

FAST ALGORITHMS FOR

SHORTEST PATHS

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Alistair Moffat

University of Canterbury

1985

QA
278.3
.M695
1985

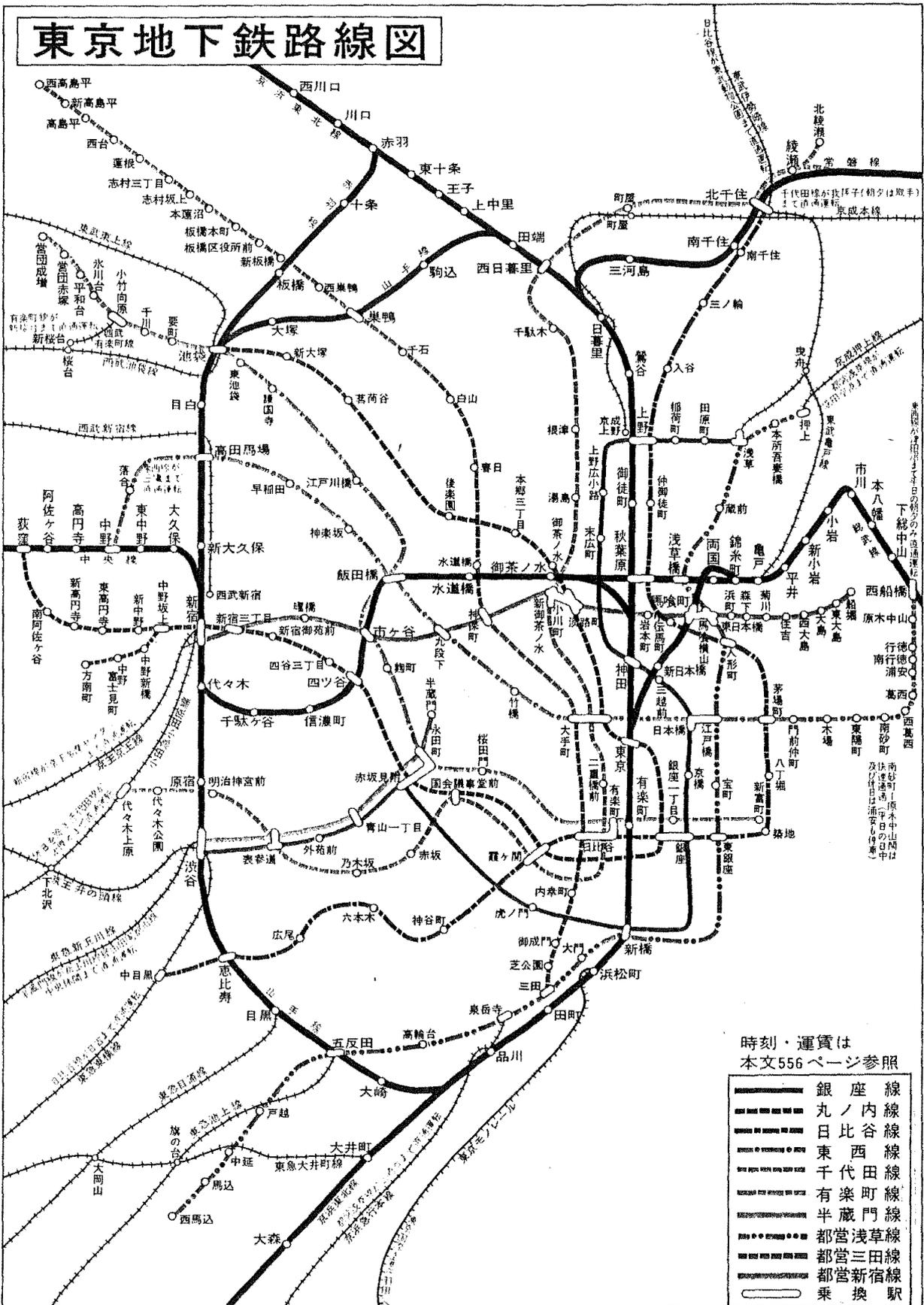
CONTENTS

Abstract	page 1
1. Introduction	2
1.1 Overview of shortest paths and algorithms	2
1.2 Applications of shortest paths	7
1.3 Definitions and notation	10
1.4 Overview of the thesis	24
2. Extant results	26
2.1 Upper bounds and algorithms	26
2.2 Lower bounds	33
3. A good worst case algorithm	37
3.1 Background	37
3.2 The greedy paradigm	40
3.3 Priority queue implementation	48
3.4 Some further observations	54
3.5 Extending the queue to Dijkstra's algorithm	59
4. An $O(n^2 \log n)$ average time algorithm	62
4.1 The Dantzig/Spira paradigm	62
4.2 Fredman's modification	74
4.3 Bloniarz's modification	78
4.4 The new algorithm	81
4.5 Experimental results	90

5. Some implementation notes	95
5.1 Background	95
5.2 The effect of unlimited scanning	96
5.3 The effect of cost-compression	105
5.4 Implementing continuous cleaning	112
5.5 Experiments on "worst case" graphs	121
6. Distance matrix multiplication	125
6.1 Background	125
6.2 An $O(n^2 \log n)$ average time dmm algorithm	128
6.3 An $O(\text{sqrtn})$ average time inner product algorithm	131
6.4 A fast hybrid algorithm for dmm	137
6.5 Experimental results	143
6.6 A further observation on calculating inner products	149
7. Summary of main results	151
Appendix 1. Experimental results for apsp algorithms	156
Appendix 2. Proof of lemma 5.2	160
Glossary	163
Acknowledgements	165
Publications	166
References	168

algorism, algorithm, ns. 1. Arabic (decimal)
notation of numbers. 2. Process or rules for
(esp. machine) calculation etc., hence algorithmic a.

The Concise Oxford Dictionary,
Sixth Edition, 1976.



Shortest paths in action -
the Tokyo subway network.

ABSTRACT

The problem of finding all shortest paths in a non-negatively weighted directed graph is addressed, and a number of new algorithms for solving this problem on a graph of n vertices and m edges are given. The first of these requires in the worst case $\min\{ 2mn, n^3 \} + O(n^{2.5})$ addition and binary comparisons on path and edge costs, improving the previous bound (Dantzig, 1960) of $n^3 + O(n^2 \log n)$ operations in a computational model where addition and comparison are the only operations permitted on path costs.

The second algorithm presented, and the main result of this thesis, has an expected running time of $O(n^2 \log n)$ on graphs with edge weights drawn from an endpoint independent probability distribution, improving asymptotically the previous bound (Bloniarz, 1980) of $O(n^2 \log n \log^* n)$, and resolving a major open problem (Bloniarz, 1983) concerning the complexity of the all pairs shortest path problem. Some variations on the new algorithm are analysed, and it is shown that two superficially good heuristics have a bad effect on the running time. A third variation reduces the worst case running time to $O(n^3)$, making the method competitive with the $O(n^3)$ classical algorithms of Dijkstra (1959) and Floyd (1962). The new algorithm is not just of theoretical interest - experimental results are given that show the algorithm to be fast for operational use, running an order of magnitude faster than the algorithms of Dijkstra and Floyd.

The closely linked problem of distance matrix multiplication is also investigated, and a number of fast average time distance matrix multiplication algorithms are given.

CHAPTER ONE
INTRODUCTION.

1.1 Overview of Shortest Paths and Algorithms.

A student stands in the lobby of Christchurch airport, carrying in one hand a bag bearing the label "Tokyo or bust", and in the other a schedule of fares for airline services in the west Pacific and Orient. His objective is to arrive in Tokyo having paid the minimum possible airfare, and as he stands in the lobby he is trying to calculate whether the first segment of his journey should be to Sydney or to Auckland; and whether then it is better to head for Singapore or Honolulu or Hong Kong or Nadi, or to fly directly to Tokyo. "Out of so many possibilities", he muses, "which is the cheapest?". [Figure 1.1].

If the problem size is small, as in the illustrated case, and involves only a handful of possible transit cities, a simple exhaustive enumeration of possible paths is enough to find the shortest. However, if tens or hundreds of transit points must be considered, and exhaustive search is the only technique used in solving the problem, the student is likely to still be in the lobby calculating after the last flight for the day has departed. Clearly, to solve anything other than trivial instances of such a shortest path problem, an efficient algorithm is required - the traveller needs a set of rules that allow for quick consideration of good candidate paths and easy elimination of uneconomic routes.

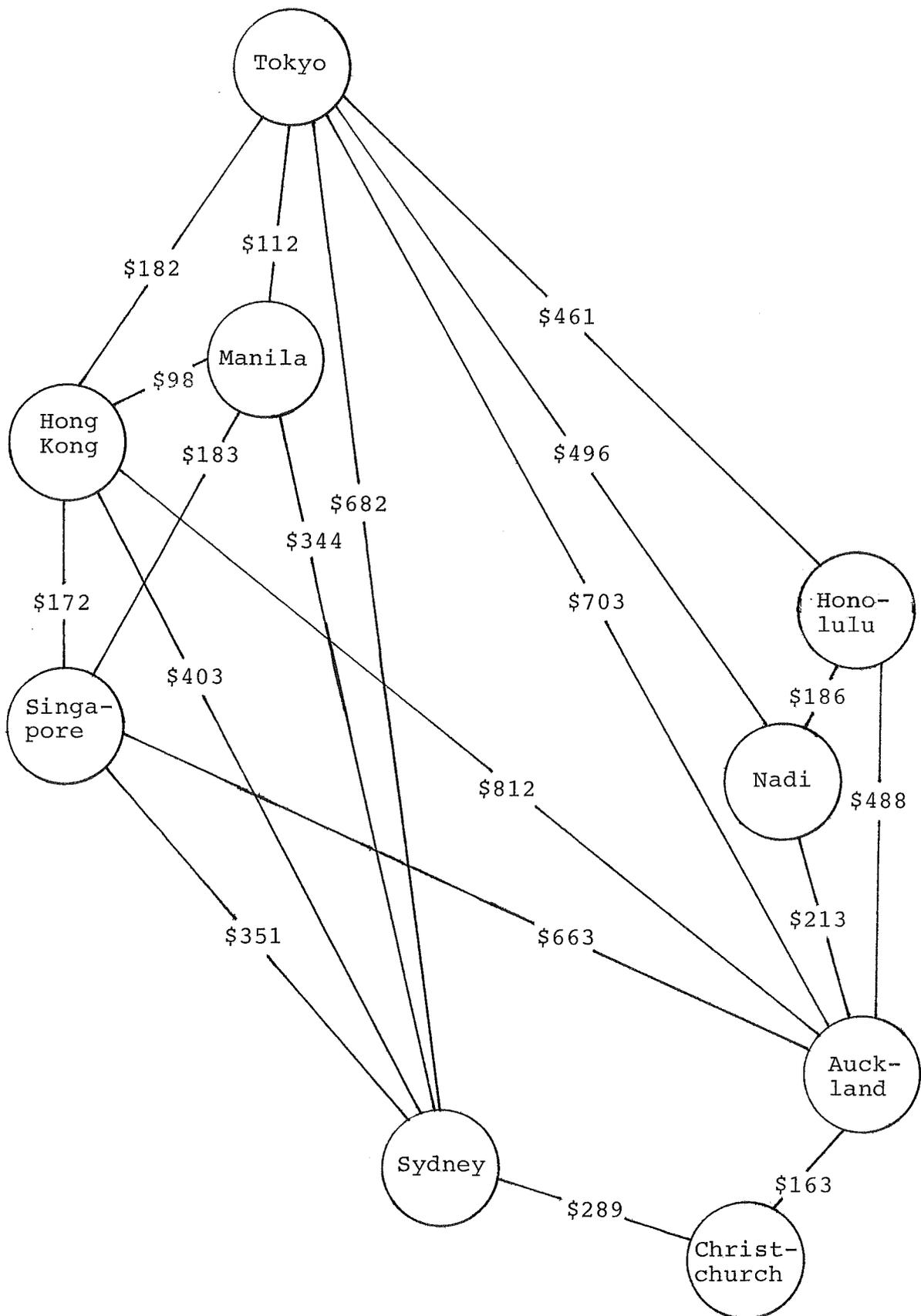


Figure 1.1 - A shortest path problem.

Of course, in the modern age such calculation is typically performed by digital computer rather than by hand. The high calculation speed and accuracy of even the smallest computer mean that within a matter of seconds of typing "CHC-TYO ?" the travelling student could be informed by a suitably programmed computer that the answer to that particular query was "CHC-SYD-MNL-TYO = \$745". Because of this speed, and with faster and faster computers being designed almost daily, it is tempting to think that the efficiency of the algorithm used to find the solution is of decreasing importance. However, the exact converse is in fact true. As the computer becomes more powerful it is more and more essential that an efficient algorithm is used, as otherwise the capacity of the machine is senselessly wasted. An inefficient algorithm will quickly absorb the computing power of even the fastest machine. As an illustration of the importance of the choice of algorithm, suppose that there are two candidate algorithms to solve such a "shortest path" problem on n cities, one requiring n^3 steps, and one requiring $n^2 \log n$ steps. Further, suppose that a current computer operates at one million steps per second, and that there is 1 second of cpu time allocated to the task:

<u>steps per second</u>	<u>max size for n^3</u>	<u>max size for $n^2 \log n$</u>
1 million	100	340

In this case, the decision should clearly be in favour of the $n^2 \log n$ algorithm. To make the example less obvious, suppose instead that the second algorithm requires $15n^2 \log n$ steps to solve a problem sized n :

<u>steps per second</u>	<u>max size for n^3</u>	<u>max size for $15n^2 \log n$</u>
1 million	100	100

Now there is no apparent reason to choose between them. But it seems rather likely that sooner or later the programmer will be given an instruction such as "Within the one second time limit problems of size 200 must now be processed; decide what new computer should be bought to achieve this." More calculations reveal the following, again for 1 second of cpu time:

<u>steps per second</u>	<u>max size for n^3</u>	<u>max size for $15n^2 \log n$</u>
5 million	170	200
8 million	200	250

The advantage of the $15n^2 \log n$ step algorithm has become quite clear as faster computers are employed and larger problems are tackled. Advances in computer software, especially in the area of algorithm efficiency, are no less important than advances in hardware, and the development of an asymptotically faster algorithm for some problem increases the effective speed of every computer on which that problem is currently being solved. Conversely, the use of a bad algorithm can make a problem intractable, no matter how powerful the computer.

This introductory section has been given with two purposes. Firstly to introduce the flavour of the shortest path problem that is addressed in this thesis, and secondly to reinforce the importance of such research into efficient algorithms. Although the area of "analysis of algorithms" falls into the domain of theoretical computer science, it is such study that

allows the practical solution of large scale problems by the powerful computers of today. This has been the aim of the research reported on in this thesis - the development of faster algorithms for finding shortest paths. One of the principal results reported herein is a shortest path algorithm which requires a number of steps proportional to $n^2 \log n$ rather than the n^3 steps of traditional algorithms.

1.2 Applications of Shortest Paths.

In this section a number of different problems are described, all of which can be solved with the use of a shortest path algorithm of some sort. The intention is to show by example the wide applicability of the shortest path problem.

The example problem given in section 1.1 requires a shortest path based on fare. It is worth noting that the "cost" can be many things - flight time, airport taxes, anxiety, and so on - any limited resource associated with the travelling steps or the points traversed. Another slight variation is the timetabled-travel problem, where for each city there is a list of connecting onward services. For example, again considering the case of "Tokyo or bust", Sydney might be represented by a list

<u>destination</u>	<u>departs</u>	<u>arrives</u>
HKG	0930	1845
SIN	1150	1815
NAN	1200	1515
SIN	1615	2255
TYO	2200	0830 + 2400

and so on, and the objective is to arrive in Tokyo as soon as possible after departure. Then the shortest route will depend not only on travel time, but also on the transit time required at the intermediate stops of each route. Such a problem will be well known to anyone who has attempted to plan long distance rail travel in Japan on the services of JNR; it can be solved as a straightforward application of a shortest path algorithm.

A second simple application appears in critical path analysis (Hu, 1982). Here a chart is made indicating which subtasks rely on which other subtasks in a large scale overall task, and for each subtask there is a time required, or cost. Then the time required by the overall task will be the total time needed on the longest path through the network. For example, if the task is building a house, a critical path analysis might reveal:

<u>task</u>	<u>time required</u>	<u>must follow</u>
a. design	4 weeks	
b. site preparation	5 weeks	a
c. foundations	2 weeks	b
d. frame construction	3 weeks	a
e. frame erection	2 weeks	c, d
f. roofing	2 weeks	e
g. interior finish	4 weeks	f
h. exterior finish	2 weeks	e

This analysis can also be translated into the following diagram:

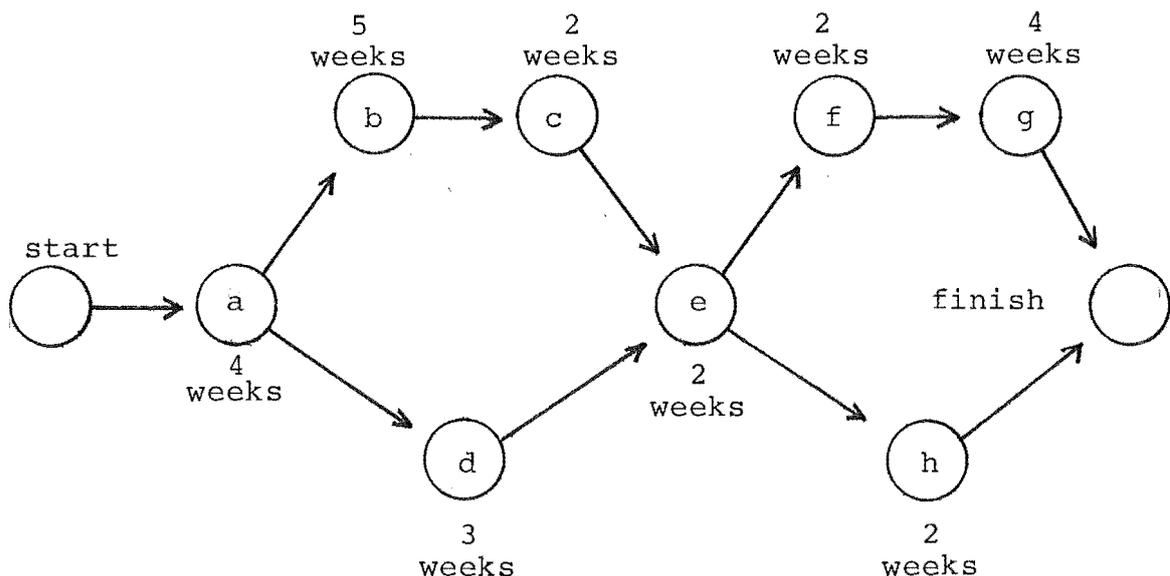


Figure 1.2 - A critical path problem.

In this simple example the critical path is a-b-c-e-f-g for a total cost of 19 weeks. The longest path in such a situation can be found with a modified shortest path algorithm.

Another problem readily solved by shortest path algorithms is the maximum reliability problem. Consider a telephone based computer network, where each host to host connection (i,j) is independently subject to loss of information at some random but known rate p_{ij} . To achieve reliable transmission of important information between pairs of nodes it is desired to find the route of maximum reliability. By taking the "cost" of each connection to be $-\log(p_{ij})$, the most reliable path can be easily found with the use of a shortest path algorithm.

Finally in this section, note that the problem of finding a maximum flow in a planar network can be reduced to a shortest path calculation, and this reduction results in an efficient and practical algorithm (Hassin, 1981; Reif, 1983).

1.3 Definitions and Notation.

The notation and terms used for various graph properties are well established, and a full description can be found in, for example, Bondy and Murty (1976), Even (1979), and Aho et al (1974). In this thesis the following descriptions of graphs, networks, and the shortest path problem will be used. For amplification of any of these definitions the reader is referred to the books mentioned.

A directed graph $G=(V,E)$ consists of a non-empty and finite set V of vertices and a finite multi-set $E = \{(u,v):u,v \text{ in } V\}$ of edges. The size of the graph will always be described by the two parameters n and m , where $n=|V|$ and $m=|E|$. The set V will often be numbered, $V=\{v_1,v_2,\dots,v_n\}$, but no ordering is implied by this; and for the sake of brevity the index will often be used to represent the vertex, so that $V=\{1,2,\dots,n\}$ and (i,j) represents the edge (v_i,v_j) . In all cases the meaning should be clear. Other terms for vertices and edges are sometimes used, notably the terms nodes for vertices and arcs for edges. For the purposes of shortest path calculation, the multi-set E can be reduced to a set of no more than $n(n-1)$ edges by the deletion of all self loops (v,v) and the replacement of multiple edges connecting the same two vertices by a single edge. Thus, without loss of generality, it will be assumed that all graphs considered have $m \leq n(n-1)$, and have no self loops and no parallel edges.

For each edge $e=(i,j)$, v_i is the source and v_j is the destination, denoted $srce(e)$ and $dest(e)$ respectively. A path P_{uv} from vertex u to vertex v is a finite sequence of k

edges $\{e_i\}$ such that $\text{srce}(e_1) = u$, $\text{dest}(e_k) = v$, and for $1 < i \leq k$, $\text{dest}(e_{i-1}) = \text{srce}(e_i)$. Then k is the length of the path. The empty path of length zero is also permitted, and links a vertex with itself. A path is simple if it contains no repeated vertices, so that any path P_{uv} that is simple has $\text{length}(P_{uv}) < n$, and the number of simple paths between any two vertices is finite. There may be an infinite number of non-simple paths between two vertices.

If there is a path from vertex u to vertex v , then v is reachable from u . A vertex is always reachable from itself. If every vertex in a graph is reachable from every other vertex then the graph is strongly connected. For a strongly connected graph it will be the case that $n \leq m$, since at least n edges must be present before every vertex can be reachable from every other vertex. The graph of figure 1.3 is strongly connected.

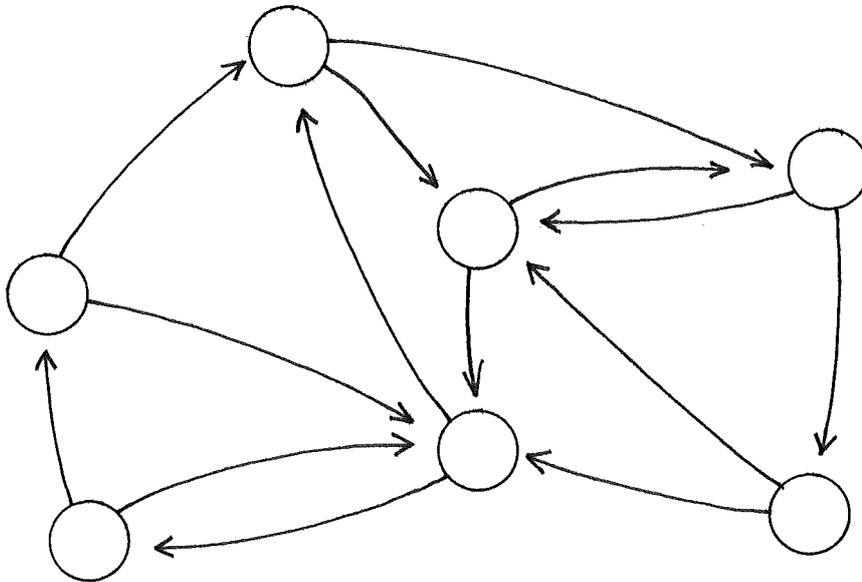


Figure 1.3 - A strongly connected graph.

For the purposes of shortest path calculations pairs (u,v) that are not present in E may be included in E as edges of arbitrarily large cost without affecting the computation, and thus it will be assumed without loss of generality that all graphs considered are strongly connected. That is, all graphs considered will have $n \leq m \leq n(n-1)$. These assumptions are in no way a restriction on the topology of the graph to be solved, as graphs that do not meet these requirements can be modified by a simple linear time preprocessing stage. The assumption that the graph is directed is also not a restriction, as undirected graphs (such as that of figure 1.1) can be considered to be directed graphs, where each undirected edge (u,v) corresponds to two directed edges (u,v) and (v,u) of identical cost.

A cycle in a directed graph is a non-empty path P_{uu} from a vertex u back to itself. An acyclic graph is a graph in which there are no cycles. The graph of figure 1.4 is acyclic.

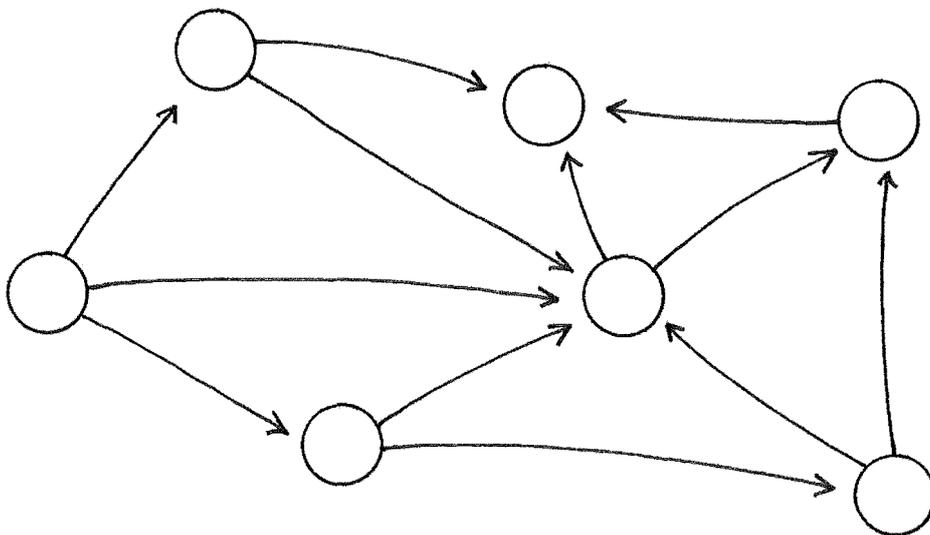


Figure 1.4 - An acyclic graph.

A shortest path from u to v is a path P_{uv} such that $C(P_{uv})$ is minimal over all possible paths from u to v . The number of edges in the path is immaterial; a path from u to v of minimal number of edges will be referred to as edge-shortest. Edge-shortest paths in a graph can be found efficiently by a breadth first search procedure. If any path from u to v contains as a subpath a cycle of negative cost, then the minimum is not defined and there is no shortest path from u to v . If no path from u to v contains a negative cost cycle and v is reachable from u then a shortest path from u to v exists. Further, if there is a shortest path from u to v then there is a shortest path that is simple, constructed by deleting all cycles from any non simple shortest path. This is valid since no cycle has negative cost. Thus, without loss of generality it may be assumed that a shortest path is simple. For a network $N=(G,C)$, define the shortest path cost function L_N to be the function $L_N:V \times V \rightarrow R$, where

$$\begin{aligned}
 L_N(u,v) &= -\infty && \text{if there is a path from } u \text{ to } v \\
 &&& \text{with a negative cycle,} \\
 L_N(u,v) &= +\infty && \text{if there is no path from } u \text{ to } v, \\
 L_N(u,v) &= C(P_{uv}) && \text{for some shortest path } P_{uv}
 \end{aligned}$$

The subscript N will normally be dropped without ambiguity. In this thesis only cost functions C that are non-negative will be considered. Then for these cost functions there can be no negative cycles, and the function L will also be entirely non-negative. This restriction is important; the algorithms discussed here are incorrect when applied to graphs with negative costs, irrespective of whether they have negative cycles. This is mentioned again in section 2.1. In

general, algorithms that solve the unrestricted shortest path problem in the presence of negative edges are less efficient than algorithms for the restricted problem where it is assumed that there are no negative arcs. Johnson (1973) considers the shortest path problem when the edge costs may be negative.

For any pair of vertices u and v let P_{uv} be a shortest path from u to v . Then any subpath of P_{uv} is also a shortest path, since if there was any shorter path connecting the two intermediate vertices, P_{uv} could be shortened, and would not be a shortest path. Hence, supposing $L(s,v)$ to be finite for all v in V , a solution to a single source shortest path problem from vertex s can always be a spanning tree rooted at vertex s . Such a spanning tree will be called a shortest path spanning tree. In figure 1.6 the edges drawn heavily form a shortest path spanning tree rooted at vertex s .

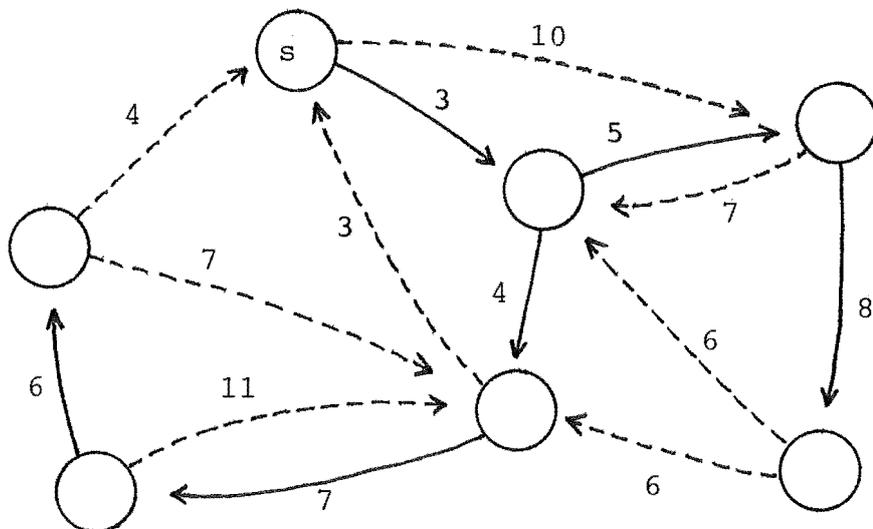


Figure 1.6 - A shortest path spanning tree.

The shortest path problem consists of a network N and a list of pairs of vertices between which the shortest paths are sought. Typically this list takes one of three forms:

In the single pair problem (spp) there is one element in the list and a single shortest path is required; in the single source problem (ssp) there are $n-1$ elements in the list and each element is of the form (s,v) where s is some fixed source vertex and v ranges over all other elements in V ; and in the all pairs problem (apsp) the list contains $n(n-1)$ elements, and shortest paths are sought between each pair of distinct vertices in the graph.

Primarily the results given here concern the all pairs problem, but many of the apsp algorithms discussed use a single source algorithm n times, once for each vertex in the graph.

The solution to a shortest path problem can also take one of three forms:

The numerical value of the cost of each shortest path may be required, that is, the appropriate values of the function L are wanted; or the cost of a shortest path together with an instance of a shortest path may be desired; or the cost of a shortest path together with a list of all possible paths that have that cost may be required.

The number of distinct shortest paths between two vertices may be exponential in n , requiring exponential time to list them, so the third solution type is outside the current discussion, which is concerned with fast polynomial time algorithms. For the purposes of this thesis, a solution to the shortest path problem will consist of one representative shortest path for each pair of vertices in the problem. Moreover, in the interests of clarity and brevity, the

programs presented here will typically only calculate shortest path costs and not a shortest path, but in all instances the algorithms are constructive and the programs are easily modified so that a shortest path as well as the shortest path cost can be recovered. The text associated with each algorithm will describe the modifications needed to recover the shortest paths; the programs themselves will compute the function L over the required range and will not include statements to record the information that would be needed to build a shortest path spanning tree.

Closely linked to the all pairs shortest path problem is the distance matrix multiplication (dmm) problem. Let $X = (x_{ij})$ and $Y = (y_{ij})$ be n by n matrices of real values. Then the distance matrix multiplication problem requires the calculation of the matrix $Z = (z_{ij}) = X * Y$, where $z_{ij} = \min\{x_{ik} + y_{kj} : 0 < k \leq n\}$, that is, multiplication in the distance matrix semi-ring (Aho et al, 1974). The link between the apsp and the dmm problems will be discussed in chapters two and six.

Of great interest is the running time of algorithms for finding the solution to a shortest path problem. In all cases analyses given here will be based on the random access machine model (Aho et al, 1974) in which all arithmetic, logical, and indexing operations take unit time. Such a model frees the analysis from consideration of the actual numeric values involved in any particular computation, and in general it will be possible for the running times to be described as functions of the graph parameters m and n . Rather than counting precise numbers of the many different

operations it is convenient to use the concept of asymptotic growth rate, and the standard notation established by Knuth (1976) is followed. Briefly,

$f(n,m) = O(g(n,m))$ when there is some constant k such that for all sufficiently large m and n , $f(n,m) \leq k \cdot g(n,m)$

$f(n,m) = \Omega(g(n,m))$ when there is some constant k such that for all sufficiently large m and n , $f(n,m) \geq k \cdot g(n,m)$

$f(n,m) = \Theta(g(n,m))$ when $f(n,m) = O(g(n,m))$ and $f(n,m) = \Omega(g(n,m))$

$f(n,m) = o(g(n,m))$ when $f(n,m) = O(g(n,m))$ and $f(n,m)$ is not $\Theta(g(n,m))$.

For example, the function $3n^3 + 5n^2 \log n$ can be described as $O(n^3)$, $\Theta(n^3)$, $\Omega(n^2 \log n)$, and $3n^3 + o(n^3)$.

Using this notation three more graph definitions are added: a family of graphs is sparse if $m = O(n)$ for members of the family; dense if $m = \Omega(n^{1+k})$ for any small positive k ; and complete if $m = n(n-1)$. In general these definitions will be misused, and a single graph (rather than a family of graphs) will be described as sparse if the number of edges is close to n , and dense if the number of edges is substantially greater than n . Planar graphs, with no more than $6n-12$ edges, are the usual example of a family of sparse graphs.

Chapter three will concentrate on the precise number of operations required by an algorithm solving the apsp problem. The operations counted as being important are comparisons between path and edges costs, and addition of path and edge

costs. These are normally the dominant components of the running time of any shortest path algorithm, and a precise bound on the number of these data operations is of interest from a theoretical point of view (Kerr, 1970). Indexing and similar operations are not counted at all. The reason for making this distinction is that the indexing operations will always be on the same data types, namely integers in the range 1 to n , but the addition and comparison of path costs may be expensive operations; they may, for example, involve multiple precision floating point numbers. Counting the number of additions and comparisons on path costs for a shortest path algorithm is similar to counting the number of comparisons required by a sorting algorithm, where again the cost of the indexing operations is ignored so long as they do not dominate the running time. These two operation types - addition and comparison of path costs - are called the active operations of the algorithm. Johnson (1973) has also used the same classification of operations for shortest path algorithms.

The analyses given here will sometimes be for the worst case, where the maximum running time that can be required by any graph of n vertices and m edges is calculated, and sometimes average case, where the time given is an expected running time. For an average case analysis some assumption must be made about the probability of each possible input configuration, and the running time given is an expected running time for the specified distribution of input probabilities. With worst case analysis there is the peace of mind that no input can require more running time than that given, but often this is wildly pessimistic. On the other

hand, the average case analysis predicts what is likely to happen on an input randomly selected from the corresponding probability distribution, but if the input network by bad luck happens to have some undesirable configuration for that algorithm then the running time might be greatly in excess of the expected value. Of the algorithms presented here, some are good in the worst case, and some are good in the average case; it will always be made clear which framework is being used at any time.

During the course of the research that resulted in this thesis many computational experiments on the different shortest path algorithms have been carried out. These empirical results have in many places been used to support analytically derived bounds on the running time of algorithms, and to compare two algorithms for operational use. To the interpretation of these results should be added a caveat - running times are volatile, and depend heavily on the architecture of the computer, the compiler and language used, the timing facilities provided by the operating system, and so on. Moreover, despite care to avoid such problems, there may also have been some unconscious bias in the implementations. Bearing in mind these two points, all running times given should be regarded as loose indications of relative performance and not as a precise measures of absolute performance. On the other hand, empirical measurements of the numbers of operations used by some algorithm - comparisons, heap operations and so on - are consistent from machine to machine and implementation to implementation. It is the time taken to execute each of these unit operations that varies from one machine to

another; the number of operations is an attribute of the algorithm and not the implementation.

The following mathematical conventions will be used. All unspecified logarithms will be to base 2; when natural logarithms are required the function $\ln()$ is used. The function $\log^2 n$ means $(\log(n))^2$; the function $\log \log n$ means $\log(\log(n))$; and the function $\log^* n$ is defined to be the minimum integer i such that $\log(\log(\dots \log(n)\dots)) < 1$, where the logarithm is taken i times. The taking of a square root will be abbreviated as $\text{sqrt}()$, and $\text{sqrt}(n)$ and $k \cdot \text{sqrt}(n)$ will be further abbreviated to sqrtn and $k\text{sqrtn}$ respectively. The function $\exp()$ indicates exponentiation base e (≈ 2.72), and the constant π represents π (≈ 3.14). The symbol $[]$ is used to mark the end of a proof. A glossary of all mathematical symbols and abbreviations appears after chapter seven.

The following lemma is sufficiently widely used in the analyses that follow that it is worth stating in the introductory section. Hereafter the result will be used without explicit reference.

Lemma 1.1 Suppose that in a sequence of independent trials the probability of success at each trial is at least p . Then the expected number of trials until the first success is not greater than $1/p$.

Proof. This follows from the standard result for the geometric distribution (Feller, 1968). $[]$

It also also worth reviewing the properties of the heap data structure (Floyd, 1962b; Williams, 1964; Floyd, 1964), as heaps are used as a data structure in many of the algorithms

discussed in this thesis, and their properties will be assumed rather than explicitly stated in each case.

A heap is an implementation the priority queue abstract data type (Sedgewick, 1983) with the properties that a heap of n items can be build in $O(n)$ time; the minimum weight element in the heap can be identified in $O(1)$ time; any element can be deleted in $O(\log n)$ time; and elements can have their weight updated in $O(\log n)$ time. All of these bounds are for the worst case.

The normal implementation of an n element heap is as an implicit binary tree stored in an n element array. In such an array the "father" of the element in position i is found in position $(i \text{ div } 2)$, and the two "sons" are in positions $2*i$ and $2*i+1$. The heap property requires that an element not exceed in weight either of its two sons, meaning that the smallest weight element can be found at the root, stored in position 1. For example, the 10 element array

position:	1	2	3	4	5	6	7	8	9	10
weight:	8	9	11	10	9	15	17	11	16	10

satisfies the heap property, and represents the binary tree shown in figure 1.7. General purpose routines for manipulating the heap elements to allow for updates of weight, insertion and deletion of elements, and heap creation, can be found in any textbook on algorithms, for example Sedgewick (1983).

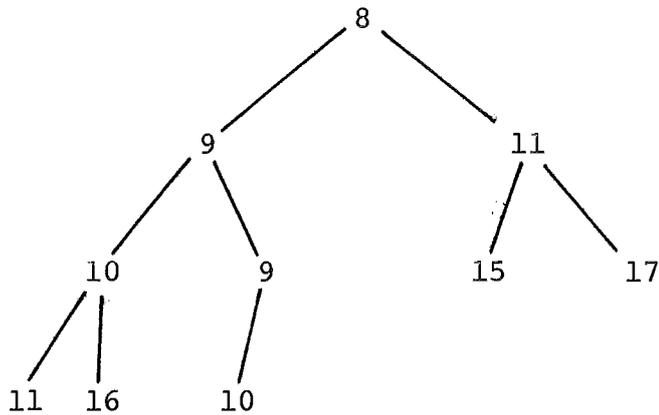


Figure 1.7 - A heap.

The Concise Oxford Dictionary (sixth edition, 1976) also provides a useful insight to the properties of a heap:

heap n. Group of things lying one on another; (in sing. or pl., colloq.) large number or quantity (a heap of people; there is heaps of time; have seen it heaps of times; he is heaps better); (colloq.) battered old motor vehicle; knock or strike all of a heap; top, or bottom, of the ~, (colloq., fig.) winner, loser.

1.4 Overview of the Thesis.

The remainder of this thesis is organised as follows. Chapter two describes existing algorithms for finding shortest paths and also discusses lower bounds on the complexity of the problem, setting the scene for the new results.

In chapter three the precise number of active operations required for a solution to the apsp problem is considered. A new priority queue data structure is developed, and leads to an apsp algorithm with an improved worst case bound on the number of active operations.

In chapter four the average running time for the all pairs problem is attacked, and a new algorithm that requires $O(n^2 \log n)$ expected running time on a wide class of random graphs is given, improving asymptotically the previous best result for this class of graph from $O(n^2 \log n \log^* n)$. The chapter includes experimental results that show that the new algorithm is fast, and suitable for operational use.

Chapter five discusses the implementation of the algorithm of chapter four, and it is shown that two apparently good implementation heuristics can have disastrous results, and should be carefully avoided. A third modification to the algorithm that improves the worst case running time from $O(n^3 \log n)$ to $O(n^3)$ is also described. The final section of the chapter gives a general construction for a "bad" graph that forces the worst case running time, and experimental results for worst case running times of shortest path algorithms.

Chapter six is concerned with distance matrix multiplication, and a hybrid algorithm for this problem that on some families of random matrices runs significantly faster than any other known algorithm is presented. Also described is a probabilistic algorithm for distance matrix multiplication that has worst case running time $O(n^3)$, and for some class of random matrices, good probability of calculating the optimal distance matrix product.

Chapter seven summarises and reiterates the main results. Chapter seven is followed by a glossary, acknowledgements, and a list of references.

CHAPTER TWO
EXTANT RESULTS.

2.1 Upper Bounds and Algorithms.

Worst case analysis.

The traditional algorithms for finding shortest paths have all been good in a worst case sense, and this area is examined first. The earliest approaches to the apsp problem were based on distance matrices. If the cost function C and the shortest path function L are considered as distance matrices, then L is the closure of C in the plus-min semiring (Aho et al, 1974), and so has the property that $C * L = L$, where $*$ represents distance matrix multiplication. Provided that the network contains no negative cycles, this closure can be found by repeated squaring of $(C+I)$, where I is a distance identity matrix. This is because no shortest path need be more than n edges long, so that $L = (C+I)^n$. To raise a matrix to the n 'th power will require $\text{ceiling}(\log n)$ repeated squarings; and by a straightforward method each squaring will require $O(n^3)$ time. This then gives an $O(n^3 \log n)$ time algorithm for the apsp problem, one of the earliest results.

Rearrangement of the calculation order for the inner products was the key to the $O(n^3)$ apsp algorithm given by Farbey et al (1967); by always using the most recently calculated value for each matrix entry they showed that only two squarings were necessary. Floyd (1962a) had already given an $O(n^3)$ apsp algorithm based on the boolean matrix closure method of Warshall (1962); his algorithm is in effect a single squaring

with the order of the "multiplications" within each inner product rearranged and again the most recent values always used.

Furman, Munroe et al have given the theorem, presented fully in Aho et al (1974), that distance matrix closure is computable in $O(T(n))$ time if and only if distance matrix multiplication is also computable in $O(T(n))$ time, provided that $T(n) = \Omega(n^2)$ and $T(n) = O(n^3)$. In removing the $O(\log n)$ overhead required by repeated squaring this result has prompted many authors to attack the distance matrix multiplication problem.

First to succeed with an $o(n^3)$ distance matrix multiplication algorithm was Fredman (1975, 1976). He showed that $O(n^{2.5})$ additions and comparisons on path costs were sufficient for the multiplication, but was unable to give a general algorithm that required this running time. Fredman then went on to show that his technique could be exploited "mildly", and was able to give an $O(n^3 (\log \log n / \log n)^{1/3})$ algorithm, which is $o(n^3)$. However this algorithm is rather complex, and no implementation of it has been reported. It seems likely that extremely large problem sizes would be required before the method could become operationally competitive. For example, the function $2n^3 (\log \log n / \log n)^{1/3}$ first becomes less than n^3 when $n = 2^{44}$, and it seems not unreasonable to expect a constant factor of at least 2 for Fredman's method when compared with Floyd's method. Even at one trillion steps per second this sized problem would still require millions of years of computing time.

Yuval (1976) gave a transformation involving exponentiation and logarithms that allows distance matrix multiplication to be encoded into matrix multiplication over the real field. This then leads to an $O(n^b)$ algorithm for the apsp problem, where b is the complexity of matrix multiplication, currently approximately 2.5 (Schönhage, 1981). However critics have pointed out that to effect this scheme, even when the path costs are integers, very high precision real arithmetic is required, and that the true complexity is exponential (Moran, 1981).

Because of this criticism, Fredman's $O(n^3)$ result is generally acknowledged as being the current worst case upper bound for distance matrix multiplication, and hence for the apsp problem on a complete graph.

The result of Munroe and Furman applies similarly to the problem of finding transitive closure in the Boolean semi-ring. In that semi-ring, boolean matrix multiplication can be carried out in $O(n^{2.81})$ time using the technique of Strassen (1969). However this approach cannot be extended to the distance matrix semi-ring, as it requires an inverse for the "min" operation. In the distance semi-ring, for arbitrary elements a there is no x such that $\min(x, a) = \text{infinity}$.

Direct "graph" approaches to shortest path problems have also yielded good algorithms. Dijkstra (1959) gave an $O(n^2)$ algorithm for the single source problem on a non-negatively weighted graph; this algorithm can be used n times to solve the apsp problem in $O(n^3)$ time. Since then there have been many suggested implementations of his idea; the most notable being the generalisation of Johnson (1977) (see also Tarjan

(1983) chapter 7) to give a running time of $O(m \log_k n)$, where $k = \max(m/n, 2)$. For dense graphs this running time is $O(m)$, which for the single source problem is optimal; for sparse graphs the behaviour is $O(n \log n)$. Recently Fredman and Tarjan (1984) gave another implementation of Dijkstra's algorithm, using a Fibonacci heap, which runs in $O(n \log n + m)$ time, a slight improvement over Johnson for graphs that are neither sparse nor dense. However all of these algorithms will work only if there are no negative arcs. If negative arcs are present then an $O(nm)$ preprocessing step is needed before these algorithms can be used (Johnson, 1973). This makes them expensive for the single source problem, but the step is only required once, even if n single source solutions are to be combined to make an apsp solution. Using the approach of Fredman and Tarjan, Dijkstra's algorithm can be used to solve the unrestricted apsp problem in $O(n^2 \log n + mn)$, which is $O(n^3)$ on a complete graph.

These worst case bounds for the apsp problem are given in the table below.

<u>algorithm</u>	<u>year</u>	<u>bound</u>
Dijkstra	1959	$O(n^3)$
Floyd	1962	$O(n^3)$
Johnson	1973	$O(nm \log_k n)$, $k = \max(m/n, 2)$
Fredman	1976	$O(n^3 (\log \log n / \log n)^{1/3})$
Yuval	1976	$O(n^b)$, $b \cong 2.5$
Fredman, Tarjan	1984	$O(n^2 \log n + nm)$

Table 2.1 - Worst case bounds for apsp algorithms.

On complete graphs only the "impractical" algorithms of Fredman and Yuval have running times $o(n^3)$.

There have been many other shortest path algorithms given in the literature that have not been listed here. Among them are algorithms by Dial (1969), who used a bucket sort technique to make an efficient implementation of Dijkstra's algorithm when the edge costs are all integers from some small range; Dantzig (1960), whose algorithm is similar to that of Dijkstra and will be discussed in detail in section 4.1; Pape (1980) who uses an interesting heuristic to achieve a fast algorithm; and so on. Dreyfus (1969) gives a survey of early work and Mahr (1981) a more recent summary; the references section of this thesis lists a number of other papers.

Average case analysis.

Since 1972 there has been a great interest in apsp algorithms that have a good average case running time. Spira (1973) pioneered this area with an algorithm he derived from Dijkstra's and Dantzig's $O(n^2)$ time single source algorithms. By using a heap data structure, and a pre-sort to order the costs on the edges from each vertex, he was able to give an algorithm that requires $O(n^2 \log^2 n)$ time on average when the edges costs are independently drawn from any fixed but arbitrary random distribution. His method involves the use of an $O(n \log^2 n)$ single source algorithm for each of n sources, but is not suitable for a single source problem because the $O(n^2 \log n)$ cost of the presort must be shared over all sources to make the technique efficient.

Corrections were made to his presentation by Carson and Law (1977), and Bloniarz, Meyer and Fisher (1979) formalised his idea of a random graph and also corrected his algorithm.

Subsequently his algorithm has been improved a number of times, with all of the improvements still using the same paradigm (Takaoka and Moffat, 1980; Bloniarz, 1980; Frieze and Grimmett, 1983). The table below lists the average running times for the members of this sequence of algorithms; the algorithms themselves will be examined in greater detail in chapter 4. One of the main results presented in this thesis is the last algorithm listed in the table, which has average running time of $O(n^2 \log n)$.

<u>algorithm</u>	<u>year</u>	<u>average case</u>	<u>worst case</u>
Dijkstra	1959	$O(n^3)$	$O(n^3)$
Dantzig	1960	$O(n^3)$	$O(n^3)$
Spira	1973	$O(n^2 \log^2 n)$	$O(n^3 \log n)$
Takaoka, Moffat	1980	$O(n^2 \log n \log \log n)$	$O(n^3 \log n)$
Bloniarz	1980	$O(n^2 \log n \log^* n)$	$O(n^3 \log n)$
new algorithm	1985	$O(n^2 \log n)$	$O(n^3)$

Table 2.2 - Average running time for apsp algorithms.

Frieze and Grimmett (1983, 1985) have also given an $O(n^2 \log n)$ average time algorithm for the apsp problem, but their method is suitable only for a narrow class of probability distributions, and is not as general as the listed algorithms.

All of these methods rely on n successive iterations of a single source algorithm, and require that the edge costs be non-negative. The preprocessing step mentioned earlier as a way of handling negative cost edges is not applicable, as on a complete graph the preprocessing time would be $\Theta(n^3)$, dominating the running time.

2.2 Lower Bounds.

An upper bound on the complexity of a problem is usually shown by giving an algorithm that solves the problem and runs in some time $T(n)$; this is sufficient to establish $T(n)$ as an upper bound for the problem. On the other hand, lower bounds are much harder to develop. Apart from the so called "trivial" bounds, where the number of inputs that must be examined by any algorithm and the number of outputs that must be produced are counted, there is no general technique for establishing lower bounds, and in general non-trivial lower bounds are scarce.

A lower bound $B(n)$ on the complexity of a problem means that every algorithm that solves the problem must require at least $B(n)$ steps for some input sized n . Because of the universal quantification over algorithms, both those invented and those not yet invented, lower bounds are normally established within a precise framework - the computational model - that states which operations may and may not be performed. For any problem, different computational models may be suitable, and the lower bound on the complexity of the problem will depend upon which model is chosen. In general, the more liberal the computational model, the less restrictive the lower bound. For example, using only $+$ and $*$, real matrix multiplication requires $O(n^3)$ operations (Kerr, 1970); but Strassen (1969) showed that if subtraction can be used, then the complexity of the problem is $o(n^3)$. A similar hierarchy exists for the shortest path problem, where a number of computational models have been proposed, and non-trivial lower bounds established in some areas. The remainder of

this section briefly surveys the different computational models in which lower bounds have been given for the apsp problem.

The most restrictive model is that in which only operations + (plus) and min are permitted, and the operations must be performed in straight-line order, that is, by a program that executes assignment statements only and has no branching based on the relative values of the input variables (Kerr, 1970). Johnson (1973) has shown that in this framework $2n(n-1)(n-2)$ active operations are required, making Floyd's algorithm optimal.

If the straight line requirement is removed to allow branching, but the operations still restricted to plus and min, then Fredman's demonstration that $O(n^{2.5})$ operations are sufficient (Fredman, 1976) is valid, meaning that the lower bound cannot exceed this. Yao et al (1977) attempted to show that the lower bound in this decision tree framework was $\Theta(n^2 \log n)$ using information theoretic techniques, but their attempt was dismissed by Graham et al (1980), who showed that the only lower bound that could be established by that approach was cn^2 . Despite Fredman's $O(n^{2.5})$ sufficiency demonstration, the only efficient algorithms known in this arena have worst case bounds of $\Theta(n^3)$. There is a wide gap between upper and lower bounds; it is in this arena that fast average time algorithms have been given, but all of these fast algorithms have worst case bounds of $\Omega(n^3)$, and do not provide information as to the worst case complexity of the apsp problem.

If the costs may be treated as quantities over the real field, with subtraction, multiplication, and division permitted in a decision tree model, then only trivial lower bounds are known; it is in this area that Fredman's $O(n^3)$ algorithm lies. Again there is a wide gap between known upper and lower bounds.

If arbitrary precision real multiplication can be permitted as a unit operation, along with exponentiation and logarithm taking, then the results of Yuval (1976) and others (Romani, 1980; Moran, 1981) can be considered to be upper bounds in this still more liberal arena. Being based on matrix multiplication, their methods are straight line.

These results are summarised in the following table:

<u>arena</u>	<u>lower bound</u>	<u>upper bound</u>
straight line, plus min only	$2n(n-1)(n-2)$ Johnson	$2n(n-1)(n-2)$ Floyd
decision tree, plus min only	cn^2 Graham	$O(n^3)$ Floyd, Dijkstra, et al
decision tree, real arithmetic	$\Omega(n^2)$ trivial	$O(n^3 (\log \log n / \log n)^{1/3})$ Fredman
arbitrary precision real arithmetic	$\Omega(n^2)$ trivial	$O(n^b)$, $b \approx 2.5$ Yuval, et al

Table 2.3 - Lower bounds for the apsp problem.

Clearly, the large differences between best known upper and lower bounds in almost all computational models leaves much

room for the development of either faster algorithms or sharper bounds or both.

For the single source problem, the situation is reversed. Here it has been shown by Spira and Pan (1975) that on a complete graph $(n-1)(n-2)$ active operations are required, even in a liberal decision tree computational model with plus, min, and subtraction permitted. Here there is little scope for improvement over the algorithm of Dijkstra, and it seems likely that $O(n \log n + m)$ running time cannot be improved for conventional uni-processing computer hardware and arbitrary graphs.

If parallel architectures or distributed processing computational models are employed then faster algorithms are possible (Hirschberg, 1976) but this area is outside the scope of the current discussion. Here all algorithms are analysed in terms of the sequential processing random access machine model given by Aho et al (1974).

CHAPTER THREE

A GOOD WORST CASE ALGORITHM.

3.1 Background.

This chapter concentrates on the worst case behaviour of apsp algorithms; in particular, on the precise number of active operations that are required when calculating a solution to an apsp problem.

Fredman (1976) showed that $O(n^{2.5})$ active operations were sufficient for a solution to the apsp problem, but did not give an algorithm realising this bound in terms of running time as well as operations. If his idea is implemented, an algorithm that requires $O(n^{2.5})$ active operations and $\Theta(n^{3.5})$ running time will result; to date there has been no successful attempt to reduce the running time of this technique to the $o(n^3)$ level while still retaining the $O(n^{2.5})$ bound on active operations. The first result presented in this chapter is a new "greedy" algorithm that when coupled with an appropriate priority queue requires $n^3 + o(n^3)$ additions and comparisons on edge costs, and $O(n^3)$ running time in the worst case. Thus, although Fredman's result means that n^3 cannot be claimed to be a best upper bound on the number of active operations, an $O(n^3)$ worst case time algorithm is given that requires only $n^3 + o(n^3)$ operations, which Fredman did not do.

Fredman's main result of his 1976 paper was to show the existence of an $O(n^3(\log\log n/\log n)^{1/3})$ worst case algorithm for distance matrix multiplication, and hence for the apsp problem. In that algorithm he relied on being able to treat

path costs as real numbers and work within the real field rather than the plus-min semiring. So although he attained $O(n^3)$ running time, he did so outside the plus-min semiring structure that is considered here. Within the semi-ring the only operations permitted on path and edge costs are addition and binary min operations, and it is these that are counted as being the active operations of an algorithm.

In an early paper Dantzig (1960) gave an algorithm for the single source problem (see section 4.1) which requires $O(n^2 + t(n))$ time, where $t(n)$ is the time needed to sort the edge lists of the graph. For a dense graph, $t(n) = O(n^2 \log n)$ and the algorithm will require $O(n^2 \log n)$ time for the single source problem, an inferior result to the ssp algorithm of Dijkstra. However, application of Dantzig's technique to the apsp problem results in an $O(n^3)$ algorithm, as the cost of sorting the edge lists is shared over all sources. Moreover, not noted by Dantzig is that a careful implementation of his technique results in an algorithm that solves the apsp problem using $n^3 + O(n^2 \log n)$ active operations. This result has been overlooked by many authors - for example Yen (1972) (see also Williams and White, 1973) reports an implementation of Dijkstra's algorithm that requires $1.5n^3$ active operations.

Because of this result by Dantzig, the greedy algorithm presented in the first part of this chapter is of interest mainly as an exercise in algorithm design. An old technique is applied to an old problem, giving a good, but not new, result. In the process of implementing this greedy algorithm a priority queue structure is developed; it is the priority

queue itself that is the main result of this chapter. This new priority queue can also be applied to Dijkstra's algorithm, giving again an algorithm that requires $n^3 + o(n^3)$ active operations on a complete graph. However this implementation also gives an $O(mn)$ bound on operations and running time on graphs that are dense but not complete, thus improving upon the result of Dantzig.

Tomizawa (1976) also did some work in this area; he gave an implementation of Dijkstra's algorithm that requires $3mn + n^2 \sqrt{2n \log(2n)}$ active operations in the worst case to solve the apsp problem. The final implementation of Dijkstra's algorithm given here is superior to Tomizawa's bound for both complete graphs and graphs that are dense but not complete.

For practical implementations none of these methods with good worst case behaviour can compete with the average case methods of Spira (1973) et al, and the $O(n^3)$ algorithms are of interest from a theoretical rather than an operational point of view. Moffat (1983) gives a computational survey of apsp algorithms that demonstrates that the fast average case methods are practically faster than any of the $O(n^3)$ methods, and, presumably, that of Fredman, although no implementation of his algorithm has been reported. The experimental results given in section 4.5 of this thesis also show this practical superiority.

Note that throughout this chapter the notation (u,v) will be used interchangeably to denote the edge (u,v) , a path P_{uv} , and an ordered pair of vertices (u,v) in $V \times V$. No ambiguity will arise from this.

3.2 The Greedy Paradigm.

The new method is based on the greedy design paradigm, and was initially given in an undeveloped form in (Moffat, 1979). There the best time bound that was obtained was $O(n^3 \log n)$. In this presentation Kruskal's (1956) algorithm for finding a minimum cost spanning tree is briefly stated as an example of the paradigm, and is then extended to the all pairs shortest path problem.

Kruskal's algorithm.

Kruskal's algorithm for finding a minimum cost spanning tree T in an undirected network is described in outline below. The method is described in detail by Tarjan (1983).

```

procedure kruskal ;
begin
  mark all edges in E unchecked ;
  T := {} ;
  while there are edges in E that remain unchecked do
  begin
    let (u,v) be the least cost unchecked edge ;
    if (u,v) can be used in the spanning tree then
      T := T + {(u,v)} ;
    mark (u,v) checked ;
  end
end {kruskal} ;

```

Algorithm 3.1 - Kruskal's Algorithm.

The algorithm is greedy in that, to build a global minimum of cost, each step involves finding a local minimum - the unchecked edge of least cost. It is this technique of repeatedly finding and using a least cost component that characterises the greedy approach to a problem.

Application to shortest paths.

An algorithm for the all pairs shortest path problem can be constructed from a similar skeleton. For any pair of vertices u and v the cost of the shortest path from u to v must be either the cost of the direct edge (u,v) or the cost of an indirect path through some intermediate vertex p . In this latter case, because all edge costs are non-negative, neither $L(u,p)$ nor $L(p,v)$ can exceed $L(u,v)$, where L is the shortest path cost function defined in section 1.3. Shortest paths can thus be created by checking paths in ascending order of cost, and, as each path (u,v) is checked, searching for longer paths that might in turn have their cost reduced if (u,v) is used as part of an indirect path. All edge costs are non-negative, so that once checked a path or edge cannot have its cost reduced, and re-scanning is not necessary. Thus it suffices for the algorithm to make a single pass over the pairs of vertices of the graph, checking each edge once. In the following program, the array "pcost" records tentative shortest path costs:

```

procedure greedy-apsp ;
begin
  {initialisation}
  for all (u,v) in V X V do
    mark (u,v) unchecked and set pcost[u,v] := C(u,v) ;
  for all u in V do
    mark (u,u) checked and set pcost[u,u] := 0 ;
  {main processing loop}
  while there are unchecked (u,v) in V X V do
    begin
      getmin: let (u,v) be such that pcost[u,v] is
              minimal over all unchecked pairs (u,v) ;
      check:  for all a,b in V X V do
              attempt to reduce pcost[a,b] using
              (u,v) as one part of an indirection ;
      mark (u,v) checked ;
    end ;
  end {greedy-apsp} ;

```

Algorithm 3.2 - Greedy-apsp.

The invariant relating the array "pcost" and the shortest path function L is as follows. If a pair (u,v) has been checked, then $\text{pcost}[u,v] = L(u,v)$, and the correct shortest path cost is recorded. If, on the other hand, (u,v) has not been checked, then $\text{pcost}[u,v]$ is either the cost of the best tentative path from u to v that consists of two checked paths, or the original cost of the edge (u,v) , whichever is smaller. In all cases $\text{pcost}[u,v]$ will be no greater than $C(u,v)$, the original cost of the edge.

The step "check:", searching for paths that might be shortened by the use of (u,v) , can be more efficient than a simple search through all of the as many as n^2 unchecked paths:

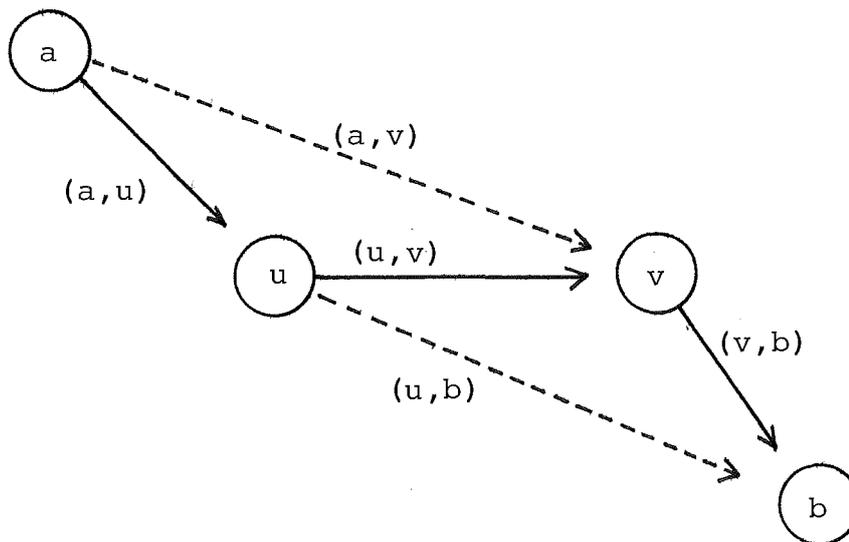


Figure 3.1 - Checking (u,v) .

Only paths that share endpoints with (u,v) can be updated by (u,v) . Thus, when (u,v) is the current pair being checked, it is only necessary to test pairs (a,v) or (u,b) , where a and b range over the vertices in $V - \{u,v\}$. Moreover,

(a,v) need only be tested if (a,v) has itself not already been checked and (a,u) has been checked. If the first of these two additional conditions is not met then $\text{pcost}[a,v]$ has already been assigned the optimal value $L(a,v)$, and the test is pointless. If the second condition is not met then the test can be deferred until $\text{pcost}[a,u]$ has been given a final value at the time when (a,u) is checked; this delay has the advantage that the test might be avoided entirely if $L(a,v) < L(a,u)$ and (a,v) is checked first. A similar pair of conditions apply to pairs (u,b) that might have their pcost updated. The following procedure describes this checking strategy, and a call to this procedure replaces the loop marked "check:" of procedure `greedy-apsp`.

```

procedure check-pair ( u,v ) ;
begin
  for a in V do
    if (a,u) is checked and (a,v) is not checked then
      begin
        newcost := pcost[a,u] + pcost[u,v] ;
        if newcost < pcost[a,v] then
          update pcost[a,v] to newcost
        end ;
      end
    for b in V do
      if (v,b) is checked and (u,b) is not checked then
        begin
          newcost := pcost[u,v] + pcost[v,b] ;
          if newcost < pcost[u,b] then
            update pcost[u,b] to newcost
          end
        end
      end
    end {check-pair} ;
end

```

Algorithm 3.3 - Checking (u,v) .

An implementation of Kruskal's algorithm requires some data structure for ordering the edges by cost, and the same is true of `greedy-apsp`. However, with `greedy-apsp` the data structure used must be capable of handling updates to the weights of the elements stored, an operation not required by the minimum spanning tree algorithm. The simple priority

queue structures that are typically used for an implementation of Kruskal's algorithm, such as a sorted list or a binary heap, are not suitable. In the initial description of the greedy-apsp method (Moffat, 1979), a binary heap was employed, leading to a bad $O(n^3 \log n)$ worst case running bound. Here the implementation uses a different priority queue - section 3.3 describes a data structure that enables successive minima to be found in $O(\text{sqrtn} + \log n)$ time each, even allowing for the updates on path costs that will be required. Given such a data structure, each execution of the main while loop of greedy-apsp can be seen to take $O(n + \text{sqrtn})$ time, and the whole algorithm will require $O(n^3)$ time in the worst case.

If the shortest paths are required, a simple modification can be made to the algorithm so that each time an entry of array "pcost" is updated the intermediate vertex that successfully reduced the path cost is also recorded. At the conclusion of the algorithm the shortest paths would then be recovered as well as the shortest path costs.

Correctness of algorithm greedy-apsp.

The following lemmas identify the ideas required to show the correctness of the method.

Lemma 3.1 For all pairs (u,v) in $V \times V$, no changes to the value of $\text{pcost}[u,v]$ will take place after (u,v) is checked.

Proof. From the guards of the program text. []

Lemma 3.2 For all pairs (u,v) in $V \times V$, $\text{pcost}[u,v]$, storing the tentative shortest path cost, is non-increasing.

Proof. From the guards of the program text. []

Lemma 3.3 The costs of the pairs checked at each iteration of the main while loop form a non decreasing sequence.

Proof. Suppose (u,v) is checked at some iteration. Then the new values assigned by any updates that take place during the call "check-pair(u,v)" will be $\text{pcost}[u,v]$ plus some non-negative quantity. On the other hand, all paths that are not updated have path values not less than $\text{pcost}[u,v]$ anyway. In either case all remaining unchecked pairs after (u,v) is checked have pcost values not less than $\text{pcost}[u,v]$. []

Corollary 3.4 Suppose that (x,y) is such that, at the conclusion of greedy-apsp, $\text{pcost}[x,y] < \text{pcost}[u,v]$. Then at the time when (u,v) is checked, pair (x,y) will have already been checked. []

Theorem 3.5 The program greedy-apsp correctly computes shortest path costs. That is, at the conclusion of greedy-apsp, $\text{pcost}[u,v] = L(u,v)$ for all (u,v) in $V \times V$.

Proof. From the constructive nature of the algorithm, a path exists from u to v of cost $\text{pcost}[u,v]$, so

$L(u,v) \leq \text{pcost}[u,v]$ for all u and v . However, suppose for some pair (u,v) that $L(u,v) < \text{pcost}[u,v]$, meaning that there is some path from u to v of cost strictly less than $\text{pcost}[u,v]$. To be specific, let (u,v) be the first path checked for which, at the conclusion of the algorithm, $L(u,v) < \text{pcost}[u,v]$. Then since $\text{pcost}[u,v]$ is initialised to $C(u,v)$ and is non-increasing, the path represented by $L(u,v)$ must contain at least one intermediate vertex. Let this intermediate node be vertex p . All edge costs are non-negative, so both $L(u,p)$ and $L(p,v)$ must also be strictly

less than $\text{pcost}[u,v]$. From this it follows that $L(u,p) = \text{pcost}[u,p]$; that $L(p,v) = \text{pcost}[p,v]$; and thus that (corollary 3.4) both were checked before (u,v) . Assume, without loss of generality, that (p,v) was checked after (u,p) , at iteration t of the main while loop. For $\text{pcost}[u,v]$ not to have been set to $\text{pcost}[u,p] + \text{pcost}[p,v]$ at iteration t would require that either (u,v) had already been checked, in which case $\text{pcost}[u,v] \leq \text{pcost}[p,v]$ (lemma 3.2), or that $\text{pcost}[u,v]$ was already less than $\text{pcost}[u,p] + \text{pcost}[p,v]$. In both cases, $\text{pcost}[u,v]$ cannot subsequently have increased (lemma 3.2) nor can $\text{pcost}[u,p]$ or $\text{pcost}[p,v]$ have decreased (lemma 3.1), and this gives the desired contradiction. Thus it cannot be that $L(u,v) < \text{pcost}[u,v]$. []

Analysis of active operations.

Given that the priority queue data structure can be implemented within the claimed bounds of $O(1)$ time per update and $O(\text{sqrtn})$ time and data operations per "getmin" operation, the running time of the whole algorithm is easily seen to be $O(n^3)$, meaning that the number of active operations must also be $O(n^3)$. To precisely count the number of operations required by procedure check-pair, consider the subgraph shown in figure 3.2. A single source vertex is considered, and any two other vertices in the graph:

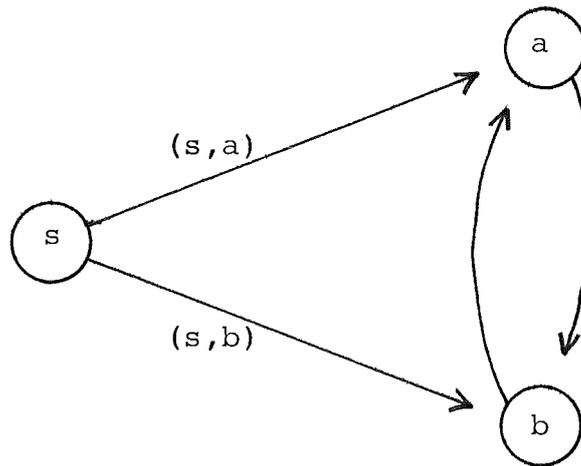


Figure 3.2 - A triple of vertices.

Lemma 3.6 In any triangle of vertices such as is depicted in figure 3.2, of the two paths (s,a) and (s,b) out of vertex s , the checking of the second will never cause a test on that checked first.

Proof. The guards in the text of procedure check-pair ensure that no pair will ever be tested after it is checked. []

Note that even in the case of paths of equal cost, one must be checked before the other.

Lemma 3.7 Over all calls to procedure check-pair there will be at most $n(n-1)(n-2)$ active operations on path and edge costs.

Proof. There are n vertices, each of which is the source in $(1/2)(n-1)(n-2)$ triangles of the form shown in figure 3.2. Lemma 3.6 bounds the number of tests per triangle at 1, and each test requires one comparison and one addition. []

Section 3.4 makes two further observations concerning the bound of lemma 3.7.

3.3 Priority Queue Implementation.

The algorithm described in section 3.2 requires a priority queue. By recognising and exploiting the relationships among the edges in the queue that get updated, it can be implemented efficiently using a two level approach.

The upper queue.

The top level of the data structure is a binary tournament tree (Knuth, 1973b) of n entries, one for each vertex in the graph. Each of the entries represents the best candidate path, in terms of cost, of the unchecked paths that are incident at that vertex. That is, each entry represents the least cost unchecked path either incoming or outgoing at that vertex, of which there may be as many as $2(n-1)$. As a consequence, each unchecked pair (u,v) appears in two places in the data structure - once in the lower queue as an incoming path of vertex v , represented in the upper queue by the candidate for v , and once in the lower queue of vertex u , as a path outgoing from u . This structure for the upper queue is shown in figure 3.3.

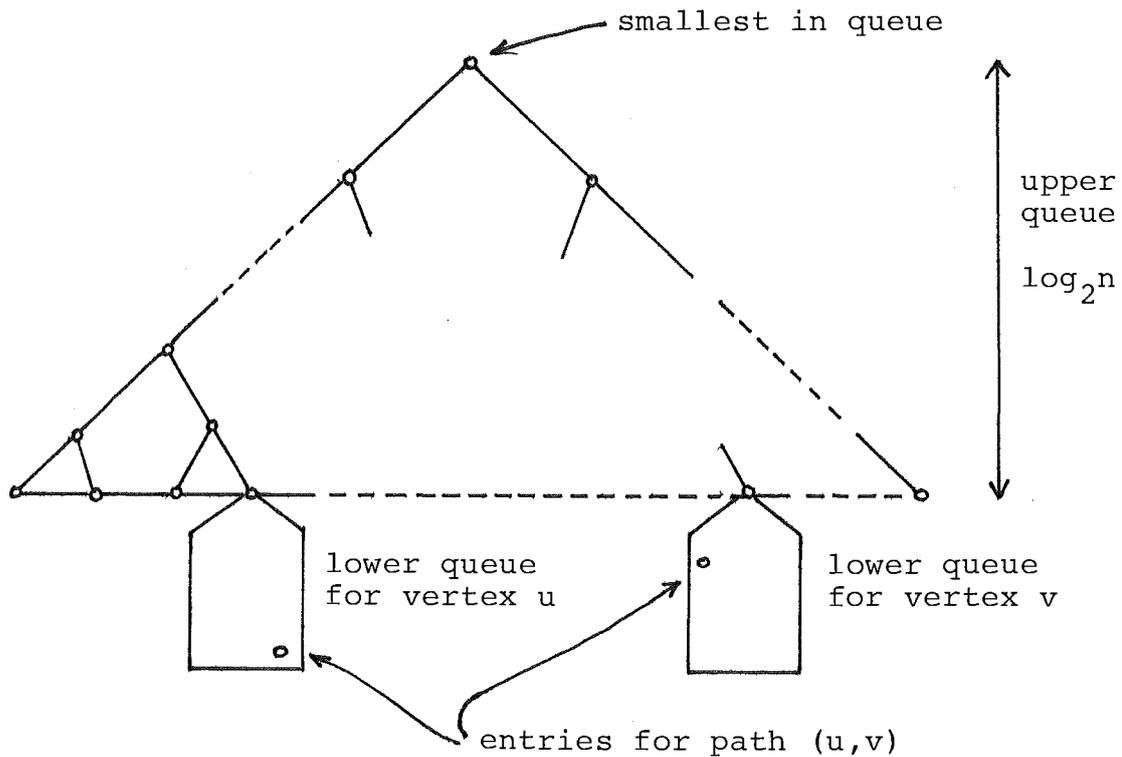


Figure 3.3 - The upper queue.

Because the cost of each path is non-increasing, it is not necessary for the entry in subqueue u for (u,v) to record the same cost as that in subqueue v , so long as one of the two records the most recent value of $\text{pcost}[u,v]$. This is safe since the priority queue "getmin" operation will return the smaller of the two different values first, which is always the most recent value. When $\text{pcost}[u,v]$ is updated, it does not matter which of the two entries in the lower queues is altered. This freedom of choice can be used to good effect.

All paths that are tested and can possibly be updated by any single call $\text{check-pair}(u,v)$ are either of the form (a,v) or of the form (u,b) . All paths of the form (u,b) are outgoing from vertex u , and can be represented in the upper queue by the candidate for u , and all paths of the form (a,v) are incoming at v , and can be represented, utilizing the freedom

of choice, in the upper queue by the candidate for v . Thus when (u,v) is being checked all updates can be confined to two of the lower queues, so that in the upper queue only two of the entries will change between successive `getmin` operations. Pair (u,v) will itself be deleted from the entire data structure, but this change is also restricted to the same two candidates in the upper queue. Thus each execution of the main while loop, involving a "getmin" operation and a call to procedure `check-pair`, will require only $O(\log n)$ time and $O(\log n)$ comparisons of path costs in the upper queue.

Construction of the upper queue will require $O(n)$ time, once the initial candidates from the lower queues have been established. Over the whole algorithm `greedy-apsp`, the upper queue will require $O(n^2 \log n)$ time and $O(n^2 \log n)$ active operations.

The lower queue.

Associated with each vertex is a lower queue. Each of the lower queues contains as many as $2n$ entries, representing all the unchecked paths incoming or outgoing at that vertex. The paths are stored as elements in no more than k doubly linked lists, where each list is in non-decreasing order from head to tail. The element of least cost in each lower queue is the smallest of the k list head elements, and this is the element passed to the upper queue.

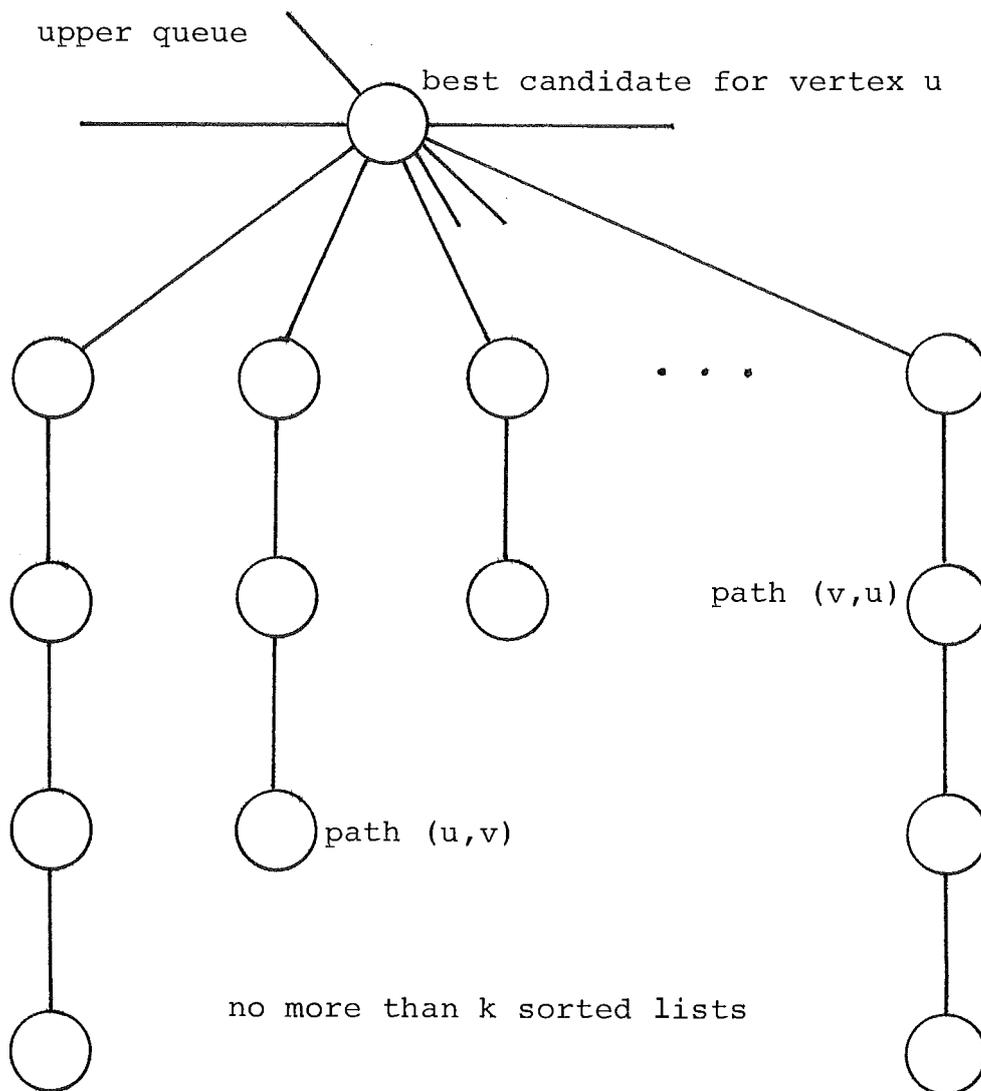


Figure 3.4 - The lower queue for vertex u.

Pointers are kept to the location of each element so that any element can be deleted in $O(1)$ time. Such a deletion may affect the validity of the upper queue candidate. However, if there is a sequence of deletions between successive getmin operations, the upper queue candidate can be re-established "lazily", that is, only when it is actually needed, and not after every deletion. Thus the deletion of a path requires only $O(1)$ time, and no active operations.

The initial construction of each lower queue requires that the list of $2(n-1)$ edges incident at a vertex be sorted and formed into a single doubly linked list; this will take for each subqueue $O(n \log n)$ time and comparisons on edge costs.

From lemma 3.3 it is known that the paths checked form a non-decreasing sequence, meaning that for each vertex a sorted list of checked outgoing paths and a sorted list of checked incoming paths can be maintained without additional data operations. If these lists are used as the ordering of the "for" loops of procedure check-pair then the sequence of new paths resulting from successful tests will also be in non-decreasing order, since in effect a constant is being added to a subset of an ordered list. So for tests that take place but do not cause an update no change is made in the lower queue; for tests that do cause an update the entry for the corresponding edge is removed from its original list, and appended with its new weight at the tail of a list of successful updates, a list that grows as the for loop proceeds. Again a lazy approach is taken to the sequence of operations, and the upper queue candidate is established only when it becomes necessary. In this way a sequence of update operations in the lower queue can be accomplished in $O(1)$ time per update, and no path operations.

When an getmin operation is called for, candidates for the two altered lower queues must be correctly established. At this point the two lists of updated paths, one for u and one for v , are added to the corresponding lower queues. This may, however, result in the number of such lists in one or both of the subqueues growing beyond k . For each of the

subqueues in which this happens the two shortest of the $k+1$ lists in that subqueue are identified, taking $O(k)$ time, and merged to make a single sorted list. The merging will require $O(n/k)$ time and $O(n/k)$ active operations. After this is done the candidate for the upper queue can be selected from amongst the no more than k list head elements using $k-1$ comparisons, and the consistency of the lower queue data structures is preserved.

Each getmin operation will thus require $O((n/k)+k)$ time and path operations in the two lower queues affected. Choosing k to grow as \sqrt{n} means that lower queue getmin operations can each be accomplished in $O(\sqrt{n})$ time and operations, dominating the time required by the operation in the upper queue. Each of the $O(n^3)$ updates will require $O(1)$ time, so the total effort in the priority queue is $O(n^3)$ time and $O(n^{2.5})$ path operations. This is the result that was claimed in section 3.2.

Theorem 3.8 Let $N=(G,C)$ be a non-negatively weighted network. Then algorithm greedy-apsp solves the apsp problem on N , and requires no more than $n^3 + o(n^3)$ active operations in the worst case.

Proof. From theorem 3.5, lemma 3.7, and the above discussion. []

3.4 Some Further Observations.

The bound on the number of tests is sharp.

It was shown in section 3.2 that $(1/2)n(n-1)(n-2)$ is an upper bound on the number of tests performed during all calls to procedure check-pair. Here it is demonstrated that this bound is tight by giving a family of graphs that requires this number of tests. In what follows, let $T(n) = (1/2)n(n-1)(n-2)$. Lemma 3.7 implies that $T(n)$ tests are performed in a graph if and only if each source, in each triangle of vertices, is the origin of exactly 1 test. To show that the bound is tight, it is necessary to construct a graph in which every source in every triple is the origin in exactly one test. One such generic n -vertex graph has a cost function given by

$$C(i,j) = (j-i) \bmod n \quad \text{for all } (i,j) \text{ in } V \times V.$$

For example, the distance matrix for such a graph with $n=5$ is

	1	2	3	4	5
1	0	1	2	3	4
2	4	0	1	2	3
3	3	4	0	1	2
4	2	3	4	0	1
5	1	2	3	4	0

A subset of the edges of this 5 vertex graph is shown in figure 3.5.

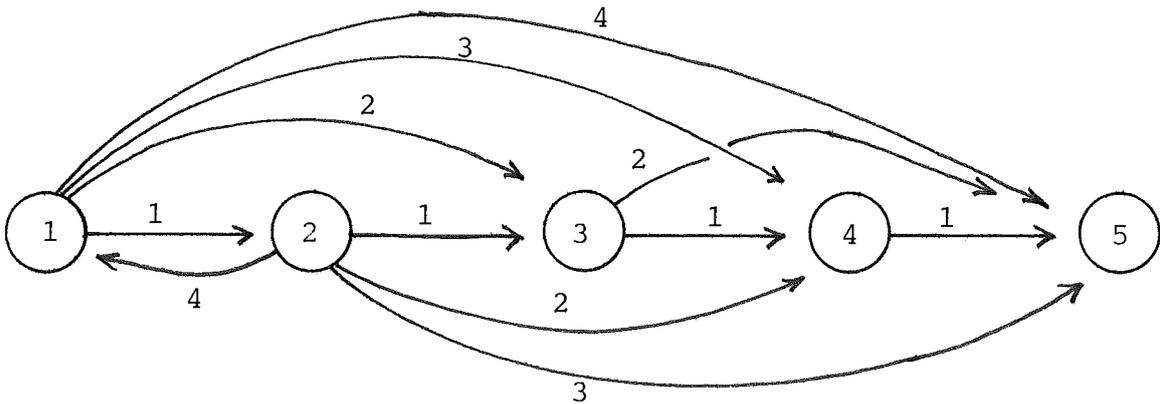


Figure 3.5 - A graph with $C(i,j) = (j-i) \bmod n$.

Lemma 3.9 Graphs of n vertices with the cost function $C(i,j) = (j-i) \bmod n$ require $T(n)$ tests under the greedy-aps algorithm.

Proof. Consider any triangle of distinct vertices s, a, b , as was shown in figure 3.2. Assume that $C(s,a) < C(s,b)$; a symmetrical case holds when $C(s,b) < C(s,a)$, and they cannot be equal.

Then

$$0 < (a-s) \bmod n < (b-s) \bmod n < n$$

ie $0 < (b-s) \bmod n - (a-s) \bmod n < n$ (a).

Hence

$$\begin{aligned} C(a,b) &= (b-a) \bmod n \\ &= ((b-s) - (a-s)) \bmod n \\ &= ((b-s) \bmod n - (a-s) \bmod n) \bmod n \\ &= (b-s) \bmod n - (a-s) \bmod n && \text{by (a)} \\ &< C(s,b) . \end{aligned}$$

That is, if (s,a) is checked before (s,b) , then so too is (a,b) and the conditions for an updating triangle will be satisfied. This holds for every source in every triangle, so

that exactly $T(n)$ tests will be required. []

It is worth noting that the cost function C defined here is also a worst case for the good expected time algorithms of Spira et al. This will be discussed further in section 5.5.

The bound on the number of updates is not sharp.

For each test, there may or may not be an update required; for the analysis of the algorithm in sections 3.2 and 3.3 the worst was assumed - that every test resulted in an update to a path cost. This assumption is unduly pessimistic; $T(n)$ updates can be performed in a graph if and only if every source in every triangle of vertices in the graph is the origin in exactly 1 update, and lemma 3.10 shows that this is not possible.

Lemma 3.10 Let $U(n)$ be the maximum number of updates required by the greedy algorithm for any graph on n vertices. Then $U(n) < T(n)$.

Proof. It is only necessary to show the existence of a triangle of vertices containing a source from which neither potential update took place. Consider the action of algorithm greedy-apsp on some graph that requires $T(n)$ tests, and assume that $U(n) = T(n)$. There are two cases:

Case 1. Suppose that from some source in the graph there are two shortest paths of length 1, that is, two edges (s,a) and (s,b) that were not updated during the course of the greedy algorithm. Then in the triangle s,a,b there were no updates performed with s as origin. Since every vertex must be the origin of at least 1 edge that

was not updated, it must be that every vertex in the graph is the origin of exactly one edge that was not updated.

Case 2. Suppose that from some vertex in the graph there is a shortest path of length 3 or more. Then there is some chain of three edges, none of which were updated, such as in figure 3.6:

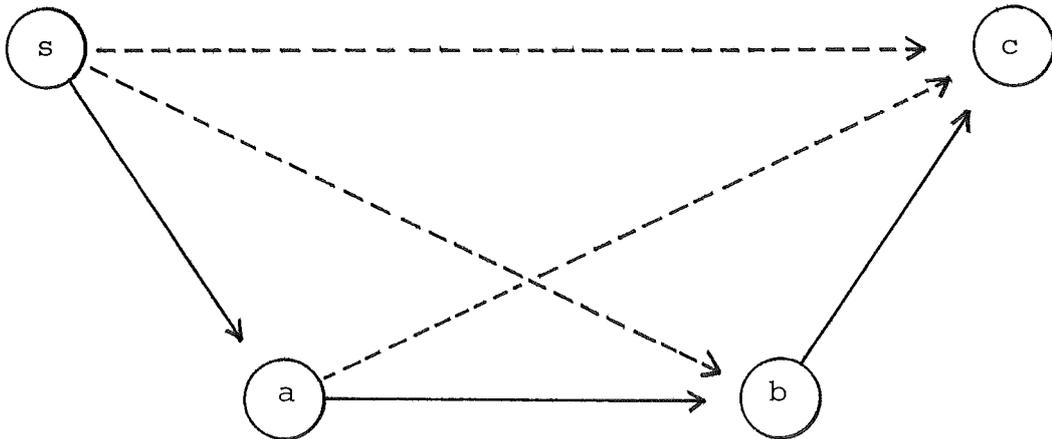


Figure 3.6 - A chain of three edges.

By lemma 3.6, in triangle s,a,c edge (s,a) was not updated, and in triangle s,b,c , (s,b) was not updated. For there to have been $T(n)$ updates it must then have been the case that (s,c) was updated in both s,a,c and s,b,c . But this is not possible, as the cost of these two paths are identical - they comprise the same three edges - and the update will only take place if the cost is to be improved. Thus if $U(n) = T(n)$ there can be no shortest paths of length three.

For the requirements of the two cases to be met, the shortest path spanning tree from every vertex s must have the form shown in figure 3.7:

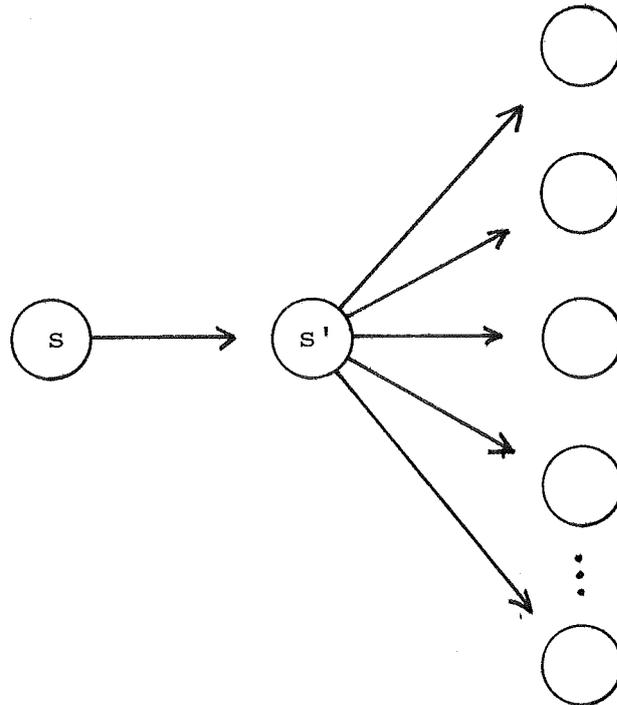


Figure 3.7 - Spanning tree from vertex s .

However, if the spanning tree for a vertex s has this shape it is not possible for the spanning tree for vertex s' to also have the required shape, and thus it cannot be that $U(n) = T(n)$. []

3.5 Extending the Queue to Dijkstra's Algorithm.

The key points that make the "parallel chains" lower queue of section 3.3 efficient are encapsulated in the following program. The parallel chains priority queue is suitable for use in any situation described by this algorithm skeleton:

```

initialise queue to contain n elements
  of the form (v,weight) ;
while queue is not empty do
  begin
    (v, weight) := getmin(queue) ;
    delete(queue, v) ;
    with (v, weight) do
      update some elements still in the queue, with
        the restriction that the sequence of updated
        weights is non-decreasing ;
  end ;

```

Algorithm 3.4 - Skeleton for priority queue algorithms.

The analysis of section 3.3 showed that if the parallel chains priority queue is used for this skeleton, the effort required in the priority queue operations will be $O(n \log n)$ time and comparisons of weights for the initial sorting, $O(\sqrt{rn})$ time and comparisons for each getmin operation, and $O(1)$ time for each delete and update operation. These bounds rely heavily on the requirement that the sequence of updates between successive getmin operations is such that the new weights of the updated elements form a non-decreasing sequence. But given that this requirement is met, if there are m update operations in the course of some algorithm built around the skeleton, the total time required will be $O(m+n^{1.5})$, and the total number of comparisons of weights will be $O(n^{1.5})$.

In section 3.3 the requirements of the lower queue for algorithm greedy-apsf fell within the scope of this

definition, and the parallel chaines priority queue was used as a feeder queue for the binary tournament tree that was used as the upper queue. Here it is shown that Dijkstra's (1959) algorithm for finding the shortest paths from one source vertex s can also be made to fit the skeleton:

```

procedure dijkstra( s ) ;
begin
  set queue to empty ;
  for v in V-{s} do
    set D[v] := infinity and add (v,D[v]) to queue ;
  set D[s] := 0 and add (s,D[s]) to queue ;
  while queue is not empty do
    begin
      (v,D[v]) := getmin(queue) ;
      delete(queue,v) ;
      for (u,D[u]) remaining in queue do
        test: if D[v]+C(v,u) < D[u] then
          update D[u] := D[v]+C(v,u)
    end ;
  end {dijkstra} ;

```

Algorithm 3.5 - Dijkstra's Algorithm.

It is necessary to show that the sequence of updates is such that between successive getmin operations the new weights are non-decreasing. This will be possible if the "for" loop performing the updates processes the elements u remaining in the queue in order of increasing $C(v,u)$, easily achieved by presorting the edge lists of the graph into non-decreasing order. By using the algorithm n times to solve the apsp problem the $O(n^2 \log n)$ effort of the presort can be spread and absorbed.

The standard analysis of Dijkstra's algorithm (Moffat and Takaoka, 1984), is that for the apsp problem on a complete graph there will be $(1/2)n(n-1)(n-2)$ executions of the statement marked "test:", accounting for n^3 active operations; to this should be added whatever is required by

the priority queue. Thus this implementation of Dijkstra's algorithm will require no more than $n^3 + O(n^{2.5})$ active operations to solve the apsp problem.

The standard analysis also shows that for each single source problem solved by Dijkstra's algorithm there can be no more than m executions of the statement marked "test:", one for each edge in the graph. This observation means that an implementation of Dijkstra's algorithm using the parallel chains data structure will require no more than $2mn + O(n^{2.5})$ active operations in the worst case. Combining these two results gives

Theorem 3.11 Let $N=(G,C)$ be a non-negatively weighted directed network of n vertices and m edges. Then an algorithm exists for solving the apsp problem on N that requires in the worst case $O(mn + n^{2.5})$ time and $\min\{ n^3, 2mn \} + O(n^{2.5})$ active operations on path and edge costs.

Proof. From the discussion above. See also Moffat and Takaoka (1984). []

This result improves the bound of Dantzig, and also improves by a constant factor the previous best bound on active operations for an implementation of Dijkstra's algorithm, namely the $1.5n^3$ bound of Yen (1972).

CHAPTER FOUR
AN $O(n^2 \log n)$ AVERAGE TIME ALGORITHM.

4.1 The Dantzig/Spira Paradigm

The new fast average time algorithm is based on the algorithm of Spira (1973). Crucial to the description of new method is an understanding of the shortest path searching paradigm first introduced by Dantzig (1960) and later exploited by Spira; with this in mind this section gives a brief overview of Dantzig's single source algorithm, and then a detailed description of Spira's 1973 result, the first of the fast average time algorithms for the amsp problem. Throughout this chapter analyses will be for average running time on complete graphs.

Dantzig's algorithm.

In Dantzig's method and in Dijkstra's algorithm (1959) an amsp solution is found by solving n single source problems. Consider the problem of finding the shortest paths from some source vertex s . To solve this ssp s is assigned a shortest path cost of zero and made the only member of a set S of labelled vertices for which the shortest path costs are known. Then, under the constraint that members of S are "closer" to s than non-members, ie that for each v in S ,

$$L(s,v) \leq L(s,u) \text{ for all } u \text{ in } V-S,$$

the set S is expanded until all vertices have been included and hence all shortest paths from the source are known. To make the expansion of S computationally easy, information is

maintained about paths from vertices already in S to vertices still outside S . This is the point at which Dantzig and Dijkstra diverged.

For each vertex c in S Dantzig maintains a "candidate". The candidate for a vertex must be outside the current S , and for vertex c is selected by scanning the sorted list of edges out of c until an edge with an unlabelled destination is found. That is, the candidate for a vertex c in S is the closest (by a single edge) vertex to c that is not yet in S . Suppose that the optimal path costs of labelled vertices are recorded in a vector D , and that t is the candidate for some vertex c . Then there are no shorter edges from c that lead to unlabelled vertices, meaning that the next vertex that can possibly be included in the shortest path spanning tree as a descendant of c must be t , with a shortest path cost of $D[c]+C(c,t)$. This expression " $D[c]+C(c,t)$ " will occur frequently; $D[c]$ is the known shortest path cost from s to c , and $C(c,t)$ is the cost of the edge from c to t , thus if c is to be the father of t in the shortest path spanning tree, $D[c]+C(c,t)$ is the shortest path cost from s to t .

Vertex t might also be the candidate of other already labelled vertices c' ; each time it is a candidate there will be some "weight" $D[c']+C(c',t)$ associated with its candidacy. At any stage of the algorithm there will be $|S|$ candidates.

Provided that these constraints have been met, at each stage of the algorithm the candidate of smallest weight can be included in S and its shortest path cost confirmed. That is, if c is the vertex such that the candidate cost $D[c]+C(c,t)$ is minimum over all labelled vertices c , then t can be

included in S and given a shortest path cost $D[t]$ of $D[c]+C(c,t)$. No other vertex can possibly label t with a smaller cost, since all edge costs are non-negative. In this way, the solution set can be expanded by one vertex. Then an onward candidate for t is added to the list of candidates, the candidates for c and any other vertices that may have had t as their candidate are revised, and the process repeats, to stop when $|S|=n$ and all vertices have been assigned shortest path costs. Figure 4.1 shows some intermediate stage during this expansion of S .

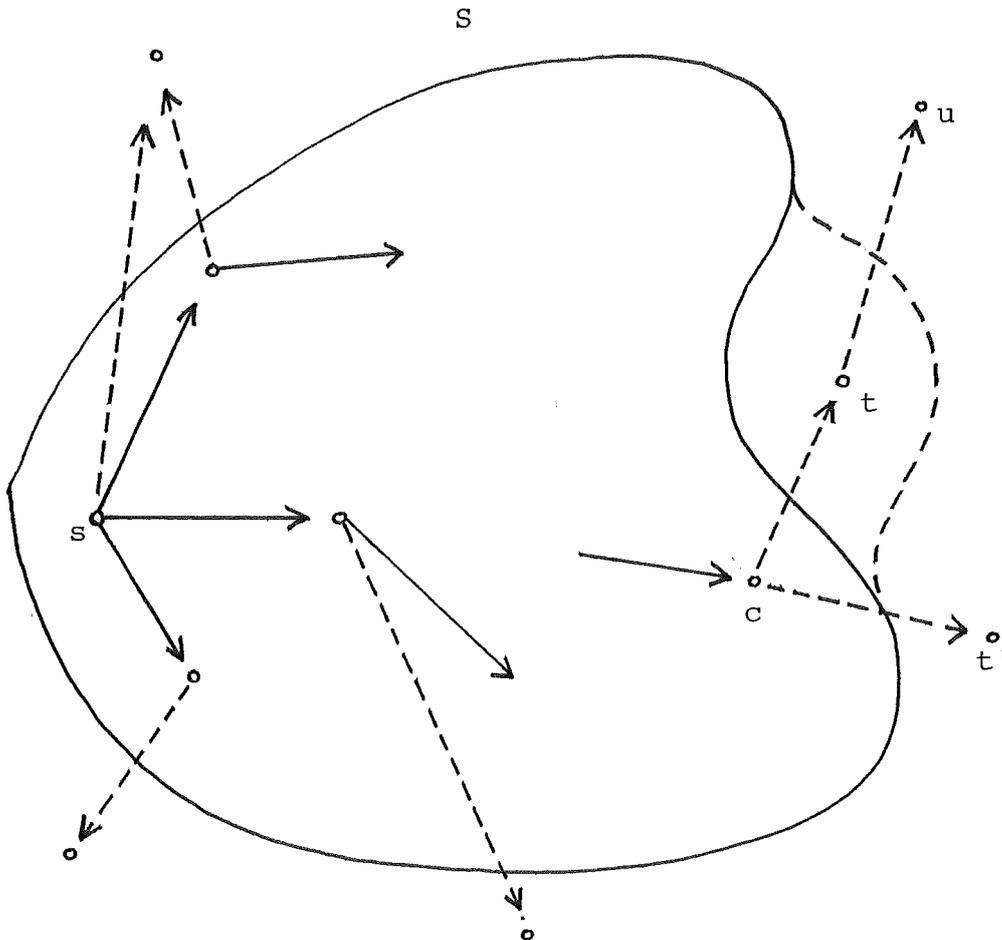


Figure 4.1 - Dantzig's algorithm.

Each element t added to S will affect the set of candidates - t itself must be given a candidate, and the candidates of other members may need to be revised, as some of them will have had t as their candidate. Provided that the edges out of each vertex can be processed in order of non-decreasing cost, these changes can be accomplished by, for each c in S , skipping edges until an edge leading to a vertex outside S is encountered.

To correctly initialise this process, the source s is made the only member of S with $D[s]$ set to 0; if t is the destination of the shortest edge out of s then the candidate for s is t with weight $D[s]+C(s,t)$.

The analysis of the method is straightforward. When $|S|=j$, $O(j)$ effort is required to find the minimum cost candidate, totalling $O(n^2)$ per source; the edge scanning to find unlabelled candidates will throughout a whole single source problem move over each edge of the graph no more than once, totalling $O(n^2)$; and when $|S|=j$ the checking of the labels to decide whether or not any scanning is required will require $O(j)$ time. Including the time needed for the ordering of the edges as $t(n)$, this gives a total running time of $O(n^2+t(n))$ for the ssp and $O(n^3+t(n))$ time for the apsp problem.

Dantzig presented his method as a solution to the single source problem. His analysis was not precise; in particular he assumed "that one can write down without effort for each node the arcs leading to other nodes in increasing order of length". Thus he claimed $O(n^2)$ running time for the single source problem, by assuming that $t(n)=0$. This is an unwarranted assumption, and in general it will be necessary

to sort the edges outgoing at each vertex. The computational effort in doing this is not small; even an efficient sorting technique such as heapsort will require $O(n \log n)$ time for each edge list of a dense graph, and thus the running time for the ssp is more correctly stated as $O(n^2 \log n)$. For the apsp problem the bound is still $O(n^3)$; the pre-sort of the edgelist needs only be done once, and when spent the effort can be shared over all sources. Dantzig's method is not efficient for the single source problem; to solve a ssp problem Dijkstra's similar algorithm, which does not require a pre-sort, should be used.

Spira's algorithm.

Seemingly independently, as he does not cite Dantzig, Spira in 1973 developed a similar algorithm. The crucial difference between the two methods is that whereas Dantzig requires that the candidate for each labelled vertex c be outside the current S , Spira does not. Dantzig's strict rule means that every change to the membership of S must be accompanied by an inspection of the destination of each of the current candidates, and forces an $\Theta(n^2)$ bound for each ssp. Spira's liberalisation of the requirement means that changes to the membership of S do not cause inspection of the destination of every candidate, but does mean that the minimum weight candidate can no longer be guaranteed to be useful for labelling purposes, as it might lead to a vertex already in S . This new situation is shown in figure 4.2.

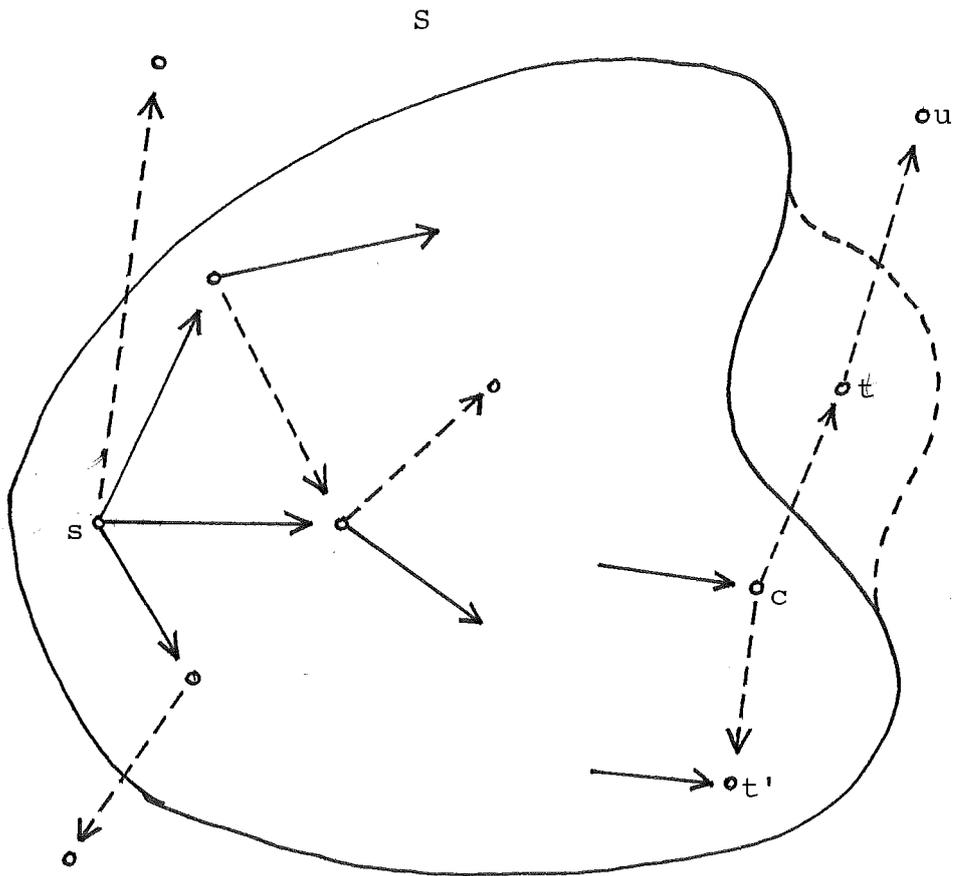


Figure 4.2 - Spira's algorithm.

Thus, more than $n-1$ "find the minimum cost candidate" stages are likely to be required. However, as the analysis below shows, when an appropriate data structure is used and the graph is "random", the trade pays off handsomely and results in asymptotically improved running time.

The following program is for Spira's single source algorithm. The edge lists are assumed to be in non-decreasing order. The variable "heap" is a binary heap of pairs (c,t) , where the weight of element (c,t) is given by $D[c]+C(c,t)$.

```

procedure spira-ssp( s ) ;
begin
  S := {s} ; D[s] := 0 ;
  initialise the heap to (s,t), where t is the endpoint
    of the shortest edge from s ;
  expand-soln( n ) ;
end {spira-ssp} ;

procedure expand-soln( stopat ) ;
begin
  while |S| < stopat do
  begin
    let (c,t) be at the root of the heap ;
    replace (c,t) by (c,t'), where t' is the endpoint
      of the next shortest edge from c, and rearrange
      the heap ;
    if t is not in S then
    begin
      {expand S, label new vertex, add a candidate}
      S := S + {t} ;
      D[t] := D[c]+C(c,t) ;
      add into the heap (t,u), where u is the
        destination of the shortest edge from t,
        and rearrange the heap ;
    end ;
  end ;
end {expand-soln} ;

```

Algorithm 4.1 - Spira single source.

To record the actual shortest paths it is sufficient to have a vector in which, if (c,t) is a candidate that results in an expansion of S , c is recorded as being the labelling vertex for t , and is thus the immediate ancestor of t in the shortest path spanning tree. Then for any vertex the path that labelled that vertex can be traced back in reverse order by following the father pointers.

Correctness.

The following argument, given substantially by Bloniarz (1983), shows that the algorithm is correct. First, the constraints on the vertices of S are repeated: for all c in S , if (c,t) is the heap entry for c , then

$$L(s,c) = D[c],$$

$$L(s,c) \leq L(s,u) \text{ for all } u \text{ in } V-S, \text{ and}$$

$$C(c,t) \leq C(c,u) \text{ for all } u \text{ in } V-S .$$

That is, D correctly records shortest path costs for labelled vertices; all unlabelled vertices are further from s than all labelled vertices; and all edges that have already been examined lead to labelled vertices.

From these and the non-negativity of the cost function Bloniarz made the observation:

Lemma 4.1 Suppose c_0 in S is such that

$$D[c_0] + C(c_0, t_0) = \text{minimum}\{ D[c] + C(c, t) : (c, t) \text{ in heap } \}.$$

Then if t_0 is in $V-S$

$$L(s, t_0) = D[c_0] + C(c_0, t_0), \text{ and}$$

$$L(s, t_0) \leq L(s, u) \text{ for all } u \text{ in } V-S .$$

Proof. See Bloniarz (1983). []

This observation shows that the algorithm will maintain the constraints on S as more and more vertices are added, and when $|S|=n$ the first constraint means that the vector D contains the required shortest path costs. Only when an unlabelled candidate is drawn as the minimum cost element in the heap will S be expanded; to see that $|S|$ will ultimately reach n (for a complete graph) and that the algorithm will terminate, observe that every iteration of the loop of procedure `expand-soln` irrevocably "consumes" one edge from one of the sorted edge lists, and so at most n^2 iterations of this loop can be required before all edges have been examined and thus all vertices labelled.

In the worst case, Spira's method can be much worse than that of Dantzig. On a complete graph that has some "bad luck" combination of edge weights, every edge might still need to be examined before the paths found can be known to be optimal. Each edge examination requires $O(\log n)$ time for the corresponding heap operation, meaning that the total running time for the apsp problem might become $\Theta(n^3 \log n)$, rather than the $O(n^3)$ worst case bound of Dantzig's method. One such bad luck graph was described in section 3.4; the subject will be explored in more detail in section 5.5.

Random graphs.

For an average case analysis it is necessary to define the type of "randomness" over which the average is to be taken. Spira analysed the expected behaviour of his algorithm in terms of a randomness model in which the edges of a dense graph are assigned costs by $n(n-1)$ independent drawings from any random distribution. The distribution itself is arbitrary. Bloniarz (1983) redefined this randomness model, widened it to a class he calls "endpoint independent" graphs, and showed that the average running time of a slightly modified Spira's algorithm on this wider class of random graphs is still $O(n^2 \log^2 n)$.

The primary property of an endpoint independent probability measure that is exploited in his analysis Bloniarz describes thus: "Suppose a particular edge is selected from the sorted edge list by virtue of either its position on the list or the value of its cost. If P [the probability measure] is endpoint independent, then the endpoint of this edge is independent of

the edge's selection; every endpoint is equally likely" (Bloniarz 1983, page 594). This requires that edges in the edge list of equal cost be stored in a random order within the overall sorted order (Bloniarz et al, 1979); once this is done each sorted edge list will be a random permutation of that edge list when ordered by destination, and when the next edge (c,t) of any sorted edge list is taken it is equally likely to be any of the $n-1$ other vertices in the graph. The repeated examination of the destinations of edges as any edge list is scanned can be taken to be a sequence of independent trials.

All of the analyses given here concerning average running time will make this same assumption - that the destination of an edge is independent of the source of the edge, the cost of the edge, and the position of the edge in the sorted edge list. This is the basic randomness assumption that describes the "average" graph that these algorithms handle well.

A number of types of edge cost assignments meet the endpoint independence requirement. The simplest situation is when each edge in the graph is independently assigned a value from any single distribution of any sort; but also within the scope of the definition is the situation in which each source vertex has associated with it some different random distribution function, and edges are assigned costs independently drawn from the distribution of the corresponding source.

Analysis of Spira's algorithm.

Given this definition of the classes of random graphs for which shortest paths are required, Spira's algorithm is analysed as follows. At the j 'th stage, when $|S|=j$, the heap contains j candidates. The minimum cost candidate is then drawn in an attempt to label a new vertex; because of the endpoint independence the destination of the minimum cost candidate is equally likely to be any of the n vertices, and of these there are only $n-j$ that are unlabelled. The probability of drawing an unlabelled vertex as the minimum cost candidate is thus $(n-j)/n$, assuming that in the case of candidates with equal and minimal weight the tie is broken randomly, without reference to the destination of the candidate.

The process is continued with $|S|=j$ until an unlabelled vertex is drawn; the expected number of such drawings until S can be expanded is $n/(n-j)$ (lemma 1.1). As S expands from one element to n elements the total number of drawings from the heap for one source will be given by $\sum_{j=1}^{n-1} (n/(n-j))$. This sum is $O(n \log n)$. Each drawing requires a heap operation, so that summed over all sources of an a-psp problem the time required is $O(n^2 \log^2 n)$, which dominates the time required by the presort.

This process of randomly selecting from amongst n equally likely elements, and continuing to select until each element has been chosen at least once, is an application of Feller's (1968) "coupon collector" problem - for example, how many boxes of cereal must be purchased before all of a set of p "Birds of New Zealand" cards have been collected. The

standard answer (Knuth, 1973a) is $p \cdot \ln(p)$, meaning that when p is 20 the hapless collector is required to purchase on average 60 packets of cereal. This coupon collector problem, and the standard solution to the summation, will be used again in the analysis of the new algorithm.

This analysis indicates that expectedly only $O(n \log n)$ edges will be examined in the course of a single source problem. Then to solve a single source problem, rather than presorting the entire edge lists for each of the n vertices, which takes $O(n^2 \log n)$ time, it is more efficient to build a heap of edges for each vertex, taking $O(n^2)$ time in total, and then each time a "next shortest" edge is required, using the appropriate heap to obtain the edge. The $O(n \log n)$ "next shortest edge" requests will take $O(n \log^2 n)$ time, so that a single source problem can be solved with $O(n^2 + n \log^2 n)$ time for ordering and $O(n \log^2 n)$ time for algorithm Spira-ssp. In this way Spira's algorithm can be applied to the single source problem with $O(n^2)$ expected running time. However, for practical purposes, Dijkstra's algorithm is much faster, and is certainly much easier to implement. Dijkstra's algorithm also has the advantage of an $O(n^2)$ worst case bound, whereas this implementation of Spira's technique would have an $O(n^2 \log n)$ worst case running time. The technique of using n heaps to order the edges cannot be efficiently applied to Dantzig's algorithm, as even for a random graph that algorithm will examine $\Theta(n^2)$ edges in the course of an ssp computation, requiring $\Theta(n^2 \log n)$ time for the ordering.

4.2 Fredman's Modification.

Fredman (1975) developed the concept of "heap cleaning" and used this idea to show that Spira's algorithm could be modified to require only $O(n^2 \log n)$ comparisons, but at the cost of increasing the running time to $\Theta(n^3)$ for the apsp problem. The heap cleaning, or removal and replacement of "dirty" candidates that lead to vertices that are already labelled, is to increase the probability of an unlabelled candidate being successfully drawn from the heap. His technique is implemented on top of the Spira program already described.

Spira's algorithm is allowed to begin normally. When the size of S reaches $n/2$ and half of the n initially unlabelled vertices have been labelled, the normal processing is suspended and a heap cleaning stage carried out. First step in a cleaning stage is to mark as purged all edges leading to labelled vertices, so that in future they can be skipped and their use can be avoided. This will require $O(n)$ time per labelled vertex. Secondly, each candidate in the heap is examined, and all candidates that are found to be labelled are replaced by unlabelled candidates, which are readily found by scanning the edge lists and skipping all purged edges. While replacing these "dirty" candidates, the heap will lose its heap property, but during the second step this is permitted. Finally, the heap is completely rebuilt, taking $O(n)$ time. The key point here is that during the cleaning stage expectedly one half of the candidates will have been dirty; had they been replaced individually as they "floated" to the root of the heap, the cleaning would take

$O(n \log n)$ comparisons rather than the $O(n)$ comparisons that are required for a complete heap rebuilding.

Once the heap has been cleaned, the normal Spira type processing is resumed, but with the added requirement that whenever a purged edge is encountered it should always be skipped. As a consequence, none of these $n/2$ vertices labelled in the first phase and purged in the first cleaning will ever reappear as a candidate.

When $|S|$ reaches $3n/4$ the processing is interrupted for a second cleaning, as again the heap will have become expectedly half dirty. This alternation of normal processing and cleaning continues, with heap cleaning taking place when $|S|=n/2, 3n/4, 7n/8, \dots, n-4, n-2, n-1$, or $\log n$ cleaning steps in all.

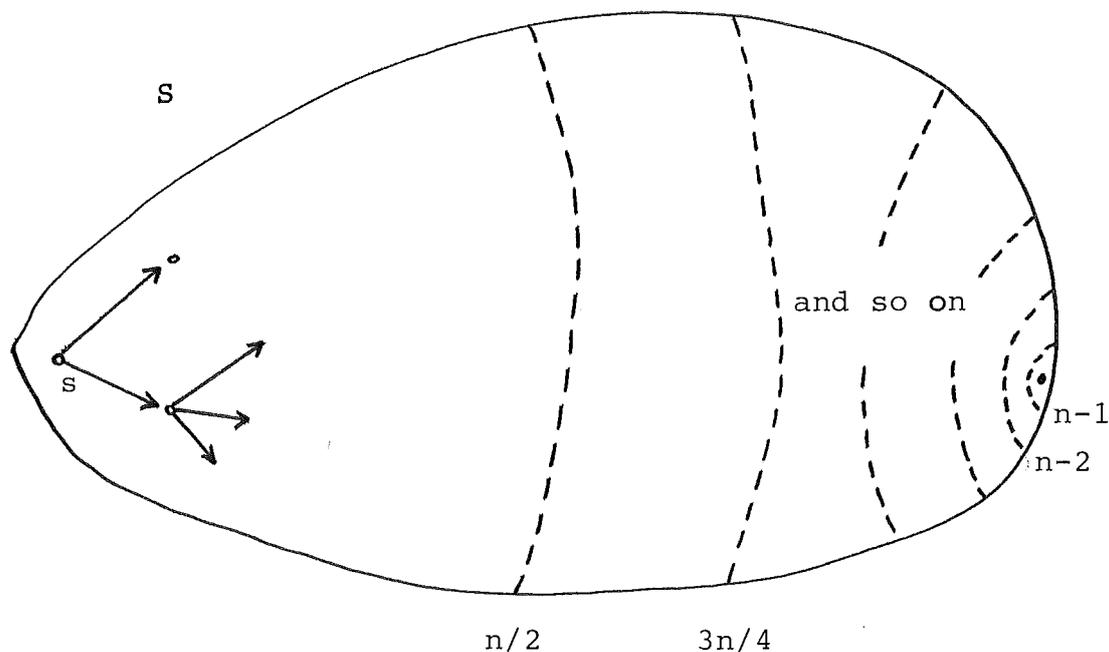


Figure 4.3 - Fredman's cleaning phases.

Each heap cleaning step will require some amount of time to mark edges as being purged, some amount of time to find the clean candidates, and $O(n)$ comparisons for the rebuilding stage. Over each single source problem the heap rebuildings will require $O(n \log n)$ comparisons.

The regular cleaning ensures that the probability of drawing an unlabelled candidate at the root of the heap is never less than $1/2$. At any stage of the processing the destination of a candidate will be among the set of vertices that were unlabelled at the most recent heap cleaning operation, since at that time all labelled candidates were removed from the heap and all edges leading to those labelled vertices were purged. Between any heap cleaning operation and the next the number of unlabelled vertices halves, so that the probability of drawing an unlabelled candidate as the minimum on the heap is never less than $1/2$. Thus the number of heap operations required before S expands from j elements to $j+1$ elements is always expectedly less than 2, and the number of heap operations required to label all n vertices in a single source problem is on average $O(n)$. Each heap operation will require $O(\log n)$ comparisons, and the presort will require $O(n^2 \log n)$ comparisons, so that for the apsp problem an average of $O(n^2 \log n)$ comparisons will be sufficient.

Each of the cleaning operations will require some time to mark edges as being purged. In the course of the cleaning stages each vertex will have all of its incoming edges marked once, taking $\Theta(n^2)$ time per source on a dense graph.

Once the edges have been marked there will also be additional time spent in the scanning of edges looking for an allowable

or unpurged candidate. By the time the heap is cleaned with $|S|=n-1$, each edge list pointer will have expectedly moved to a position midway along the edge list, as the last remaining vertex will be the candidate of all of the other $n-1$ labelled vertices. So for a single source problem the total scanning effort must be $\Theta(n^2)$, as for Dantzig's algorithm.

Fredman gave this algorithm as a demonstration that on average $O(n^2 \log n)$ comparisons were sufficient, but the algorithm itself takes $\Theta(n^3)$ time. He was unable to improve the running time to give an efficient algorithm, so this result is of theoretical interest only. However the idea of heap cleaning has been used in the new algorithm, giving an $O(n^2 \log n)$ bound for time as well as comparisons. Fredman's " $O(n^2 \log n)$ comparisons are sufficient" has been turned into a practical algorithm that attains the bound.

In their recent paper Frieze and Grimmett (1983, 1985) have re-invented this algorithm of Fredman's, and with a slightly different analysis show that the running time will expectedly be $O(n \log n + m)$ for a single source problem, not counting the cost of the pre-sort. Thus they obtained the same bound as Fredman, who only considered dense graphs.

4.3 Bloniarz's Modification.

The heuristic which to date has had the most effectiveness in reducing the asymptotic complexity of Spira's algorithm is that of limited scanning, developed independently by Takaoka and Moffat (1980) and Bloniarz (1980). Under this strategy, when a candidate edge is required from the edge list of some vertex, the next element is not blindly taken to be the candidate. Instead, a sequence of elements is examined until an edge with a good likelihood of being useful is encountered. To avoid possibly high scanning times the number of edges examined is limited. This technique greatly improves the probability of an unlabelled element being drawn as the minimum in the heap, and leads to asymptotically improved running time.

The effectiveness of the technique comes from the fact that the limited scanning, in terms of the asymptotic behaviour of the algorithm, is free. To understand this point, consider this small code fragment taken from procedure `expand-soln` of algorithm `Spira-ssp`:

```
let (c,t) be at the root of the heap ;  
replace (c,t) by (c,t'), where t' is the endpoint of the  
next shortest edge from c, and rearrange the heap ;
```

The "rearrange the heap" operation will take $O(\log n)$ time, and dominates the running time of this small fragment. The statement prior to this selects a new candidate for vertex c ; in Spira's algorithm, as can be seen from this section, the next edge is used as the new candidate without consideration of its destination. This is rather wasteful, as the next edge might lead to a vertex that is already labelled.

Inspection of the label of the potential candidate is cheap, and if it is labelled, the edge can be safely skipped, and an expensive heap operation avoided:

```

let (c,t) be at the root of the heap ;
let t' be the endpoint of the next shortest edge from c ;
while t' is in S do
  let t' be the endpoint of the next "next shortest"
  edge from c ;
  replace (c,t) by (c,t'), and rearrange the heap ;

```

However this code is a bit dangerous in that the running time of the while loop will dominate the time for the heap operation (section 5.2). Much safer is

```

let (c,t) be at the root of the heap ;
let t' be the endpoint of the next shortest edge from c ;
cnt := 0 ;
while (t' is in S) and (cnt < logn) do
  begin
    let t' be the endpoint of the next "next shortest"
    edge from c ;
    cnt := cnt+1 ;
  end ;
  replace (c,t) by (c,t'), and rearrange the heap ;

```

Now the while loop cannot dominate the cost of the heap operation, and the limited scanning is effectively free, completely absorbed by the time of the heap operation. But when as many as $\log n$ edges may be examined there is a good probability that an unlabelled destination can be encountered before the $\log n$ 'th edge must be accepted irrespective of destination, and the limitation is nowhere near as harsh as Spira's "take the next edge".

The analysis of the improved algorithm is complex. Takaoka and Moffat used a slightly different function to bound the maximum number of edges inspected at any scanning step, and were able to show that their version of this technique required $O(n \log \log n)$ heap operations per source. Bloniarz

used the $\log n$ bound function described above and a different analysis technique and obtained a slightly sharper bound of $O(n \log^* n)$ expected heap operations. In both cases the analysis is for endpoint independent probability measures. For the details of these analyses the reader is referred to Moffat (1979) and Bloniarz (1983). The point made here is that limited edge scanning is an important heuristic for fast average time shortest path algorithms, and the new algorithm presented below will make use of a similar heuristic.

4.4 The New Algorithm.

It is often the case that a good algorithm for some problem is the result of a carefully balanced juxtaposition of two or more different ideas. In this way, for example, Hoare's quicksort is the result of combining the concept of recursion with an efficient partitioning scheme. In this section a new algorithm for the apsp problem is presented, and it is shown that it has an $O(n^2 \log n)$ running time on graphs drawn from an endpoint independent probability distribution. The algorithm results from combining and balancing the techniques of Dantzig, Spira, Fredman and Takaoka/Moffat/Bloniarz; the computationally expensive parts of each of these methods have been eliminated to improve upon all. The analysis of the algorithm is surprisingly simple, and the implementation straightforward, a very satisfactory situation. A "good" algorithm should always be simple.

Spira improved upon Dantzig's method by avoiding unnecessary examination of long paths in the graph. Takaoka/Moffat and Bloniarz improved upon Spira by avoiding the examination of redundant short paths in the graph. Fredman also avoided unnecessarily considering short paths in the graph, but his algorithm paid a heavy price in that the total cost of the scanning of edges became $\Theta(n^3)$.

The crucial observation concerning Fredman's algorithm that is made here is that the scanning for a good candidate is expensive only in the closing stages of the algorithm. Even when there are as few as $n/\log n$ vertices remaining unlabelled, scanning for a good candidate in a randomly ordered edge list - a geometric distribution, with

probability of success $(n/\log n)/n$, or $1/\log n$ - will expectedly require the skipping of only $\log n$ edges. Beyond this critical point, however, the scanning required to clean the heap becomes more and more expensive - Fredman's last cleaning stage when $|S| = n-1$ will expectedly examine $(1/6)n^2$ edges. This observation leads to the simple rule that the heap cleaning must be ceased when $|S| = n-n/\log n$. In the concluding stages of the algorithm, as the last $n/\log n$ vertices are labelled, the heap must be allowed to become dirty. Stopping the cleaning when $|S| = n-n/\log n$ means that there will only be $\log \log n$ cleaning stages.

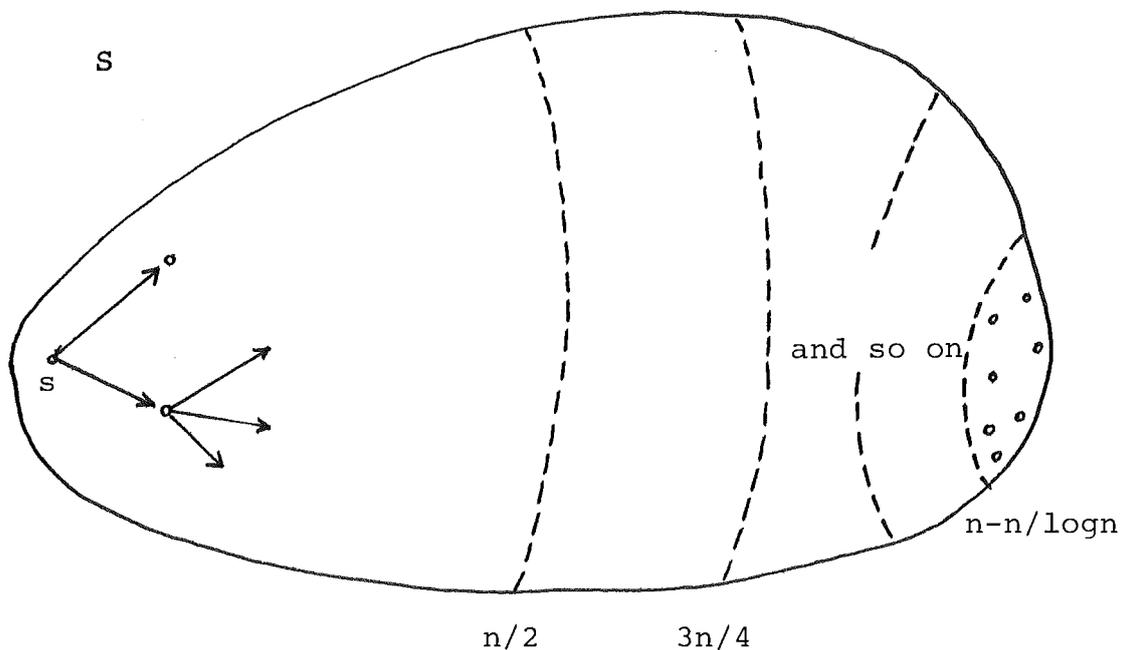


Figure 4.4 - Revised cleaning phases.

A more obvious observation, but no less important, is that it is not necessary to mark edges as being purged (Fredman, 1975) or unlink them from the edge lists (Frieze and Grimmett, 1983). These techniques also contributed to the $\Theta(n^2)$ time per source required by Fredman's scheme. Instead

of purging edges, it is sufficient to mark vertices as being purged. Then each time an edge is examined, instead of examining a flag associated with the edge, it suffices to examine a flag associated with the destination of the edge. This simple change means that the $\Theta(n^2)$ purging time of Fredman can be reduced to an $O(n)$ time per source spent on purging.

The processing must then be necessarily different as the last $n/\log n$ vertices are labelled. It is no longer possible to regularly clean the heap, nor is it possible to have unlimited scanning, and so the heap will become increasingly dirty. The third important observation is that $n/\log n$ is functionally smaller than $n - n/\log n$ by a factor of $\log n$, and so to label these $n/\log n$ vertices and still retain an $O(n)$ bound on the total number of heap operations there are now $O(\log n)$ heap operations available per labelling.

From these three ideas the algorithm follows. The notation and names of variables are the same as those used in algorithm Spira-ssp.

```

procedure fast-ssp( s ) ;
begin
  S := {s} ; purged := {s} ; D[s] := 0 ;
  initialise the heap to (s,t), where t is the
  endpoint of the shortest edge from s ;
  for k := 1 to iloglog(n) do
  begin
    expand-soln( n - n/2k ) ;
    clean-up-heap ;
  end ;
  expand-soln( n ) ;
end {fast-ssp} ;

```

```

procedure expand-soln( stopat ) ;
begin
  while |S| < stopat do
    begin
      let (c,t) be at the root of the heap ;
      replace (c,t) by (c,t'), where t' is the endpoint
        of the next shortest edge from c such that t'
        is unpurged, and rearrange the heap ;
      if t is not in S then
        begin
          S := S + {t} ;
          D[t] := D[c]+C(c,t) ;
          add into the heap (t,u), where u is the
            endpoint of the shortest edge from t such
            that u is unpurged, and rearrange the
            heap ;
        end ;
      end ;
    end {expand-soln} ;

procedure clean-up-heap ;
begin
  purged := S ;
  for each element (c,t) in the heap do
    if t is in S then
      replace (c,t) by (c,t'), where t' is the endpoint
        of the next shortest edge from c such that
        t' is unpurged ;
    completely rebuild the heap ;
  end {clean-up-heap} ;

```

Algorithm 4.2 - The new algorithm.

A new procedure has been added: clean-up-heap performs the $\log \log n$ heap cleaning operations, checking the label of each candidate and then completely rebuilding the heap. The function "iloglog(n)" should return 0 for $n=1$, and $\text{ceiling}(\log \log n)$ otherwise. This definition means that for non-trivial values of n , $\log n \leq 2^{\text{iloglog}(n)} < 2 \log n$. Shortest paths may be recovered using the same technique as was described for Spira's algorithm.

The first $\log \log n$ "phases", each followed by a cleaning operation, are contained in the for loop of the main procedure; calls to expand-soln with a target that gets closer and closer to n alternate with calls to clean-up-heap.

The final phase, to label the last $n/\log n$ vertices, is carried out by the call to `expand-soln` after the termination of the main for loop.

The fourth and final observation that makes the algorithm simple and elegant, yet completely effective, is that having $O(\log n)$ heap operations available per labelling in the last phase is enough of an advantage that these vertices can be labelled by continuing with the normal processing. To handle the last phase differently, it is sufficient to process it in exactly the same manner as the preceding $\log \log n$ phases. That this is true will become clear in the analysis.

Correctness of the algorithm.

The correctness of the algorithm follows directly from the correctness of Spira's algorithm. In the edge scanning, only edges leading to labelled vertices will ever be skipped and thus not placed into the heap as candidates. Then all candidates skipped, at whatever point, be it during heap cleaning or normal processing, would have been discarded as useless by Spira's algorithm if taken from the heap. To make this idea clear, note that the heap manipulation routines of both `Spira-ssp` and `Fast-ssp` could be modified by the addition of a tie breaking rule to the effect that if two candidates have equal cost, for example the candidates from v_1 and v_2 , then the "smaller" candidate will be that of the smaller of v_1 and v_2 , using the index of the labelling vertex as a secondary key. If this were to be done then the sequence of edges used for labelling vertices would be identical for both algorithms, and the shortest path tree

generated by the new algorithm would be identical to the shortest path tree of Spira's algorithm. Without such a modification, the same path costs will be assigned by the two algorithms, but if there are paths of equal cost a different spanning tree may result. Inclusion of this tie breaking rule will not violate the previous requirement (section 4.1) that ties for the minimal cost candidate be resolved without reference to the destination of the candidate. Bloniarz (1983) discusses this point and gives a slightly different tie breaking rule that also allows the relaxation of the "random ordering of edges of equal cost" requirement.

Analysis of the algorithm.

The algorithm is analysed as a sequence of phases. During the first phase the size of S expands from 1 to $n/2$, during the second phase from $n/2$ to $3n/4$, and during the $\log \log n$ 'th phase the solution set will expand from $n - 2n/\log n$ entries to $n - n/\log n$. In the final $\log \log n + 1$ 'th phase the solution set grows to include all of the remaining $n/\log n$ vertices. That phase is somewhat different, and will be analysed separately. The dominant components in the running time are the number of update and insert operations on the heap (hops) and the total number of edges inspected during the course of the algorithm (scans). The heap is rebuilt only $\log \log n$ times, and time spent on the heap rebuildings is not dominant in the running time.

At the end of each of the first $\log \log n$ phases the heap will be cleaned. To find the good candidates with which the heap is rebuilt requires some scanning effort. At the end of the

k 'th phase there are $n/2^k$ unlabelled vertices, so that the probability of discovering an unlabelled vertex by inspecting one edge in a randomly ordered edge list is $1/2^k$, and expectedly 2^k edges need to be examined before a clean candidate is found (lemma 1.1). This must be done for at most n heap entries, so the scanning effort for the k 'th rebuilding is expectedly $2^k n$.

During the k 'th phase $n/2^k$ vertices will be added to the solution set. Each successful labelling of a vertex involves one addition to the heap; and to achieve each successful labelling there will be some number of update operations caused by useless candidates, the number depending on the probability that a candidate drawn from the heap leads to an unlabelled destination. But because of the heap cleaning, this probability is never less than $1/2$, so that during the whole phase the number of heap operations is expectedly less than $3n/2^k$.

Each of these heap operations has associated with it some edge scanning. During the k 'th phase there are $n/2^{k-1}$ "acceptably clean" non-purged vertices, so that the expected number of edges skipped for each heap operation is 2^{k-1} . There are $3n/2^k$ such operations, so the total edge scanning requirement caused by the k 'th phase is expectedly $3n/2$.

The total cost of the first $\log \log n$ phases is given by

$$\begin{aligned} \text{hops} &= \sum_{k=1, \log \log n} [3n/2^k] \\ &< 3n \\ \text{scans} &= \sum_{k=1, \log \log n} [2^k n + 3n/2] \\ &< 4n \log n + O(n \log \log n) \end{aligned} .$$

At the critical point of $n - n/\log n$ the regular heap cleaning operation is discontinued. Beyond this point the cost of the heap rebuilding would still be acceptable; it is the cost of the edge scanning to find clean candidates that becomes too high - a total scanning effort $\Theta(n^2)$ would be required. Instead a final heap cleaning operation is performed when $n - n/\log n$ vertices have been labelled, thereafter, while labelling the last $n/\log n$ vertices, the heap is permitted to become increasingly dirty, and compared with the earlier phases, asymptotically more heap operations per labelling are required .

During the whole of the last phase there are $n/\log n$ "acceptably clean" non-purged candidates; hence at any stage during the last phase a heap operation will require a corresponding scanning effort of expectedly $\log n$ edges. During the phase every element in the heap will have as its candidate one of these $n/\log n$ acceptably clean destinations, and no other vertices will appear as candidates; moreover, the randomness assumption means that each time the minimum cost candidate is drawn from the heap each of the $n/\log n$ possible destinations is equally likely.

This situation is once again a coupon collector problem. In the final phase of fast-ssp a successful event is when an unlabelled vertex is drawn at the root of the heap, and there are $p = n/\log n$ such equally likely events. The standard solution to the coupon collector problem for p equally likely events is that $p \cdot \ln(p)$ trials will expectedly be necessary before all events have occurred, and so, including the operations needed to insert candidates for newly labelled

vertices, the total number of heap operations in the final phase is given by

$$\begin{aligned} \text{hops} &= n/\log n + (n/\log n)\ln(n/\log n) \\ &< 2n \end{aligned}$$

This analysis justifies the earlier claim that although the last phase is different, the last $n/\log n$ vertices can be labelled by the same procedure as was used in the earlier phases, and the target of $O(\log n)$ heap operations per labelling during the last phase can be seen to be realistic. Because of this bound on heap operations, for the last phase

$$\text{scans} < 2n \log n \quad .$$

Counting all $\lceil \log \log n \rceil + 1$ phases,

$$\begin{aligned} \text{hops} &< 3n + 2n &= O(n) \\ \text{scans} &< 4n \log n + 2n \log n + O(n \log \log n) &= O(n \log n) \end{aligned}$$

and the expected running time for one source becomes

$$\begin{aligned} \text{time} &= \text{scans} * O(1) + \text{hops} * O(\log n) + \text{builds} * O(n) \\ &= O(n \log n) \quad . \end{aligned}$$

Multiplying by the number of sources that must be processed, and adding $O(n^2 \log n)$ time for the presort gives the main result of this chapter:

Theorem 4.2 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Then the apsp problem on N can be solved in $O(n^2 \log n)$ expected running time.

Proof. From the above discussion. []

4.5 Experimental Results.

The new algorithm can be implemented quite easily, with the description of the algorithm given above forming the core of such a program and requiring very little extension. To actually make a program for the apsp problem it was only necessary to add routines for the presort, and provide the heap manipulation routines that are called by the program Fast-ssp. Quicksort, widely accepted as being a fast sorting method, was used for the presort. As the results below show, the sorting takes less than 25% of the total running time. The heap routines were carefully coded, and the usual tricks (Knuth, 1973b) to improve the speed of these were employed. The main purpose of these experiments was to compare the new algorithm with that of Bloniarz, and since both use the same presort and heap routines, both derived equal benefit from the careful coding.

The Fast-ssp program was written in Pascal, compiled with a DEC Pascal compiler, and run with range checking disabled under VMS on a Digital Equipment VAX 11/785. The program for Bloniarz's method was then created by copying the source file, deleting the heap cleaning, and changing slightly the procedure for edge skipping. In all other respects the programs were identical.

Experiments were carried out on complete graphs, with the edge costs floating point values generated randomly and independently on the interval $[0,1)$. Each of the values below records the mean of 10 experiments.

<u>n</u>	<u>sort time</u>	<u>apsp time</u>	<u>apsp/n²logn</u>
45	0.31 sec	0.95 sec	8.5 *10 ⁻⁵
64	0.65	2.11	8.6
91	1.35	4.44	8.2
128	2.83	9.8	8.5
181	5.99	20.8	8.5

Table 4.1 - Running Time for Fast-ssp.

These numerical results for the running time can be seen to be in agreement with the analytic results - the near constancy of the running time when divided by $n^2 \log n$ is confirmation that the running time is $O(n^2 \log n)$.

For comparison, the running times for the algorithms of Bloniarz and Dijkstra were compared with the total running time for algorithm Fast-ssp:

<u>n</u>	<u>Dijkstra</u>	<u>Bloniarz</u>	<u>Fast-ssp</u>
45	1.06 sec	1.18 sec	1.26 sec
64	3.04	2.65	2.76
91	8.66	5.77	5.79
128	24.1	12.5	12.6
181	-	26.5	26.8

Table 4.2 - Running times for apsp algorithms.

The methods of Bloniarz and Dijkstra were chosen for the comparison as a result of earlier experiments (Moffat, 1983) in which the author discovered that for complete graphs Bloniarz's method ran faster than the algorithms of Floyd, Dijkstra, Spira, and Takaoka when $n > 100$, while Dijkstra's

method was fastest for smaller values of n . The running times of table 4.2 are illustrated in figure 4.5.

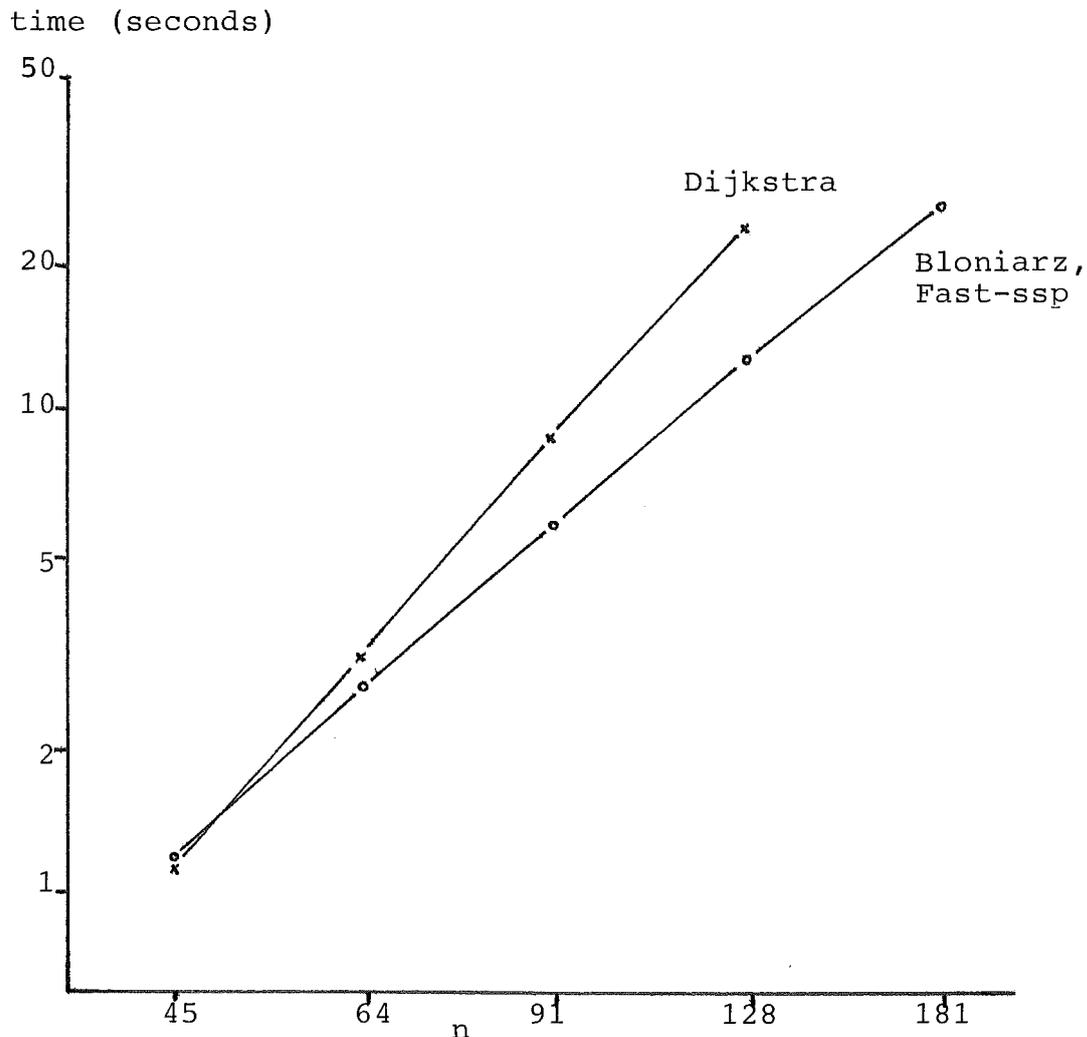


Figure 4.5 - Running times for apsp.

The graph and table 4.2 show that this set of experiments was unable to distinguish between the new algorithm and that of Bloniarz, with the slight difference between them well within the range of the experimental error. A more detailed set of experimental results is reported in Appendix 1; there experiments on a Prime 750 give the result that for values of n between 45 and 256 the new algorithm is about 8% faster than that of Bloniarz. Again the difference is within the experimental error. For the range of values of n tested in

these two sets of experiments it was not possible to detect a major difference in running time between the two methods.

The total numbers of comparisons for the two Spira type methods were also recorded. The numbers given do not include the comparisons used during the pre-sort. The units are thousands.

<u>n</u>	<u>Bloniarz</u>	<u>Fast-ssp</u>	<u>Fast-ssp/n²logn</u>
45	31	24	2.2
64	73	59	2.4
91	160	120	2.3
128	360	290	2.5
181	750	640	2.6

Table 4.3 - Comparisons for amsp problem.

The new algorithm is clearly superior to that of Bloniarz if the number of comparisons used is important, as expected. The number of comparisons has been decreased, but at the expense of more overhead during the heap cleaning stages.

That the running times of the two methods are very similar is no surprise - all of the values of n tested lie in the same zone of the discrete $\text{iloglog}(n)$ function and have 3 heap cleanings per source; in this range the step function \log^*n is also 3. Memory limitations meant that it was not possible to test larger values on n on either of the two test machines. Even at $n=256$ the VAX produced erratic and variable running times as a consequence of the virtual memory paging costs being included in the running time of the program, and these results were discarded. Thus, the only

observation made here about the practical usefulness of the new algorithm is that for tractable values of n it is dramatically superior to the algorithms of Dijkstra and Floyd and is not measurably inferior to the algorithm of Bloniarz. Moreover the asymptotically better bound means that, as the graphs to be processed become larger, Fast-ssp will become more and more efficient relative to the algorithm of Bloniarz. The new algorithm is not merely a theoretical curiosity. It is a practical and operationally useful algorithm.

CHAPTER FIVE

SOME IMPLEMENTATION NOTES.

5.1 Background.

This chapter is also concerned with the average time complexity of the apsp problem. Chapter 4 gave a detailed description of several fast algorithms; here some variations of those algorithms are investigated. Two of the modifications examined are applicable to any of the "Spira class" algorithms discussed in chapter 4, and to any implementer of these algorithms the changes discussed will be quite tempting and seemingly beneficial. However, the results concerning these alterations show quite clearly that the converse is in fact true - that the "new" algorithms created are asymptotically inefficient relative to the original schemes. The third mutation considered applies only to the algorithm Fast-ssp, and has the effect of reducing the worst case performance of the method from $O(n^3 \log n)$, a bound shared by all of the Spira class of algorithms, to a much more respectable $O(n^3)$ bound. This third mutation will be useful in a critical situation where fast performance is wanted, but the risk of $O(n^3 \log n)$ behaviour is unacceptable.

The final section of this chapter discusses worst case running times in more detail by considering the actions of the various Spira class algorithms on the "bad" graph that was first introduced in section 3.4.

5.2 The Effect of Unlimited Scanning.

Bloniarz (1980) and Takaoka and Moffat (1980) improved Spira's algorithm asymptotically by observing (section 4.3) that when replacing a candidate in the heap, the "shadow" of the heap operation is sufficiently long that a sequence of edges can be safely examined in an attempt to find a clean candidate. Thus, each time a candidate t' is sought from some vertex c , Bloniarz scans as many as $\log n$ edges, trying to find an edge that leads to a vertex that is not already labelled. The code fragment describing this scanning was given in section 4.3.

Here a more dangerous code fragment, repeated below, is considered. Instead of stopping the scanning when $\log n$ edges have been examined, which ensures that the scanning effort remains inside the $O(\log n)$ shadow of the heap operation but possibly chooses a dirty candidate, edges are examined until a "good" candidate is found. This increases the general cleanliness of the heap and reduces the number of heap operations and comparisons, but does so at the expense of increased scanning effort.

```

let (c,t) be at the root of the heap ;
let t' be the endpoint of the next shortest edge from c
  such that t' is not in S ;
replace (c,t) by (c,t'), and rearrange the heap ;

```

Algorithm 5.1 - Unlimited Scanning.

This code fragment is considered in the context of Spira's algorithm. The intention of this section is to show that with this scanning strategy the running time of algorithm Spira-ssp will become $\Omega(n^{1.5})$ for each source, asymptotically

worse even than Spira's simple "take the next edge" regime.

Assume throughout what follows that algorithm Spira-ssp has been implemented with unlimited scanning, and that the network being processed is endpoint independent. Let the k 'th stage be the processing that is involved from the time that $|S|$ becomes k to the time when an unlabelled candidate appears at the root of the heap and $|S|$ changes to $k+1$. The last, or $n-1$ 'th, stage will be of particular interest in the analysis. This includes all of the processing involved in the attempt to label the last remaining vertex. Throughout this last stage there is a single unlabelled vertex; let this be vertex v . Define any heap entry to be valid if v is the candidate of the entry and, were the entry to become that of minimum weight, v would be labelled and the algorithm terminated.

Suppose that t is the vertex labelled at the end of the second to last phase. The first action in the last phase is that the candidate (c,t) that labelled t is replaced; then an onward candidate for vertex t will be added into the heap. Because of the unlimited scanning, both of these two new heap entries will have v as their candidate, and so there will always be at least two valid candidates when the algorithm terminates. Thereafter, each time the root of the heap is drawn and is not valid, the unsuccessful entry is replaced by a valid candidate and the processing continued. By the conclusion of the last stage there will be some number of valid candidates, rf, consisting of two parts - candidates that were valid before the last stage, rb, and some number rd of candidates that were made valid during the last stage:

$rf = rb + rd$. The following lemmas show that rd and rb are sufficiently inter-related that regardless of the initial value of rb the final expected value of rf will be approximately $c \cdot \sqrt{n}$ for some constant c . Note that rb can only take on values $0 \leq rb \leq n-3$, since rf cannot exceed $n-1$ and rd is at least 2.

Lemma 5.1 For i in the range $0 \leq i \leq n-3$ the minimal value of the conditional expectation of rf , given that $rb=i$, occurs when $i=0$. That is,

$$\min_{i=0, n-3} E(rf|rb=i) = E(rf|rb=0)$$

Proof. Since rd is at least 2 and rf can be no more than $n-1$,

$$E(rf|rb=n-3) = n-1 .$$

That is, the labelling loop will execute once, with a valid candidate guaranteed to be the heap element of minimum cost. If rb is initially $n-4$ then there are two possibilities: either a valid candidate will be removed from the heap at the first trial and the stage will end with $rf=n-2$; or a non-valid candidate will be processed first, rd will increase by 1, and the labelling loop will be executed again. The first alternative will occur with probability $(n-2)/(n-1)$, and the second with probability $1/(n-1)$, since only 1 of the $n-1$ candidates are non-valid, and the graph is endpoint independent. In the latter case, when the loop is executed again, there will be $n-1$ valid candidates present, exactly as if rb had initially been $n-3$, so that

$$E(rf|rb=n-4) = [(n-2)/(n-1)]*(n-2) + [1/(n-1)]*E(rf|rb=n-3)$$

Using a similar argument

$$E(rf|rb=i) = \text{Pr}[\text{success at first iteration}]*(i+2) + \text{Pr}[\text{failure at first iteration}]*E(rf|rb=i+1)$$

where

$$\begin{aligned} \text{Pr}[\text{success at first iteration}] &= (i+2)/(n-1) \\ \text{Pr}[\text{failure at first iteration}] &= (n-i-3)/(n-1) \end{aligned}$$

To show that a minimal value of $E(rf|rb=i)$ occurs at $i=0$, it is sufficient to show that

$$E(rf|rb=i) \leq E(rf|rb=i+1) \quad \text{for } 0 \leq i \leq n-4 \text{ .}$$

Assume the converse, so that for some $0 \leq i \leq n-4$,

$$E(rf|rb=i) > E(rf|rb=i+1) \text{ .}$$

Then

$$\begin{aligned} [(i+2)/(n-1)]*(i+2) + [(n-i-3)/(n-1)]*E(rf|rb=i+1) \\ > E(rf|rb=i+1) \end{aligned}$$

so

$$\begin{aligned} (i+2)^2 &> E(rf|rb=i+1) * [(n-1)-(n-i-3)] \\ (i+2)^2 &> E(rf|rb=i+1) * (i+2) \\ i+2 &> E(rf|rb=i+1) \text{ .} \end{aligned}$$

But, for all i in the range $0 \leq i \leq n-4$, because $rd \geq 2$,

$$E(rf|rb=i+1) \geq i+3$$

establishing the contradiction and proving the lemma. []

Lemma 5.2 $E(rf|rb=0)$ is approximately $c \cdot \sqrt{n}$, for some constant c .

Proof. The proof of this lemma is due to T. Takaoka. The proof is given in appendix 2. []

Given these two lemmas it is straightforward to establish the first target:

Lemma 5.3 $E(rf) \geq c \cdot \sqrt{n}$ for some constant c .

Proof. To find an expected value for rf , use the theorem of total probability

$$\begin{aligned}
 E(rf) &= \sum_{i=0, n-3} E(rf|rb=i) * Pr[rb=i] \\
 &\geq \min_{i=0, n-3} E(rf|rb=i) \\
 &\geq E(rf|rb=0) && \text{(lemma 5.1)} \\
 &\geq c \cdot \sqrt{n} \quad \text{for some constant } c && \text{(lemma 5.2) []}
 \end{aligned}$$

The destinations in the sorted edge lists form a random permutation of the set of vertices, meaning that v , the last vertex labelled, lies expectedly in the middle of each edge list. Endpoint independence implies that the valid candidates represented by rd (those made valid during the phase) become valid independently of the position of v in the unscanned portion of the edge lists, and so the position of vertex v in these lists is expectedly the midpoint of the unscanned portion. This is not the case for the candidates represented by rb (those valid before the last phase), as when they were established as candidates more than one vertex remained unlabelled.

The scanning strategy will thus expectedly move rd of the edge lists pointers beyond the mid-way position, performing $O(1)$ work for each intervening edge on the list that was

skipped, and bringing the total number of edges inspected from such a list during the course of the algorithm to at least $n/2$. The total computing time required for one source is thus $\Omega(n*rd)$. To show that this component will dominate the running time of the algorithm it is sufficient to show that rd is some fixed fraction of rf .

Lemma 5.4 Suppose that rf and rd are the expected values respectively of the number of final valid candidates, and the number of these that were established during the last phase, as discussed above. Then $rd > rf/2$.

Proof. Consider the action of the algorithm on a complete network $N=(G,C)$ drawn from an endpoint independent probability measure. The vertex v labelled at the end of the last phase will have expectedly $rf = c*\sqrt{n}$ valid candidates associated with it at the conclusion of the algorithm. Create from N a new network $N'=(G',C)$, where $G'=(V',E')$ is created by setting $V'=V-\{v\}$ and then taking the induced subgraph. Then N' is a complete network on $n-1$ vertices, and again is endpoint independent. If the mutant algorithm were to be applied to N' , the last vertex labelled, v' say, will expectedly be valid for $c*\sqrt{n-1}$ candidates by the conclusion of the processing on N' . But in the processing on N , v' is only the second to last vertex labelled, and symmetry demands that of the $c*\sqrt{n-1}$ valid candidates for v' in N' , $(1/2)*c*\sqrt{n-1}$ of them are valid for v in N , since in expectedly half of the edge lists of N vertex v will precede vertex v' . Moreover, v' in N' will be labelled after exactly the same number of steps as v' in N , since v does not become the minimum cost candidate at any time before the $n-1$ 'th stage of the processing on N . Thus, at the point in

the processing on N where v' is labelled, there are expectedly $(1/2)c*\sqrt{n-1}$ valid candidates for v . But this quantity is exactly the description of rb , meaning that rd is expectedly greater than $(1/2)c*\sqrt{n}$. []

Lemmas 5.3 and 5.4 lead directly to the following theorem:

Theorem 5.5 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Suppose that Spira's apsp algorithm is altered by the addition of unlimited scanning. Then the expected running time of the resulting algorithm on N is $\Omega(n^{2.5})$.

Proof. From the above discussion and lemmas 5.3 and 5.4. []

Experimental verification.

The algorithm with the unlimited scanning mutation was implemented in Pascal on a Prime 750 computer. Measurements were made of the total number of valid candidates (rf), the number of these that were established during the last phase (rd), and the total number of edges examined during the course of the computation ($scans$). The results given are the average of 20 experiments for the smaller values of n , and the average of 16 experiments for $n=80$, where each experiment consists of solving n single source problems. The edge costs were assigned by independent drawings from a uniform distribution.

<u>n</u>	<u>rf</u>	<u>rd</u>	<u>rd/rf</u>	<u>scans</u>	<u>rf/n^{1.5}</u>	<u>scans/n^{2.5}</u>
10	58	33	0.57	417	1.8	1.3
20	191	95	0.50	2950	2.1	1.6
40	588	315	0.54	18600	2.3	1.8
80	1925	913	0.47	118500	2.7	2.1

Table 5.1 - Unlimited Scanning.

From table 5.1 it can be seen that the experimental ratio of rd to rf is about 1/2, as expected; and the non-decreasing values in the last two columns serve as indications that the growth rate of rf is $\Omega(n^{1.5})$ - that is, $\Omega(\text{sqrtn})$ per source - and that the total number of edges examined is $\Omega(n^{2.5})$, in agreement with the analysis.

In terms of running time, the method is still fast on small sized problems - the "mutation" does not get the chance to become "cancerous". But for larger sized problems, the running time begins to grow. The values in the next table resulted from experiments on a Vax 11/785 computer, with the programs written in Pascal. Again, a uniform distribution was used. The results are the mean of 10 experiments; times are in seconds and do not include the time required for the pre-sort.

<u>n</u>	<u>Bloniarz time</u>	<u>Unlimited time</u>	<u>/$n^{2.5}$</u>
45	0.87 sec	1.04 sec	$7.7 * 10^{-5}$
64	2.00	2.41	7.4
91	4.42	5.81	7.4
128	9.64	14.2	7.7

Table 5.2 - Running Times for Unlimited Scanning.

From this table it can be seen that for all values of n the scheme of Bloniarz runs faster. The approximate constancy of the final column is consistent with the predicted bound of $\Omega(n^{2.5})$; it is possible that the running time of the mutant is $\Theta(n^{2.5})$.

The conclusion to be drawn from this section is that to have a good expected time asps algorithm some form of limited scanning strategy is essential.

5.3 The Effect of Cost-compression.

The mutation considered in this section is superficially even more attractive than that of limited scanning, and it seems likely that many implementations have included this strategy, as it is easier to include in a program than it is to exclude. The idea is very simple and relates to the fact that a single source algorithm will be used n times when finding a solution for the a-psp problem. Each time a single source problem is solved, the shortest paths from that source are available. Surely, it might be suggested, when solving for subsequent sources in the same graph it will be of advantage to use the shortest paths found so far instead of the original costs.

In the context of the Spira class of algorithms this strategy will be called "cost-compression" - once the shortest path costs from a source have been found they are used in all subsequent calculation, and the original edge costs from solved sources are ignored. The use of these costs does not entail a second sorting step, as the shortest paths are found by the algorithm in order of increasing cost. Here it is shown that this strategy can be of no positive benefit, and is far more likely to be harmful. Although Spira's algorithm will be used in the following development the same logic will apply to any algorithm in the class. Spira's algorithm is referred to as SP, and the mutant derived from it by adding cost-compression as SPCOST.

In the processing for one source, let edge (c,t) be said to have been completely processed if (c,t) has at some point entered the heap as a candidate and, before the termination

of the algorithm, been removed from the heap as the minimum cost candidate.

Lemma 5.6 Let algorithm SP be used to find the shortest paths from source s , and let v be the last vertex labelled. Then every edge (c,t) such that $C(c,t) < L(s,v) - L(s,c)$ will have been completely processed.

Proof. In Spira's algorithm there is no skipping, so every edge scanned becomes a candidate. Any candidate (c,t) for which $L(s,c) + C(c,t) < L(s,v)$ will be removed from the heap as the minimum cost candidate before the termination of the algorithm, since the weight of candidate (c,t) is less than the weight of the candidate that labels vertex v . Moreover, any edge (c,t) for which $L(s,c) + C(c,t) < L(s,v)$ will be scanned and become a candidate, since inductively its predecessor in the edge list for c will have been completely processed and replaced in the heap by candidate (c,t) . The first entry in every edge list will always be scanned and inserted as a candidate, so the base of the induction is established. []

While solving for source s , for each edge list there is some fixed value $L(s,v) - L(s,c)$, dependent on the shortest path costs but not the individual edge costs, that determines whether or not an edge will be completely processed. That is, the running time of the algorithm depends largely upon the number of edges completely processed, and the number of edges completely processed depends on a value along the ordered edge list, not a position.

For any network, SPCOST must yield the same shortest path costs as SP. Moreover, SPCOST never increases edge costs,

only decreases them. Thus cost compression cannot decrease the number of edges satisfying the conditions of lemma 5.6, and so the processing effort of algorithm SPCOST will be at least as great as the processing effort of algorithm SP. In fact the effort required is likely to increase. The cost-compression introduces spurious edges of low cost that serve as distractions to the processing - any vertex that could be labelled by a compressed edge can be labelled by a "true" edge, and the "true" edges that are actually useful for labelling purposes are all direct connections and will consequently never have their cost decreased. The other edges that do have their cost reduced will move toward the head of their edge list, meaning that the useful edges will drift by default toward the tail of the lists, and an increased amount of scanning effort will be necessary to find them. The point of lemma 5.6 is that the "true" edges, of invariant cost under the cost-compression, must still be processed before the algorithm can terminate, and so the spurious edges only add to the processing required.

When applied to one of the algorithms that employs limited scanning the potential for the cost-compression to create extra work is even greater. This can be seen in the following example, in which it is assumed that edge (a,b) was originally of high cost, but has been compressed, so that when solving from source s the cost function is such that $C(a,b) = C(a,u) + C(u,b)$:

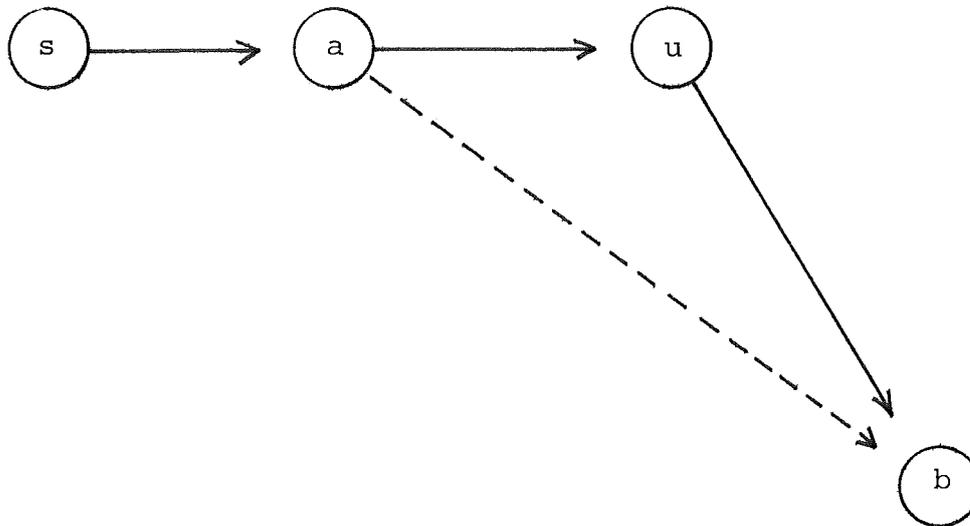


Figure 5.1 - A compressed edge.

Suppose that algorithms B and BCOST represent some limited scanning Spira type algorithm with and without cost-compression. If, solving from s , b is labelled from u by B, then, so long as $C(a,b)$ is beyond the "barrier" value $L(s,v) - L(s,a)$, edge (a,b) will not be scanned. But because of the limited scanning, even if edge (a,b) is inside the barrier value and is scanned by B, it is unlikely to cause a heap operation, for it is probable that it will be skipped over. But in BCOST the cost-compression means that not only will the edge be scanned, but that a heap operation will take place to establish it as a candidate, since now it is not possible for b to be labelled before (a,b) is scanned. The cost compression has reduced the cost of (a,b) so that now there is no chance of the heap operation being avoided, and the whole purpose of the limited scanning is jeopardised.

Quantitative analysis of this effect would be rather difficult, as the increased running time will build as sources are processed, and depends on many factors such as the arc-length of shortest paths, the distribution of edge costs, the distribution of shortest path costs, and so on. Thus, a precise analysis is beyond the scope of this discussion.

Lemma 5.7 The use of cost-compression with a Spira class algorithm will not decrease the running time of the algorithm; rather, the running time is likely to increase. Proof. From the above discussion. []

Experimental verification.

Cost-compression was implemented on top of the standard program (section 4.5) for Bloniarz's algorithm, and the effect of the strategy on the running time was measured. The table below gives the time required by the two strategies in seconds, measured on a VAX 11/785 running Pascal under VMS. The average of 10 experiments were taken; graphs were generated with each edge drawn independently from a uniform distribution. The times listed do not include the time required by the presort, which is identical for the two variants.

<u>n</u>	<u>without</u>	<u>with compression</u>	<u>ratio</u>
45	0.87 sec	1.70 sec	1.95
64	2.00	4.75	2.38
91	4.42	12.18	2.76
128	9.64	34.7	3.59

Table 5.3 - Effect of cost-compression.

As can be seen from the running times, the strategy should not be used under any circumstances - the running time grows very rapidly relative to that of the standard non-compression method. These results give empirical support to the claim made by lemma 5.7. The effect of the cost compression is cumulative, and is most pronounced as the last few ssp problems are being solved. So, although the running times listed are much less than the time required in the worst case (section 5.5), it is conjectured that the running time of the last ssp solved is $\Theta(n^2 \log n)$, the worst case for the algorithm, meaning that as the graph size grows the handicap would become asymptotically large.

The performance of this mutation and the unlimited scanning mutation described in section 5.1 are also shown in the following graph, to highlight the claim that these mutations should be avoided at all costs.

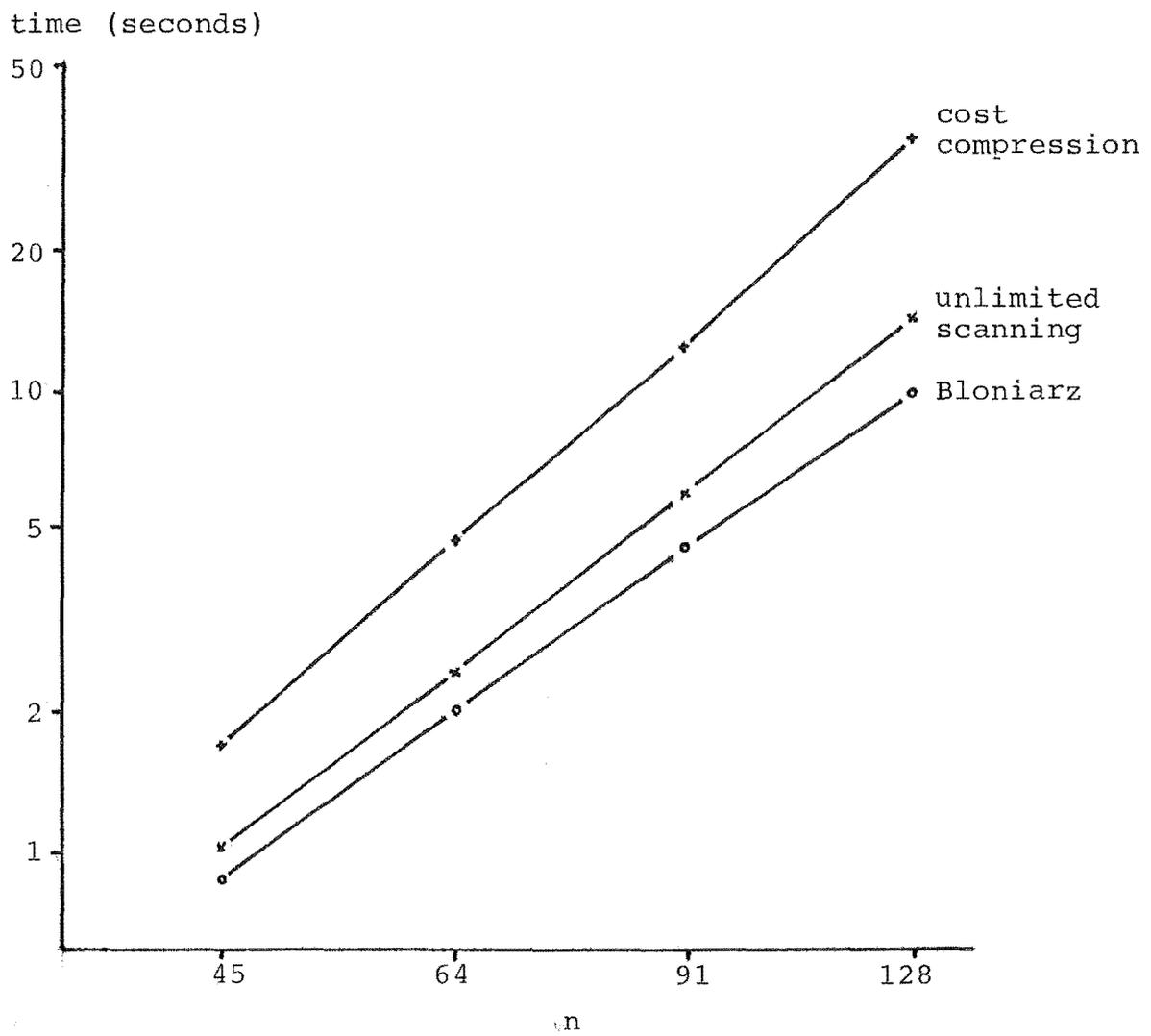


Figure 5.2 - Running time of mutations.

5.4 Implementing Continuous Cleaning.

Algorithm Fast-ssp of section 4.4 may, in the worst case, require $O(n^3 \log n)$ running time. For each single source problem each edge in the graph might be processed through the heap, and each heap operation requires an overhead of $O(\log n)$ time. The inferior worst case bound of these heap based methods was pointed out originally by Spira when presenting his fast algorithm. To bring the worst case running time back to $O(n^3)$ he suggested running Dijkstra's algorithm in parallel with an implementation of his algorithm. In this scheme the two algorithms are executed alternately, one step, or one cpu-second, at a time. Because the average time of Spira's method is $O(n^2 \log^2 n)$ and the worst case time of Dijkstra's method is $O(n^3)$, and because such a composite algorithm can be halted as soon as either of the components terminates, the average running time of the scheme becomes $O(n^2 \log^2 n)$ and the worst case becomes $O(n^3)$. Naturally, however, such a composite will require twice the running time of Spira's method on an "average" graph, and twice the running time of Dijkstra's algorithm on a "bad" graph. Thus the scheme, whilst avoiding the $O(n^3 \log n)$ worst case bound, is neither particularly elegant nor very practical, and Spira's method is usually described as having an $O(n^3 \log n)$ worst case analysis. Section 5.5 below shows shows how large the worst case running time can become.

The algorithms of Takaoka and Moffat and Bloniarz inherited the worst case bound from Spira; and the Fast-ssp algorithm presented in section 4.4 does not escape the $O(n^3 \log n)$ worst case bound either. However in this section it is shown that

by making a slight change to Fast-ssp the worst case bound can be cut to $O(n^3)$ without resorting to clumsy composition, and without adding a large overhead. This then equals the worst case bounds of the classical algorithms of Floyd, Dantzig, and Dijkstra.

Consider one of the heap cleaning stages. The objective in cleaning the heap is to replace all dirty candidates by clean candidates, so that the probability of drawing a clean candidate at the root of the heap remains high. The cleaning operates intermittently - when $|S|=n/2, 3n/4$ and so on. The reason the cleaning is organised this way is to take advantage of the speed of heap building as compared with heap maintenance - an n element heap can be completely rebuilt in time $O(n)$, or $O(1)$ time per changed element, but if a change is made to the root of the heap and the heap structure must be maintained, $O(\log n)$ time is required. Thus, to obtain a good bound on the time spent replacing dirty candidates, all such replacements were grouped into the $\log \log n$ cleaning phases to make rebuilding of the entire heap, rather than individual replacement, economical. However, this "batch" mode processing is not strictly necessary.

The key observation, pointed out by T. Takaoka, is that if a dirty candidate is replaced by a clean candidate, the weight of the heap element will always increase, and can never decrease.

Lemma 5.8 In a heap of n elements, let one of the elements be chosen at random and its weight increased. Then the manipulations required to re-establish the heap property will expectedly require $O(1)$ time.

at level 1, two elements are at level 2 and so on. The $\log n$ lists "q[i]" store, for each of the $\text{floor}(\log n) + 1$ heap levels, the indices of entries in that level that need rearrangement in relation to their sons. The running time of this algorithm is described by the following lemma.

Lemma 5.9 In a heap of n elements, let the weights of any k element subset be increased. Then the heap property can be re-established requiring $O(n)$ time in the worst case, and, if the heap positions of $k-1$ of the elements are random within the heap and the k 'th element is the root, $O(k + \log n)$ time on average .

Proof. The worst thing that can happen in procedure mass-update is that every heap position is rebuilt downwards, taking the same $O(n)$ time as the standard heap building algorithm. Provided that the elements that have had their weights updated are processed in order from the bottom of the heap to the root, none of the elements processed early can swap other elements into the expensive positions near the root of the heap. This is why the procedure first distributes the elements by heap level. Doing this rather than simply scanning the heap means that the average time for updating a set of $k-1$ elements will be retained at $O(k)$ (lemma 5.8); the $\log n$ term comes from the initialisation step and that fact that one of the elements updated is assumed to be the root.
[]

The application of the ideas of these two lemmas is as follows. Rather than having batch cleaning of the heap at specific times, suppose that the heap is continually kept in a state of 100% cleanliness, and, to travel full circle,

Dantzig's requirement that every candidate lie outside the current solution set is returned to. Then each time an element t is labelled, all vertices v that have t as their candidate, of which there may be many, must be identified so that their candidate can be revised. Each labelling will thus require rather more than the replacement of a single item in the heap; it will involve a mass update. These concepts lead to a new variant of procedure `expand-soln`:

```

procedure cc-expand-soln( stopat ) ;
  begin
    while |S| < stopat do
      begin
        let (c,t) be at the root of the heap ;
        {heap is 100% clean, so t is unlabelled}
        S := S + {t} ;
        D[t] := D[c]+C(c,t) ;
        add into the heap (t,u), where u is the endpoint
          of the shortest edge from t such that u is not
          in S, and rearrange the heap ;
        updatelist := emptylist ;
        for each v such that t is the candidate of v do
          begin
            replace (v,t) by (v,t'), where t' is the
              endpoint of the next shortest edge from v
              such that t' is not in S ;
            append (v) to updatelist ;
          end ;
        mass-update( updatelist ) ;
      end ;
    end {cc-expand-soln} ;
  end ;

```

Algorithm 5.3 - Labelling with continuous cleaning.

Again, a little more information must now be stored - for each vertex t it is necessary to maintain a list of vertices that have t as a candidate; and each time a candidate is placed into the heap the candidacy should be recorded on the appropriate list. Note that the scanning is now of the unlimited variety - edges are examined until a candidate outside the current solution set is found.

The concept of the critical point is still very important, and this "on the fly cleaning" is only continued while $|S| < n - n/\log n$. In the final phase the heap is again allowed to become dirty, and restricted scanning is used. The following program describes a revised version of Fast-ssp; the correctness of this algorithm is immediate from the correctness of Fast-ssp.

```

procedure cc-fast-ssp( s ) ;
begin
  S := {s} ; D[s] := 0 ;
  initialise the heap to (s,t), where t is the endpoint
    of the shortest edge from s ;
  cc-expand-soln( n - n/log n ) ;
  purged := S ;
  expand-soln( n ) ; {this procedure in section 4.4}
end {cc-fast-ssp} ;

```

Algorithm 5.4 - Fast-ssp with continuous cleaning.

Average case analysis.

Before considering the worst case running time, it is necessary to show that the expected running time is still $O(n^2 \log n)$ for an endpoint independent network. There are now only two phases. The final phase is identical to that considered before, and requires no further consideration - when it starts the heap is clean and when it finishes all vertices are labelled.

The procedure cc-expand-soln includes all of the $\log \log n$ early phases of Fast-ssp. The total scanning effort must still be $O(n \log n)$ per source, as when cc-expand-soln terminates the heap and edge lists are in exactly the same configuration as at the end of the $\log \log n$ 'th phase of the original Fast-ssp. Thus, the only point of concern is the

effort spent in the heap.

When $|S|=j$ there are $n-j$ unlabelled vertices and j entries in the heap, and so, because of the endpoint independence, expectedly $j/(n-j)$ of the heap entries will be pointing at the vertex just labelled and must be replaced by the mass update operation. The following lemma shows that these $j/(n-j)$ entries form a random subset of the elements in the heap.

Lemma 5.10 Suppose that at stage j of the processing of cc-Fast-ssp on an endpoint independent network all heap candidates (v,t) , for some fixed t , are marked. Then each position in the heap is equally likely to be so marked.

Proof. The candidates in the heap can, because of the independence of cost and destination forced by the randomness assumption, and because the heap is 100% clean, be taken to be a collection of j independent samples drawn from a variable that is equally likely to take on the value of any of the $(n-j)$ unlabelled vertices. None of the heap operations can affect this independence, since the destination of any candidate is independent of the cost of the candidate, and the criteria used for permuting heap positions are cost, and, should a tie breaker be needed, source. Thus, when candidates (v,t) are marked, each position in the heap, even after some heap operations, will still be pointing to each unlabelled destination vertex with equal probability, and is equally likely to be marked. []

Effectively this lemma states that if the network is such that the root of the heap is a random candidate, then every position in the heap will be random. Then application of

lemmas 5.9 and 5.10 shows that the average effort required to replace the dirty candidates at each labelling is $O(j/(n-j) + \log n)$. Throughout the phase $j < n - n/\log n < n$, so that $j/(n-j) < \log n$. Thus, each call to mass-update will require $O(\log n)$ time, and the total expected time required over the $n - n/\log n$ iterations will be $O(n \log n)$. This falls within the limit of $O(n \log n)$ established previously, and so the running time of cc-Fast-ssp on an endpoint independent graph is still expectedly $O(n \log n)$ per source.

Worst case analysis.

In the worst case, every edge must be examined at some stage, so the scanning effort can be as high as $O(n^2)$ per source. In the long first phase, the effort spent in the heap at each stage is bounded above by $O(n)$ (lemma 5.9), and so the total heap effort with the continuous cleaning through this stage is $O(n^2)$.

In the final phase each edge list contains only $n/\log n$ entries that have not been purged. There are n edge lists, meaning that there are $n^2/\log n$ unpurged edges that might be processed through the heap. Purged edges will still always be skipped. Thus, even if every unpurged edge causes a heap operation, there will be no more than $n^2/\log n$ heap operations during the last phase, and allowing $O(\log n)$ at most for each heap operation, the worst case running time of the last phase is $O(n^2)$.

Since the scanning for each ssp is bounded by $O(n^2)$, and the effort in the heap for both phases is bounded by $O(n^2)$, algorithm cc-Fast-ssp provides the proof of the following:

Theorem 5.11 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Then the a-psp problem on N can be solved by a single algorithm with expected running time $O(n^2 \log n)$ and worst case running time $O(n^3)$.

Proof. From the above discussion. []

Experimental results.

In the implementation of cc-Fast-ssp there is some overhead caused by the recording of candidates and heap positions, and the more complex heap rebuilding procedures. To measure the extent of the overhead algorithm cc-Fast-ssp was compared with Fast-ssp of chapter 4.5. The experimental technique is identical to that discussed there. The running times given do not include the allowance for the presort.

<u>n</u>	<u>Fast-ssp</u>	<u>cc-Fast-ssp</u>
45	0.95 sec	1.08 sec
64	2.11	2.32
91	4.44	5.23
128	9.76	11.1

Table 5.4 - Average Running time of cc-Fast-ssp.

From these the handicap can be seen to be about 15%, which might, in some critical application, be worth paying for peace of mind.

5.5 Experiments on "Worst Case" Graphs.

In section 3.4 a generic cost function for a "bad" graph was described, and it was claimed that it forced the worst case running time for the Spira class algorithms. Here that claim is proved, and experimental evidence is given to show how extremely bad the running time can become.

The cost function given was $C(i,j) = (j-i) \bmod n$. An example distance matrix when $n=5$ is shown in section 3.4. With this function the distance matrix closure is invariant, so that $L(i,j) = (j-i) \bmod n$. Consider the following lemma as it applies to this graph:

Lemma 5.12 Suppose that network $N=(G,C)$ is such that

$$C(u,v) > 0 \text{ for all } u,v \text{ in } V$$

$$C(u,v) \neq C(u,w) \text{ for all edges } (u,v) \text{ and } (u,w).$$

Then for each edge (c,t) such that

$$L(s,c)+C(c,t) = L(s,t) \quad ,$$

any Spira class algorithm will place (c,t) into the heap as a candidate.

Proof. At the time when edge (c,t) is considered in the edge list and a decision is made whether to skip it or to use it, vertex t cannot be labelled, since the value of the current root of the heap must be less than $L(s,t)$ by the assumptions on the cost function - either c is freshly labelled and the weight of the root is $L(s,c)$, which is less than $L(s,t)$; or some lesser cost candidate (c,u) from c is being replaced, and in this case $C(c,u) < C(c,t)$. Thus any scanning strategy based on the label of the destination of an edge will scan

(c,t) while t is still unlabelled; and so (c,t) will be accepted as useful and placed into the heap as the candidate for vertex c . []

For each single source problem on a graph with this cost function there are $(1/2)(n-1)(n-2)$ edges that satisfy the requirements of the lemma. Thus,

Theorem 5.13 Let $N=(G,C)$ be a directed network of n vertices, with the cost function given by $C(i,j) = (j-i) \bmod n$. Then any Spira class algorithm solving the apsp problem on N will require $(1/2)n^3$ changes to the weights of heap elements, meaning that the implementations of Spira and Bloniarz will both require $\Theta(n^3 \log n)$ running time.

Proof. From the discussion above and lemma 5.12. []

Note that the continuous cleaning variant of algorithm Fast-ssp also obeys this theorem. However in that implementation most of the candidates are installed into the heap with $O(1)$ effort rather than the $O(\log n)$ heap rearrangement required by implementations of Spira and Bloniarz.

Experimental results.

A number of algorithms were run on networks that were generated with the cost function $C(i,j) = (j-i) \bmod n$. The running times caused by this cost function were very large, and the figures given below are typically the results of only one or two experiments - the trends were clear and repetition unnecessary. For the Spira class algorithms these times include the time required for the pre-sorting of the edges.

<u>n</u>	<u>Dijkstra</u>	<u>Bloniarz</u>	<u>cc-Fast-ssp</u>
45	1.1 sec	8.4 sec	6.0 sec
64	3.0	24	16.6
91	8.7	75	51
128	24.1	230	150

Table 5.5 - Worst case running times.

These figures are also shown in the graph of figure 5.3. The dramatic difference between the average case and the worst case can be clearly seen. The worst case of cc-Fast-ssp is significantly faster than the worst case of Bloniarz's algorithm, but both are well beaten by Dijkstra's algorithm. If good worst case performance is critical, then Dijkstra's algorithm should be used.

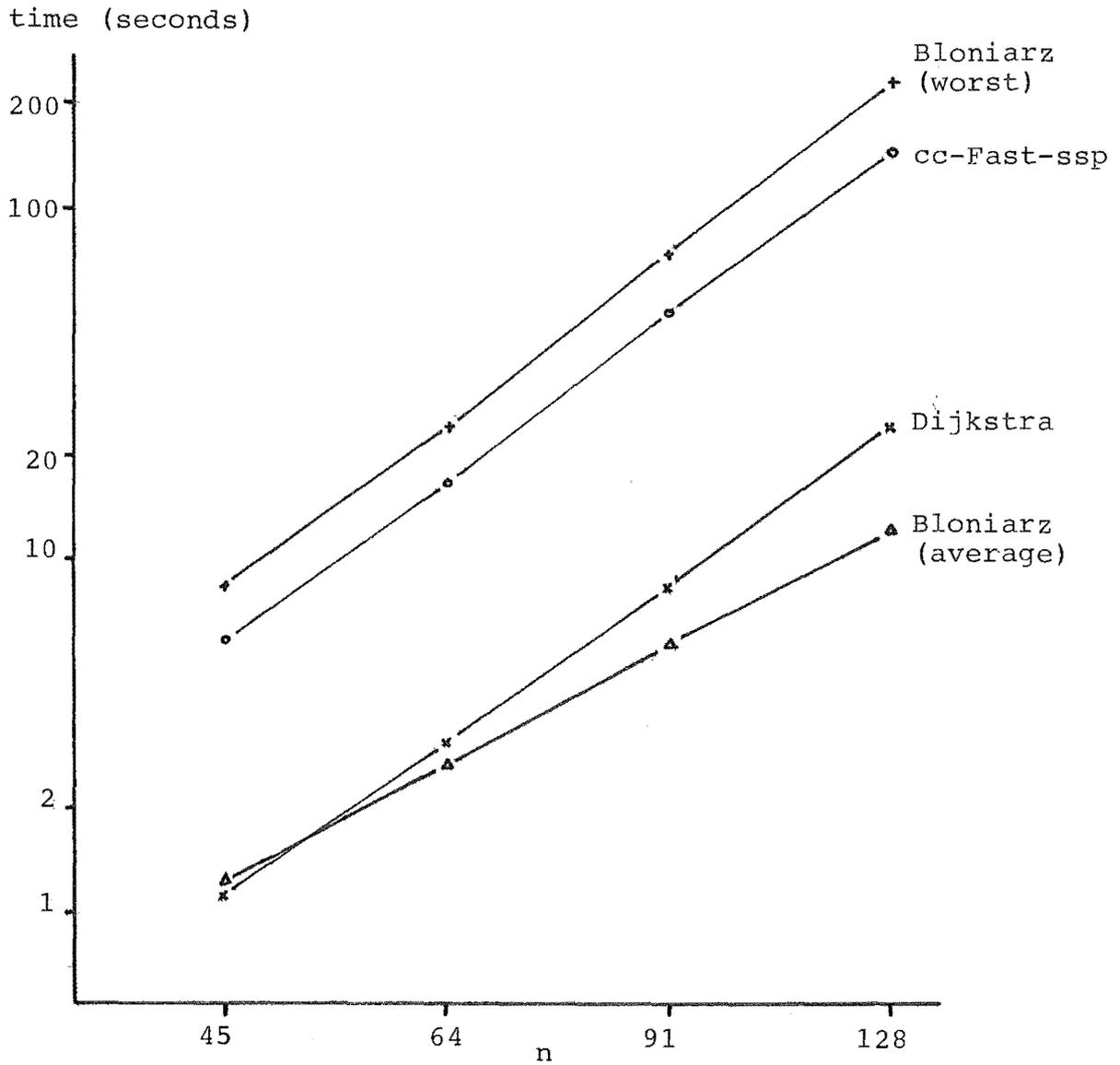


Figure 5.3 - Worst case running times.

CHAPTER SIX
DISTANCE MATRIX MULTIPLICATION.

6.1 Background.

The distance matrix multiplication problem (dmm) is closely linked to the problem of all pairs shortest path calculation, and most algorithms for one of these problems can be applied to the other problem after some slight modification. For example, Floyd's algorithm for the apsp problem is very similar to the simple "row on column" $O(n^3)$ distance matrix multiplication algorithm. In a similar fashion, a distance matrix multiplication algorithm can be constructed from the fast apsp algorithm of chapter four, establishing a new best upper bound for endpoint independent distance matrices. In section 6.2 the required transformation is described, and the resulting $O(n^2 \log n)$ average time distance multiplication algorithm given.

In section 6.3 another approach to dmm is explored - an algorithm for computing k inner products of pre-sorted distance matrices in time $O(k \sqrt{rn} + n)$ is described. This result is of minor importance when taken in isolation, but in section 6.4 the technique is used as a part of the main result of this chapter - a hybrid distance matrix multiplication of $O(n^{2.5})$ average time. This complexity is greater than the algorithm derived in section 6.2, but as the experiments of section 6.5 show, the operational running time is very fast. Although asymptotically less efficient, for tractable sized matrices the new hybrid algorithm is faster than any other dmm algorithm.

In section 6.6 another use is made of the inner product algorithm of section 6.3, and a probabilistic dmm algorithm with $o(n^3)$ worst case running time is described.

The following notation will be used consistently throughout this chapter. In finding the product Z of two n by n distance matrices X and Y , where

$$z_{ij} = \min\{ x_{ik} + y_{kj} : 0 < k \leq n \}$$

the quantity $x_{ik} + y_{kj}$ for variables i, j, k will occur frequently, and will always be abbreviated to $\text{cost}(i, k, j)$. For convenience in the pseudo-code programs describing dmm algorithms matrix subscripting will be indicated by square brackets, so that $Z[s, t]$ represents z_{st} . No distinction is intended by the use of this alternative notation.

The three matrices X , Y and Z will always be assumed to be n by n , so each entry z_{ij} of Z is the result of one n element distance inner product. Without ambiguity this will be abbreviated to inner product, and similarly the phrase "distance matrix multiplication" will be abbreviated to "matrix multiplication" or "dmm". The distance matrix semi-ring should always be taken to be the default algebraic structure; at no stage will multiplication of real matrices or inner products be discussed. Note that distance matrix multiplication is not commutative.

In section 6.2 the underlying randomness assumption will be that the second of the two distance matrices is drawn from an endpoint independent distribution. Only the second of the two matrices needs to be random, for reasons that will be made clear in the analysis. This means that the first matrix may be chosen arbitrarily.

In sections 6.3, 6.4 and 6.6 analyses will again be for average case running time, but with the randomness requirement being that either the first matrix X is endpoint independent, or that (an analogous definition is omitted) the second matrix Y is source independent.

6.2 An $O(n^2 \log n)$ Average Time DMM Algorithm.

The similarity between the amsp problem and the dmm problem can be seen by considering figure 6.1, taken from Aho et al (1974, page 201). In the diagram, edges in the left half of the graph represent entries in the matrix X , and edges in the right half of the graph represent entries in the second matrix Y . The value of the inner product z_{ij} is the cost of a shortest path from the vertex marked x_i to the vertex y_j , so a solution to the amsp problem for this graph of $3n$ vertices gives a solution to the dmm problem.

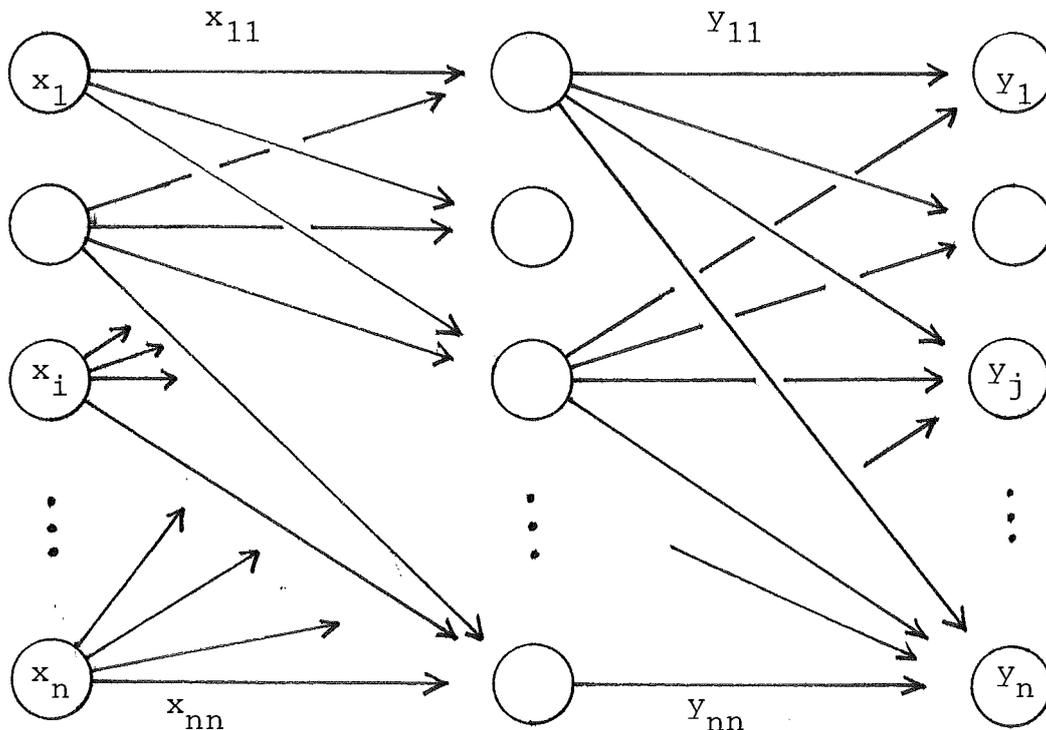


Figure 6.1 - Graph for dmm problem.

With this construction in mind, the algorithm Fast-ssp can be modified to take advantage of the special graph structure, and a solution to the dmm problem can be given by the

following program. The same conventions and variables as were used in describing algorithms Spira-ssp and Fast-ssp (chapter 4) have been used, except that the heap uses a slightly different weight function. The algorithm computes the shortest paths from the vertex x_s , and so calculates one row of the matrix Z . To find the complete matrix Z it should be applied n times, once for each source.

```

procedure fast-dmm ( s ) ;
begin
  S := {} ; purged := {} ;
  for c in 1 to n do
    place into the heap (c,t), where t is the endpoint
      of the shortest edge from c in Y ;
  buildheap ;
  for k := 1 to iloglog(n) do
    begin
      expand-soln( n - n/2k ) ;
      clean-up-heap ;
    end ;
  expand-soln( n ) ;
end {fast-dmm} ;

procedure expand-soln( stopat ) ;
begin
  while |S| < stopat do
    begin
      let (c,t) be at the root of the heap
        when weighted by cost(s,c,t) ;
      replace (c,t) by (c,t'), where t' is the endpoint
        in Y of the next shortest edge from c such that
        t' is unpurged, and rearrange the heap ;
      if t is not in S then
        begin
          S := S + {t} ;
          Z[s,t] := cost(s,c,t) ;
        end ;
      end ;
    end {expand-soln} ;

procedure clean-up-heap ;
begin
  purged := S ;
  for each element (c,t) in the heap do
    if t is in S then
      replace (c,t) by (c,t'), where t' is the endpoint
        of the next shortest edge from c in Y such that
        t' is unpurged ;
  buildheap ;
end {clean-up-heap} ;

```

Algorithm 6.1 - Fast-dmm.

In this variant of algorithm Fast-ssp there are initially no labelled vertices. The heap is initialised to contain n entries, with the weight of each entry (c,t) given by $\text{cost}(s,c,t)$; and when a vertex is freshly labelled there are no outward edges, so the heap does not get expanded. The size of the heap is fixed at n elements throughout the execution of the algorithm. Only the second of the two distance matrices needs to be presorted.

The correctness proof and analysis of this method are identical to those of algorithm Fast-ssp of chapter 4, and the arguments are not repeated. The running time of Fast-dmm will be $O(n^2 \log n)$ whenever the second matrix is endpoint independent; because the first matrix is treated one row at a time and is not presorted there is no need to impose any randomness constraints upon it. Only the second matrix needs to be random to attain the desired $O(n^2 \log n)$ running time.

Theorem 6.1 Let X and Y be n by n distance matrices, with Y drawn from an endpoint independent probability distribution. Then the matrix product $Z=X*Y$ can be computed in $O(n^2 \log n)$ expected time.

Proof. From the above discussion and theorem 4.2. []

All of the results given in chapter 5 regarding the implementation of algorithm Fast-ssp also apply to algorithm Fast-dmm.

6.3 An $O(\sqrt{rn})$ Average Time Inner Product Algorithm.

This section describes a simple technique for computing the inner product of two pre-sorted n element distance vectors in time $O(\sqrt{rn})$, after an initial $O(n)$ time that is required for initialising the data structure. That is, k of the inner products of two pre-sorted distance matrices can be computed in time $O(k\sqrt{rn}+n)$.

A graphical interpretation of the inner product of two distance vectors is shown in figure 6.2.

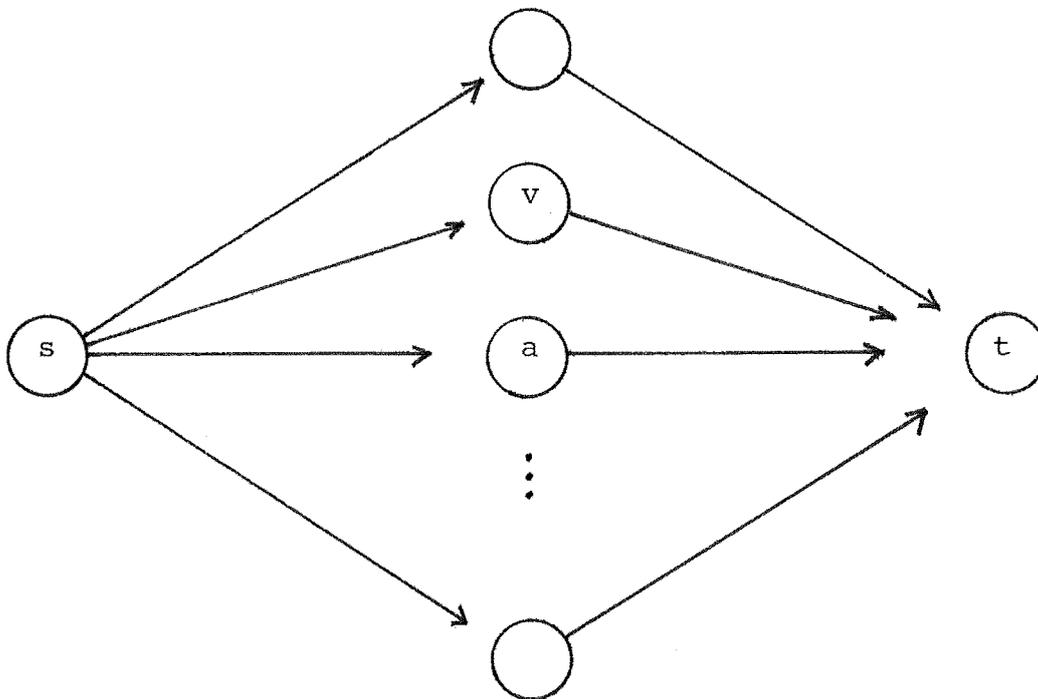


Figure 6.2 - A distance inner product.

The value of the inner product is the shortest path from the source s to the sink t . For any intermediate vertex a , $\text{cost}(s,a,t)$ gives an upper bound on the value of the inner

product; and for any other intermediate vertex v ,
 $\text{cost}(s,v,t) < \text{cost}(s,a,t)$ only if either $x_{sv} < x_{sa}$ or
 $y_{vt} < y_{at}$. If $x_{sv} \geq x_{va}$ and $y_{vt} \geq y_{at}$ then v need not be
considered as an intermediate point when calculating the
inner product. From this idea the algorithm follows.

Two sets are grown, S the set of intermediate points that are
the p closest to s , and T the set of intermediate points that
are the p closest to t . Initially S and T are empty, so that
the intersection of S and T is also empty. Then p is
increased and S and T expanded by the addition to them of
"next" vertices from the presorted vectors. When the
intersection of S and T becomes non empty for the first time,
 S and T are frozen, and the vertex that is common to both
establishes an upper bound on the value of the inner product.
It then remains to check the other members of the union of S
and T , to see if any of these intermediate points can improve
the upper bound given by $\text{cost}(s,a,t)$, where a is the vertex
in S intersection T . No vertices outside the union of S and T
can possibly improve upon $\text{cost}(s,a,t)$, and so they need not
be considered for this inner product.

```

real function innerprod( s,t ) ;
  {assume: S[i]=false, and T[i]=false, for all i}
  begin
    stack := empty ; p := 0 ;
    repeat
      begin
        p := p+1 ;
        let a be endpoint of p'th shortest edge from s,
            b be source of p'th shortest edge into t ;
        S[a] := true ; push(stack, a) ;
        T[b] := true ; push(stack, b) ;
      end
    until S[b] or T[a] ;
    bestsofar := averylargenumber ;
    while stack not empty do
      begin
        v := pop(stack) ;
        S[v] := false ; T[v] := false ;
        if cost(s,v,t) < bestsofar then
          bestsofar := cost(s,v,t) ;
        end ;
      innerprod := bestsofar
    end {innerprod} ;
  end

```

Algorithm 6.2 - Innerprod.

Correctness.

The correctness of the algorithm follows from the discussion above. When the p 'th shortest edge from s is among the p shortest edges into t , the optimal value of the inner-product can be found with one of the vertices in $S \cup T$ as an intermediate point.

Analysis of algorithm Innerprod.

Each of the two while loops will take $O(1)$ time per iteration, assuming that the two distance matrices are both pre-sorted, the first by row, and the second by column. Thus, if $E(p)$ is the expected value of p at the conclusion of the first loop, then the time required by the algorithm is $O(E(p))$. Lemma 6.2 shows that for suitably random matrices

$E(p) = O(\sqrt{n})$.

Lemma 6.2 If either matrix X is endpoint independent, or matrix Y is source independent, or both, then $E(p) < 2\sqrt{n}$.

Proof. $E(p)$ is broken into two manageable parts with the use of conditional expectations:

$$E(p) = E(p | p < \sqrt{n}) * \Pr[p < \sqrt{n}] + E(p | p \geq \sqrt{n}) * \Pr[p \geq \sqrt{n}] .$$

The first term, representing the expected value of p given that no more than \sqrt{n} elements are added to S and T , is clearly less than \sqrt{n} . The second conditional expectation represents the expected value of p , given that S and T have each grown to include \sqrt{n} elements without there being any intersection. Beyond this threshold point of \sqrt{n} the probability of the intersection becoming non-null for each further element added is at least \sqrt{n}/n , since there are at least \sqrt{n} elements in each of S and T , and the destination (or source) of the next shortest edge is random amongst n possible vertices for S (or T , or both). Moreover, each addition of a vertex is an independent event, so the expected number of such events before the intersection becomes non-null is less than \sqrt{n} (lemma 1.1). So even in the case where p reaches \sqrt{n} , expectedly fewer than \sqrt{n} more vertices will be added to each of S and T before the intersection becomes non-null. Thus the second conditional expectation is less than $2\sqrt{n}$; combining these two terms gives the desired result. []

The following lemma concerning the value of p will also be of later use:

Lemma 6.3 Suppose that either matrix X is endpoint independent, or that matrix Y is source independent, or both. Let $\Pr[p > m]$ represent the probability that the final value of p in algorithm Innerprod is greater than some value m. Then

$$\Pr[p > k \cdot \sqrt{n}] \leq \exp(-k^2) .$$

Proof. To bound the probability it is only necessary to consider the final result - that $k \cdot \sqrt{n}$ elements were taken randomly from a set of n elements, and none of them were among a subset of $k \cdot \sqrt{n}$ special elements of the set. The probability of doing this with one trial is given by $(1 - k/\sqrt{n})$; so the probability of doing it $k \cdot \sqrt{n}$ times without "success" is less than

$$(1 - k/\sqrt{n})^{k \cdot \sqrt{n}} .$$

The following Taylor's expansion is valid for $0 \leq x < 1$,

$$\begin{aligned} \ln(1-x) &= -x - x^2/2 - x^3/3 - \dots \\ &\leq -x \end{aligned}$$

so that $\ln[(1-x)^r] \leq -rx$

and $(1-x)^r \leq \exp(-rx)$.

Replacing x by k/\sqrt{n} and r by $k \cdot \sqrt{n}$ gives the desired result - that the probability of $k \cdot \sqrt{n}$ successive trials each missing a target set of $k \cdot \sqrt{n}$ out of n elements is not greater than $\exp(-k^2)$. []

Neglected by algorithm Innerprod is the initialisation of the boolean vectors S and T. This will take $O(n)$ time. The algorithm assumes that the vectors are correctly initialised, and employs a stack to record the indices for which the

vectors are set to true. Thus the vectors can be reinitialised - returned to their initial state - in $O(p)$ time.

Theorem 6.4 Let X and Y be n by n pre-sorted distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then k of the elements of $Z = X*Y$ can be calculated in $O(k\sqrt{rn}+n)$ expected time.

Proof. From the above discussion and lemma 6.2. []

Corollary 6.5 Let X and Y be n by n distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then the matrix product $Z = X*Y$ can be calculated in $O(n^{2.5})$ expected time. []

Although this gives an algorithm that is faster asymptotically than the standard $O(n^3)$ method, the result is not as sharp theoretically as the $O(n^2 \log n)$ expected time bound of section 6.2. But in section 6.4 algorithm Innerprod is built upon to give an algorithm for dmm that is very fast for practical use.

6.4 A Fast Hybrid Algorithm for DMM.

Here a second $O(n^{2.5})$ expected time algorithm for dmm is described. The advantage of this algorithm is speed - in a first very fast pass tentative values are assigned to each inner-product, and then in a second pass any inner-products that cannot be proven optimal are re-evaluated using algorithm Innerprod. The first pass assigns tentative values and not necessarily optimal inner-product values, and because of this "flexibility" it can be implemented to run very quickly using a depth first search technique. The uncertainty as to the optimality of the values assigned by this pass comes from the fact that it is "guessed" that the optimal intermediate point of each inner product (s,t) can be found within the $b(n)$ shortest edges from s and within the $b(n)$ shortest edges into t (figure 6.3). The bound function $b(n)$ and the validity of the guess will be discussed below.

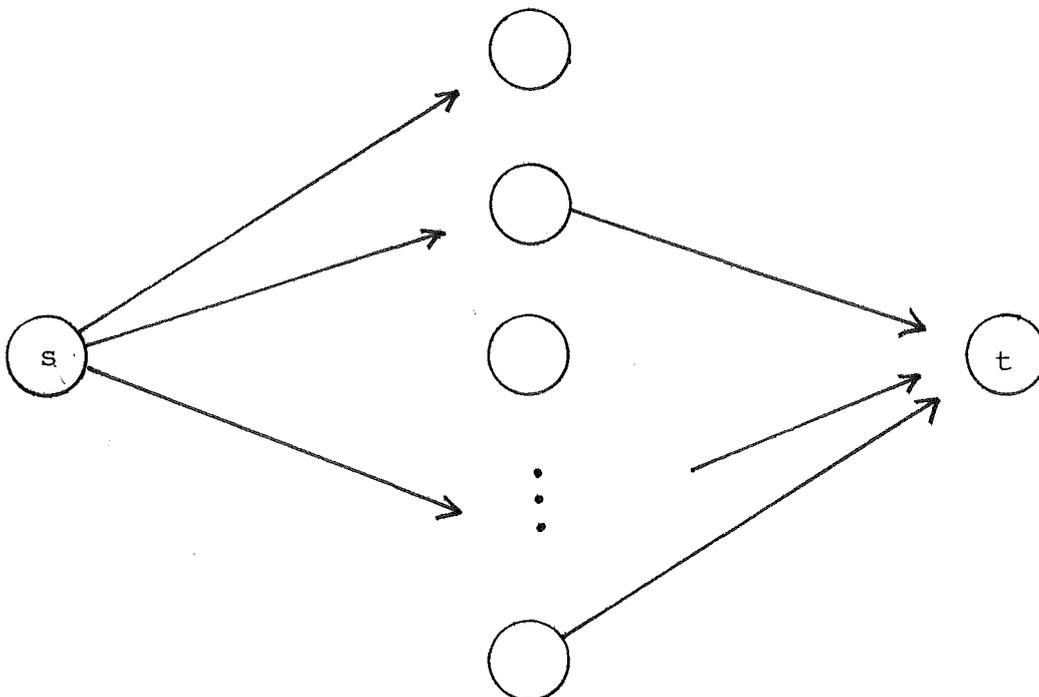


Figure 6.3 - The $b(n)$ shortest edges.

The algorithm to implement this guessing approach is straightforward. One row of the matrix Z will be calculated by a call to procedure `hybrid-dmm`;

```

procedure hybrid-dmm( s ) ;
begin
  for i := 1 to n do Z[s,i] := averylargenumber ;
  {probabilistic pass, using dfs and limited edges}
  for i := 1 to b(n) do
    begin
      let v be endpoint of i'th shortest edge from s ;
      for each t such that v is the source of one of
        the b(n) shortest edges into t do
        if cost(s,v,t) < Z[s,t] then
          Z[s,t] := cost(s,v,t) ;
        end ;
      {second checking pass}
    end
  for t := 1 to n do
    if Z[s,t] is not optimal then
      Z[s,t] := innerprod( s,t ) ;
    end {hybrid-dmm} ;
end

```

Algorithm 6.3 - Hybrid-dmm.

Here it is assumed not only that each row of the first matrix and each column of the second matrix have been sorted, but also that, after this sorting, the second matrix has been stored into an auxiliary form where the $b(n)$ shortest edges (v,t) into each vertex t of Y have been distributed by their source vertex v . This step will take $O(n*b(n))$ time, but only once; thereafter the loop described by "for each t such that v is the source of one of the $b(n)$ shortest edges into t " will require expectedly $b(n)$ time.

The second pass will require no more than $O(n^{1.5})$ expected time, and possibly much less. The worst thing that can happen is that each of the n inner products that makes up this row of the result matrix fails, but with good luck almost all the the inner products will have been given optimal values by the first pass. The first probabilistic assignment pass will

require $O([b(n)]^2)$ time, provided that the edges have been preprocessed as described. The bound $b(n)$ can thus be chosen to be functionally a little larger than \sqrt{rn} without disturbing the $O(n^{1.5})$ average running time for hybrid-dmm, and the function tested was $b(n) = \sqrt{n \cdot \sqrt{rn}} = n^{0.75}$. This choice of $b(n)$ means that the running time of the first pass is also $O(n^{1.5})$.

The big difficulty of this scheme is the test for optimality. The first pass does not necessarily assign an optimal value, so, given some tentative value for the inner product (s,t) , it must be decided if this value is optimal. To err on the side of caution, and say that some optimal value is dubious and use algorithm Innerprod is acceptable, but to claim to be optimal some tentative value z_{st} which in fact is not optimal cannot be allowed.

One suitable optimality test is described by this function:

```
boolean function optimal ( s,t ) ;
begin
  let s1 be the endpoint of the shortest edge from s,
      sb be the endpoint of the b(n)+1'th
          shortest edge from s,
      t1 be the source of the shortest edge into t, and
      tb be the source of the b(n)+1'th
          shortest edge into t ;
  optimal := ( Z[s,t] < X[s,s1] + Y[tb,t] ) and
             ( Z[s,t] < X[s,sb] + Y[t1,t] ) ;
end {optimal} ;
```

Algorithm 6.4 - Optimality testing.

If the tentative guess is such that every unconsidered path must be longer, then it can be declared optimal. To know that every unconsidered path is longer, a lower bound on the value of unconsidered paths is established. This lower bound is given by

$$\min(x_{s,sl} + y_{tb,t}, x_{s,sb} + y_{tl,t})$$

If z_{st} is less than this bound then the inner product (s,t) can be known to have been correctly guessed by only using the shortest $b(n)$ edges, and so the second pass need not call the more expensive Innerprod procedure for this pair. If z_{st} is greater than the lower bound then it may or may not be optimal, and so Innerprod should be used to return a value that is guaranteed to be optimal.

The proportion of inner-products that fall through to the second pass is dependent upon the distribution function used for assigning costs to the edges. But because both the first and second passes have running time $O(n^{2.5})$, for the asymptotic analysis this fraction is not important - the expected running time of the hybrid algorithm is still $O(n^{2.5})$.

To show that the guess that $b(n)$ edges are sufficient is reasonable for some class of distribution, consider the case when the two matrices X and Y have their entries drawn independently from a uniform distribution.

Lemma 6.6 Let n values be drawn independently from the uniform distribution on $[0,1)$. Then the k 'th smallest value is expectedly $k/(n+1)$.

Proof. Let the n values be drawn and ordered to make the sorted list x_i . Define l_k to be the difference $x_k - x_{k-1}$, where $x_0 = 0$. Feller (1966, page 22) gives the result that the lengths l_i are independent and described by the density function

$$\Pr[L < t] = 1 - (1-t)^n$$

From this the expected value of each l_i is deduced to be $1/(n+1)$, and the value of the k 'th smallest value can be seen to be the sum of k drawings from the variable L . Application of the Law of Large Numbers (Feller 1968, pg 243) then gives the desired result. []

Corollary 6.7 Let two n element distance vectors be created by $2n$ independent drawings from the uniform distribution on $[0,1)$. Define $I(n)$ to be the expected value of the inner product of the two vectors. Then the expected value of $I(n)$ is $O(1/\sqrt{n})$.

Proof. Lemma 6.6 shows that the value of the element in the p 'th position of the sorted vector is expectedly $p/(n+1)$; and lemma 6.2 and algorithm Innerprod showed that to establish an upper bound on the value of an inner product expectedly no more than the shortest p edges need to be inspected, where $p=O(\sqrt{n})$. []

Let $B(n)$ be the expected value of the $b(n)$ 'th smallest of n independent drawings from $[0,1)$. Then $B(n) = b(n)/(n+1)$, which, for $b(n) = n^{0.75}$, is very close to $n^{-0.25}$, and $I(n) = o(B(n))$. Thus, for the uniform distribution, by examining $b(n)$ edges it becomes increasingly likely that the optimal value of the inner product will in fact be assigned by the first pass, and again, since $I(n) = o(B(n))$, it becomes increasingly likely that if an optimal value is assigned it can be detected as being optimal. The exact probabilities of these two events depend upon higher moments of the order statistics and are not relevant here. The point made here is that, for a uniform distribution, the

probability of an optimal value not being assigned decreases as n increases.

For other distributions the first pass may or may not be cost effective. Certainly it seems likely that a pathological distribution could be contrived for which a major fraction of the inner products needed to be referred to the second pass for confirmation. However, as the experimental results below show, even if algorithm Innerprod is used in its original form to calculate all n^2 inner products, the result is a fast algorithm. Even on a particularly bad distribution the hybrid-dmm algorithm will have an $O(n^{2.5})$ average case running time and will be fast for practical use; that for some (perhaps many) distributions the first pass dramatically reduces the time required is an additional bonus.

6.5 Experimental Results.

All of the algorithms described in this chapter were implemented in Pascal and run on a VAX 11/785 computer under VMS. Matrices were generated randomly, with each element taken independently from the uniform distribution on $[0,1)$. All listed results are the average of ten separate experiments. None of the computing times listed in tables 6.1, 6.2 and 6.3 include the presorting of the distance matrices; the subject of the presort will be discussed in detail below. All running times are in seconds.

The first table gives the running time required by algorithm Fast-dmm of section 6.2.

<u>n</u>	<u>dmm time</u>	<u>dmm time/n²logn</u>
45	1.09 sec	9.8 *10 ⁻⁵
64	2.27	9.2
91	4.83	9.0
128	10.0	8.7

Table 6.1 - Running times for algorithm Fast-dmm.

For this apsp based method, the running times can be seen to be almost identical with the time required for the same sized apsp problem (table 4.1). This is not unexpected, considering the origin of the algorithm. A Bloniarz based dmm algorithm would be expected to run fractionally faster than the times given here.

The second algorithm tested was the $O(n^{2.5})$ method implied by corollary 6.5, in which n^2 inner products are independently

calculated by algorithm Innerprod.

<u>n</u>	<u>dmm time</u>	<u>dmm time/n^{2.5}</u>
45	0.48 sec	3.5 *10 ⁻⁵
64	1.25	3.8
91	2.87	3.6
128	6.56	3.5

Table 6.2 - Running times for algorithm Innerprod.

The times given here also represent the worst case running time of the second pass of algorithm Hybrid-dmm. The hybrid algorithm was also implemented and tested, and, as can be seen from the next table, for the uniform distribution the actual running time of the second pass is much less than the "worst situation", as a decreasingly small proportion of the inner products actually need to be solved by the second pass. In these trials most of the inner products were assigned optimal values in the first pass, and because of the simplicity of the first pass, the majority of optimal values are being assigned very quickly.

<u>n</u>	<u>first pass</u>	<u>second pass</u>	<u>total</u>	<u>total/n^{2.5}</u>
45	0.20 sec	0.12 sec	0.32 sec	2.3 *10 ⁻⁵
64	0.45	0.18	0.63	1.9
91	1.03	0.33	1.36	1.7
128	2.49	0.59	3.08	1.7

Table 6.3 - Running time for Hybrid-dmm.

The number of inner products that were referred to the second pass was also counted, and these results are listed in table 6.4. The second column lists the number of referrals as a fraction of n^2 , the number of inner products, and this fraction can be seen to be decreasing rapidly, confirming the qualitative analysis of section 6.3.

<u>n</u>	<u>referrals</u>	<u>referrals/n²</u>
45	156	7.7 %
64	169	4.1
91	159	1.9
128	117	0.7

Table 6.4 - Calls to Innerprod by Hybrid-dmm.

For these experiments the distance matrices were drawn from a uniform distribution and the bound function used was $b(n) = \sqrt{n \cdot \sqrt{n}} = n^{0.75}$; for this combination the hybrid algorithm is very successful.

The times for the dmm algorithms were recorded without including the sort times. The algorithm Fast-dmm requires that one matrix be presorted, and the algorithm Hybrid-dmm requires the sorting of two distance matrices. In the case of the amsp algorithms of chapter four, the time required by the sort was less than 25% of the total running time of the algorithm. Here, however, for algorithm Hybrid-dmm the sorting time for the two matrices dominates the running time, and some effort was invested to improve the running time of the sort. The first improvement made to the sort was to reorganise it as suggested by Sedgewick (1978), and the

addition of an insertion sort pass reduced the sort time by about 40%. However, a much better way of reducing the sorting time is to realise that not all of the edges need to be sorted. The first pass of Hybrid-dmm requires only that the shortest $b(n)$ edges be identified; the second pass requires that expectedly $O(\sqrt{n})$ edges from each vertex be available in sorted order. The expectation here is sufficiently tight (lemma 6.3) that the probability of more than $b(n)$ edges being required in any single use of algorithm Innerprod is $O(\exp(-[b(n)]^2/n))$; with $b(n)=n^{0.75}$ this means that $b(n)$ edges will not suffice with probability $\exp(-\sqrt{n})$, which is $O(n^{-k})$, for all k . To partition out and then sort only the first $b(n)$ edges with quicksort is quite straightforward, and only a minor change in the program is required to reduce the running time by a large factor. The running times for these three sorting strategies are given in the following table.

<u>n</u>	<u>quicksort</u>	<u>Sedgewick</u>	<u>Sedgewick</u>
	all n edges	all n edges	$b(n)$ edges
45	0.31 sec	0.16 sec	0.13 sec
64	0.65	0.37	0.26
91	1.35	0.81	0.49
128	2.83	1.71	0.90

Table 6.5 - Sorting times for n and $b(n)$ edges.

Clearly, a large amount of time can be saved by sorting only the first $b(n)$ edges, and then, should more edges be required by some instance of the second pass, simply extracting the required edges by a linear search of the remainder of the edge list. This approach will not affect the asymptotic

complexity of the algorithm, but makes the practical implementation much faster. In all of the experiments on this algorithm, only one pair of edges was required to be extracted in this way.

This technique of sorting only a fraction of the edgelist can also be applied to the algorithm Fast-dmm. However, although the total number of edges used by that method is $O(n \log n)$ per source, the edges used are not evenly distributed across the edge lists, and experiments showed that for each single row multiplication there was some random vertex from which a large number of the edges were used. Typically one half to two thirds of the edges were commonly being examined from some of the edgelists, and it is less advantageous to sort only a fraction of the edge lists.

With these sorting times - one full Sedgewick sort for the Fast-dmm method, and two abbreviated sorts for each of Innerprod-dmm and Hybrid-dmm, the total running times given below are obtained. The simple row on column dmm algorithm is also tested, and the running times are listed in the table for comparison:

<u>n</u>	<u>simple</u>	<u>Fast-dmm</u>	<u>Innerprod</u>	<u>Hybrid-dmm</u>
45	0.70 sec	1.25 sec	0.74 sec	0.58 sec
64	2.12	2.64	1.77	1.15
91	6.24	5.64	3.85	2.34
128	17.9	11.7	8.36	4.88

Table 6.6 - Total running times for dmm.

As can be seen from the table, despite the inferior theoretical bound of the hybrid method when compared to Fast-dmm of section 6.2, it is very fast in terms of practical running time. Extrapolation (figure 6.4) shows that this speed advantage would continue until about $n=2000$, at which point the $O(n^2 \log n)$ algorithm would catch up, but both would be requiring over one hour of cpu time and many megabytes of memory. Thus, for practical purposes, the hybrid algorithm runs faster than the $O(n^2 \log n)$ algorithm and is no more difficult to implement.

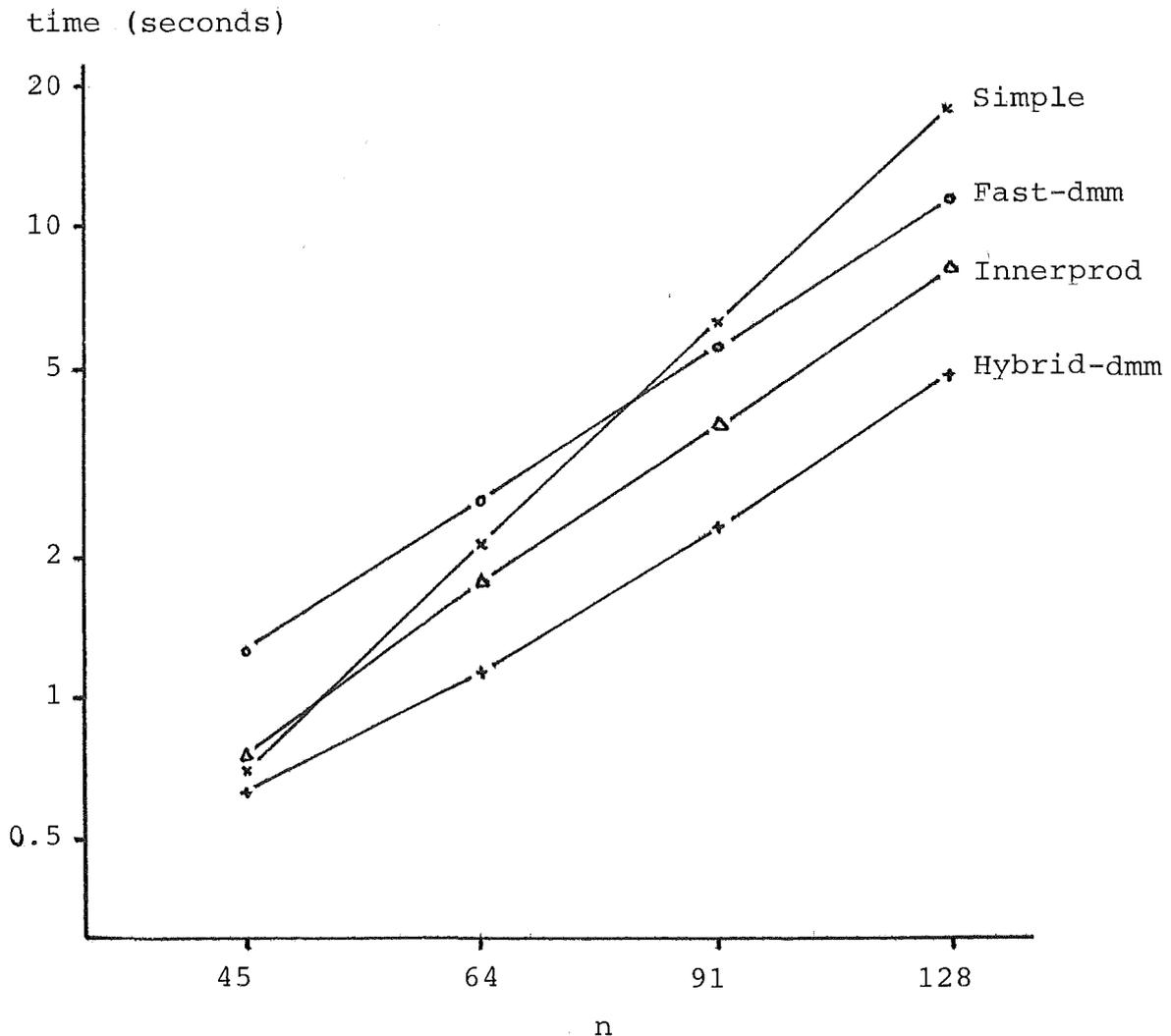


Figure 6.4 - Running times for dmm.

6.6 A Further Observation on Calculating Inner Products.

The probability bound given in lemma 6.3 has an interesting side effect. Algorithm Innerprod, when applied to suitably random distance matrices, guarantees to assign optimal values, and has an average running time of $O(n^{2.5})$, but a worst case running time of $O(n^3)$. By trading the certainty of optimality for certainty of running time a probabilistic variant of algorithm Innerprod can be made that on suitable random graphs has a worst case bound of $O(n^2b(n))$ time but some non-zero probability of assigning an incorrect inner product value. The choice of $b(n)$ and lemma 6.3 establish the extent of the tradeoff between running time and accuracy.

The algorithm proposed is a simple change from Innerprod. The repeat loop of that algorithm is revised so that it also terminates when p reaches $b(n)$, for some bound function b :

```

repeat
  begin
    p := p+1 ;
    let a be endpoint of p'th shortest edge from s,
        b be source of p'th shortest edge into t ;
    S[a] := true ; push(stack, a) ;
    T[b] := true ; push(stack, b) ;
  end
until S[b] or T[a] or p=b(n) ;

```

Algorithm 6.5 - Revised repeat loop.

Now, instead of continuing until the intersection of S and T becomes non empty and an upper bound on the value of the inner-product is established, a bounded number of edges is examined, with the hope that such an upper bound can be set, but always stopping if p reaches $b(n)$. The analysis of lemma 6.3 shows that, for endpoint independent matrices, the probability of this revised algorithm not assigning an

optimal value is less than $\exp(-[b(n)]^2/n)$. For example, with $b(n) = \sqrt{k*n*\ln(n)}$ the probability of not assigning an optimal inner product is $O(n^{-k})$, and the algorithm has an average running time of $O(n^{0.5})$ and a worst case running time of $O(n^{0.5}(\log n)^{0.5})$.

Using this technique and the same bound function $b(n)$ for an entire dmm problem means that the average time for the calculation of the product will be $O(n^{2.5})$ and the worst case time will be only $O(n^{2.5}(\log n)^{0.5})$. By choosing k to be greater than 3 the probability of failure becomes less than n^{-3} , so that over the n^2 inner products of a dmm problem, as n grows, it becomes more and more unlikely that a non-optimal value will be assigned (provided that one of the matrices is suitably random, required by lemma 6.3). Note that the worst case running time of $O(n^{2.5}(\log n)^{0.5})$ achieves Fredman's first sufficiency bound (1976), although the modified algorithm is an approximate algorithm and does not necessarily assign an optimal value for the product.

Theorem 6.8 Let X and Y be n by n distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then the matrix product $Z = X*Y$ can be calculated probabilistically in average time $O(n^{2.5})$ and worst case time $O(n^{2.5}(\log n)^{0.5})$, where the probability of each individual inner product being assigned an optimal value is $1-O(n^{-k})$ for any arbitrary, but fixed, value k .

Proof. From lemma 6.3 and the above discussion. []

CHAPTER SEVEN

SUMMARY.

Extracted from the previous text are the main theorems that have been shown concerning the complexity of the all pairs shortest path problem on a non-negatively weighted network. These are the best summary of the work reported in this thesis; to each of these theorems corresponds an algorithm that has been developed, analysed, and in most cases, empirically tested.

Theorem 3.11 Let $N=(G,C)$ be a non-negatively weighted directed network of n vertices and m edges. Then an algorithm exists for solving the apsp problem on N that requires in the worst case $O(mn + n^{2.5})$ time and $\min\{ n^3, 2mn \} + O(n^{2.5})$ active operations on path and edge costs. []

Theorem 4.2 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Then the apsp problem on N can be solved in $O(n^2 \log n)$ expected running time. []

Theorem 5.5 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Suppose that Spira's algorithm is altered by the addition of unlimited scanning. Then the expected running time of the resulting algorithm on N is $\Omega(n^{2.5})$. []

Lemma 5.7 The use of cost-compression with a Spira class algorithm will not decrease the running time of the algorithm; rather, the running time is likely to increase. []

Theorem 5.11 Let $N=(G,C)$ be a non-negatively weighted network of n vertices, with C a cost function drawn from an endpoint independent probability distribution. Then the apsp problem on N can be solved by a single algorithm with expected running time $O(n^2 \log n)$ and worst case running time $O(n^3)$. []

Theorem 5.13 Let $N=(G,C)$ be a directed network of n vertices, with the cost function given by $C(i,j) = (j-i) \bmod n$. Then any Spira class algorithm solving the apsp problem on N will require $1/2n^3$ changes to the weights of heap elements, meaning that the implementations of Spira and Bloniarz will both require $\Theta(n^3 \log n)$ running time. []

Theorem 6.1 Let X and Y be n by n distance matrices, with Y drawn from an endpoint independent probability distribution. Then the matrix product $Z=X*Y$ can be computed in $O(n^2 \log n)$ expected time. []

Theorem 6.4 Let X and Y be n by n pre-sorted distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then k of the elements of $Z = X*Y$ can be calculated in $O(k\sqrt{rn}+n)$ expected time. []

Corollary 6.5 Let X and Y be n by n distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then the matrix product $Z = X*Y$ can be calculated in $O(n^{2.5})$ expected time. []

Theorem 6.8 Let X and Y be n by n distance matrices, with either X drawn from an endpoint independent distribution, or Y drawn from a source independent distribution, or both. Then the matrix product $Z = X*Y$ can be calculated probabilistically in average time $O(n^{2.5})$ and worst case time $O(n^{2.5}(\log n)^{0.5})$, where the probability of each individual inner product being assigned an optimal value is $1-O(n^{-k})$ for any arbitrary, but fixed, value k . []

The single main result of this thesis is the $O(n^2 \log n)$ algorithm for the all pairs shortest path problem on a non-negatively weighted endpoint independent directed network. That algorithm, developed in sections 4.4 and 5.4, attacks two existing algorithms.

Prior to the discovery of the new algorithm, Bloniarz's $O(n^2 \log n \log^* n)$ algorithm was the best known result for graphs drawn from an endpoint independent probability measure. The result here improves upon his algorithm by a factor of $\log^* n$, and, although $\log^* n$ is a slowly growing function, the result is an asymptotic improvement, the importance of which is unquestionable. In terms of operational running time the new algorithm required the same or slightly less time than that of Bloniarz, and both are significantly better than all

previous methods for even quite small graphs, making the new algorithm suitable for operational use. As an added advantage the new algorithm can be implemented with a worst case bound of only $O(n^3)$, making it asymptotically superior to the algorithm of Bloniarz for both average running time and worst case running time.

The second algorithm that has been attacked is the recent proposal by Frieze and Grimmett (1983, 1985). Their algorithm can be used only when the edges of the graph are drawn independently from some probability distribution F , where F must be positive in every near neighbourhood of zero. That is, F must be such that

$$F(0) > 0, \text{ or } F(0)=0 \text{ and } \lim(x \rightarrow 0+) (F(x)/x) > 0.$$

The uniform distribution on $[0,1)$ meets this requirement, and can be handled by their algorithm, but even a simple change, such as a uniform distribution on $[0.5,1.5)$ cannot, as it is not possible to translate the distribution in a shortest path problem. Thus, although they achieved $O(n^2 \log n)$ running time on some class of random graphs, this class is contained as a proper subset in the endpoint independent class of probability measure handled by the new algorithm, and the new algorithm completely subsumes their previous result.

It is possible that $O(n^2 \log n)$ is optimal for this problem; this seems to have been the feeling of several authors (Yao et al, 1977; Graham et al, 1980; Bloniarz, 1983) though none have been able to prove it. Further, no wider class of random graphs containing as a proper subset the class of

endpoint independent graphs has been given, and endpoint independence seems to be a very useful concept for shortest path applications. It is possible that a "random graph" defined by endpoint independence is as strong a concept as that of a "random unsorted list" defined by equally likely permutations. If these two conjectures as to optimality and generality are correct then the new algorithm presented here will be important both theoretically and operationally for many years to come.

APPENDIX ONE

EXPERIMENTAL RESULTS FOR APSP ALGORITHMS.

Section 4.5 gives the result of experiments in which the new Fast-ssp algorithm was compared with the algorithm of Bloniarz. Those experiments were carried out using VAX 11/785 hardware, at the University of Ibaraki in Japan. All of the experimental times reported in the main body of this thesis were collected on that hardware, and so the results in section 4.5 can be compared with other running times reported for other algorithms.

The results given there for the two algorithms were almost identical, and certainly within the experimental error. Here the results of a second set of experiments comparing Fast-ssp and Bloniarz's algorithm are given. These experiments were carried out on Prime 750 hardware at the University of Canterbury, using the University of Sheffield Pascals compiler with range checking disabled. The two programs measured were identical to the two programs used in Japan in the first set of experiments; the programs were ported to New Zealand on magnetic tape and compiled again, with the only difference being in the system call for random number generation.

A wider range of experiments was carried out than in the initial trials in Japan, and a more detailed description of the running times of the two algorithms is given here.

The numbers below are the result of 20 experiments for $n \geq 91$, and 40 experiments for the two smaller values of n . Edge costs were again independently assigned from a uniform distribution.

<u>n</u>	<u>minimum</u>	<u>maximum</u>	<u>mean</u>	<u>deviation</u>
45	1.9 sec	2.3 sec	2.0 sec	0.1 sec
64	4.0	5.5	4.6	0.3
91	8.7	11.5	9.9	0.8
128	19.2	22.7	20.8	1.1
181	41.2	53.6	46.3	3.3
256	85.8	100.5	93.3	4.6

Table A1.1 - Running times for Fast-ssp on Prime 750.

From the table it can be seen that the standard deviation of the recorded values is typically about 5% of the average, and that the maximum and minimum values recorded over the 20 experiments varied from the mean by about 10%. The distribution of running times is discussed further below.

The second table gives the same information for the implementation of Bloniarz's algorithm that was tested. The final column gives the ratio of the mean running times for the two methods:

<u>n</u>	<u>minimum</u>	<u>maximum</u>	<u>mean</u>	<u>deviation</u>	<u>ratio</u>
45	1.8 sec	2.6 sec	2.1 sec	0.2 sec	1.05
64	4.0	5.6	4.7	0.4	1.02
91	9.1	12.3	10.8	0.9	1.09
128	20.2	26.9	22.4	1.8	1.08
181	44.2	59.4	50.3	4.2	1.09
256	94.6	114.8	102.5	5.5	1.10

Table A2.2 - Running times for Bloniarz on Prime 750.

From the table it can be seen that for this range of experiments the new Fast-ssp algorithm recorded a mean running time consistently better than Bloniarz's algorithm. However the difference between them is only about 10%, and the standard deviations of the two sets of recorded measurements are 5% each. Thus, although this second set of experiments is also statistically inconclusive, again it can be observed that there is no reason to believe that the new algorithm is inferior to that of Bloniarz for operational use.

To get a "feel" for the distribution of running times, the two algorithms were then run 100 times each with $n=91$, and the scatter of the running times recorded. The distribution of the running times for the two algorithms at $n=91$ is shown in figure A1.1:

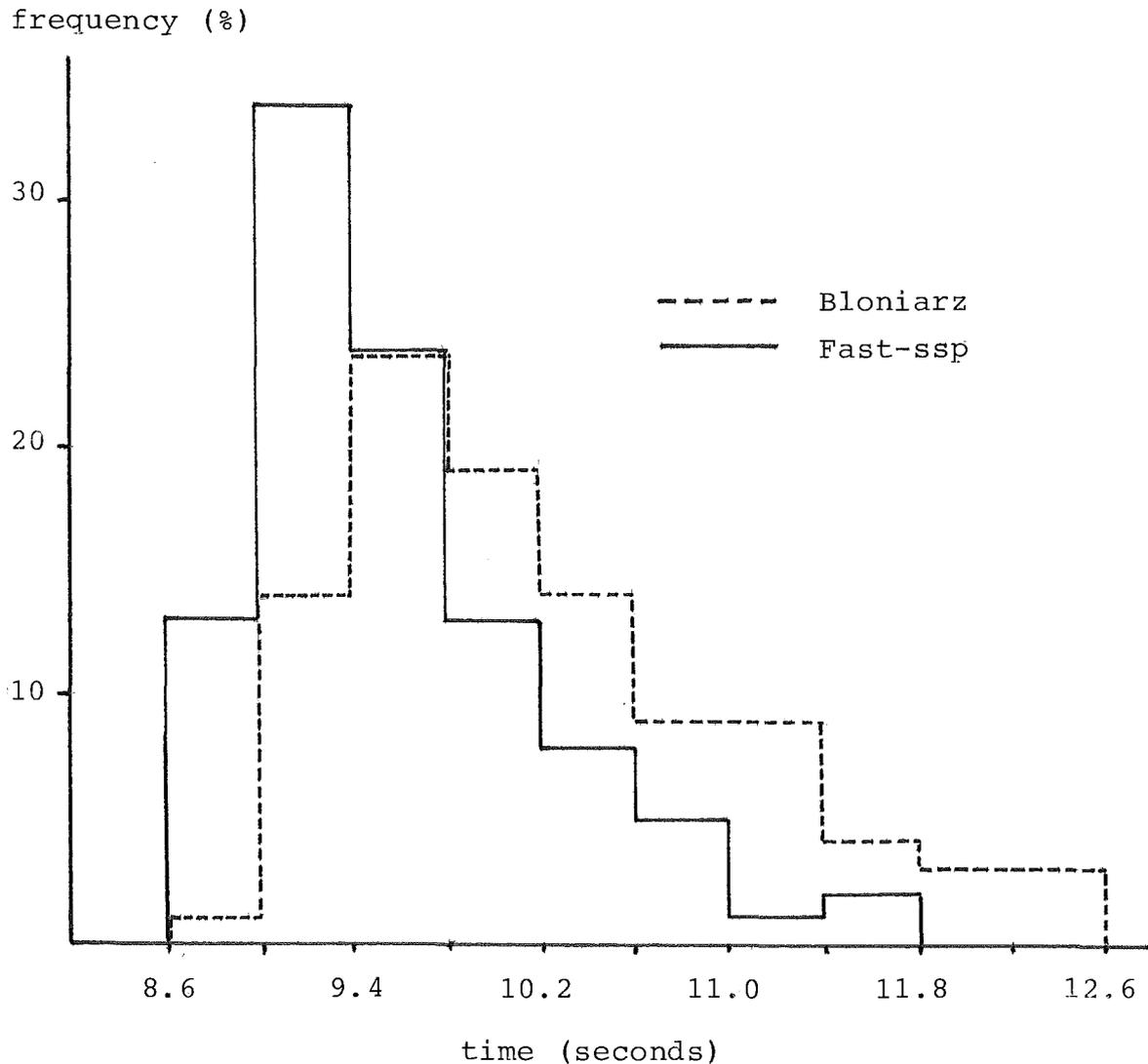


Figure A1.1 - Running times for n=91.

From the graph it again can be seen that the Fast-ssp has an empirically faster running time, but cannot be claimed unilaterally to be superior.

That one set of experiments recorded almost no difference between the two algorithms, and a second experiment recorded Fast-ssp to perhaps be 10% faster is an illustration of the need for care when comparing running times of two algorithms.

APPENDIX TWO

PROOF OF LEMMA 5.2.

Lemma 5.2 $E(rf|rb=0)$ is approximately $c \cdot \sqrt{rn}$, for some constant c .

Proof. This proof was supplied by T. Takaoka. For the purposes of brevity and clarity, throughout what follows the quantity $(n-1)$ is abbreviated to t . The recursive formula for $E(rf|rb=i)$ is used with increasing i :

$$\begin{aligned}
 E(rf|rb=0) &= \Pr[\text{success at first iteration}] * 2 + \\
 &\quad \Pr[\text{failure at first iteration}] * E(rf|rb=1) \\
 &= 2^2/t + [(n-3)/t] * E(rf|rb=1) \\
 &= 2^2/t + (n-3)/t * [3^2/t + ((n-4)/t) * E(rf|rb=2)] \\
 &= 2^2/t + 3^2(n-3)/t^2 + 4^2(n-3)(n-4)/t^3 + \\
 &\quad \dots + (n-1)^2(n-3)!/t^{n-2}
 \end{aligned}$$

so that

$$\begin{aligned}
 E(rf|rb=0) + n/(n-2) &= \sigma(i=1,t) [i^2(n-3)! / [t^{i-1}(t-i)!]] + (n-1)/(n-2) \\
 &= (n-3)!/t^{n-2} * \\
 &\quad [\sigma(i=1,t) [i^2 t^{t-i} / (t-i)!] + t^t / (n-2)!] \\
 &= F(n) * G(n),
 \end{aligned}$$

where

$$\begin{aligned}
 F(n) &= (n-3)!/t^{n-2} \quad \text{and} \\
 G(n) &= \sigma(i=1,t) [i^2 t^{t-i} / (t-i)!] + t^t / (n-2)!
 \end{aligned}$$

Using Stirling's approximation,

$$\begin{aligned} F(n) &\cong [\text{sqrt}(2\pi(n-3)) * (n-3)^{n-3} * \exp(-(n-3))] / t^{n-2} \\ &\cong c_1 * \exp(-n) / \text{sqrt}n \quad \text{for some constant } c_1. \end{aligned}$$

Now

$$\begin{aligned} G(n) &= \text{sigma}(i=1, t) [i^2 t^{t-i} / (t-i)!] + M(n) \\ &= \text{sigma}(k=0, t) [(t-k)^2 t^k / k!] + M(n), \\ &= \text{sigma}(k=0, t) [(t^2 - 2k * t + k^2) t^k / k!] + M(n) \end{aligned}$$

where

$$M(n) = t^t / (n-2)!$$

Define

$$H(n) = \text{sigma}(k=0, t) t^k / k!$$

Then

$$\begin{aligned} t * H(n) &= \text{sigma}(k=0, n) k * t^k / k! \\ t^2 * H(n) &= \text{sigma}(k=0, n+1) k(k-1) * t^k / k! \end{aligned}$$

and

$$\begin{aligned} G(n) &= \text{sigma}(k=0, n+1) k(k-1) t^k / k! - \\ &\quad 2 * \text{sigma}(k=0, n) k(k-1) t^k / k! + \\ &\quad \text{sigma}(k=0, n-1) k^2 t^k / k! + M(n) \\ &= \text{sigma}(k=0, n-1) k * t^k / k! + M(n) \\ &= \text{sigma}(k=0, n) k * t^k / k! - n * t^n / n! + t^t / (n-2)! \\ &= t * H(n) \end{aligned}$$

To determine the size of $H(n)$, consider the expansions

$$\begin{aligned} \text{sqrt}(\pi * t / 2) &\cong \\ &\quad 1 + t/n + t^2 / [n(n+1)] + t^3 / [n(n+1)(n+2)] + \dots \\ &\quad \text{(Knuth, 1973a, p112,117)} \end{aligned}$$

$$\begin{aligned} \exp(t) &= 1 + t + t^2 / 2! + \dots + t^t / t! + t^n / n! + \dots \\ H(n) &= 1 + t + t^2 / 2! + \dots + t^t / t! \end{aligned}$$

$$t! \cong \sqrt{2\pi t} * (t/e)^t$$

(Stirling's approximation)

so that

$$\begin{aligned} \exp(t) - H(n) + t^t/t! \\ &= t^t/t! * [1 + t/n + t^2/[n(n+1)] + \dots] \\ &\cong \exp(t) * \sqrt{\pi t/2} / \sqrt{2\pi t} \\ &\cong \exp(t) / 2 \end{aligned}$$

Then

$$\begin{aligned} H(n) &\cong \exp(t)/2 + t^t/t! \\ &\cong \exp(t)/2 + O(\exp(n)/\sqrt{tn}) \end{aligned}$$

Consequently

$$\begin{aligned} E(rf|rb=0) + n/(n-2) \\ &= F(n) * G(n) \\ &= F(n) * t * H(n) \\ &\cong [c_1 * \exp(-n)/\sqrt{tn}] * t * \\ &\hspace{15em} [\exp(t)/2 + O(\exp(n)/\sqrt{tn})] \\ &\cong c * \sqrt{tn} \hspace{10em} \text{for some suitable constant } c. \quad [] \end{aligned}$$

GLOSSARY

The following abbreviations and symbols have been used. For each a brief explanation has been given; for a more detailed explanation refer to the indicated section, or, if no section is given, section 1.3.

apsp	all pairs shortest path
$C(u,v)$	cost of direct edge from vertex u to vertex v
ceiling()	the smallest integer greater than or equal to the argument
$\text{cost}(s,a,t)$	the quantity $x_{sa} + y_{at}$ (6.1)
D	records shortest path costs for vertices in S for shortest path algorithms (4.1)
dmm	distance matrix multiplication
E	set of edges of a graph
$E()$	the expected value of the indicated variable
exp()	exponentiation base $e = 2.71\dots$
floor()	the largest integer less than or equal to the argument
$G=(V,E)$	graph G , consisting of vertex set V and edge set E
iloglog()	integer loglog function; $\text{iloglog}(n) = \text{ceiling}(\text{loglog}n)$
$L(u,v)$	shortest path cost from vertex u to vertex v
$\ln()$	logarithms base e
log()	logarithms base 2
$\log^*(n)$	$\log^*n = \text{minimum } i \text{ such that } \log(\log\dots(n)) < 1,$ where the logarithm is applied i times
m	number of edges in E for shortest paths
min()	minimum, returns the smallest of the arguments

n	number of vertices in V for shortest paths, size of matrices for distance matrix multiplication
$N=(G,C)$	network N, consisting of graph G and cost function C
$O(), o(), \Omega(), \Theta()$	order notation for indicating asymptotic growth rates
π	the quantity $\pi = 3.14\dots$
$Pr[]$	the probability of the indicated event
s	source node from which shortest paths are to be found (4.1)
S	growing solution set for single source algorithms (4.1)
sigma	operator of summation over the indicated range
spp	single pair problem
sqrt()	square root extraction
ssp	single source problem
V	set of vertices of a graph
$Z[i,j]$	the element z_{ij} of the matrix Z
S	the number of elements in the set S
(u,v)	the edge from vertex u to vertex v
(c,t)	candidate for vertex c, where c is a member of the current S and t is the destination of the shortest unexamined edge from vertex c (4.1)
[]	used to mark the end of a proof

ACKNOWLEDGEMENT

Much of the research reported in this thesis was undertaken during three periods totalling 10 months that I was a visiting researcher at the Department of Information Science of Ibaraki University in Japan, and a great debt is owed to the members of that Department for making such visits not only possible, but enjoyable. In particular I would like to thank Hisao Tamaki and Manabu Iwasaki for their willingness to discuss in a foreign language any subject I cared to raise; the three secretaries, Yasuko Matsudaira, Reiko Endo, and Kazuko Hata, who helped with the daily trivia of living in a foreign country; and finally, Professor Tadao Takaoka, who was my supervisor while I was in Japan and my conscience when I was not. My indebtedness to Professor Takaoka, for his support, advice, and encouragement, is beyond measure.

Thanks are also due to my colleagues at the University of Canterbury, especially Dr K. Pawlikowski, Dr B.J. McKenzie, and Dr W. Kreutzer, who read drafts of this thesis and assisted the debugging effort with many helpful comments and observations; and to the Departmental secretary, Mrs A. Marshall, who always had a smile when I needed one, and always knew how to spell those difficult words.

PUBLICATIONS.

The following sections of this thesis have appeared in published form:

The greedy algorithm of theorem 3.8 was described in a preliminary form in:

MOFFAT A.M. (1983), "A greedy algorithm for the all pairs shortest path problem", Proceedings of the sixth Australian Computer Science Conference, Sydney, 134-143.

The application of the parallel chains priority queue to Dijkstra's algorithm (theorem 3.11) was described in

MOFFAT A.M. (1984), with Takaoka T., "A priority queue for the all pairs shortest path problem", Information Processing Letters, 18: 189-193.

Algorithm Fast-ssp of section 4.4 was described in preliminary form in

MOFFAT A.M. (1985a), with Takaoka T., "An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$ ", Proceedings of the Japanese Institute of Electronics and Communication Engineers Symposium on Automata and Languages, Tokyo, 17 July 1985, 19-25,

and in a revised form, including the continuous cleaning ideas of section 5.4, in

MOFFAT A.M. (1985b), with Takaoka T., "An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$ ", 26th IEEE Symposium on the Foundations of Computer Science, Portland, Oregon, October 21-23 1985.

It is expected that a further revision of this last paper will be submitted for journal publication in the near future.

REFERENCES.

- AHO A.V. (1974), with Hopcroft J.E., Ullman J.D., "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Massachusetts.
- BLONIARZ P.A. (1979), with Meyer A., Fischer M., "Some observations on Spira's shortest path algorithm", Tech. Rep. 79-6, Comput. Sc. Dept., State Univ. New York at Albany.
- BLONIARZ P.A. (1980), "A shortest path algorithm with expected time $O(n^2 \log \log n)$ ", 12th ACM Symp. Theory Comput., 378-384.
- BLONIARZ P.A. (1983), "A shortest path algorithm with expected time $O(n^2 \log \log n)$ ", SIAM J. Comput., 12: 588-600.
- BONDY J.A. (1976), with Murty U.S.R., "Graph Theory with Applications", North-Holland, New York.
- CARSON J.S. (1977), with Law A.M., "A note on Spira's algorithm for the all-pairs shortest path problem", SIAM J. Comput., 6: 696-699.
- DANTZIG G.B. (1960), "On the shortest route through a network", Management Sc., 6: 187-190.
- DIAL R.B. (1969), "Algorithm 360: shortest path forest with topological ordering", Comm. ACM, 12: 623-633.
- DIJKSTRA E.W. (1959), "A note on two problems in connection with graphs", Numer. Math., 1: 269-271.

- DREYFUS S.E. (1969), "An appraisal of some shortest path algorithms", *Op. Res.*, 17: 395-312.
- EVEN S. (1979), "Graph Algorithms", Pitman, London.
- FARBEY B.A. (1967), with Land A.H., Murchland J.D., "The cascade algorithm for finding all shortest distances in a directed graph", *Management Sc.*, 14: 19-28.
- FELLER W.H. (1968), "An Introduction to Probability Theory and its Applications, Volume 1", 3rd edition, John Wiley, New York.
- FELLER W.H. (1966), "An Introduction to Probability Theory and its Applications, Volume 2", John Wiley, New York.
- FLOYD R.W. (1962a), "Algorithm 97: shortest path", *Comm. ACM*, 5: 345.
- FLOYD R.W. (1962b), "Algorithm 113: treesort", *Comm. ACM*, 5: 434.
- FLOYD R.W. (1964), "Algorithm 241: treesort 3", *Comm. ACM*, 7: 701.
- FREDMAN M.L. (1975), "On the decision tree complexity of the shortest path problem", 16th IEEE Conf. Found. Comput. Sc., 98-99.
- FREDMAN M.L. (1976), "New bounds on the complexity of the shortest path problem", *SIAM J. Comput.*, 5: 83-89.
- FREDMAN M.L. (1984), with Tarjan R.E., "Fibonacci heaps and their uses in improved network optimization algorithms", 25th IEEE Conf. Found. Comput. Sc., 338-346.

- FRIEZE A.M. (1983), with Grimmett G.R., "The shortest path problem for graphs with random arc lengths", Tech. Rep. S-83-02, Dept. Comput. Sc. and Stat., Queen Mary College, University of London.
- FRIEZE A.M. (1985), with Grimmett G.R., "The shortest-path problem for graphs with random arc-lengths", *Discrete Applied Mathematics*, 10: 57-77.
- GRAHAM R.L. (1980), with Yao A.C., Yao F.F., "Information bounds are weak in the shortest distance problem", *J. ACM*, 27: 428-344.
- HASSIN R. (1981), "Maximum flow in (s-t) planar networks", *Inf. Proc. Lett.*, 13: 107.
- HIRSCHBERG D.C. (1976), "Parallel algorithms for the transitive closure and connected components problems", 8th ACM Symp. Theory Comput., 55-57.
- HU T.C. (1982), "Combinatorial Algorithms", Addison-Wesley, Reading, Massachusetts.
- JOHNSON D.B. (1973), "Algorithms for Shortest Paths", TR-73-169, Ph.D. Thesis, Dept. Comput. Sc., Cornell University, Ithaca, New York.
- JOHNSON D.B. (1977), "Efficient algorithms for shortest paths in sparse networks", *J. ACM*, 24: 1-13.
- KERR L.R. (1970), "The effect of algebraic structure on the Computational Complexity of Matrix Multiplication", Ph.D. Thesis, Dept. Comput. Sc., Cornell University, Ithaca, New York.

- KNUTH D.E. (1973a), "The Art of Computer Programming, Volume 1 - Fundamental Algorithms", 2nd edition, Addison-Wesley, Reading, Massachusetts.
- KNUTH D.E. (1973b), "The Art of Computer Programming, Volume 3 - Sorting and Searching", Addison-Wesley, Reading, Massachusetts.
- KNUTH D.E. (1976), "Big omega and big omicron and big theta", SIGACT News, 8: 18-24.
- KRUSKAL J.B. (1956), "On the shortest spanning subtree of a graph and the travelling salesman problem", Proc. Am. Math. Soc., 7: 48-50.
- MAHR B. (1981), "A birds eye view to path problems", in Noltemeier H. (ed), "Graphtheoretic Concepts in Computer Science", Springer-Verlag, Berlin, 335-353 (Lecture Notes in Computer Science volume 100).
- MOFFAT A.M. (1979), "The All Pairs Shortest Path Problem and Algorithms to Solve It", Honours Report, Dept. Comput. Sc., Univ. Canterbury, New Zealand.
- MOFFAT A.M. (1983), "An empirical survey of shortest path algorithms", NZ Op. Res., 11: 153-163.
- MOFFAT A.M. (1984), with Takaoka T., "A priority queue for the all pairs shortest path problem", Inf. Proc. Lett., 18: 189-193.
- MORAN S. (1981), "A note on 'Is the shortest path problem not harder than matrix multiplication?'", Inf. Proc. Lett., 13: 85-86.

- PAPE U. (1980), "Algorithm 562: Shortest path lengths", ACM Trans. Math. Software, 6: 450-355.
- REIF J.H. (1983), "Minimum s-t cut of a planar undirected network in $O(n^2 \log^2 n)$ time", SIAM J. Comput., 12: 71-81.
- ROMANI F. (1980), "Shortest path problem is not harder than matrix multiplication", Inf. Proc. Lett., 11: 134-136.
- SCHONHAGE A. (1981), "Partial and total matrix multiplication", SIAM J. Comput., 10: 434-355.
- SEGEWICK R. (1978), "Implementing quicksort programs", Comm. ACM, 21: 847-857.
- SEGEWICK R. (1983), "Algorithms", Addison-Wesley, Reading, Massachusetts.
- SPIRA P.M. (1973), "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$ ", SIAM J. Comput., 2: 28-32.
- SPIRA P.M. (1975), with Pan A., "On finding and updating spanning trees and shortest paths", SIAM J. Comput., 4: 375-380.
- STRASSEN V. (1969), "Gaussian elimination is not optimal", Numer. Math., 13: 354-356.
- TAKAOKA T. (1980), with Moffat A.M., "An $O(n^2 \log n \log \log n)$ expected time algorithm for the all shortest distance problem", in Dembinski P. (ed), "Mathematical Foundations of Computer Science", Springer-Verlag, Berlin, 643-655 (Lecture Notes in Computer Science volume 88).

- TARJAN R.E. (1983), "Data Structures and Network Algorithms", Society for Industrial and Applied Mathematics, Philadelphia.
- TOMIZAWA N. (1976), "An efficient algorithm for solving the shortest route problem", Electronics and Communications in Japan, 59A: 1-8 (in Japanese).
- WARSHALL S. (1962), "A theorem on boolean matrices", J. ACM, 9: 11-13.
- WILLIAMS J.W.J. (1964), "Algorithm 232: heapsort", Comm. ACM, 7: 347-348.
- WILLIAMS T.A. (1973), with White G.P., "A note on Yen's algorithm for finding the length of all shortest paths in N-node non-negative distance networks", J. ACM, 20: 389-390.
- YAO A.C. (1977), with Avis D.M., Rivest R.M., "An $\Omega(n^2 \log n)$ lower bound to the shortest path problem", 9th ACM Symp. Theory Comput., 1977, 11-17.
- YEN J.Y. (1972), Finding the lengths of all shortest paths in N-node nonnegative distance complete networks using $(1/2)n^3$ additions and n^3 comparisons", J. ACM, 19: 423-324.
- YUVAL G. (1976), "An algorithm for finding all the shortest paths using $n^{2.81}$ infinite precision multiplications", Inf. Proc. Lett., 4: 155-156.