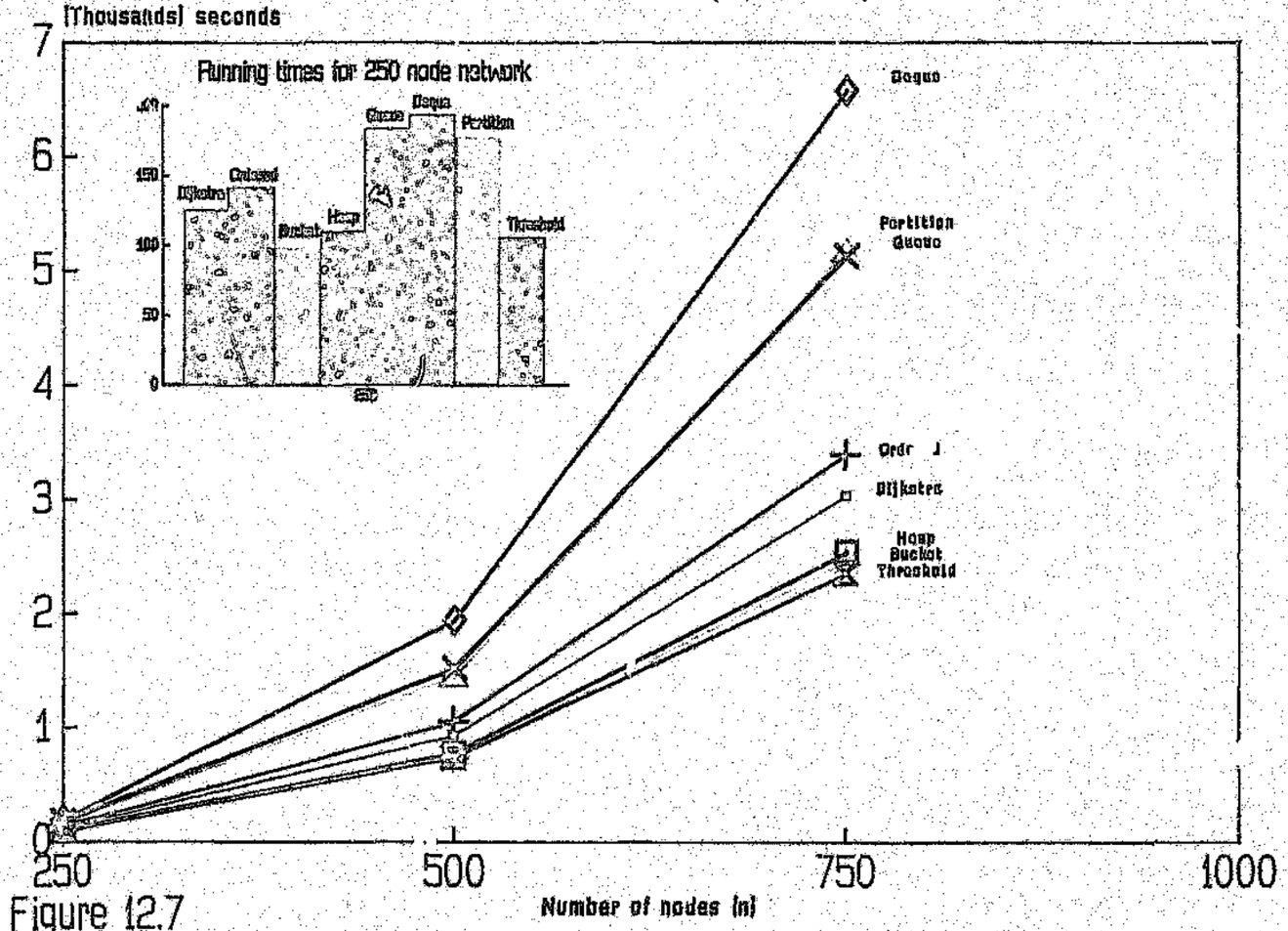

Random Networks (m/n = 40)
(Thousands) seconds
7
6
5
4
3
2
1
0
Running times for 250 node network
Dijkstra
Ordered
Bucket
Heap
Order
Dequo
Partition
Threshold
250
500
750
1000
Number of nodes (n)
Dequo
Partition
Quoko
Order
Dijkstra
Heap
Bucket
Threshold

Figure 12.7

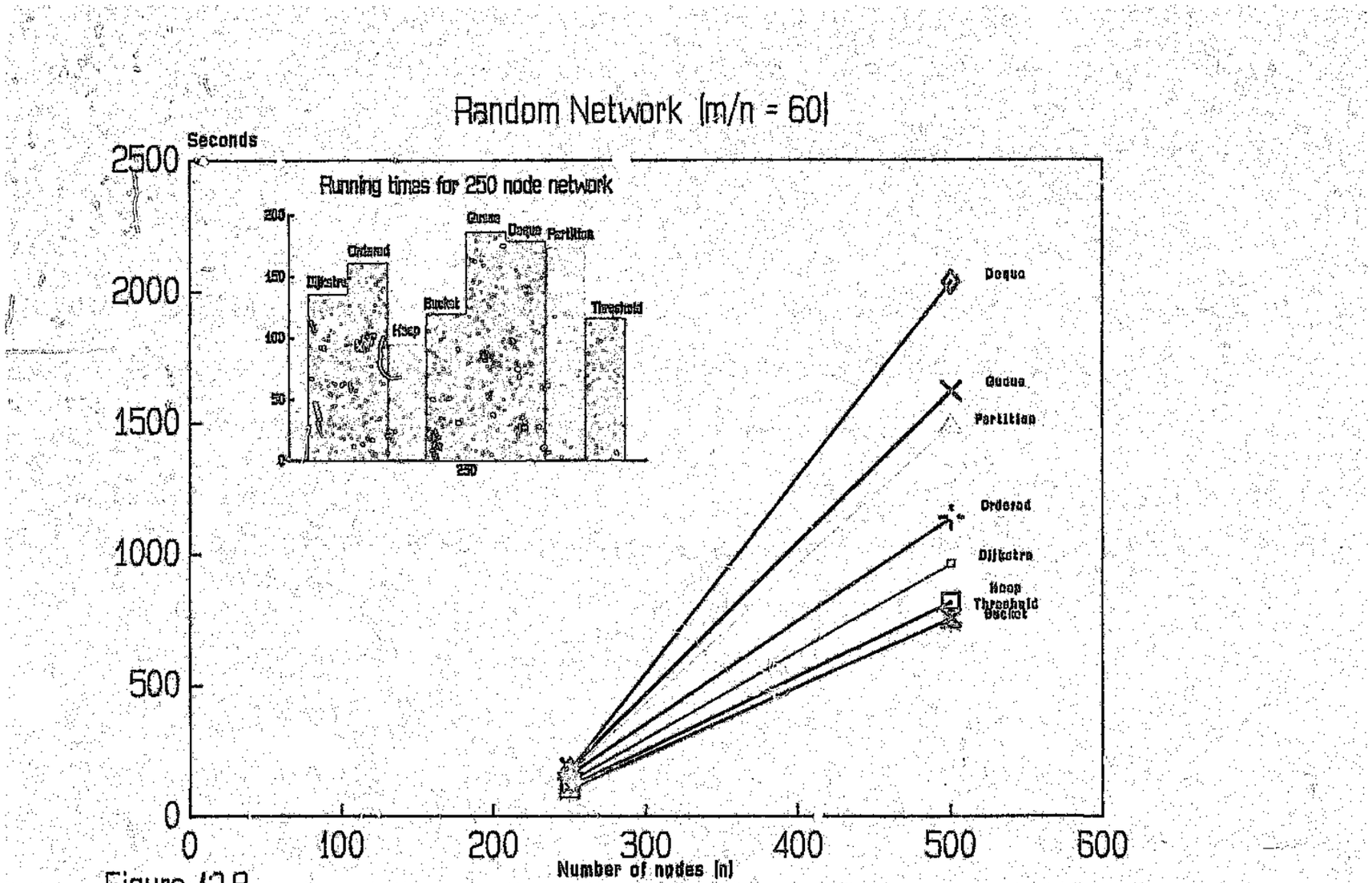

Random Network (m/n = 60)
Seconds
2500
2000
1500
1000
500
0
Running times for 250 node network
200
150
100
50
0
Dijkstra
Ordered
Heap
Bucket
Queue
Deque
Partition
Threshold
250
Deque
Queue
Partition
Ordered
Dijkstra
Heap
Threshold
Bucket
0
100
200
300
400
500
600
Number of nodes (n)

Figure 12.8

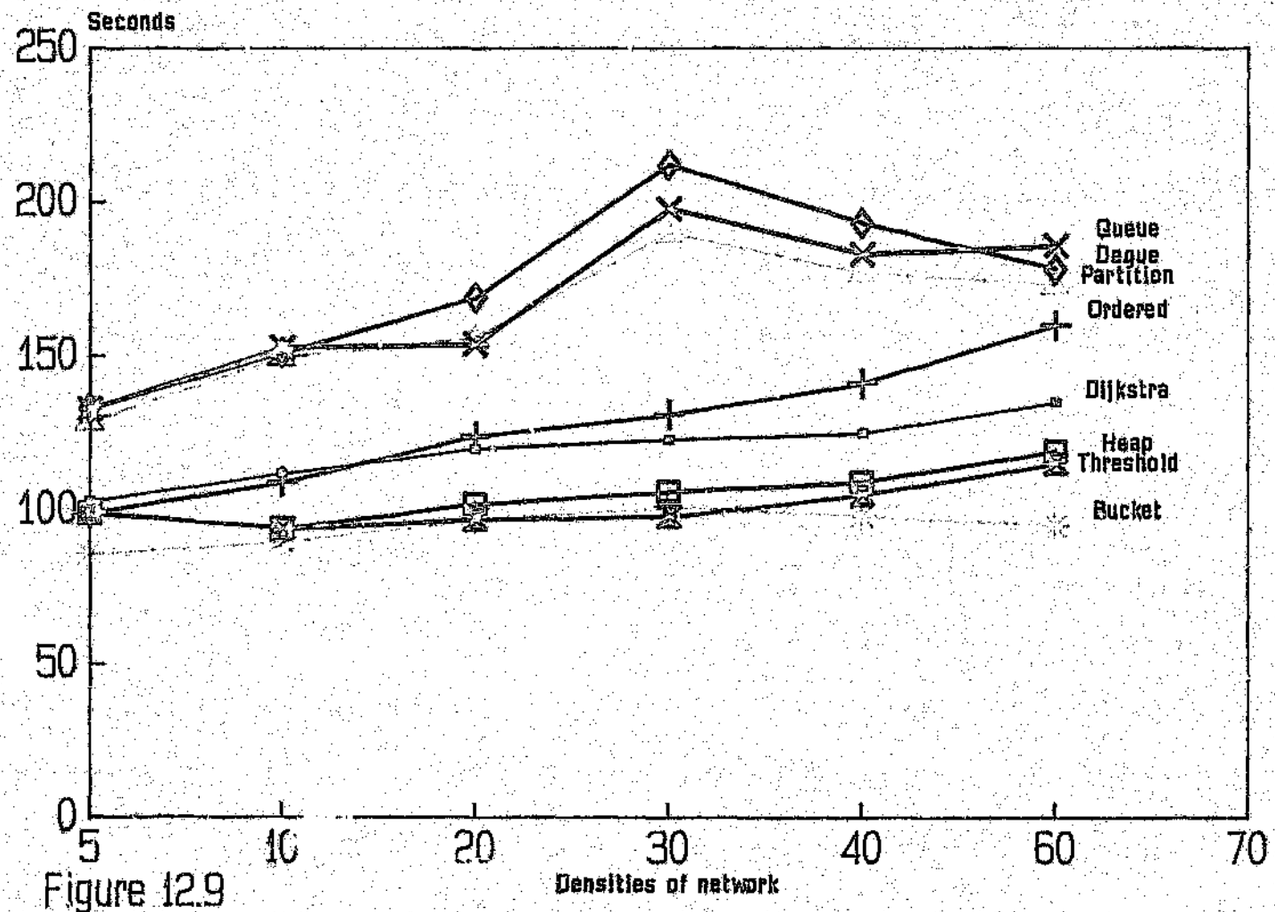

Effect of Density change on algorithms
Number of nodes 250
Seconds
250
200
150
100
50
0
5
10
20
30
40
60
70
Densities of network
Queue
Deque
Partition
Ordered
Dijkstra
Heap
Threshold
Bucket

Figure 12.9

Effect of Density change on algorithms
Number of nodes 500

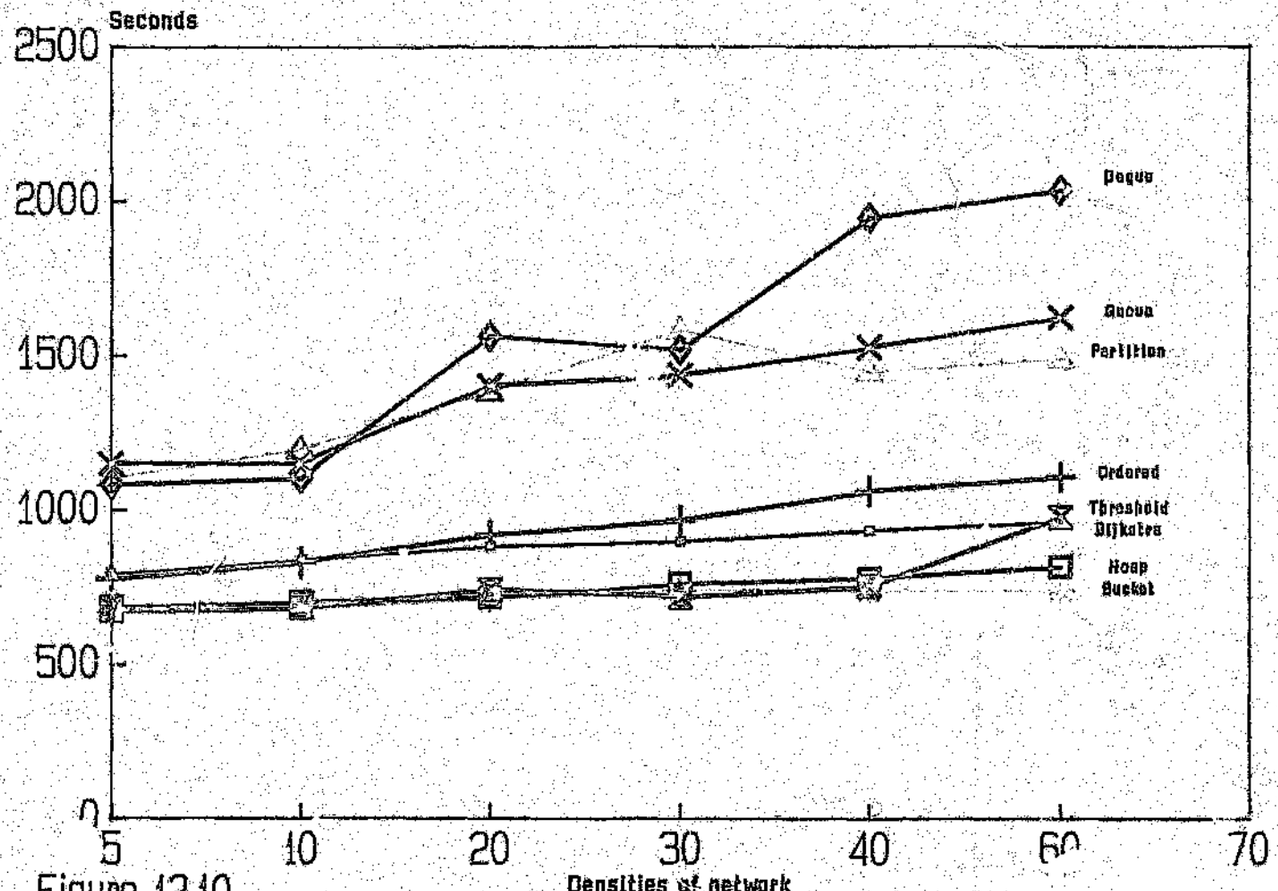


Figure 12.10

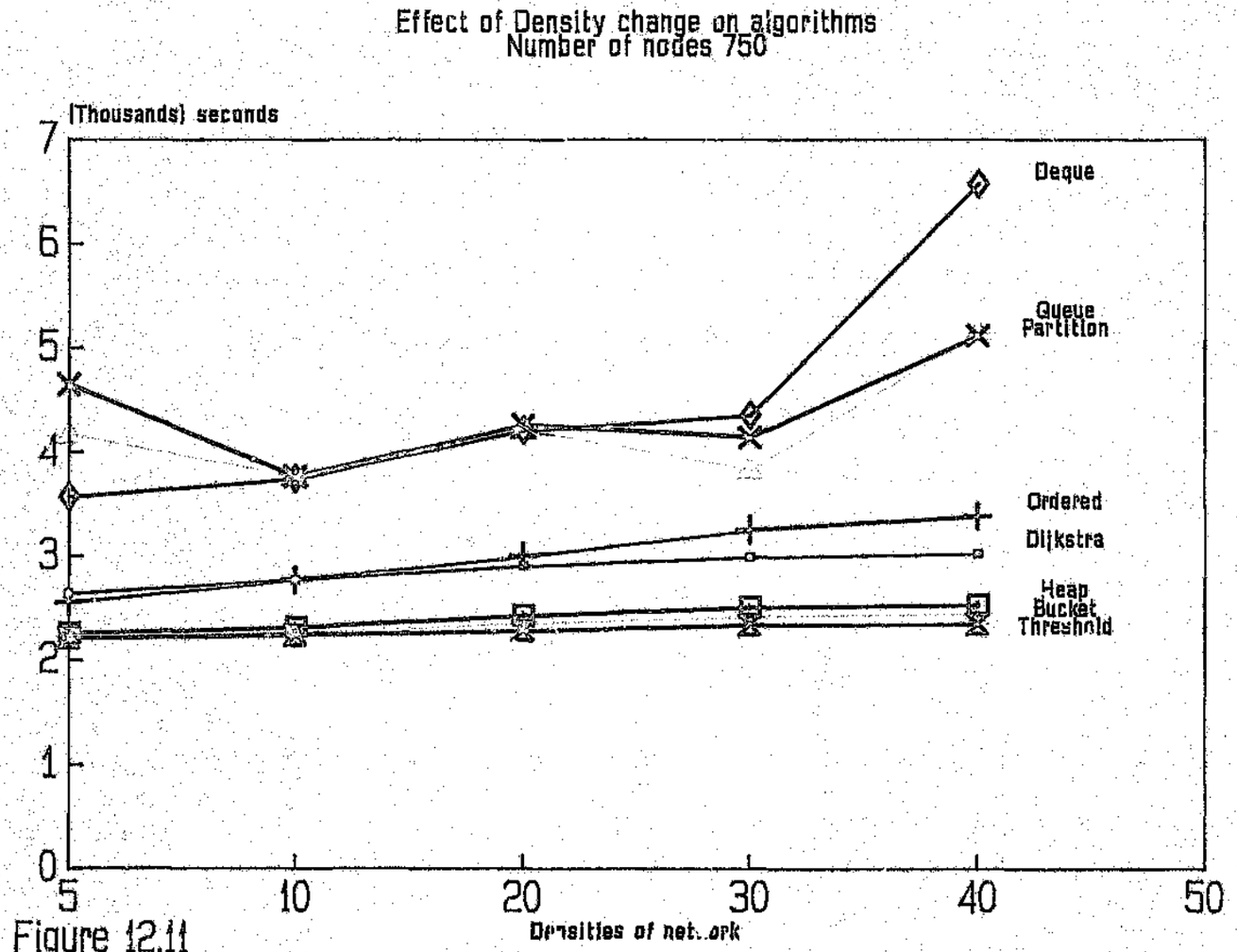

Effect of Density change on algorithms
Number of nodes 750
(Thousands) seconds
7
6
5
4
3
2
1
0
5
10
20
30
40
50
Densities of network
Deque
Queue Partition
Ordered
Dijkstra
Heap
Bucket
Threshold

Figure 12.11

Table 12.1 Smaller Random Networks

m/n	Nodes	Dijkstra	Ord	Bucket	Heap	Queue	Deque	PSP	Thsh I	Thsh P
20	100	10	10	7	8	13	14	13	8	9
20	200	62	66	51	54	85	85	95	51	53
20	300	199	207	162	168	265	299	270	160	162
20	400	469	499	382	389	759	777	715	396	377
20	500	885	919	716	742	1404	1564	1386	742	705
10	200	39	59	47	50	90	73	91	50	51
10	400	432	436	357	362	586	554	530	360	359
10	300	3185	3142	2643	2663	5165	5065	4827	4980	4980
10	1000	5880	5820	5291	5091	9426	8550	9053	5379	5439

Table 12.2 Random Networks With Varying Density

m/n	Nodes	Dijkstra	Ord	Bucket	Heap	Queue	Deque	PSP	Thsh I	Thsh P
5	250	103	99	86	99	133	133	129	99	102
5	500	794	775	670	686	1150	1084	1107	673	702
5	750	2643	2562	2224	2263	4652	3574	4173	2216	2274
5	1000	6167	5969	5236	5303	10056	9419	9265	5205	5377
5	1250	12160	11660	10156	10287	20107	18042	17941	10110	10536
5	1500	19810	19054	16734	16953	30733	24091	26117	16703	17225
10	250	112	109	90	94	153	151	110	94	92
10	500	841	831	685	703	1154	1105	1198	682	702
10	750	2778	2783	2273	2319	3768	3751	3765	2248	2297
10	1000	6534	6446	5347	5429	10382	10483	10510	5256	5362
10	1250	12282	122[illegible]	10021	10191	19992	20982	18683	12960	12960
20	250	120	124	97	102	154	160	156	97	96
20	500	885	919	716	742	1404	1564	1386	742	705
20	750	2911	3009	2362	2433	4261	4213	4222	2291	2364
20	1000	6844	7194	5547	5683	9563	9984	9582	5341	5341
30	250	123	131	101	106	198	212	190	98	98
30	500	900	966	743	764	1437	1579	1583	717	710
30	750	2996	3261	2426	2513	4142	4356	3834	2348	2341
30	1000	7808	8405	6315	6513	10968	11557	11432	6071	6058
40	250	125	141	96	109	183	193	177	105	102
40	500	933	1059	738	781	1525	1946	1448	752	737
40	750	3031	3397	2458	2536	5126	6576	5158	2357	2358
60	250	135	160	95	119	186	178	173	115	109
60	500	962	1106	739	818	1623	2036	1490	976	754

13.0 Conclusion

The purpose of this research report was to survey the most recent state-of-the-art shortest path algorithms from both a theoretical and a computational point of view. This type of survey has been done and published by Dial, Glover, Karney and Klingman (1979) and Gallo and Pallottino (1984), (1986) and (1988). The aim of this survey was to re-explore the prominent methods with their latest extensions and to verify previous computational results.

To conclude the survey, the user is brought to mind, as it is he/she who have to choose the algorithm to explore the network(s) with which they are confronted. The survey has analysed the problem in two ways, namely theoretically (priori analysis) and computationally (Posteriori analysis).

The advantage of a theoretical analysis is that it is software and hardware independent. Theoretically the label setting algorithms seem to be far superior. The bucket and heap algorithms are of a linear order for low density, sparse networks ($m << n^2$ and usually $m = O(n)$). Where as all the label correcting algorithms are at least of an exponential order. The user may be tempted to rule out the label correcting algorithms altogether. However since using worst case computational analysis to judge the performance of an algorithm may be misleading, we turn to the experimental approach of computational analyses.

The disadvantage of computational analysis is that it is software and hardware dependent. The application and environment used to code and solve the problem may to some degree limit the users choice. The computational results of this survey, at times, do not correlate with previously published computational results. A reasonable explanation for this, it is felt is the different software and hardware used. Taking this factor into account, these results, together with most of the published results, found that the threshold algorithm was the only algorithm universally favoured. Ironically this algorithm has one of the worst computational complexities.

From the results of the computational analysis the bucket algorithm and the threshold algorithm seem to be on a par. It was found that both algorithms may suffer if there are few arcs with a large *dmax* unevenly distributed within the network. The heap algorithm may be a good stable choice as although it may not have been the best performer, it was consistent and its running time seemed to be most certainly within reach of the threshold and bucket algorithm. The heap algorithm is not affected by any abnormal large arc lengths. The heap algorithm has the best computational complexity.

The prominence displayed by the threshold algorithm should render it to further study and refinement. A further possibility is to combine the more prominent label correcting algorithms, the bucket and heap algorithms with the threshold algorithm. This may done by making NEXT a priority queue.

14.0 Notation Used

n - Number of nodes in a given network

m - Number of arcs in a given network

Q - Queue or list of Scan Eligible nodes

n_q - Cardinallity of Q

r - Root (Start or Source) node.

A - Set of arcs

N - Set of nodes

d_{ij} - Distance from node i to node

$FS(u)$ - Forward star of node u

k - Width of the buckets

$dmax$ - Upper Bound of arc lengths

p - Predecessor vector

q - Vector of pointers to buckets

Q', Q'' - Subsets of Q

NEXT, NOW, NOW' - Partitioned subsets of Q

t - Threshold value

15.0 References

A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Compu:. Algorithms* (Addison-Wesley, Reading, Mass., 1974)

R. Ahuja, K Melhorn, J. Orlin and R.E. Tarjan, Faster Algorithms for the shortest path problem, Princeton University (1988)

R. Bellman, On a Routing problem, Quart. Appl. Math. 16(1958)88.

G.B.Dantzig, On the shortest root through a network, *Studies in Graph Theory Part I* (The Mathematical Association of America, 1975)

E.V. Denardo, B.L. Fox, Shortest Route Methods : 1 Reaching Pruning and Buckets, Oper. Res. 27(1979)161-186

R.B. Dial, F. Glover, D. Karney and D. Klingman, A computational analysis of alternative algorithms and labelling techniques for finding shortest path trees, Networks 9, 3(1979)215.

M.Florian, S. Nguyen and S. Pallottino, A dual simplex algorithm for finding all shortest paths, Networks 11(1981)367.

M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in Improved Network Optimization Algorithms, Journal of the ACM (1987)596-615

G. Gallo and S. Pallottino, A new algorithm to find the shortest paths between all pairs of nodes, Discr. Appl. Math. 4(1982)23

G. Gallo and S. Pallottino, Shortest path methods in Transportation Planning Models, (North Holland) (1984)

G. Gallo and S. Pallottino, Shortest path methods: A unifying approach, Mathematical Programming Study 26(1986)38

G. Gallo and S. Pallottino, Shortest path algorithms, Annals of Operations Research 13(1988)3-79

F. Glover, R. Glover and D. Klingman, Computational study of an improved shortest path algorithm, Networks 14(1984)25

F. Glover, D. Klingman and N. Phillips, A new polynomially bounded shortest path algorithm, Oper. Res. 33(1985)65

F. Glover, D. Klingman, N. Phillips and R. Schneider, New polynomial shortest path algorithms and their computational attributes, Management Science 31(1985)

F. Glover, R. Glover and D. Klingman, Threshold Assignment Algorithm, Mathematical Programming Study 26(1986)12-37

F. Glover and D. Klingman New Sharpness properties, Algorithms and Complexity Bounds for Partitioning Shortest path procedures, Oper. Res. 37(1989)

P. Hansen An O(m.log D) algorithm for shortest paths, Discrete Applied Maths 2(1980)151-153

D.B. Johnson, A note on Dijkstra's shortest path algorithm, J.A.C.M. 20,3(1973)385

D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, J.A.C.M. 24,1(1977)1.

A. Kershenbaum, A note on finding shortest path trees. Networks 11(1981)399

R.G. Karlsson and P.V. Poblete, An O(m log log D algorithm for shortest paths. Discrete Applied Maths. 6(1983)91-93

D.E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms* (Addison-Wesley, Reading, Mass., 1973)

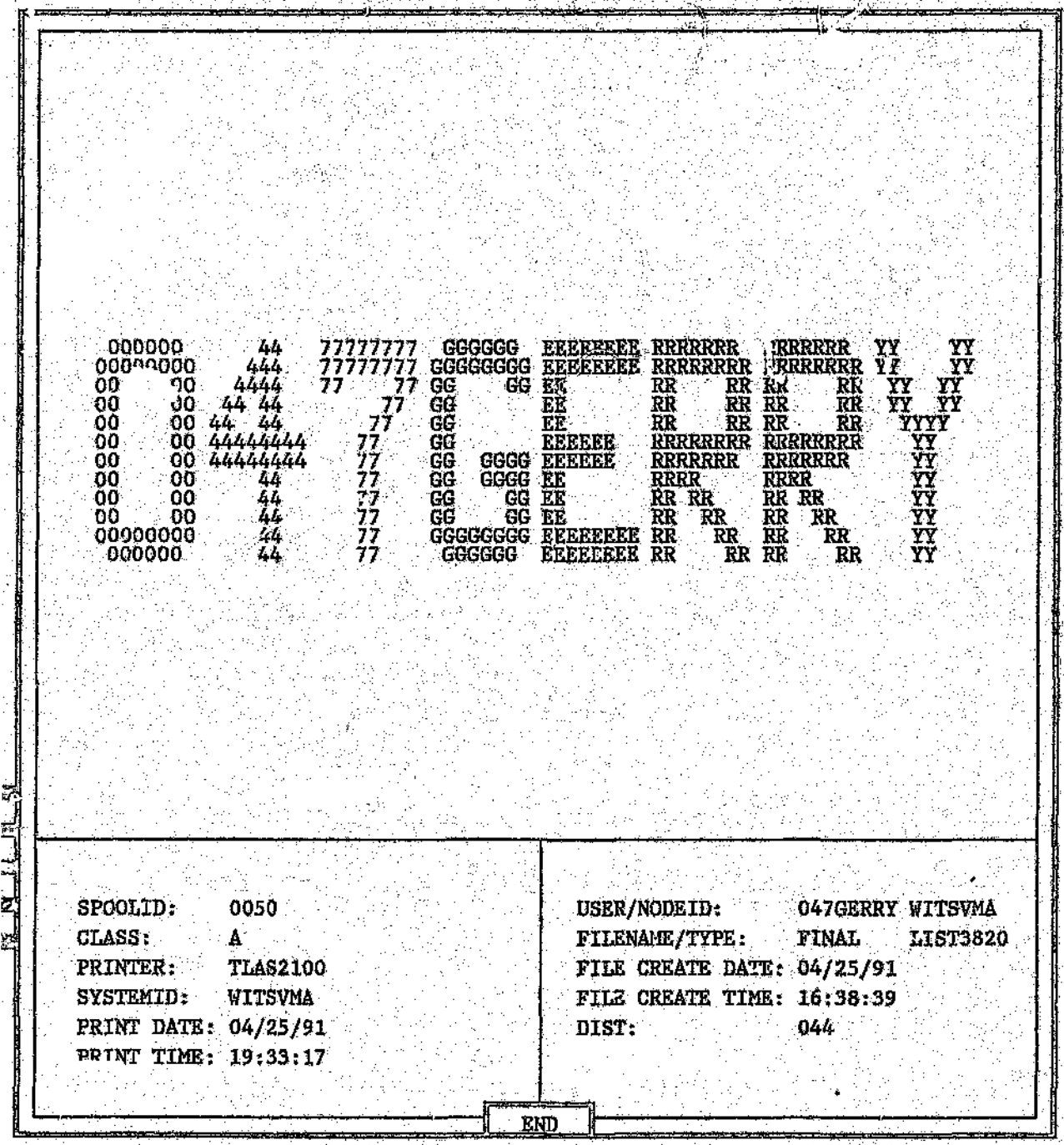
D.E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching* (Addison-Wesley, Reading, Mass., 1973)

S. Pallottino, Shortest Path Methods: Complexity, interrelations and new propositions, Networks 14(1984)257

U. Pape, Algorithm 562: Shortest Path lengths, A.C.M. Transactions on Mathematical Software 6(1980)450

U. Pape, Remark on Algorithm 562, A.C.M. Transactions on Mathematical Software 26(1983)260

J.Y. Yen, Finding the length of all shortest paths in n-node nonnegative-distance complete networks using $n^3/2$ additions and n^3 comparisons, Journal of the ACM 19(1972)423-424



SPOOLID: 0050
CLASS: A
PRINTER: TLAS2100
SYSTEMID: WITSVMA
PRINT DATE: 04/25/91
PRINT TIME: 19:33:17

USER/NODEID: 047GERRY WITSVMA
FILENAME/TYPE: FINAL LIST3820
FILE CREATE DATE: 04/25/91
FILE CREATE TIME: 16:38:39
DIST: 044

END

Author: Comninos Gerry.
Name of thesis: A Survey Of The Most Recent State-of-the Art Shortest Path Algorithms And Their Applications To Different Types Of Networks.

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2015

LEGALNOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page)for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.