COMPRESSION METHODS FOR GRAPH ALGORITHMS

A thesis

submitted for the Degree

of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

S.A. Freeth

University of Canterbury

1985

TO

MY FATHER

ACKNOWLEDGEMENTS

I wish to thank Professor J.P. Penny for accepting me as a research student, my supervisors Dr. Tadeo Takaoka for introducing me to the subject, and Dr. B.J. McKenzie for his guidance and encouragement in preparing this thesis. I would also like to thank Mr. A.M. Moffat for many helpful comments on the drafts.

I wish to thank Mr. G. Smith of Gabites, Porter and Partners (CH-CH) Ltd., for the opportunity of working with transportation networks 'in the real world' and also Mr. G. Bellis who ran the transportation models. I wish to thank the Christchurch Transport Board for permission to use the map in Appendix A.

I wish to thank Mr. C. Hadlee for drafting many of the graphs.

Finally, I would like to thank my husband, without whose constant encouragement and support this would not have been possible.

ABSTRACT

Two compression methods for representing graphs are presented, in conjunction with algorithms applying these methods.

A decomposition technique for networks that can be generated in O(m) time is presented. The components of the decomposition and the shortest path matrix of the compressed network can be used to find the shortest path between any pair of vertices in the original network in linear time.

A compression method for boolean matrices and a method for applying the compression to boolean matrix multiplication is developed. The algorithms have an expected running time of $O(n^2*\log_2 n)$. From this compression method a simple heuristic that may be applied to any algorithm for boolean matrix multiplication has been developed. This heuristic will improve the average running time of boolean matrix multiplication algorithms.

An order of magnitude analysis of the results published by Loukakis and Tsouris [1981], on the efficiency of algorithms for finding all maximal independent sets of a graph has been performed. This analysis showed that their conclusions, which are based on a direct comparison of the running times of the algorithms, do not take into account implementation factors.

An average constant factor improvement is developed for the algorithm of Tsukiyama, Ide, Ariyoshi and Shirakawa [1977] for finding all maximal independent sets of a graph. Analysis of the running time results from the algorithm comparisons presented in this thesis show that the Bron-Kerbosch algorithm has the smallest rate of increase in running time as the size of the graphs increase.

TABLE OF CONTENTS

LIST OF DEFINITIONS

LIST OF THEOREMS

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

> "Although algorithmic graph theory was started by Euler, if not earlier, its development in the last ten years has been dramatic." [Even, 1979]

Graph theory has become recognised as a suitable paradigm for embedding many problems from engineering and science, and therefore algorithms for the solution of graph problems have been intensively studied. Efficient data representations are as important as the techniques used in algorithm design.

Read [1969] published a survey paper on the methods of representing a graph in a computer, but still today the amassed knowledge of graph representation is mainly heuristic and far from being unified in any profound mathematical theory. Lack of structure, except at a most simplistic level, is inherent from the definition of a graph. This lack of predetermined structure gives graphs the power to represent any structured object. Graphs are the most suitable representation of data where the relationships between the data items are as significant as the items themselves. Graphs can also reflect complex relationships that cannot be represented by other data structures.

Examination of graph algorithms shows, in almost all cases, that the complexity is dependent on the choice of graph representation. Various constraints, for example the type of operation to be performed, determine the most appropriate graph representation for any particular graph problem. It is possible no single representation will provide a global solution for the representation of graphs for all graph algorithms. However, the reported successes, from applying new data structures, in the development of new and efficient algorithms suggests graph representations are worthy of further investigation. Carre [1979] states:

> "Since 1970, computer scientists have made important contributions to graph theory . . . they have achieved remarkable improvements in the performance of graph algorithms, mainly through the clever manipulation of data structures."

While no formal theory exists as a basis for selecting and applying graph representations, continuing research in this area has led to improved algorithms for individual graph problems.

This thesis reports the results of an investigation of graph representations and their applications. First, two distinct compression methods for representing graphs are developed in conjunction with algorithms applying these compression methods to specific graph problems. Second, an examination of graph representations has led to the development of an average constant factor improvement to the Tsukiyama et al algorithm for finding all maximal independent sets of a graph.

The following organisation has been used in this thesis.

Chapter 2 presents the background of graph theory and terminology used in this thesis. Traditional graph representations and the theoretical relationship between directed graphs and boolean matrices is described. Trees play an extremely important role in algorithm design, so, although they are a class of graph, it is their application to algorithm design that has been considered in this thesis. A probability distribution for graphs is described so that the expected behaviour of algorithms which accept graphs as input may be analysed. Finally, a special class of subgraphs, the maximal independent sets of graphs and their probability distribution is described.

Chapter 3 surveys graph representations, presented in the literature, for the development efficient graph algorithms. These representations include the traditional graph representations described in Chapter 2, which have been used for the efficient solution of some graph problems and some of the data structures developed in computer science. Also surveyed are the representations of a graph based on the identification of features generally found in graphs, despite their essentially unstructured nature. Another method for efficiently representing a graph is based on decomposing the graph into smaller graphs and several decomposition methods which have appeared in the literature are also surveyed.

A decomposition of networks, with emphasis on path problems, is presented in Chapter 4. A theory for the decomposition is developed and an algorithm to find the decomposition of any network in O(m) time (where m is the number of edges in the network) is presented. Path problems may be solved more efficiently on the compressed network generated by the decomposition than on the original network. From such solutions the recomposition algorithm that is also presented in Chapter 4, will find, for example, shortest paths between any pair of vertices in the original network in linear time. The application of this decomposition to shortest path problems on a local transportation network is reported.

One of the most common representations of a graph is a matrix and the adjacency matrix of a graph is known to be equivalent to a boolean matrix. Chapter 5 presents a method for compressing the elements of a row or column of a boolean matrix into a single value. A tree of these values then represents the row or column of the boolean matrix and a family of algorithms for boolean matrix multiplication based on tree search methods is developed. The results from testing these algorithms are presented and one of the algorithms is compared with other methods for boolean matrix multiplication.

A simple, easily implemented heuristic that may be applied to most boolean matrix multiplication algorithms has been developed, based on the compression method developed in Chapter 5. The 'summation heuristic' has been applied to many previously reported algorithms for boolean matrix

multiplication, and results from testing the modified and original algorithms are presented in Chapter 5.

Suggestions for representing a graph by its cliques are discussed in Chapter 3 and Chapter 6. Tsukiyama, Ide, Ariyoshi and Shirakawa [1977] presented the only algorithm for finding the maximal independent sets of a graph (equivalent to a clique found in the complement graph) with a proven time bound of O(n*m*c) (where n is the number of vertices, m the number of edges and c the number of maximal independent sets in a graph). Chapter 6 presents a modification to this algorithm that, on average, gives a constant time improvement. The algorithms are compared with other algorithms for finding the maximal independent sets or cliques of a graph and the results are presented. In an attempt to resolve an apparent conflict between these results and the previously reported conclusions of Loukakis and Tsouris [1981], further analysis of the results published by Loukakis and Tsouris was undertaken and is presented in Chapter 6.

Chapter 7 presents the conclusion.

CHAPTER 2

BACKGROUND

2.1 INTRODUCTION

A graph algorithm is an algorithm for solving problems formulated in terms of graphs. Improving the efficiency of algorithms has lead to the observation that the choice of representation has a strong influence on the complexity of many graph algorithms.

2.2 GRAPHS

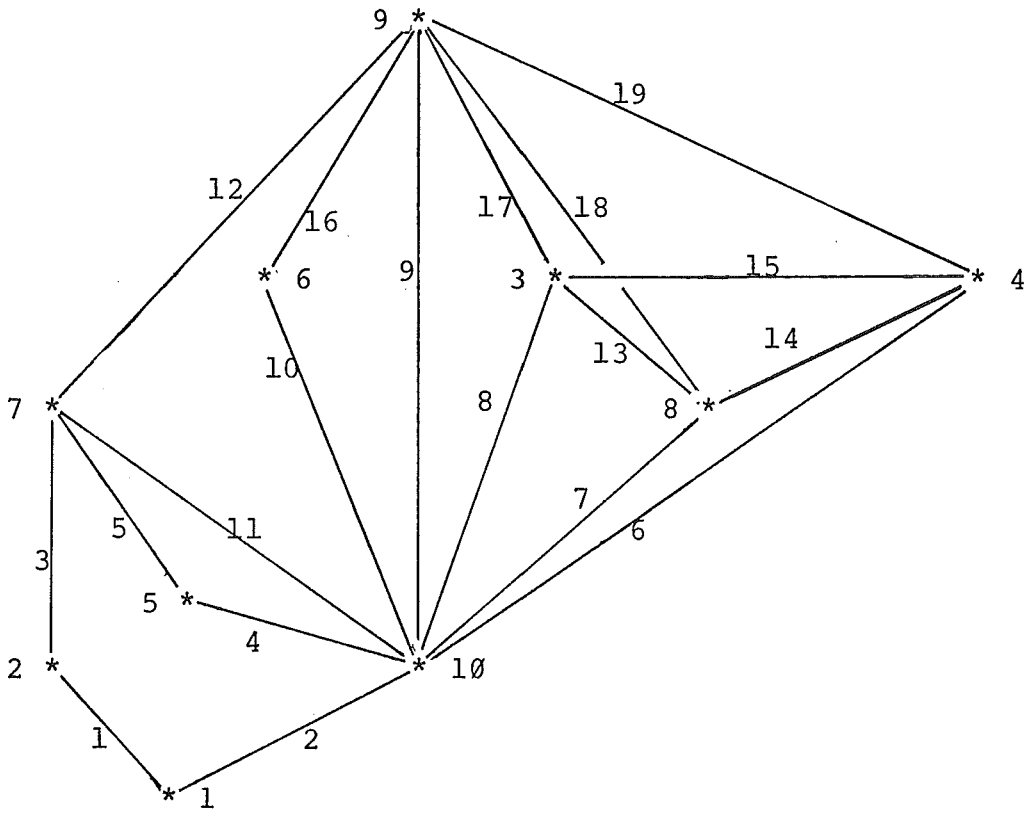
Full details on the graph theory summarised in the following definitions may be found in Bollobas [1979] and Tutte [1966].

Definition 2.1

A simple undirected graph G is a pair (V,E) where V is a finite set of graph vertices and E is a (finite) set of unordered pairs (v,w) where v and w are distinct vertices. The elements of E are called the edges of the graph and the edge (v,w) is said to be incident to the vertices v and w.

Example 2.1

The graph shown in Figure 2.1 below is an undirected graph with 1Ø vertices and 19 edges. The vertices of the graph have been labelled 1 to 1Ø and the edges have been labelled 1 to 19. This graph will be used as an example graph in Chapters 2, 5 and 6.



A GRAPH ON 1Ø VERTICES AND 19 EDGES **FIG 2.1**

(end of example)

The vertices of two distinct graphs G_1 and G_2, will be distinguished as $V(G_1)$ and $V(G_2)$. Similarly, the edges will be distinguished as $E(G_1)$ and $E(G_2)$.

Definition 2.2

The set r(v) for any v in V is the set of vertices adjacent to v in G. That is

$$r(v) = \{w \mid (v,w) \in G\}.$$

Definition 2.3

The degree of a vertex v, d(v) is the number of vertices adjacent to v. Thus the degree of any vertex v is

$$d(v) = |r(v)|.$$

The vertices of degree 2 are said to be divalent.

Definition 2.4

A subgraph G(W) of a graph G is a graph G(W) = (W,E(W)) where the vertices W are a subset of V and the edges are a subset of E on the vertices of W.

$$W \subseteq V$$

$$E(W) = \{(u,v) \in E \mid u,v \in W\}$$

Definition 2.5

A path on a graph is a sequence $v_1, v_2, .., v_i, ..., v_{k+1}$ such that there is an edge $e=(v_i, v_{i+1})$ for all vertices v_i for $1<=i<=k$.

Definition 2.6

A simple path on a graph is a path such that no vertex occurs more than once.

Definition 2.7

A graph is connected if every pair of vertices is joined by a path.

Definition 2.8

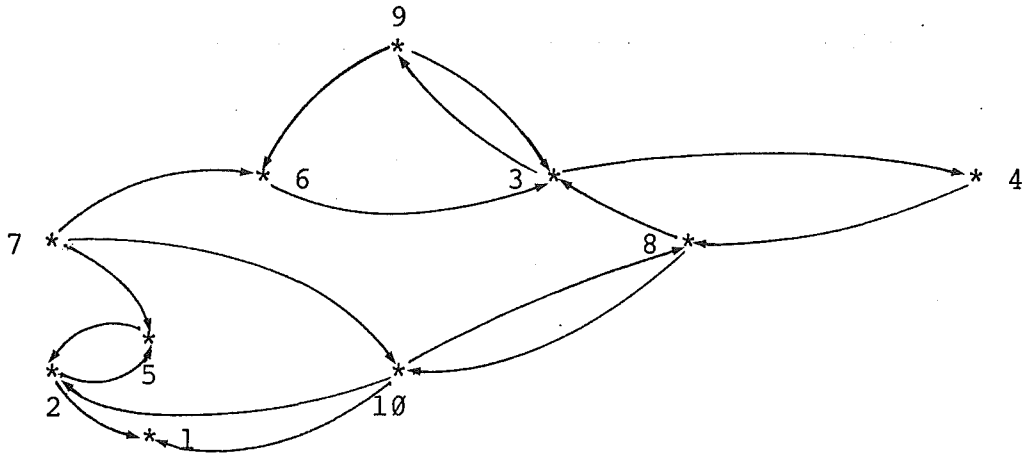
A polygon is a connected graph such that every vertex has degree 2.

Definition 2.9

A directed graph or digraph D is a pair (V,E) where V is a finite set of graph vertices and E is a (finite) set of ordered pairs (v,w). The elements of E are called the edges of the graph and are directed from v to w. The edge is said to be incident out of v and incident into w. The vertex w is the successor of v and v is the predecessor of w. The set of all successors of a vertex v is denoted by $r^+(v)$ and the set of all predecessors is denoted by $r^-(v)$.

Example 2.2

A directed graph with 10 vertices and 17 edges is shown in Figure 2.2. Note there are no predecssors of vertex 7 and no successors of vertex 1, $r^-(7) = \{\}$ and $r^+(1) = \{\}$, and for vertex 4, $r^-(4) = r^+(4) = 1$.



A DIGRAPH ON 10 VERTICES AND 17 EDGES **FIG 2.2**

(end of example)

Definition 2.10

A <u>self loop</u> in a digraph is an edge incident out of a vertex v and incident into the same vertex v.

Definition 2.11

Two distinct edges $e_i = (v_i, w_i)$ and $e_j = (v_j, w_j)$ in a digraph are <u>parallel</u> if $v_i = v_j$ and $w_i = w_j$.

Many graph definitions allow parallel edges, but they are specifically excluded by the definition of a graph used here and in the following definition of a network.

Definition 2.12

A <u>network</u> is a directed graph G(V,E) with no self loops and no parallel edges. Every edge e in a network is assigned a non-negative number c(e) called the distance or cost.

Definition 2.13

A path on a network is a sequence of edges $e_1, e_2, \ldots e_k$ such that if e_i is an edge (v,w) incident out of a vertex v and incident into a vertex w and if e_i is not the first or last edge then e_{i-1} is incident into v and e_{i+1} is incident out of w. The length of the path is the number of edges in the path, k.

Definition 2.14

Two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ are isomorphic if there exists a 1-1 mapping $f:V_1 \rightarrow V_2$ such that (v,w) in E_1 if and only if (f(v),f(w)) in E_2.

Definition 2.15

An automorphism of a graph G=(V,E) is an isomorphism of G onto itself.

Definition 2.16

A homomorphism h of $G_1=(V_1,E_1)$ onto $G_2=(V_2,E_2)$ is a pair (h,σ) of a mapping $\sigma:V_1 \rightarrow V_2$ and a mapping $h:E_1 \rightarrow E_2$ such that for any e in E_1 if e=(u,v) with u,v in V_1 the $h(e)=(\sigma(u),\sigma(v))$.

The following definitions were used by Tutte to define the nodes and branches of a graph. A more direct definition is presented in Chapter 4 to define a decomposition of a network.

Definition 2.17

Let G(H) be a subgraph of a graph G=(V,E). A vertex of attachment is a vertex v_i in H such that there exists an edge $e=(v_i,v_j)$ not in E(H) and v_j is not in H. Denote the vertices of attachment of G(H) in G by W(G,H). G(H) is J-detached if $W(G,H) \subseteq V(J)$ where G(J) is a subgraph of G. G is J-connected if it has no J-detached subgraph other than G itself and the subgraphs of J and the subgraph G(H) is J-connected if it is $H \cap J$-connected. G(H) is called a J-component of G if it is J-detached and J-connected but not a subgraph of J.

Definition 2.18

Let G=(V,E) be a graph. A node of a graph is any vertex of G that is not divalent. The set of all nodes of G are denoted by N.

Definition 2.19

The N-components of G are termed the branches of G. The set of all branches of G is denoted by B. A node x and a branch X are incident if x in V(X).

Definition 2.20

A homeomorphism f of a graph $G_1=(V_1,E_1)$ with nodes N_1 and branches B_1 onto a graph $G_2=(V_2,E_2)$ with nodes N_2 and branches B_2 is a pair (g,h) where g is a 1-1 mapping of N_1 onto N_2 and h is a 1-1 mapping of B_1 onto B_2 such that a node x is incident with a branch X in G_1 if and only if g(x) is incident with h(X) in G_2.

2.3 REPRESENTATION OF GRAPHS

Prior to the application of computing to graph theory, the need for a non-topological representation of a graph had been realised. In response to this, the use of matrices became well established. Korfhage [1974] gives a comprehensive list of the many different matrix representations used in graph theory.

The most commonly used matrices are the Vertex Adjacency Matrix, usually known simply as the Adjacency Matrix, and the Incidence Matrix.

Definition 2.21

A Vertex Adjacency Matrix $A = a_{ij}$

where $a_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \text{ is an edge in } G \\ \emptyset \text{ otherwise} \end{cases}$

Definition 2.22

An Incidence Matrix $C = c_{ij}$

where $c_{ij} = \begin{cases} 1 \text{ if } e_j \text{ incident with } v_i,\ v_i \text{ in } V,\ e_j \text{ in } E \\ \emptyset \text{ otherwise} \end{cases}$

Let $n=|V|$ be the number of vertices in a graph G and $m=|E|$ be the number of edges. Then an adjacency matrix representation of a graph requires $O(n^2)$ storage space, and the incidence matrix representation requires $O(n*m)$ storage space. Furthermore, each of the m columns of the incidence matrix contains exactly two non-zero elements.

Example 2.3

Figure 2.3 shows the Adjacency Matrix and Incidence Matrix of the graph shown in Figure 2.1.

```
0100000001          11000000000000000000
1000001000          10100000000000000000
0001000111          00000001000010101000
0010000111          00000100000001100001
0000001001          00011000000000000000
0000000011          00000000010000010000
0100100011          00101000001100000000
0011000011          00000010000011000010
0011011101          00000000010010001111
1011111110          01010111111100000000
```

(a) (b)

(a) **ADJACENCY MATRIX**
(b) **INCIDENCE MATRIX**

FIG 2.3

(end of example)

More efficient storage of a graph may be achieved through the use of a list structure. A simple list is a finite set of items that can be totally ordered.

Definition 2.23

An Adjacency List $H_i = h_{ij}$

where $h_{ij} = v_k$ if v_k is the jth vertex adjacent to v_i in the sequence v_1 to v_n.

A graph can be represented by n adjacency lists, one for each vertex.

$H' = H_i$ for $1<=i<=n$.

Definition 2.24

An Edge List $L = l_i$

where $l_i = (v_j, v_k)$ for $1<i<m$

An adjacency list H_i for a vertex v_i with degree $d(v_i)$, stores each of the $d(v_i)$ values ranging from 1 to n. The n adjacency lists will store the m edges so will require O(n+m) storage.

An edge list L will store the vertex pair of each of m edges in the graph, so will require O(m) storage. The vertices of the graph are not explicity represented.

Example 2.4

Figure 2.4 shows the Adjacency List and Edge List for the graph shown in Figure 2.1.

```
N (1) = 2,1Ø                    1: (1,2)     11: (7,1Ø)
N (2) = 1,7                     2: (1,1Ø)    12: (7,9)
N (3) = 4,8,9,1Ø                3: (2,7)     13: (3,8)
N (4) = 3,8,9,1Ø                4: (5,1Ø)    14: (4,8)
N (5) = 7,1Ø                    5: (5,7)     15: (3,4)
N (6) = 9,1Ø                    6: (4,1Ø)    16: (6,9)
N (7) = 2,5,9,1Ø                7: (8,1Ø)    17: (3,9)
N (8) = 3,4,9,1Ø                8: (3,1Ø)    18: (8,9)
N (9) = 3,4,6,7,8,1Ø            9: (9,1Ø)    19: (4,9)
N(1Ø) = 1,3,4,5,6,7,8,9        1Ø: (6,1Ø)

      (a)                             (b)
```

(a) **ADJACENCY LIST** **FIG 2.4**
(b) **EDGE LIST**

(end of example)

2.4 GRAPHS AND BOOLEAN MATRICES

Definition 2.25

A closed semiring is an algebraic structure (S,+,.,Ø,1) on a set S which defines two binary operations, denoted by + and . respectively, such that for all a,b,c in S

1. a+b is in S and a.b is in S
   (+ and . are closed)
2. (a+b)+c = a+(b+c) and (a.b).c = a.(b.c)
   (+ and . are associative)
3. a+b = b+a (+ is commutative)
4. a+a = a (+ is idempotent)
5. a+Ø = Ø+a = a for every a in S
   (Ø is the + identity)
6. a.1 = 1.a = a for every a in S
   (1 is the . identity)
7. a.Ø = Ø.a = Ø (Ø is an annihilator)
8. a.(b+c) = a.b + a.c and (a+b).c = a.c + b.c
   (. distributes over +)
9. If $a_1,a_2,..,a_i,...$ is a countable sequence of elements then $a_1+a_2+..+a_i+...$ exists and is unique
10. . distributes over countably infinite sums as well as finite sums

A closed semiring of particular importance for this work is the semiring $S'=(\{Ø,1\},+,.,Ø,1)$ with addition and multiplication tables as follows:

+	Ø	1
Ø	Ø	1
1	1	1

.	Ø	1
Ø	Ø	Ø
1	Ø	1

Boolean matrices are defined over this semiring and so they can be related to the adjacency matrix of a graph.

Let G=(V,E) be a directed graph, where $V=\{v_1,v_2,..,v_n\}$. For a labelling function L:(VxV)->{Ø,1} over the elements of S' such that

$$L(v_i,v_j) = \begin{cases} 1 \text{ if } (v_i,v_j) \text{ is in } E \\ Ø \text{ otherwise} \end{cases}$$

the graph G can be represented by the nxn matrix A_G whose ijth entry is $L(v_i,v_j)$. The matrices A_G are also defined over the semiring S'.

2.5 TREES

Trees have been defined and analysed by Knuth [1973] and their use in many algorithms is described by Aho, Hopcroft and Ullman [1974].

Definition 2.26

A connected acyclic directed graph is a directed tree if

1. There is exactly one node, called the root, which no edges enter.
2. Every node except the root has exactly one entering edge.
3. There is a unique path from the root to each node.

A tree with n nodes has exactly n-1 edges.

Definition 2.27

If (v,w) is an edge in a tree, then v is the parent of w and w is a child of v. If there is a path from v to w then v is an ancestor of w and w is a descendent of v. A node with no

descendents is called a leaf.

Definition 2.28

The depth of a vertex v in a tree is the length of the path from the root to v. The height of a vertex v in a tree is the length of a longest path from v to a leaf. The level of a vertex v in a tree is the length of the path from a leaf to v.

Definition 2.29

An ordered tree is a tree in which the children of each node are ordered. A ordered binary tree is an tree such that:

1. each child of the node is distinguished as either the left child or the right child and
2. no node has more than one left child and one right child.

An ordered log tree is an tree such that no node has more than $\log_2 n$ children.

When drawing an ordered tree, the children of each vertex are assumed to be ordered from left to right.

Definition 2.30

A traversal of a tree visits each node of the tree and many algorithms which use trees traverse them. A depth first traversal of an ordered tree starts at the root of the tree and recursively visits all children of a node before visiting any sibling of the node. A breadth first traversal of an ordered tree starts at the root and visits all nodes at the same depth in the tree before visiting any node at a greater depth.

2.6 PROBABILITY DISTRIBUTION

A probability distribution for random graphs is used in the analysis of the average time complexity. Erdos and Renyi [1959] present two equivalent graphical distributions.

Definition 2.31

$G_{n,p}$ are the graphs of n vertices where each edge is chosen independently of other edges and is present with probability p.

$G_{n,N}$ are the graphs of n vertices and N edges where the N eges are distributed at random through n(n-1) possible edges.

The complexity analysis of Chapter 5 will use the first distribution. For an adjacency matrix A of a graph G=(V,E), the probability that an edge exists is

$$\text{Probability}(a_{ij} = 1) = p$$

and the probability that an edge does not exists is given by

$$\text{Probability}(a_{ij} = \emptyset) = 1-p$$

From which it follows that the probabiltiy of k independent edges existing is

$$\text{Probability(k independent edges exist)} = p^k$$

and the probability of k independent edges not existing is

$$\text{Probability(k independent edges not existing)} = (1-p)^k$$

Furthermore, the probability of m edges existing from a partition of the graph that has z possible edges is

$$\text{Probability(m edges in a z partition)} = \binom{z}{m} p^m (1-p)^{z-m}$$

2.7 MAXIMAL INDEPENDENT SETS

Definition 2.32

A set $C \subseteq V$ such that for all $v,w \in C$ implies $(v,w) \notin E$ is an independent set. If C is not contained in any other independent set then C is called a maximal independent set. The set $C' \subseteq V$ such that for all v,w in C implies (v,w) in E is a complete subgraph. If C' is not contained in any other complete subgraph then C' is a clique.

Moon and Moser [1965] characterised the graphs having the maximum possible number of cliques for any given number of vertices. They showed that for a graph of size n the maximum number of cliques c(n) is exponential in n and is given by

$$c(n) = \begin{cases} 3^{n/3} & n=0 \pmod 3 \\ 4*3^{(n-4)/3} & n=1 \pmod 3 \\ 2*3^{(n-2)/3} & n=2 \pmod 3 \end{cases}$$

There exists only one (not including all possible isomorphisms) such graph for each value of n. The minimum possible number of independent sets in a graph is one and the graph has n vertices and 0 edges.

The complements of these graphs will have the same number of maximal independent sets as cliques in the original graphs, and this will also be the maximum possible number. For any value n, the graphs contain $\lfloor n/3 \rfloor$ disjoint polygons on three vertices and, if n is not a multiple of 3, the remaining vertices are connected but disjoint from the polygons of the graph. These complement graphs will be used to test algorithms for finding all maximal independent sets of a graph and the original graphs will be used to test algorithms for finding all cliques of a graph. The graphs will be collectively called the Moon-Moser graphs.

However, these graphs give no indication of the average number of independent sets in a graph, or of the probability of the graph's occurrance. Matula [1970] and Bollobas and Erdos [1976] have studied complete subgraphs and cliques in random graphs. Bollobas and Erdos use the random graph G_n on n vertices, where each edge is present with probability p independently of all other edges, where $0 < p < 1$ and p is fixed. If $Z_r = Z_r(G_n)$ is the number of cliques of order r in G_n and if $d = dn$ is the positive real number for which

$$\binom{n}{d} p^{\binom{d}{2}} = 1$$

$$\text{then} \quad d(n) = \frac{2*\log n}{\log(1/p)} + O(\log_{1/p}\log_{1/p} n)$$

Bollobas and Erdos state

> "that cliques of order essentially greater than dn are unlikely to occur . . . and that cliques of order roughly less than 1/2dn are also unlikely to occur, but every other value is likely to be the order of a clique . . .that the orders of cliques occurring are almost exactly the numbers between . . . ~1/2d(n) and d(n)".

Since Probability(r given vertices form a clique in G_n)

$$= (1-p^r)^{n-r} p^{\binom{r}{2}}$$

the expected number of cliques of order r in G_n (the expectation of Z_r) is

$$E(Zr) = \binom{n}{r} (1-p^r)^{n-r} p^{\binom{r}{2}}$$

Then, the expected total number of cliques in the random graph G_n, will be the sum of the expected number of cliques of each possible size.

$$E(c) = \sum_{r=1/2d}^{d} E(Zr)$$

$$= \sum_{r=\log_{1/p} n}^{2\log_{1/p} n + O(\log_{1/p}\log_{1/p} n)} \binom{n}{r} (1-p^r)^{n-r} p^{\binom{r}{2}}$$

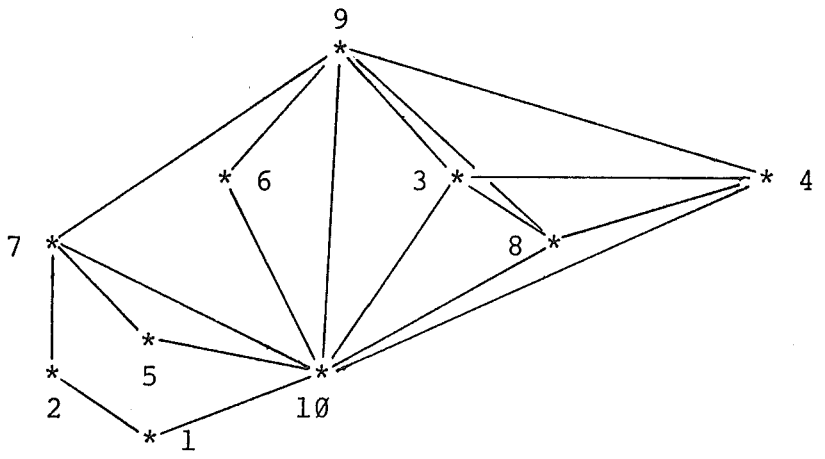
Clearly, $E(c) << 3^{n/3}$.

Thus, the bound for the expected number of cliques or maximal independent sets in the random graph is polynomial and much less than the exponential bound for the worst case, however the number is still very large for small n.

Example 2.5

The graph shown in Example 2.1 has the following independent sets.

(2,1Ø)
(1,5,9)
(2,5,9)
(1,3,5,6)
(1,3,6,7)
(1,4,5,6)
(1,4,6,7)
(1,5,6,8)
(1,6,7,8)
(2,3,5,6)
(2,4,5,6)
(2,5,6,8)



(a) (b)

(a) **THE 12 MAXIMAL INDEPENDENT SETS OF A** **FIG 2.5**
(b) **GRAPH ON 1Ø VERTICES AND 19 EDGES**

(end of example)

CHAPTER 3

A SURVEY OF GRAPH REPRESENTATION

3.1 INTRODUCTION

All non-topological representations of a graph are based on the implicit assumption that the elements of the graph can be arbitrarily labelled. Usually the vertices of the graph $G=(V,E)$ with $n=|V|$ and $m=|E|$, are labelled with the natural numbers 1..n and the edges are labelled with the natural numbers 1..m. The traditional graph representations used prior to the application of computing are based on the above assumption and have been successfully transfered to computer algorithms to give the best known representations for some graph problems.

The data structures used to represent a graph are fundamental to the efficiency of the solution of graph problems. Data structures such as lists, trees and queues, which have been extensively used throughout computer science, have also been successfully applied to graph problems. In particular, some of the many applications of tree structures to graph algorithms are surveyed.

Two closely related methods for solving graph problems, decomposition and shrinking, are related to the algebraic concepts of subalgebra and homomorphism. Both methods are generally termed decompositions. These methods are not restricted to graph problems, as decomposition techniques have been presented for varying combinatorial classes. There are a number of combinatorial optimisation problems which can be solved more efficiently on the composition graph by solving similiar problem instances on the smaller graphs. Decompositions of graphs, based on both decomposition and homomorphisms of graphs, and their application to graph algorithms are surveyed in this chapter.

3.2 REPRESENTATION OF GRAPHS

The graph representations discussed in Chapter 2, using matricies and lists, are conveniently adapted to data structures within conventional programming languages using, in the simplest form, only arrays. Furthermore, in section 2.5 the relationship of adjacency matrices to the closed semiring of boolean algebras was detailed, so the adjacency matrix can be manipulated according to the laws of algebra which are far better established than the laws of graph manipulation.

Matrix or list data structures are efficient and 'natural' representations of some graph problems. Multiplying boolean matrices is equivalent to computing the transitive closure of a graph [Fisher and Meyer, 1971], although there are far

less laborious methods for computing the closure than computing the successive powers of the adjacency matrix, and many path problems are algebraically equivalent to the determination of one or more elements of the weak or strong closure of an adjacency matrix [Carre, 1979]. The solution methods discussed by Carre range from substitution through Gaussian and Jordan elimination to Dijkstra's algorithm and all use matrix or list data structures. The theoretical framework used for the formulation and solution of path problems is a path algebra. Path algebra are defined for accessible sets, shortest paths, critical paths, most reliable paths, paths of greatest capacity and enumeration of paths in a graph. Mahr [1981] surveys shortest path problems and their general solution techniques and the adjacency matrix also dominates these solutions.

A matrix representation does not give the best known solutions for all graph problems, for many of which an efficient computer representation requires data structures tailored to the problem area. The history of algorithm design has many milestones marked by the development of new data structures which have led to new and efficient algorithms [Aho, Hopcroft and Ullman, 1974]. The most recent was the development of the Fibonacci-heaps applied to shortest path problems [Fredman and Tarjan, 1984] and since used for efficient algorithms for finding minimum spanning trees, maximum weighted matchings and optimum branchings [Gabow, Galil and Spencer, 1984].

Among the earliest data structures to have a fundamental effect on the design of efficient algorithms was the tree. Nievergelt [1979] describes trees as:

> "Thus, trees have emerged as THE class of list structures which are most widely used, and are understood best from a theoretical point of view. Knuth . . . collected and classified the accumulated knowledge on data structures [1968,1973] . . . Of all the data structures considered, trees claim the lions share of the space."

The tree data structure is closely linked to an algorithm that has been used extensively to systematically search a graph, the depth first search which derives an underlying tree substructure from the graph. The tree created by depth first search was termed a palm tree with fronds. The fronds are back links in the tree and they determine the connectivity of the original graph. Depth first search has been used to find the biconnected components of an undirected graph in O(m) time [Hopcroft, 1971], the triply connected components in linear time [Hopcroft and Tarjan, 1973], and the strongly connected components of a graph in O(max(n,m)) time by adding cross links [Tarjan, 1972]. It has also been used in the path addition algorithm for testing graph planarity in linear time [Hopcroft and Tarjan, 1974], the best known algorithm for finding dominators in graphs [Tarjan, 1974] and a test for flow-graph reducibility in linear time by adding path compression which redefines the root of the tree to achieve a balanced tree structure [Tarjan, 1973].

In general, problems on trees are simpler to solve than problems on more general graphs. Indeed, some NP-complete problems have polynomial time solutions when restricted to trees.

The identification of particular properties of a class of graphs establishes a restriction on the complexity of the graphs that can be used to develop efficient algorithms for that class of graph. This is apparent in the clique problem. The COVERING BY CLIQUES problem, finding all the cliques in a graph, is NP-complete. The PARTITION INTO CLIQUES problem, is solvable in polynomial time for graphs containing no complete subgraphs on 3 vertices, for circle arc graphs, chordal graphs and for comparability graphs, and the CLIQUE subgraph problem is solvable in polynomial time for graphs obeying any fixed degree bound, for planar graphs, edge graphs, chordal graphs, comparability graphs, circle graphs and for circle arc graphs. [Garey and Johnson, 1979].

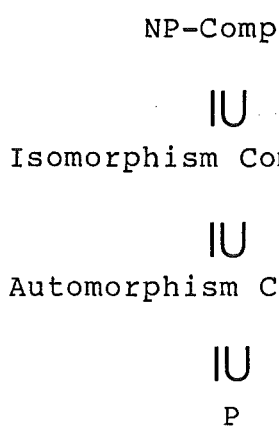
3.3 GRAPH DECOMPOSITION

The limitation to a specific class (or classes) of graphs has applied to all decomposition theorems developed to date. Any well-defined decompostion theory will be valid for all graphs, however in the worst case the only possible decompositions will be to single vertices or the whole graph and for these graphs the decomposition method will not provide an effective decomposition. For the class of graphs which can be effectively decomposed, the method can provide

efficient solutions. The algorithms of Gabow, Galil and Spencer [1984] discussed in section 3.2, use Fibonacci-heaps to maintain and merge the components of a graph and from this some very fast and efficient contraction algorithms have been developed.

The use of totally ordered label sets implies an ordering relationship within the graph which does not exist. Furthermore, since any labelling is valid, any ordering relationship can be established. This reordering is equivalent to the graph isomorphism problem. The graph structure is lost within these representations as are all graph parameters except the number of nodes and the number of edges and even the latter is lost in an adjacency matrix. Frucht [1970] introduced a notational representation of graphs that is independent of any labelling. The graph is represented by the orbits under the cyclic subgroups of the automorphism groups of the graph. The method is succienct and readily translated to computer storage.

The isomorphism and automorpism problems hold special places in computational complexity. They are known to be in NP but have never been shown to be NP-complete. Unfortunately, they are not known to be solvable in polynomial time. The relationship is shown in Figure 3.1. Furthermore, the automorphism partition problem is polynomially equivalent to the isomorphism problem and hence is isomorphisn complete. Thus, there are practical problems associated with an automorphism partition of graphs.



THE COMPLEXITY HIERARCHY **FIG 3.1**

A simplical decomposition of a graph is based on the cliques of a graph. The components of the decomposition are simplicies where a simplex is a complete graph. Simplical decompositions were discussed by Halin [1978] and have been used by Halin [1982] to solve some colouring and connectivity problems and he reports that they were first used extensivly by Wagner studying the Four-Colour Problem in 1937. The representation of a graph by its simplical decompostion is considered by Atkin [1972] and Atkin and Casti [1977]. No algorithm for finding the decomposition of a graph is given, so the available methods are those for finding the complete subgraphs (cliques) or the maximal independent sets of a graph. This problem is known to be NP-complete for general graphs, but solvable in polynomial time for specific classes of graphs [Garey and Johnson, 1979].

3.4 GRAPH HOMOMORPHISM

More successful attempts at decomposing a graph have been based on the homeomorphisms and homomorphisms of a graph. If part of a graph is shrunk to a single vertex and a problem is solved on the shrunken graph then the solution on the original graph can be computed from the solution on the smaller graph. The shrinking corresponds to taking a homeomorphic image of the graph. A homomorphism usually can be defined only on the paths of a graph, rather than the edges of a graph.

The usual graph homeomorphism is an adjacency homeomorphism as used in Kuratowski's theorem. Perl [1980] used this to define a reachability homomorphism on directed graphs. However, this definition allows the introduction of new reachablities and Ebert and Perl [1981] point out that this problem has not yet been solved.

A digraph homomorphism is described by Robinson [1983] and is based on the algebra of series and parallel relations. While this homomorphism maintains reachabilities and flows, the algebra of series and parallel relations has no distributive laws and thus the application of the homomorphism may not, in general, achieve a minimum. Lawler [1976] successfully used this homomorphism to prove tractable results for many NP-complete problems when restricted to the transitive series parallel digraphs.

Sabuddissi [1961] defined the X-join of a family of graphs. The X-join is a composition operator on graphs that obtains, from the graph X, a composition graph by substituting each vertex of X by the graph G_x in the set. Every graph is isomorphic to an X-join of some family of graphs, but possibly only a trival set. That is, either X is trival or all G_x are. If a graph is isomorphic to an X-join of some family of graphs and both X and some member of the family are non-trival then the graph is decomposable.

James, Stanton and Cowan [1972] reported an algorithm for decomposing graphs with respect to the X-join. The algorithm requires $O(n^4)$ time. Habib and Maurer [1979] presented an $O(n^3)$ algorithm for decomposing undirected graphs.

Pfaltz [1972] defined a homomorphism on directed acyclic graphs, and showed how to retain the ability to reconstruct the original graph from the sequence of contracted graphs. The homomorphism developed from the study of an equivalence relation, over nodes of the graph, that is dependent on preserving adjacencies.

Eftimie and Eftimie [1977] observed that the graph homomorphism used by Pfaltz did not preserve the basis of graphs and that its kernal was not a congruence. They redefined the homomorphism used by Pfaltz to capture these properties, thereby embedding the results in the general framework of universal algebra. The representation becomes an application of the decomposition theorems for algebraic structure by considering a graph as an algebra whose

operator domain consists of its relations.

A similar approach was used by Cardon and Crochmere [1982]. They further gave an algorithm for finding the graph homomorphism in $O(m*\log_2 n)$.

Cunningham and Edmonds [1980] generalises these theorems in a unifying decomposition theory. Non-separable graphs are defined and a simple decomposition operator is developed from which a unique minimal decomposition can be defined. Cunningham [1982] presents the composition for digraphs and shows that a general decomposition theory can be applied to the resulting digraph decomposition which is unique as each member of the decomposition is either not decomposable or belongs to a class of special digraphs which are completely characterised. Algorithms for the decompostion requiring $O(n^4)$ time and $O(m)$ space are also presented. For special classes of graphs the complexity is reduced to $O(n^3)$ time and order $O(n+m)$ space.

The algebraic nature of graphs and the relations between graphs and groups has also been studied, but no new represntations have developed [Hestenes, 1973]. The relations of a graph (that is the edges) can be more succinctly represented than having each as an algebraic operator. Mahr [1981] defines an equivalence relation over which several classes of graphs can be characterised, but the extent of the application to all graphs and graph representation remains open.

3.5 SUMMARY

Many graph representations translated easily to computational structures for the solution of graph problems and data structures such as trees which are extensively used in other areas of computing have been used to develop the best known solutions for many graph problems.

The search for specifically designed graph structures, and the theoretical development of graph decompositions which has been the subject of much research, have not, in general, led to applications in graph algorithms. In practice, decompositions have generally been restricted to certain classes of graphs, while graph representations based on the automorphism or clique structure of a graph are also restricted to the graphs which do not require exponential time to find the graph structure.

The power and usefulness of graphs is in part due to the lack of restrictions in their definition. Hence, the study of graph representations is often the search for a structure that reflects some specific property of the graph or the problem area. Since this choice depends on the operations that have to be performed it seems unlikely that there is any global solution that is a best representation.

CHAPTER 4

A DECOMPOSITION ALGORITHM FOR SHORTEST PATHS ON NETWORKS

4.1 INTRODUCTION

Many specialised problems within the general framework of path problems have been studied. Christofides [1975] and Carre [1979] give comprehensive studies of the shortest path problem and algebraic path problems respectively. The most common forms of shortest path problems are the 'all pairs' and 'single source' problems. The problem studied in this chapter is a variation of these shortest path problems to find the shortest paths for a selected set of vertices in the network.

A transportation network is a network simulating an existing transportation system and subject to modification, reflecting the changes that occur the system [Dubois, Bel, Llibre 1979]. Generally, a transportation network is a planar graph and the costs on the edges of the network are related to the distances between the vertices in their geometric space. There is frequently more than one cost on the edges of the network and in addition there may be costs attached to the vertices of the network. Movements within the network may be restricted to certain paths upon which a

paticular flow may exist, and to change between these may also incur extra costs. In many applications within transportation studies only a subset of the distance matrix which is output by an all pairs shortest path algorithm is required. Typically, transportation networks contain many hundreds of vertices and only a representative sample of some 5 to 10 percent of these vertices will be required in a shortest paths matrix. Clearly, generating the all pairs solution or a multiple single source solution and extracting a submatrix will involve calculations on many otherwise unused vertices in the network.

Cunningham [1982] states that:

> "There are a number of combinatorial optimisation problems which can be solved (more efficiently) on the composition graph by solving similar problem instances on the smaller graphs."

This chapter examines a decomposition for a network. It presents the decomposition theory and defines a compressed (shrunken) network of the original network. Generating shortest paths on the original network is performed in three stages. First, an algorithm is presented which generates the compressed network and also calculates the shortest paths on the components of the decomposition in O(m) time where m is the number of edges in the network. Second, the all pairs shortest path matrix for the compressed network must be calculated, and any of the efficient established algorithms for shortest paths may be used. For a well defined class of networks, this will require less time and space than for the original network. The components of the decomposition and

the all pairs shortest path matrix of the compressed network can then be combined in the third stage. An algorithm is presented that generates a single pair shortest path on the composition network in linear time, so for any selected set S of vertices of size $s=|S|$, the shortest path matrix for the set can be generated in $O(s^2)$ time.

This chapter also presents the application of the decomposition to transportation networks. The decomposition presented in this chapter is valid for both directed and undirected graphs.

4.2 DECOMPOSITION OF A NETWORK

In this section a decomposition is defined on networks. The decomposition is similar to the homeomorphism used by Kuratowski's Theorem for planar graphs [Even 1979]. It is also similar to the o contractions described by Pfaltz [Pfaltz 1972]. The development followed here is that presented by Tutte [1966]. Tutte defined a decomposition on a graph that partitioned the graph into nodes and branches. For the decomposition presented in this chapter the definitions are first extended to a network. To avoid confusion with the more common use of the term node as a vertex in a directed graph, the term link vertex will be used.

Definition 4.1

Let G=(V,E) be a network. Divide the vertices of G into two classes:

(1) the simple vertices are adjacent to exactly two other vertices and either $r^+(v) = r^-(v) = 1$ or $r^+(v) = r^-(v) = 2$.

(2) the link vertices are the remaining vertices in the network.

The set of all link vertices of G is denoted by L.

Definition 4.2

The edges of the network are partitioned into simple paths between a link vertex x and a link vertex y in G and all vertices in the path other than x and y are simple vertices and for every pair of simple vertices v_i and v_j in the same simple path partition, $r^+(v_i) = r^+(v_j)$ and $r^-(v_i) = r^-(v_j)$. The edges and vertices in each partition are called the branches of G. Divide the branches of G into three classes:

(1) the simple paths of length 1 are the trivial branches of G.

(2) the simple paths of length greater then 1 which are polygons, that is link vertex x = link vertex y, so the edges form a cycle from a link vertex x L back to x.

and (3) the simple paths of length greater than 1 which are not polygons, are the connecting branches of G.

The set of all branches of G is denoted by B.

Each branch of a network starts at a link vertex. If it is a trivial branch then there will be exactly two vertices (both link vertices) and either one or two edges in the branch. If the branch is a connecting branch or a polygon it will have an edge to an adjacent simple vertex and the branch will include the simple path from the starting link vertex through simple vertices until any other link vertex is reached. If there is an edge from the simple vertex to the starting link vertex then all simple vertices in the branch will include edges incident to both adjacent vertices.

Example 4.1

Figure 4.1 shows a portion of a transportation network that includes 5 link vertices and 9 branches, 1 of which is a polygon and 8 connecting branches. The bidirectional edges of the network are shown as undirected edges in Figure 4.1.

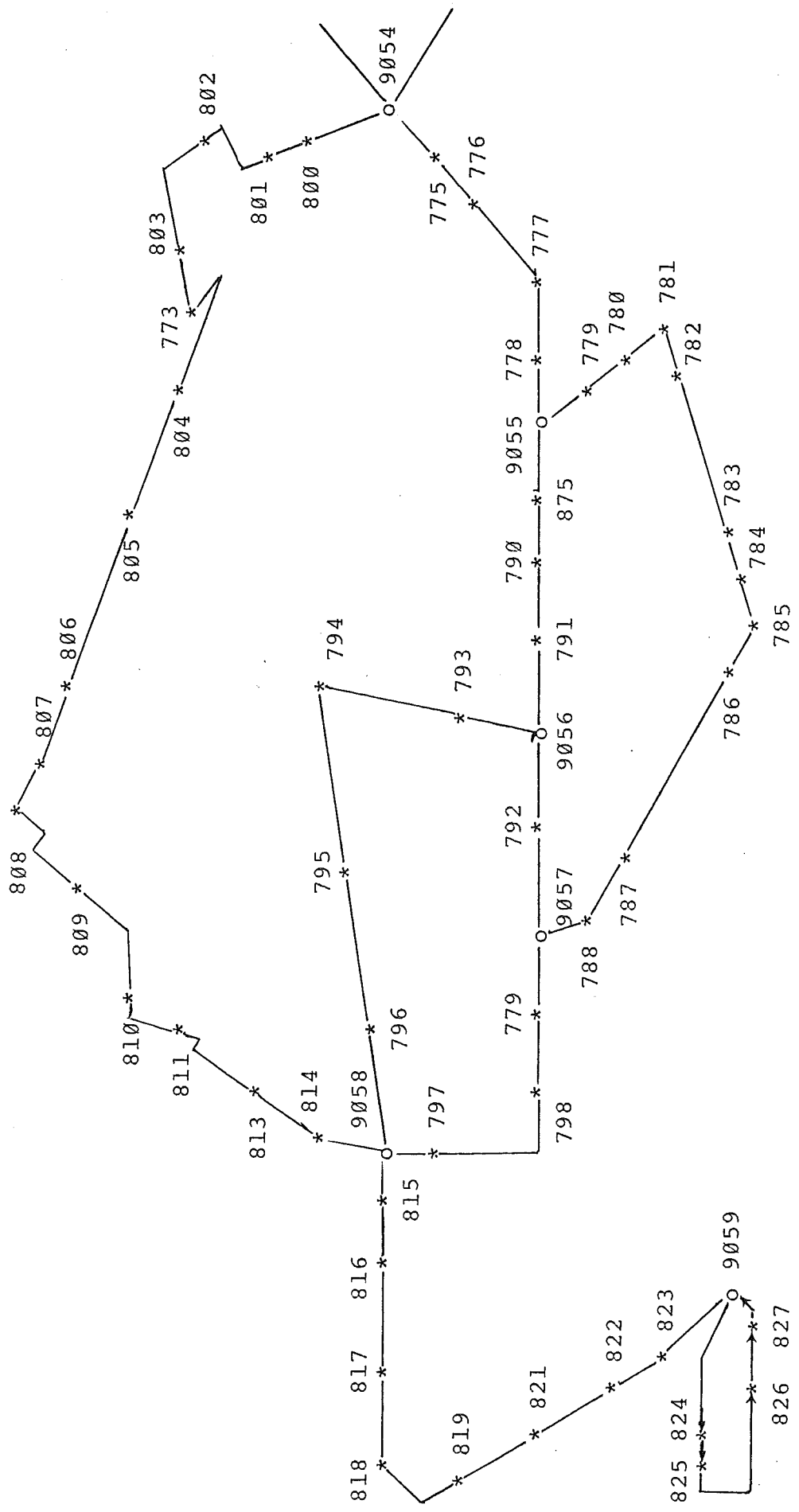
The link vertices of Figure 4.1 are

{9Ø54,9Ø55,9Ø56,9Ø57,9Ø58,9Ø59}

The branches of a network include vertices and edges. Two branches of the network portion of Figure 4.1 are shown in Figure 4.2. These are the branches whose vertices are:

{9Ø55,779,78Ø,781,782,783,784,785,786,787,788,9Ø57}

and {9Ø59,824,825,826,827,9Ø59}.



A PORTION OF A TRANSPORTATION NETWORK

FIG 4.1

The vertices of the remaining branches which are shown completely in Figure 4.1 are:

{9Ø54,775,776,777,778,9Ø55}

{9Ø55,875,79Ø,791,9Ø56}

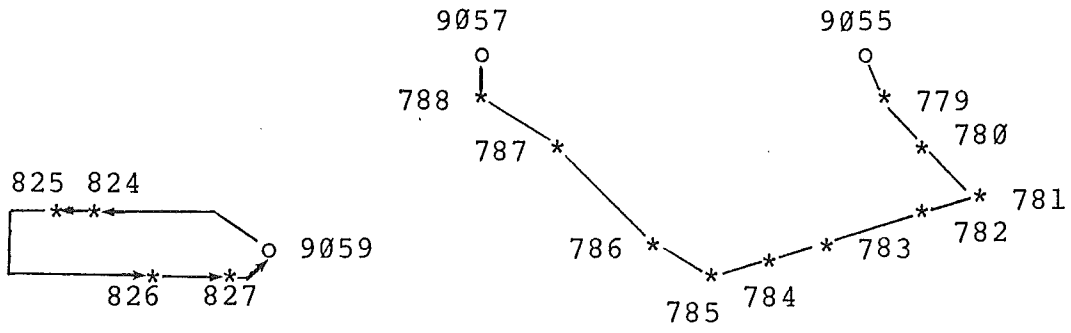
{9Ø56,792,9Ø57}

{9Ø56,793,794,795,796,9Ø58}

{9Ø57,799,798,797,9Ø58}

{9Ø54,8Ø0,8Ø1,8Ø2,8Ø3,773,8Ø4,8Ø5,8Ø6,8Ø7,8Ø8,8Ø9,81Ø,
811,813,814,9Ø58}

{9Ø58,815,816,817,818,819,821,822,823,9Ø59}



TWO BRANCHES IN THE NETWORK **FIG 4.2**

(end of example)

Definition 4.3

Let the vertices of a network G=(V,E) be labelled 1..n, where n=|V|. For any trivial or connecting branch that has a path from link node x to link node y and from y to x where x < y, the edges from x to y form the forward path, and the edges from y to x form the backward path. For a polygon the choice of forward and backward paths is arbitrary.

The decomposition can now be defined on the vertices and edges of the network.

Definition 4.4

A decomposition D of a network G=(V,E) divides the vertices and edges of G into the branches of G. The components of the decomposition are each of the branches of G.

Each distinct branch of the decomposition is itself a network. Several theorems on the nodes and branches of a graph were proved by Tutte and these extend naturally to the link vertices and branches defined on a network.

Theorem 4.1

Let X be any branch of a graph G. Then $| V(X) \cap L | <= 2$. If X is incident with two distinct (nodes) link vertices x and y of L, then X is a simple path with ends x and y. In the remaining case X is a polygon.

Proof. Due to Tutte.

(end of proof)

By theorem 4.1 the number of link vertices of G incident with a given branch X is either 2, 1 or Ø. In the first case the branch is a simple path joining two link vertices. In the second case the branch is a polygon and there is only one link vertex in the branch. The third case cannot occur in a connected network.

Theorem 4.2

Let G be a graph without divalent edges. The graph H whose vertices are the (nodes) link vertices of G and whose edges are the branches of G is a true graph and there exists an isomorphism f of G onto H. The graph H is the branch-graph of G.

Proof. Due to Tutte.
(end of proof)

From theorem 4.2 it is possible to show that the decomposition of definition 4.3 is sufficiently general to apply to all networks.

Theorem 4.3

For all networks G=(V,E) there exists a decomposition D of G onto the branches of G.

Proof.
Each vertex of a network is either a simple vertex or a link vertex by definition 4.1. Every edge in a network either connects two link vertices in which case it is a trivial branch, or it connects a simple vertex with another simple vertex or a link vertex, in which case it is on a simple path that is either a polygon or connecting branch of G. Every vertex and edge of any network exists on a branch of the network and the branches of G form a decomposition of G.
(end of proof)

Tutte defines a homeomorphism f of a graph G onto a graph H as a pair {g,h}, where g is a 1-1 mapping of the (nodes) link vertices L(G) onto L(H) and h is a 1-1 mapping of B(G) onto B(H), such that a (node) link vertex x is incident with a branch X in G if and only if gx is incident with hX in H. The relationship between a graph G and its branch graph is a homeomorphism and so is valid for all graphs, although in the trivial case L=V and the edges of the branch graph are simply the edges of the original graph. Tutte defines the properties of the homeomorphism. These include the existence of an identity homeomorphism and an inverse homeomorphism. The extension to a network retains the homeomorphic relationship between a network G and its branch graph and this graph is unique. So, the components of the decomposition are unique and irreducible.

Theorem 4.4

Every network has a unique decomposition D consisting of the branches of G and every branch b_r in B is irreducible.

Proof.

A branch of a network G is a simple path from a link vertex x to a link vertex y. If x and y are distinct then the branch must have at least one edge. If x=y then the branch is a polygon of length > 1, so must have at least one edge. The branches of G partition the edges of G, so no edge can occur in more than one branch. So, each branch of G is uniquely defined by the edges it contains and the decomposition D is unique.

By theorem 4.1 $| V(b_r) \cap L | <= 2$ and the remaining vertices

of b_r, namely $V(b_r) - (V(b_r) \cap L)$, are simple vertices. As every branch of a network must contain at least one edge and be a simple path between link vertices, there is no other branch in b_r so b_r is irreducible.
(end of proof)

The final theorem that must be proved on the decomposition is that the original network can be reconstructed from the components of the decomposition, so a composition on the branches of G must be defined.

Definition 4.5

Let b_1 and b_2 be networks having vertex sets $V(b_1)$ and $V(b_2)$ and edge sets $E(b_1)$ and $E(b_2)$ respectively. A composition of b_1 and b_2 is the composed network

$$N = b_1 \circ b_2 = (V(b_1) \cup V(b_2), E(b_1) \cup E(b_2)).$$

If $| V(b_1) \cap V(b_2) | >= 1$ then N is a connected network.

Theorem 4.5

The original network G can be reconstructed from the components of the decomposition B.

Proof.

Each branch of G is a network and so the composition of definition 4.4 may be applied to two branches of G to form a composed network. Since the branches form a partition on the edges of G, no edge will occur more than once in a composed network. If the branches contain a common vertex, it will only occur once in the union $V(b_1) \cup V(b_2)$. The repeated application of the composition to all the branches

of G will combine all the edges of G that have been partitioned by the branches. Since the network is connected every vertex will be the end-point of some edge, and so all vertices will be included in the composition of all the branches of G.

(end of proof)

4.3 A COMPRESSED NETWORK

For the solution of shortest path problems that will be presented in this chapter, the branch graph of Tutte although it extends to a network, will be altered to give a more efficient compressed network. The compressed network of definition 4.7 differs from the homeomorphism of Tutte by the removal of the polygons from the branch graph of G.

Definition 4.6

For each path in a connecting branch b_c of a network G, from a link node x to a link node y, define a directed link edge $p_c = (x,y)$ from x to y. The cost of the new edge $c(p_c)$ is the sum of the costs of the edges of path in the branch b_c. For the set B of branches of G, the set of edges p_c is denoted by P. If there is more than one link edge $p_c = p_{c'} = (x,y)$ then the cost of the edge (x,y) is $\min(c(p_c),c(p_{c'}))$ and there exists only one edge (x,y) in P.

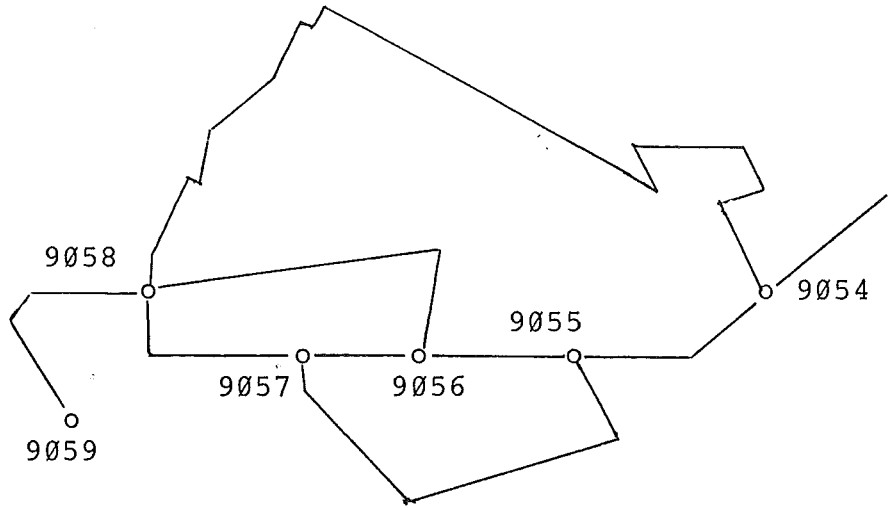
Definition 4.7

A network formed on the link vertices of the network G=(V,E) is the compressed network $G_c=(L, E(L) \cup P)$. The vertices are the link vertices L and the edges are the edges of the subgraph formed on L (equivalently the trivial branches of G) plus a directed link edge p_c replacing every path in the connecting branches of G. The costs of the edges of G_c are:

(1) if $e=(x,y)$ for $x,y \in L$ and $e \in E$ and $e_c=(x,y)$ for $x,y \in L$ and $e_c \in E(L)$ then $c(e_c)=c(e)$.

(2) if $p_c \in P$ then $c(p_c)= \sum c(e)$ for all e in the simple path b_c.

Example 4.2



A PORTION OF THE COMPRESSED NETWORK **FIG 4.3**

The compressed network of the network portion shown in Figure 4.1 is shown in Figure 4.3. The vertices of the compressed network are the link vertices of the original network, namely {9054,9055,9056,905,9058,9059}. Note the polygon at link vertex 9059 does not cause a self loop in

the compressed network.

(end of example)

The size of the compressed network is determined by the original network and the number of link vertices it contains.

Theorem 4.6

Let G=(V,E) be a network with n = |V| the number of vertices and m = |E| the number of edges in G. If the number of link vertices in G is t, then the number of vertices in the compressed network is t and the number of edges u is

$$m-2*(n-t+1) <= u <= m-(n-t)$$

Proof.

The vertices of the compressed network are the link vertices of G, so there will be t vertices in the compressed network. If a branch of G is a trivial branch then for each edge in the branch (forward and/or backward) there will be exactly one edge in the compressed network. If a branch of G is a polygon containing p simple vertices and 1 link vertex then either p+1 or 2*(p+1) edges of the original network will not occur in the compressed network. A connecting branch of G containing p simple vertices and 2 link vertices with p+1 edges will be replaced by 1 edge and if the branch contains 2*(p+1) it will be replaced by 2 edges. A compressed network with t link vertices has n-t simple vertices deleted so at least n-t edges are deleted and at most 2*(n-t+1) edges are deleted.

(end of proof)

The decomposition that has been defined provides for two essential requirements; namely, that the components of the decomposition are unique and irreducible and that the original network can be reconstructed from the components of the decomposition. The algorithm to generate the compressed network that has been defined on the components of the decomposition will also calculate the costs of the shortest paths on the components of the decomposition and is given in Algorithm 4.1.

A planar network is sparse and so will be presented most efficiently as an adjacency list. The costs of the input network G may either be presented as a matrix or attached to the adjacency list. Every simple vertex occurs in only one branch of the network and on at most two simple paths, one forward and one backward. Every edge will be traversed exactly once forward and once backward by the algorithm, so O(m) time will be required.

Algorithm 4.1

Generating the Compressed Network and the Shortest Paths of the Components of the Decomposition.

Input: A network G and the costs c associated with each edge in the network.

Output: A compressed network and the costs for the components of the decomposition.

Method:

Find the link vertices of G:

for every vertex v_i in G

if not (($r^-(v_i) = r^+(v_i) = 1$) or ($r^-(v_i) = r^+(v_i) = 2$))

then v_i is a link vertex

Find the costs for the link edges and the components:

for every link vertex v_i do

for every $v_{i'}$ in $r^+(v_i)$ do begin

follow a simple path
calculating the cost of the shortest path
from v_i to every simple vertex in the path

(and ultimately the cost of the link edge)
until another link vertex $v_{i''}$ is reached

if the path is not a polygon then
store the total cost of the path as the cost of the
link edge (v_i,$v_{i''}$) in the compressed network

trace backward through the simple path from $v_{i''}$ to v_i

calculating the cost of the shortest path
from every simple vertex in the path to $v_{i''}$

until link vertex v_i is reached
end

(end of method)

The cost of the shortest path from a link vertex to a simple vertex and from the simple vertex to a link vertex may be attached to the vertex, so all the shortest paths of the components of the decomposition can be stored in O(n) space. Algorithm 4.1 steps through the link vertices in numerical order, so the forward path of any branch will be traversed first. Therefore, while the direction of the path is unknown at the start of its traversal, if no previous path has been traversed it can be assumed to be forward. Storing the start link vertex with the cost of the shortest path from the start link vertex to the simple vertex and the ending link

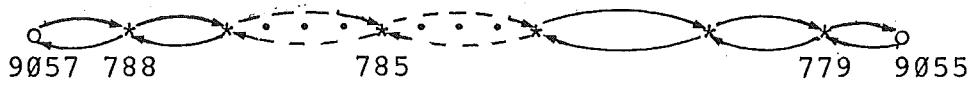
vertex with the cost of the shortest path from the simple vertex to the end link vertex will completely define the path direction.

Two distinct branches b_1 and b_2 may occur with the same link vertices; that is $|V(b_1) \cap V(b_2)| = 2$. For all simple vertices in both b_1 and b_2 the start link and end link of the vertex will be the same although they do not occur in the same branch. During the decomposition process each branch of simple vertices is traced, so if two vertices are in the same branch they may be labelled with the same path identification. This will distinguish between two vertices with the same link vertices, but on different branches of the decomposition.

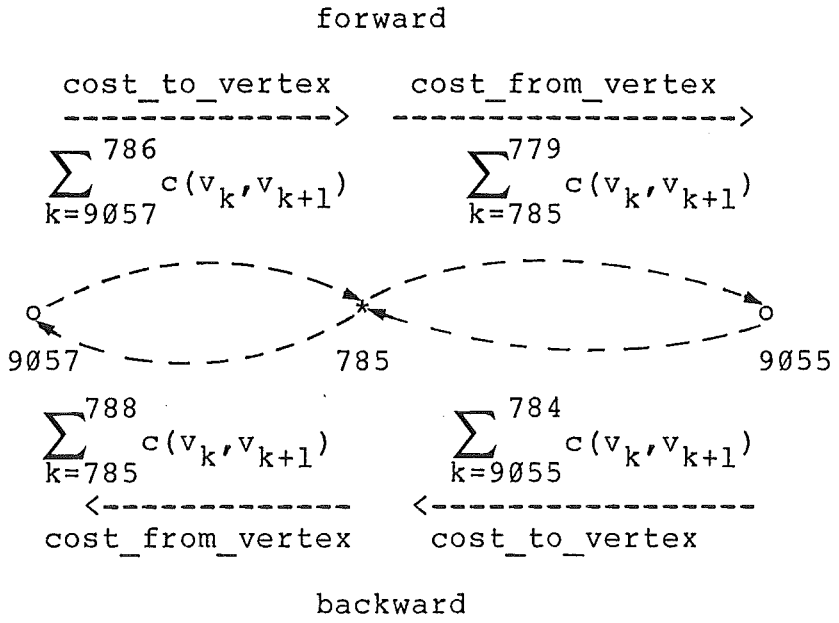
Example 4.3

Figure 4.4 (a) shows the simple vertices on a forward and backward path between two link vertices 9Ø55 and 9Ø57. The shortest paths from each link vertex to the simple vertex 785 are shown in Figure 4.4 (b) as the 'cost_to_vertex', and the shortest paths from the simple vertex to the link vertices are shown as the 'cost_from_vertex'.

The shortest path costs for simple vertices may be stored for each vertex in a record with the starting and ending link vertices for the path and a path identification number. Figure 4.5 (a) shows a record layout suitable for the component_costs and Figure 4.5 (b) shows the storage of forward and backward costs using this data structure for two



(a)



(b)

SHORTEST PATHS IN THE COMPONENTS OF THE DECOMPOSITION **FIG 4.4**

of the simple vertices shown in Figure 4.4. The data structure for the shortest path costs is shown as the two dimensional array partial_cost of the record of component_costs. The first dimension of partial_cost is the vertex the costs relate to, and the second dimension is for the forward or backward path.

```
record component_costs =
                start_link       : link vertex ;
                cost_to_vertex   : real ;
                end_link         : link vertex ;
                cost_from_vertex : real ;
                path_id          : integer
                end
```

(a)

partial_cost[v_s,forward]

	start_link	cost_to_vertex	end_link	cost_from_vertex	path_id
779	9055	9055-->779	9057	779-->9057	1
785	9055	9055-->785	9057	785-->9057	1
787	9055	9055-->787	9057	787-->9057	1

partial_cost[v_s,backward]

	start_link	cost_to_vertex	end_link	cost_from_vertex	path_id
779	9057	9057-->779	9055	779-->9055	2
785	9057	9057-->785	9055	785-->9055	2
788	9057	9057-->788	9055	788-->9055	2

(b)

DATA STRUCTURE OF PARTIAL COSTS FOR COMPONENTS OF THE DECOMPOSITION **FIG 4.5**

(end of example)

Defining the data structures for the component of the decomposition allows Algorithm 4.1 to be stated explicitly. Algorithm 4.2 shows the calculation and storage of the shortest path costs for the decomposition and the creation of the compressed network.

Algorithm 4.2

Generating the Compressed Network and the Shortest Paths of the Components of the Decomposition.

Input: A network G and the cost c associated with each edge in the network.

Output: A compressed network G_c with cost c' associated with each edge in the compressed network and the costs for the components of the decomposition partial_cost.

Initialisation:

```
for every vertex v_i do
  for each direction do
    partial_cost[v_i,direction].start_link := ∅
    partial_cost[v_i,direction].cost_to_vertex := ∞
    partial_cost[v_i,direction].end_link := ∅
    partial_cost[v_i,direction].cost_from_vertex := ∞
    partial_cost[v_i,direction].path_id := ∅
path_count := ∅
```

Method:

```
for every vertex v_i in G do

  if not ((r⁻(v_i) = r⁺(v_i) = 1) or (r⁻(v_i) = r⁺(v_i) = 2))
    then begin
     add v_i to the vertices of G_c

     for j := 1 to |r⁺(v_i)| do c'_{i,j} := ∞
     end

for every link vertex v_i do begin

  for every vertex in r⁺(v_i) do begin

    let the vertex be called v_{i+1}

    cost := c_{v_i,v_{i+1}}

    path_count := path_count + 1
    if partial_cost[v_{i+1},forward].start_link = ∅ then
                                        direction:= forward
    else direction := backward
    k := ∅
    do begin
       k := k + 1
       find (a vertex in r⁺(v_{i+k})) ≠ v_{i+k-1}

       let the vertex be called v_{i+k+1}

       cost := cost + c_{v_{i+k},v_{i+k+1}}

       partial_cost[v_{i+k},direction].start_link := v_i

       partial_cost[v_{i+k},direction].cost_to_vertex := cost

       partial_cost[v_{i+k},direction].path_id := path_count
       end
      until v_{i+k+1} in G_c

    if v_{i+k+1} ≠ v_i then

      add the edge (v_i,v_{i+k+1}) to G_c with edge cost
            c'_{v_i,v_{i+k+1}} := min(c'_{v_i,v_{i+k+1}},cost)

    v_j := v_{i+k+1}

    do begin
       k := k - 1
       partial_cost[v_{i+k},direction].end_link := v_j

       partial_cost[v_{i+k},direction].cost_from_vertex := cost

                - partial_cost[v_{i+k},direction].cost_to_vertex
       end
      until v_i is reached
    end
  end
(end of method)
```

The compressed network that is generated by Algorithm 4.2 will contain only the link vertices of the original network and so will be no larger than the original network and for some networks will be considerably smaller. If the original network is planar then by Kuratowski's Theorem the compressed network will also be planar and hence sparse. The all pairs shortest path algorithms most efficient for sparse graphs or networks is the repeated application of Dijkstra's single source shortest path algorithm which requires $O(n^2+m)$ time and $O(n)$ space per source vertex. Using the network sizes defined in Theorem 4.5, Dijkstra's Algorithm applied to the compressed network will produce the shortest path cost matrix (compressed_cost) in $O(t*(t^2+u))$ time and $O(t^2)$ space. Recently, Fredman and Tarjan [1984] have presented an $O(n^2*\log n)$ algorithm for shortest paths, but it has not been implemented with this decomposition.

4.4 SELECTED SHORTEST PATHS

The shortest paths from the compressed network and the components of the decomposition of the original network can be combined by a very efficient and simple algorithm to generate the shortest path from any source vertex v_s to any destination vertex v_d in the network. A vertex v_s in a component of the decomposition is on a branch with at most two link vertices and these are stored in the start_link and end_link of v_s in the array partial_cost. If v_s is a link vertex then it has no entries in partial_cost and these will be interpreted as zero values. The generation of a shortest

path from any source to any destination vertex is shown in Algorithm 4.3.

Algorithm 4.3

Outline of Algorithm for Generating the Shortest Paths

Input: Partial costs for components of decomposition,
Shortest path matrix for compressed network,
Source vertex v_s and destination vertex v_d.

Output: Shortest path cost from v_s to v_d.

Initialisation:

cost := ∞

Method:

if v_s and v_d are in the same component then
cost := minimum of two possible paths
else cost := minimum of four possible paths

(end of method)

Two vertices v_s and v_d are in the same component if they have the same path number. They will also have the same start and end link vertices.

If the vertices v_s and v_d are in the same component then they will have identical link vertices. There are two possible paths that may be the shortest path from v_s to v_d. Firstly, the path that lies completely within the component from v_s to v_d. This may be either a forward or backward path. The cost of the path can be calculated as the difference between the partial cost of the path from v_s through v_d to a link vertex and the partial cost of the path from v_d to the same link vertex. The cost from v_s to the

link vertex must be greater than the cost from v_d to the link vertex so the direction of the path is immediately established. Secondly, the shortest path may be the path from v_s to the link vertex that does not pass through v_d followed by the shortest path in the compressed network between the two link vertices followed by the path from the other link vertex to v_d. The cost of this alternate route is the sum of the costs of each of the sections of the path. If v_s is a link vertex then the cost from v_s through v_d to the other link vertex will not be available if the branch is not the shortest path from v_s to the other link vertex of v_d, but the cost of the path is exactly the cost from the link vertex v_s to v_d. If v_d is a link vertex then the cost from the destination to the link vertex does not exist, but the cost of the path is exactly the cost from v_s to the link vertex v_d.

Example 4.4

Figure 4.6 shows the two possible paths that may be the shortest path from a source to a destination vertex in the same component. The component illustrated is the connecting branch between link vertices 9Ø55 and 9Ø57 from Figure 4.2. When the source vertex is 788 and the destination vertex is 779 the first cost in Figure 4.6 (a) is the difference

```
partial_cost[788,backward].cost_from_vertex -
        partial_cost[779,backward].cost_from_vertex
```

and the second cost in Figure 4.6 (b) is the sum

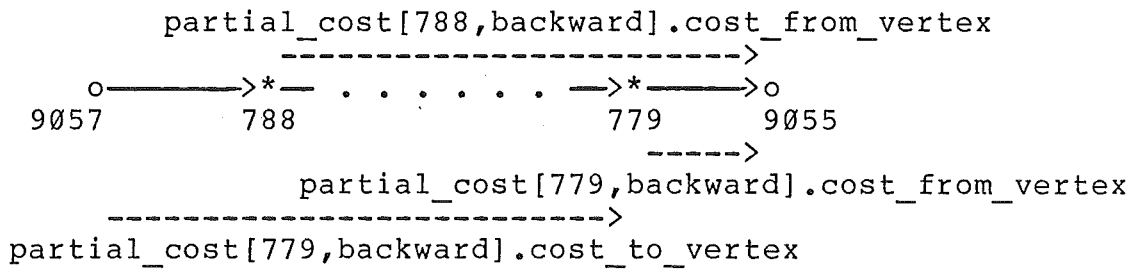
```
partial_cost[788,forward].cost_from_vertex +
        compressed_cost[9Ø57,9Ø55] +
        partial_cost[779,forward].cost_to_vertex
```

When the source vertex is 9Ø57 and the destination vertex is 779 the first cost is

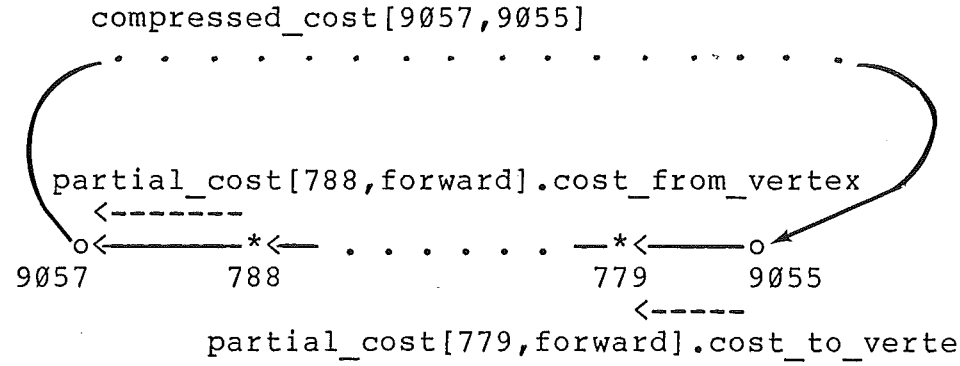
partial_cost[779,backward].cost_to_vertex

and the second cost is calculated as above but the partial_cost for a link vertex does not exist so is deemed to be zero. So the cost is

compressed_cost[9Ø57,9Ø55] +
partial_cost[779,forward].cost_to_vertex



(a)



(b)

SHORTEST PATHS FOR VERTICES IN THE SAME COMPONENT **FIG 4.6**

When the source vertex is 788 and the destination vertex is 9055 the the costs are respectively

```
partial_cost[788,backward].cost_from_vertex
```

and

```
partial_cost[788,forward].cost_from_vertex +
        compressed_cost[9057,9055]
```

(end of example)

If the component is a polygon then the two link vertices are identical and the shortest path between them in the compressed network is a null path with a zero cost. The shortest path from v_s to v_d is the path with minimum cost of these two possible paths.

Example 4.5

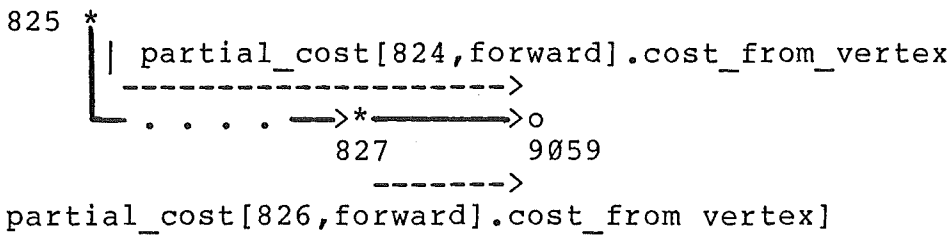
Figure 4.7 shows the two possible paths that may be the shortest path from a source vertex to a destination vertex in a polygon. The source vertex is 825 and the destination vertex is 827 in the polygon shown in Figure 4.2. The first cost in Figure 4.7 (a) is the difference

```
partial_cost[825,forward].cost_from_vertex -
        partial_cost[827,forward].cost_from_vertex
```

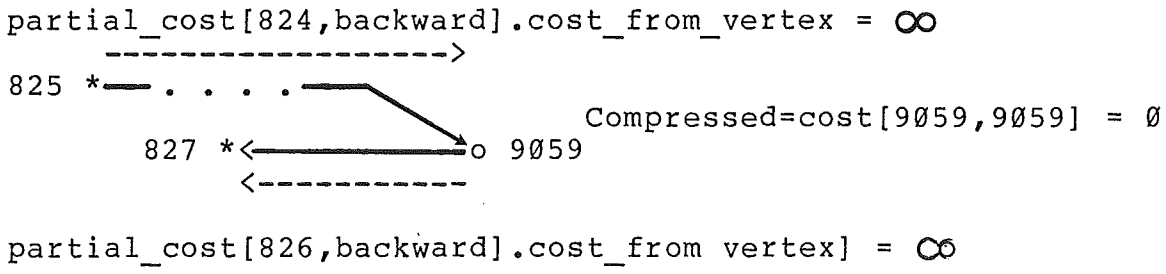
The second cost in Figure 4.6 (b) is the sum

```
partial_cost[825,backward].cost_from_vertex +
        compressed_cost[9059,9059] +
        partial_cost[827,backward].cost_to_vertex
```

If either the source or destination is a link vertex the costs are as shown in Example 4.4.



(a)



(b)

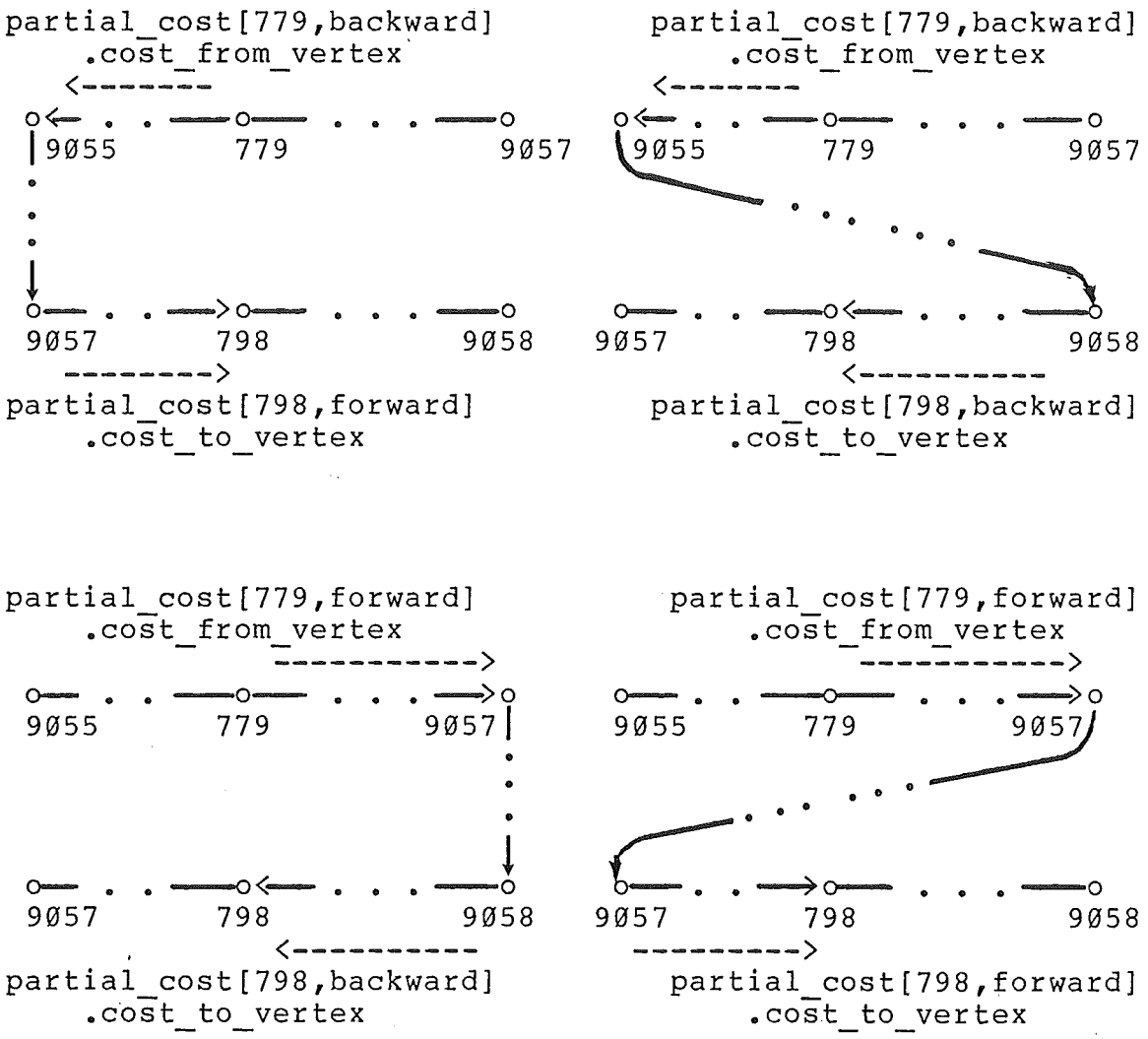
SHORTEST PATHS IN A POLYGON **FIG 4.7**

(end of example)

If the vertices v_s and v_d are not in the same component then there are at most four possible paths from v_s to v_d that may be shortest paths. Each vertex is in a component with at most 2 link vertices and so there are at most 4 paths in the compressed network from one of the link vertices of the source vertex to one of the link vertices of the destination vertex. The shortest path is from v_s to a link vertex followed by a path in the compressed network from this link vertex to one of the link vertices in the component of v_d followed by the path in the component from the link vertex

to v_d. The shortest path between v_s and v_d is the path with minimum cost of these paths. When either a forward or backward path does not exist in the network then following the technique used by Dijkstra's algorithm, the cost of the nonexistent path is deemed to be infinite for the calculation of the costs of the shortest paths. If the source and destination have a common link vertex the shortest path in the compressed network will be zero.

Example 4.6



SHORTEST PATH BETWEEN TWO VERTICES **FIG 4.8**

Figure 4.8 shows the four possible shortest paths from a source vertex to a destination vertex when they do not occur in the same component of the decomposition. The components are from the network portion shown in Figure 4.1. The source vertex is 779 which is in a component with link vertices 9055 and 9057. The first pair of path samples shows the path from 799 to 9055 and thence to either link vertex of the destination vertex. The second pair of path samples shows the path from 779 to 9057 and thence to either link vertex of the destination. The destination vertex is 798 which is in a component with link vertices 9057 and 9058. The last path shown in Figure 4.8 is through the common link vertex of vertices 779 and 798 and the shortest path cost in the compressed network will be zero.
(end of example)

When either the source or destination vertex is in a polygon there are still four possible paths as although there are only two possible paths through the compressed network to or from the link vertex of a polygon, there are still two ways to traverse the path from the source vertex to the link vertex or from the link vertex to the destination vertex. When either the source or destination vertex is a link vertex the number of possible paths is reduced, but the cost from the source to the link vertex or from the link vertex to the destination is deemed to be zero and the analysis for shortest paths between two vertices still applies.

With all possible shortest paths well defined, Algorithm 4.3 can be detailed explicity using the data structure detailed in section 4.3. Algorithm 4.4 generates a matrix of shortest paths for a set of vertices S selected from the vertices of the network.

Algorithm 4.4

Generating the Shortest Paths

Input: Partial costs for components of decomposition,
Shortest path matrix for compressed network,
Set S of vertices for which shortest paths are required.

Output: Shortest path costs matrix 'cost' for vertices of S.

Initialisation:

```
For each selected vertex v_s do
 For each selected vertex v_d do
    cost[i,j] := ∞
```

Method:

```
For each selected vertex v_s do
 For each selected vertex v_d do
   if partial_cost[v_s,forward].path_id =
            partial_cost[v_d,forward].path_id then
    if v_s = v_d then cost[v_s,v_d] := ∅
```

```
else begin
  if v_s < v_d then direction := forward
                    reverse := backward
               else direction := backward
                    reverse := forward

  if v_s is a link vertex then first_cost :=
                partial_cost[v_d,direction].cost_to_vertex

  else first_cost :=
              partial_cost[v_s,direction,cost_from_node] -

                partial_cost[v_d,direction,cost_from_node]


  start := partial_cost[v_s,direction].start_link

  end := partial_cost[v_s,direction].end_link

  if v_d is a link vertex then second_cost :=
              compressed_cost[start,end] +
              partial_cost[v_d,reverse].cost_to_vertex

  else second_cost :=
            partial_cost[v_s,reverse].cost_from_vertex +

              compressed_cost[start,end] +
              partial_cost[v_d,reverse].cost_to_vertex

  c[v_s,v_d] := min (first_cost,second_cost)

  end

else begin
  if v_s is a link vertex then start1 := v_s
                               start2 := v_s

  else start1 := partial_cost[v_s,forward].start_link
       start2 := partial_cost[v_s,forward].end_link

  if v_d is a link vertex then end1 := v_d
                               end2 := v_d

  else end1 := partial_cost[v_d,forward].start_link
       end2 := partial_cost[v_d,forward].end_link

  first_cost := compressed_cost[start1,end1]
  second_cost := compressed_cost[start1,end2]
  third_cost := compressed_cost[start2,end1]
  fourth_cost := compressed_cost[start2,end2]

  if start ≠ v_s then

      first_cost := first_cost +
            partial_cost[v_s,backward].cost_from_vertex

      second_cost := second_cost +
            partial_cost[v_s,backward].cost_from_vertex
```

```
        third_cost := third_cost +
              partial_cost[v_s,forward].cost_from_vertex

        fourth_cost := fourth_cost +
              partial_cost[v_s,forward].cost_from_vertex

    if end ≠ v_d then

         first_cost := first_cost +
                partial_cost[v_d,forward].cost_to_vertex

        second_cost := second_cost +
                partial_cost[v_d,backward].cost_to_vertex

        third_cost := third_cost +
                partial_cost[v_d,forward].cost_to_vertex

        fourth_cost := fourth_cost +
                partial_cost[v_d,backward].cost_to_vertex

    cost[v_s,v_d] := min (first_cost, second_cost,
                          third_cost, fourth_cost)
    end
```

(end of method)

4.5 ANALYSIS OF COMPLEXITY

Let G(V,E) be a network on n vertices and m edges. Let G contain t (<=n) link vertices. Then the compressed graph G_c=(L, E(L) U P) has t nodes and let it have u edges.

For Algorithm 4.2, the initialisation requires 5*2*n operations to set the elements of the record of the array partial_cost to either ∞ or $\emptyset$. Finding the vertices of the compressed network requires O(n) operations and a maximum of u = m - (n-t) operations are required to set the costs of the compressed network, c', to ∞.

Every edge e of G has been partitioned into a branch of G. If e is in a trivial branch of G then no tracing through the simple vertices of G is performed and O(1) operations are

required to set the cost of the edge u_p in c'. For each connecting branch or polygon of G having p simple vertices and p+1 edges, each edge in the branch will be traced by Algorithm 4.2 and for each simple vertex v_s in the branch, the table of partial_cost will be assigned 2 values on the first tracing, and when the tracing is reversed a further 2 values will be assigned to partial_cost. A total of 5*p assignments, plus one further assignment for the cost of the link edge in the compressed network if the branch is not a polygon. Therefore the n-t edges of the simple vertices will require O(5) operations and the remaining m-(n-t) edges will require O(1) operation. So, overall O(m) operations will be required.

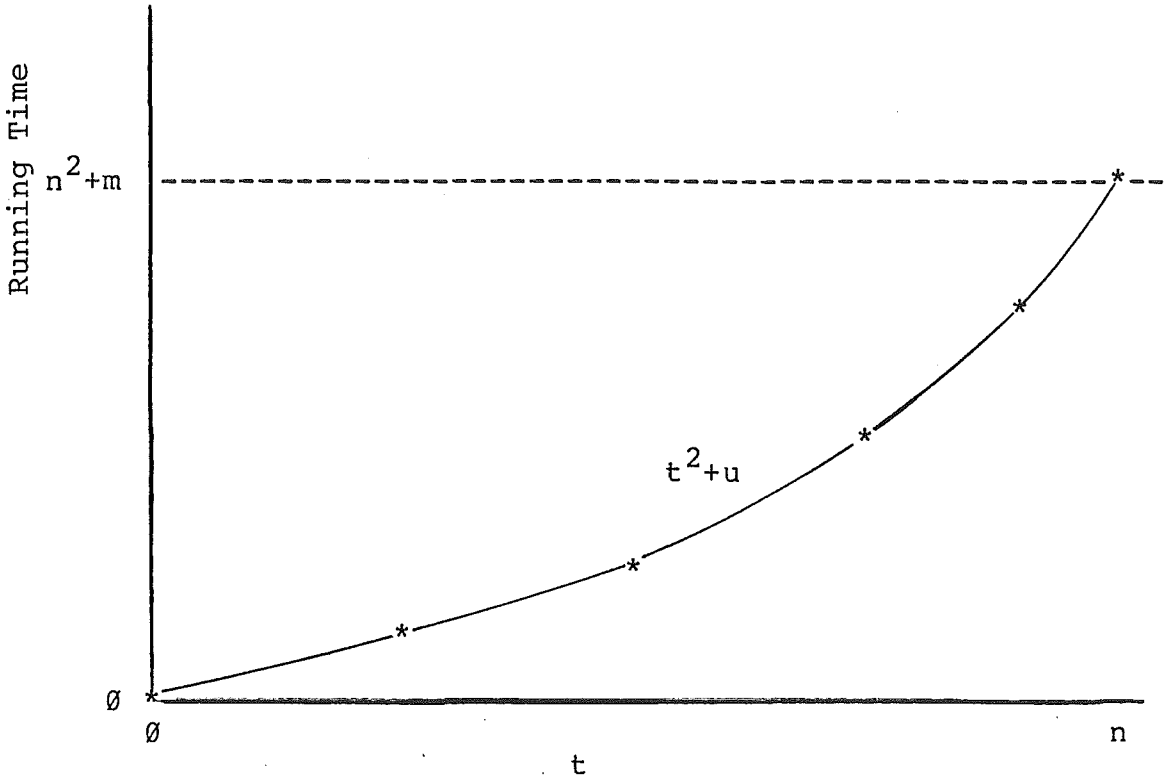
The array partial_cost will require O(n-t) space. This array contains the full shortest path costs for the components of the decomposition. The compressed network can be stored in O(t+u) space in an adjacency list containing the u edges and t vertices of G_c. An all pairs shortest path algorithm applied to the compressed network will require $O(t*(t^2+u))$ time and $O(t^2)$ space.

For Algorithm 4.4, finding the shortest path between any pair of nodes requires a fixed number of arithmetic operations and comparisons, so requires linear time. Let S be the set of selected nodes with $s = |S|$. Then the total running time of the algorithm is $O(s^2)$.

The number of operations to generate the shortest paths for the compressed network using Dijkstra's algorithm and the shortest paths for the components of the decomposition using Algorithm 4.2 is $O(n)+O(u)+O(m)+O(t*(t^2+u))$. The final calculation of the shortest paths requires $O(s^2)$ operations. The space requirements are $O(n+m)$ for the original network and $O(n-t)+O(t+u)+O(t^2)$ for the decomposition. The final matrix of shortest path costs requires $O(s^2)$ space.

In comparison a straight forward application of Dijkstra's algorithm to the original network would require $O(s*(n^2+m))$ operations to find the shortest path costs from each source in the set of selected vertices, S. Generating the shortest path matrix will require $O(s*n)$ space and the selected vertices shortest path matrix will require $O(s^2)$. space. The decomposition defined in Section 4.2 may be applied to any graph, but the advantages of the decomposition will increase as the number of simple vertices increases relative to the number of link vertices.

The major cost component of the algorithm is building the shortest path matrix of the compressed network. This will require $O(t^2+u)$ operations for every source vertex compared with the $O(n^2+m)$ operations required for the original network. Figure 4.9 shows the growth of t^2+u where t is a function of n. The value of $m=O(n)$ and $u=O(t)$.



GRAPH OF t^2+u VERSUS t AS t INCREASES FROM Ø TO n **FIG 4.9**

For a planar graph with no parallel edges and no self loops, the number of edges is limited to m <= 3*n-6 [Even] and for a network in which possibly two directed edges may exist for every edge of an undirected graph, this limit on the number of edges can be doubled. For a connected network the minimum number of edges is m >= n+1. Clearly, for the compressed network these limits still apply so t+1 <= u <= 6*t-12, although a bound on u in terms of n has been established by Theorem 4.5. Substituting m=O(n) and u=O(t) the time requirements for generating the shortest paths is $O(n) + O(t) + O(t^3) + O(s^2)$ and the space requirements are $O(n) + O(t) + O(t^2) + O(s^2)$.

4.6 COMPARISION WITH OTHER ALGORITHMS

An early motivation for the development of methods for decomposing the matrix used for graph representation (and hence the graph) was the limitation of computational power as larger graphs were investigated. Many of the methods were specifically designed for shortest path problems.

Farley, Land and Murchland [1967] developed a cascade algorithm for finding all shortest paths in a directed graph in two passes over the matrix. This was extended by Land and Stairs [1967] who introduced a partitioning of a graph to allow the analysis of larger graphs than had previously been possible. This partitioning identified a set of boundary vertices B and a division of the remaining vertices into N disjoint sets. Thus the vertices belonging to any set p have edges only to or from vertices in the same set or in the boundary B. The decomposition generates centrally overlapping subnetworks.

Hu [1968], Hu and Torres [1968], Yen [1971], Glover, Klingman and Napier [1973] and Jarvis and Tufekci [1982] suggested methods for partitioning graphs into linearly overlapping subnetworks.

Shier [1973] defines a directed graph G = (N,A) where N is a finite set of nodes and A is a finite set of arcs (directed edges) with $a_{ij} \in S$ the value of the arc (v_i, v_j) on the binoid (S,+,*) and partitions a graph into a collection of subgraphs linked to form a tree structure. This

decomposition generalised the earlier methods which are included as special cases.

These decomposition methods are all based on a similiar approach. The decomposed graph has a simple overall structure, for example the tree structure of Shier's method and the even more restricted form of the earlier methods, while the components of the decomposition are a complex network structure. Once the structure of the network is obtained, the full shortest path problem need only be solved on limited submatrices and these solutions combined to form the final shortest path matrix.

Jarvis and Tufekci give the cost of generating a linear decompositon as defined by Hu as $O(n^2)$ and the costs of calculating the shortest path from the decomposition using Hu's algorithm as $O(k^2*u^3)$ where k is the number of components of the decomposition and u the size of each component. Jarvis and Tufekci improved this to $O(k*u^3)$. Using the same notation, Shier defined (m) k disjoint sets of vertices $N_1 \ldots N_k$ and the size of each set is $u_1 \ldots u_k$ respectively, where $\max(|u_i|) < n$. Generating the shortest path submatrices requires $O(k*\max(|n_i|)^3)$ operations compared to $O(n^3)$ operations without the decomposition.

All the methods in this group were concerned with the decomposition of the matrix used to represent the graph and no author has presented an algorthm for generating an optimal decomposition of the graph. The method to be used is described and left to the user. For example, Hu and Torres

state:

> "To obtain (the decomposition) we can proceed as follows. Take any subset of nodes as A, then its minimum cut set is X_A. Let B be a cut set (not necessarily a minimum cut set) of AUX_A, . . . Repeat until no more decomposition is desired."

Shier is no more explicit in defining the decomposition. He states:

> "It is now supposed that the nodes N of the original network G can be partitioned into m >= 2 disjoint sets $N_1, N_2, \ldots N_m$ which exhibit a tree structure when viewed as an undirected graph T(G). . . . Therefore, given the tree graph T(G) . . ."

The decomposition is not well defined and so will be neither unique nor irreducible. Therefore obtaining the optimal solution cannot be ensured and the efficiency of the methods depend on the chance of choosing a good decomposition.

4.7 APPLICATIONS

Theorem 4.4 proved that the decomposition described in this chapter can be applied to all networks, but it will not be the best solution method for every network. The networks that the decomposition is best adapted to are sparse graphs and the planar graphs are included in this group. Within the framework of random graphs, planar graphs are infrequent. Erdos and Renyi [1959] showed that for a graph on n vertices the probabilty that the random graph $G_{n,N}$ is planar tends to 1 when $N < n/2$ and tends to Ø when $N > n/2$. But the

requirement that a graph be connected, will reduce the possible number of graphs further, as a connected graph on n vertices must have at least n-1 vertices and each vertex must be incident by at least one edge. For a connected planar graph the number of edges must lie in the range n-1 <= m <= 3n-6. The first n-1 edges must connect the graph and give every vertex a minimum degree of 1 and an average degree of 2. The remaining 2n-6 edges are then distributed randomly. Although most of the

$$\sum_{N=n-1}^{3n-6} \binom{\binom{n}{2}}{N}$$ graphs are not planar connected

graphs, the number of divalent vertices in any planar connected graph will be in the range Ø to n, and the remaining vertices will be link vertices. In Figure 4.9 the running time improvement possible using the decomposition method is shown relative to the proportion of divalent vertices.

The decomposition was intended to apply particularly to transportation networks rather than random graphs. The method was applied to a local transportation application, namely the bus network of the Christchurch Transport Board [Gabites, Porter and Partners, 1982]. The vertices of the network are the bus stops and the intersection points of the bus routes. The bus routes also determine the direction of an edge in the graph. The costs on the edges of the network are the distance and the time to travel the distance. As well, the costs of the bus fare, determined by the number of 'sections' travelled, are included and so there are 'section' bus stops at which the fare rate changes. Paths

within the bus network are further determined by the bus routes, and changing from one route to another incurs a penalty transfer cost. The original network contains 951 vertices and 1966 edges and is shown in Figure A.1 in Appendix A. The network portion used in the examples of this chapter is from the extreme left of the network. The application of the decomposition algorithm generated a compressed network on 101 vertices and 266 edges. The shortest path matrix was required over a representative sample of 120 'centroid' bus stops spread over the entire network. The selected set was chosen randomly from the simple and link vertices of the network.

For this application the relative number of operations required to calculate the shortest path cost matrix are

For the original network: $s*(n^2+m) = 10876400 = 108 \times 10^6$

For the compressed network: $s*(t^2+u) = 125600 = 1 \times 10^6$

With the selected set: $s*s = 14400$

4.8 SUMMARY

A decomposition method for solving shortest paths on networks has been presented. The decomposition has been defined and two important features for the use of the decomposition have been proved, that is that the decomposition produces unique and irreducible components and that the original network is reconstructable from the

decomposition. The decomposition is valid for all graphs and networks but the class of graphs for which it is most effective are graphs with many divalent vertices and the networks with many simple vertices. The compressed network defined by the components of the decomposition can be obtained in O(m) time and will be smaller than the original network for this class of networks so, a shortest path matrix can be obtained more efficiently.

The full analysis of possible paths has shown that the reconstruction of the shortest paths in the original network requires a single calculation on the cost of the compressed network and the costs of the components of the decomposition. The number of operations required by each part of the method has been analysed. Generating the decomposition requires O(m) operations and it produces the shortest path costs on the components of the decomposition and the compressed network. Generating the shortest path costs on the compressed network will depend on the size of the compressed network. The costs of the $O(t^2+u)$ operations on the compressed network and the $O(n^2+m)$ operations on the original network have been compared for the range of possible values of t.

Earlier decomposition methods have been examined. They defined several subnetworks for which the shortest paths had to be solved and generated the overall shortest path costs on the simple structure connecting the subnetworks. The opposite approach has been taken in the decomposition method presented here. The single compressed network is composed of

the simple paths in the components of the decomposition. This approach led to a decomposition that is well defined and for any network will always generate the same decomposition. Comparing the efficiency of the methods is subject to inaccuracies as the decompositions of the earlier methods are not explicity defined.

The decomposition has been used in the transportation planning model of the Christchurch Transport Board. This transportation network is reduced and simplified by the decomposition.

CHAPTER 5

BOOLEAN MATRIX MULTIPLICATION

5.1 INTRODUCTION

One of the most common graph representations is the adjacency matrix which is widely used in computer and other applications. An adjacency matrix can be interpreted as a boolean matrix and so for many problems in graph theory there is an equivalent problem in matrix theory. For example, the multiplication of boolean matrices is equivalent to the computation of the transitive closure of a graph. Algorithms for boolean matrix multiplication and transitive closure have been extensively studied for nearly 30 years and many efficient algorithms have been published. Data structures used in these algorithms have varied from boolean matrices to list representations.

This chapter investigates a compression of the elements of a boolean matrix and algorithms to use the compression for boolean matrix multiplication. The compressed data are formed into a tree and a family of algorithms based on different tree structures and techniques for searching trees is presented. Tree data structures have not previously been applied to boolean matrix multiplication. The worst and average case running times of the algorithms are

investigated and the method is compared to existing methods for boolean matrix multiplication. The algorithms developed in this chapter suggest a simple heuristic that may be added to existing algorithms to improve their running time.

The worst case running time of algorithms for boolean matrix multiplication is readily established. Recent research has concentrated on the average case time complexity and this approach is continued in this chapter.

5.2 TREE SUMMATION METHODS FOR BOOLEAN MATRIX MULTIPLICATION

For an nxn matrix n trees can be constructed, one for each row or column of the matrix. Each tree, representing one row of the matrix has n leaves and these leaves have the same value as the elements of the row of the matrix. At each level of the tree each parent vertex is assigned a value which is the arithmetic sum of all the values of its children. Thus, each tree contains the sum and partial sums of the number of occurrances of a '1' in the matrix row or column.

Suppose that A and B are two nxn matrices over the semiring $S'=(\{\emptyset,1\},+,.,\emptyset,1)$ then, the product C = A.B is the matrix whose elements are

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The elements of A,B and C are all elements of a semiring on the set {∅,1}, and so

$$c_{ij} = 1 \quad \text{if} \quad a_{ik}.b_{kj} = 1 \quad \text{for any k.}$$

Let A' and B' be two nxn matrices over the ring of integers. The elements of the semiring S' are a subset of the elements of the ring of integers.

$$\{\emptyset,1\} \subset I$$

Define the elements of A' and B' to be equal to the elements of A and B respectively.

$$a'_{ik} = a_{ik}$$
$$b'_{kj} = b_{kj}$$

Theorem 5.1

If

$$\sum_{k=1}^{n} a'_{ik} + \sum_{k=1}^{n} b'_{kj} > n$$

then there exists k such that

$$a_{ik}.b_{kj} = 1$$

Proof: by contradiction

The number of elements in any row or column of an nxn matrix is n. Let

$$\sum_{k=1}^{n} a'_{ik} + \sum_{k=1}^{n} b'_{kj} >= n+1$$

Each non-zero $a'_{ik} = 1$ and similarly each non-zero $b'_{kj} = 1$. In the i^{th} row of A' and the j^{th} column of B' there are at least n+1 non-zero entries and at most n-1 zero entries.

If

$$\sum_{k=1}^{n} a_{ik} \cdot b_{kj} = \emptyset$$

then for all k either $a_{ik} = \emptyset$ or $b_{kj} = \emptyset$. In the i^{th} row of A and the j^{th} column of B there are at least n zero entries, but in the i^{th} row of A' and the j^{th} row of B' there are at most n-1 zero entries. Since $a'_{ij} = a_{ij}$ and $b'_{ij} = b_{ij}$ then this forms a contradiction.

(end of proof)

Example 5.1

Let A and B be the two matrices illustrated in Figure 5.1.

	1	2	3	4
1	1	Ø	1	1
2	1	Ø	1	1
3	1	Ø	1	1
4	1	Ø	1	1

A

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	Ø	Ø	Ø	Ø
4	Ø	Ø	Ø	Ø

B

TWO EXAMPLE MATRICES **FIG 5.1**

Then for all i and j

$$\sum_{k=1}^{n} a'_{ik} = 3 \quad \text{and} \quad \sum_{k=1}^{n} b'_{kj} = 2$$

so

$$\sum_{k=1}^{n} a'_{ik} + \sum_{k=1}^{n} b'_{kj} = 5 > 4$$

and

$$\sum_{k=1}^{n} a_{ik} \cdot b_{kj} = 1$$

(end of example)

Define a partition ${}^{k}P$ of size k on a row i of a matrix A as a division of the elements of the row into $|\bar{}n/k\bar{}|$ sets

$${}^{k}P_{i1} = \{a_{ij} \mid 1 <= j <= k\}$$
$${}^{k}P_{i2} = \{a_{ij} \mid k+1 <= j <= 2k\}$$

.

.

.

$${}^{k}P_{i,|\_n/k\_|} = \{a_{ij} \mid (|\bar{}n/k\bar{}|-1)*k <= j <= (|\_n/k\_|)*k\}$$

and if $|\bar{}n/k\bar{}| \neq |\_n/k\_|$ then

$${}^{k}P_{i,|\bar{}n/k\bar{}|} = \{a_{ij} \mid (|\_n/k\_|)*k+1 <= j <= n\}$$

Each set ${}^{k}P_{im}$ $1 <= m <= |\_n/k\_|$ contains exactly k elements of the row of matrix A. Set ${}^{k}P_{i,|\bar{}n/k\bar{}|}$ may contain less than k elements, but this will not affect any use of the partition. Let each set defined by a partition be called an element of the partition. Similarly, a partition can be defined on a column of a matrix. In the following discussion, partitions will be used for boolean matrix multiplication, so any partition defined on the rows of the matrix A must correspond to a partition on the columns of the matrix B.

Theorem 5.2

Let the rows of an nxn matrix A and the columns of an nxn matrix B be partitioned. For an element of the partition on

row i of A $\{a_{ik} \mid r <= k <= s\}$ there exists a corresponding element on column j of B $\{b_{kj} \mid r <= k <= s\}$ and if

$$\sum_{k=r}^{s} a'_{ik} + \sum_{k=r}^{s} b'_{kj} > s-r+1$$

then

$$\sum_{k=r}^{s} a_{ik} \cdot b_{kj} = 1$$

Proof:

1. The corresponding partition element exists on B.

Each a_{ik} is an element of the matrix so $1 <= r,s <= n$. The division of the elements of the rows of A and the columns of B is determined by the size of the partition. Therefore there exist elements of the j^{th} column of matrix B $\{b_{kj} \mid r <= k <= s\}$ and these elements form an element of a partition of the j^{th} column of B.

2. Summation on partitions.

By Theorem 5.1,

$$\sum_{k=1}^{n} a'_{ik} + \sum_{k=1}^{n} b'_{kj} > n \text{ implies } \sum_{k=1}^{n} a_{ik} \cdot b_{kj} = 1$$

then similarly, for the smaller matrix defined from r to s,

$$\sum_{k=r}^{s} a'_{ik} + \sum_{k=r}^{s} b'_{kj} > s-r+1 \text{ implies } \sum_{k=r}^{s} a_{ik} \cdot b_{kj} = 1$$

(end of proof)

A series of partitions can be defined for each row of A and column of B. The partition sizes can range from n which is a

partition whose single element covers the entire row or column, down to a partition size of 1 which is a partition in which each of the n elements of the partition will contain a single element of the row or column.

Definition 5.1

Let A be an nxn matrix. For each row or column of A there is a partition series as follows:

1. There is one partition of size 1, which has n elements, each of which is a set of one of the n elements of the row or column of the matrix. The elements of the partition of size 1 are called the leaves of the partition series.
2. There is one partition of size n, which has one element that includes all the elements of the row or column of the matrix. The element of the partition is called the root of the partition series.
3. For each partition in the series that is not the root partition, there exists a partition whose elements completely include 2 or more its elements. The number of elements of a partition that are combined to form the elements of a new partition is called the summation size of the partition series and is fixed for every partition in the partition series that is not a leaf.

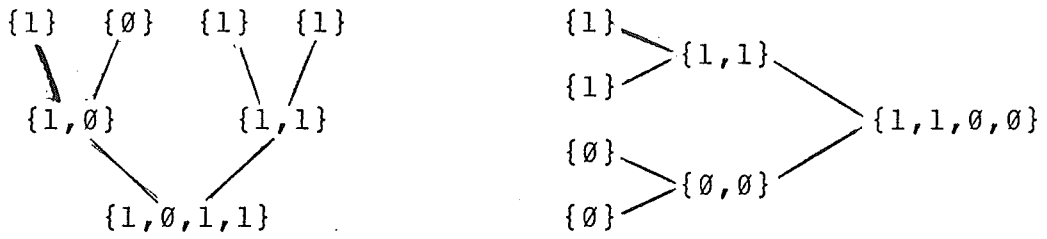
The elements of the partitions in the partition series will be called the members of the partition series.

The partition series defines a recursive division of the elements of the rows and columns of a matrix and so it can

be described by a tree structure. The smallest and simplest summation size for a partition series is 2 and the partition series that is generated is called a binary summation. A partition series for a binary summation can be illustrated by a binary tree.

Example 5.2

A binary summation constructed on the rows and columns respectively of the two matrices of Figure 5.1 is illustrated by the binary trees in Figure 5.2. The rows of matrix A are identical so the same partition series is generated for every row. Similarly, the columns of matrix B are identical so the same partition series is generated for every column.



BINARY SUMMATION PARTITION SERIES ON TWO MATRICES **FIG 5.2**

(end of example)

Definition 5.2

For each element $kP_{im} = \{a_{ik} \mid r <= k <= s\}$ of a partition

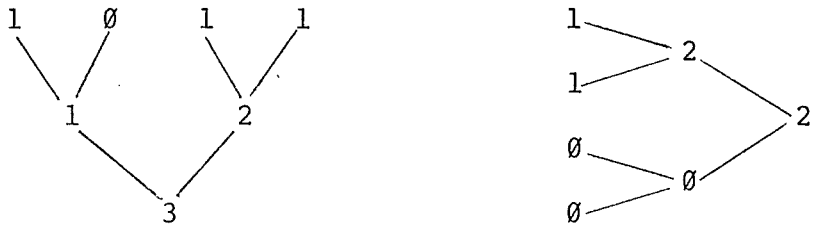
define a value $T_{im} = \sum_{k=r}^{s} a'_{ik}$

and a size $S_{im} = s-r+1$.

For any partition series there exist two equivalent series of values and sizes respectively, whose structure is defined by the partition series and whose elements are the values and sizes defined by Definition 5.2. These series will be called the summation values and summation sizes respectively of the partition series.

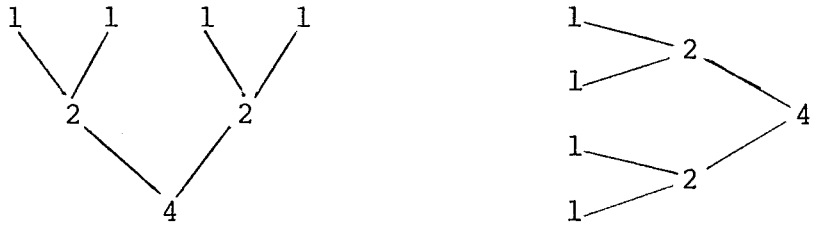
Example 5.3

The summation values corresponding to the partition series in Example 5.2 are shown in Figure 5.3.



SUMMATION VALUES OF A BINARY SUMMATION PARTITION SERIES **FIG 5.3**

The summation sizes corresponding to the partition series in Example 5.2 are shown in Figure 5.4.



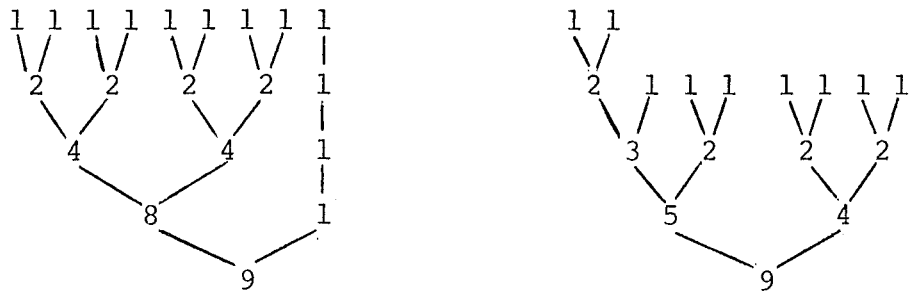
SUMMATION SIZES OF A BINARY SUMMATION PARTITION SERIES **FIG 5.4**

(end of example)

A series of partitions described by a tree structure can be referenced by the same terminology as general tree structures. Each member of a partition series is a vertex of the tree. A partition element which completely includes smaller partition elements is a parent vertex and the smaller partitions are its children vertices. The members of the partition series will be indexed as for standard tree numbering systems. The root of the tree which is a partition of size n will have index 1 and the children of any partition will be numbered from left to right, or for partitions on a matrix column, from top to bottom.

There are two distinct methods for building the recursive structure of the partition series for the matrices and traversing this series. First, the leaves of the partition series may all be at the same depth. Alternatively, a tree with fewer vertices will occur if every parent vertex has the maximum possible number of children (with the possible exception of the parent of the last leaf in the tree).

Example 5.4



(a) (b)

ALTERNATIVE RECURSION STRUCTURES **FIG 5.5**

The two trees illustrated in Figure 5.5 show the summation sizes for two possible recursion structures for a binary summation partition series on a row of 9 elements. The tree in (a) shows that a binary summation partition series with 9 leaves at the same depth requires 20 vertices. The tree in (b) shows that a binary summation partition series with 9 leaves and all parent vertices having exactly 2 children requires 17 vertices.
(end of example)

5.3 THE ALGORITHM

An algorithm for determining the product matrix from the partition series defined on the rows and columns of the matrices was developed. The tree structures of summation values generated by the partition series are searched until the sum of values from a row of A and a column of B are greater than the summation size for that member of the partition series. When such a sum is found, the product may be assigned a value of '1', and the search can terminate. Any search technique may be applied.

Algorithm 5.1

Outline of Algorithm for Boolean Matrix Multiplication

Input: A and B two nxn boolean matrices.

Output: C = A.B an nxn boolean matrix.

Initialisation:

Build the series of summation values for A and B, respectively tree_A and tree_B.

Build the series of summation sizes tree_sum.

Method:

```
For i := 1 to n do
  For j := 1 to n do
    C_ij := Ø
    Tree_scan(1,i,j)
```

(end of method)

The procedure Tree_scan referenced in Algorithm 5.1 searches the partition trees and is further defined by the search technique used. The procedures for depth first and breadth first search are described in Algorithm 5.2 and 5.3 respectively. The algorithms are both presented as a recursive algorithm to emphasise the similarities and differences between the two methods.

Algorithm 5.2

Outline of Procedure Tree_scan using Depth First Search

```
Procedure Tree_scan(k,i,j : integer);
 if tree_A_ik + tree_B_kj > tree_sum_k then c_ij := 1
 else
   for each child h of vertex k in the series of
                                   summation values do
         Tree_scan(h,i,j)
```

(end of method)

Algorithm 5.3

Outline of Procedure Tree_scan using Breadth First Search

```
Procedure Tree_scan(k,i,j : integer);
 if tree_A_ik + tree_B_kj > tree_sum_k then c_ij := 1
 else Tree_scan(k+1,i,j)
```

(end of method)

For either the depth first search or the breadth first search version of Tree_scan, the efficiency of Algorithm 5.1 will be determined by the number of recursive calls made to the procedure Tree_scan. This number is limited by the number of elements in the partition series which defines the size of the recursion tree that will be searched. The recursion tree illustrated in Figure 5.5 (b) gives a smaller tree size than the tree of Figure 5.5 (a).

The number of calls to the procedure Tree_scan can also be reduced by strengthening the termination conditions of the algorithm. Once a summation value greater than the summation size has been found in the search procedure the search will terminate. If any member of the partition series has a value of zero then all the children of the partition vertex will have the value zero, and so, these vertices do not need to be searched. The extended procedure Tree_scan for a depth first search is shown in Algorithm 5.4.

Algorithm 5.4

Terminating conditions added to the Outline of Procedure Tree_scan using Depth First Search.

Input: A and B two nxn boolean matrices.

Output: C = A.B an nxn boolean matrix.

Initialisation:

Build the series of summation values for A and B, respectively tree_A and tree_B.

Build the series of summation sizes tree_sum.

Method:

```
Procedure Tree_scan(k,i,j : integer);
  if tree_A_ik + tree_B_kj > tree_sum_k then
     c_ij := 1
  else
    for each child h of vertex k in the series of
                                   summation values do
      if tree_A_ih≠Ø and tree_B_hj≠Ø and c_ij=Ø do
         Tree_scan(h,i,j)
{end of Tree_scan}

  For i := 1 to n do
    For j := 1 to n do
      c_ij := Ø
      Tree_scan(1,i,j)
```

(end of method)

The recursion method applied to the Breadth First Search version of the Tree_scan algorithm should in practice be replaced by the iterative version outlined in Algorithm 5.5. The terminating conditions have also been added.

Algorithm 5.5

Terminating Conditions Added to theOutline of Iterative Procedure Tree_scan using Breadth First Search

Outline of Algorithm for Boolean Matrix Multiplication

Input: A and B two nxn boolean matrices.

Output: C = A.B an nxn boolean matrix.

Initialisation:

Build the series of summation values for A and B, respectively tree_A and tree_B.

Build the series of summation sizes tree_sum.

Method:

```
Procedure Tree_scan(k,i,j : integer);

  while c_ij = Ø and k<=n do
   if tree_A_ik + tree_B_kj > tree_sum_k then c_ij := 1
   else begin
      for each child h of vertex k in the series of
                                   summation values do

          if tree_A_ih≠Ø and tree_B_hj≠Ø and c_ij=Ø do
             add h to the queue to vertices to search

      k := vertex at top of queue
      end

For i := 1 to n do
  For j := 1 to n do
    c_ij := Ø

  Tree_scan(1,i,j)
```

(end of method)

Reordering the elements of the matrices can cluster the elements with a value of 1 together and the elements with a value of Ø together. Such a cluster may allow a subtree to be eliminated from the depth first search. Maximising the number of 1's in the lower indices may lead to the successful termination of a depth first search in the

subtrees searched earliest. The reordering must be performed over the entire matrix, not per row and column, or the correspondence of partition elements in a series will be lost. Advantages will be obtained from a reordering if the input data is not randomly distributed.

The complete algorithm for boolean matrix multiplication using the binary summation partition series with depth first search is presented in Algorithm 5.6. The recursion structure used in the algorithm has all parent vertices having the maximum possbile number of children. An example of this recursion structure is shown in Figure 5.5 (b).

Algorithm 5.6

Algorithm for Boolean Matrix Multiplication

Input: A and B two nxn boolean matrices.

Output: C = A.B an nxn boolean matrix.

Initialisation:

Build the summation values for each row i of A and each column i of B, respectively tree_A and tree_B.

```
for i := 1 to n do begin
  for j := 1 to n do begin
    k := n+j-1
    tree_A[i,k] := A[i,j]
    tree_B[i,k] := B[j,i]
    end;
  for j := n-1 downto 1 do begin
    tree_A[i,j] := tree_A[i,2*j] + tree_A[i,2*j+1]
    tree_B[i,j] := tree_B[i,2*j] + tree_B[i,2*j+1]
    end
  end
```

Build a sequence of summation sizes tree_sum.

```
  for j := n to 2n-1 do tree_sum[j] := 1
  for j := n-1 downto 1 do
    tree_sum[j] := tree_sum[2*j] + tree_sum[2*j+1]
```

Method:

```
    procedure Tree_scan(k,i,j : integer);
    begin
      if tree_A[i,k] + tree_B[j,k] > tree_sum[k] then begin
        c[i,j] := 1
        finished := true
        end
      else if (k := 2*k) <= 2*n-1 then begin
        if tree_A[i,k]>Ø and tree_B[j,k]>Ø then
                                  Tree_scan(k,i,j)
        if not finished and (k := k+1) <= 2*n-1 then
         if tree_A[i,k]>Ø and tree_B[j,k]>Ø then
                                  Tree_scan(k,i,j)
        end
      end { of Tree_scan }

   for i := 1 to n do
      if tree_A[i,1]>Ø then
        for j := 1 to n do
            if tree_B[j,1]>Ø then begin
               finished := false
               Tree_scan(1,i,j)
               if not finished then C[i,j] := Ø
               end
            else c[i,j] := Ø
      else for j := 1 to n do c[i,j] := Ø
```

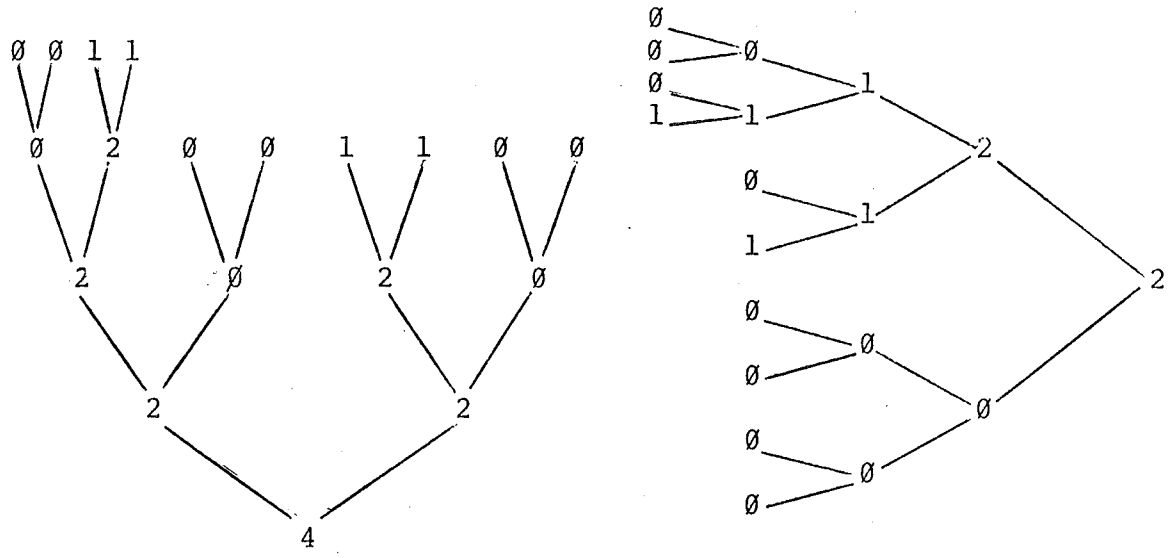
(end of method)

Example 5.5

Let A and B be the 1Øx1Ø adjacency matrix in Figure 2.3 (a). This example will calculate the product element

$$c_{81} = \sum_{k=1}^{1Ø} a_{8k} \cdot b_{k1}$$

using the depth first search algorithm 5.5. Figure 5.6 shows the summation values for a binary summation partition series on row 8 of matrix A and column 1 of matrix B.



SUMMATION VALUES **FIG 5.6**

Using a depth first search the following sequence of calls tØ the procedure Tree_scan would occur:

k = 1

tree_A[8,1] + tree_B[1,1] = 6 $\not>$ 1Ø

k = 2 * k = 2

tree_A[8,2] + tree_B[1,2] = 4 $\not>$ 6

k = 2 * k = 4

tree_A[8,4] + tree_B[1,4] = 3 $\not>$ 4

k = 2 * k = 8 tree_A[8,8] = Ø & tree_B[1,8] = Ø No call

k = k + 1 = 9

tree_A[8,9] + tree_B[1,9] = 3 $>$ 2

Therefore after the initial call to Tree_scan a further 3 calls were required to establish the value of c_{81}. A total of 4 calls.

Using a breadth first search to calculate the product element c_{81} the following search sequence would occur:

k = 1

tree_A[8,1] + tree_B[1,1] = 6 $\not>$ 10

k = 2

tree_A[8,2] + tree_B[1,2] = 4 $\not>$ 6

k = 4

tree_A[8,4] + tree_B[1,4] = 3 $\not>$ 4

k = 5

tree_A[8,5] + tree_B[1,5] = 1 $\not>$ 2

k = 9

tree_A[8,9] + tree_B[1,9] = 3 $>$ 2

A total of 5 vertices must be searched to calculate the product element c_{81}.
(end of example)

5.4 ALTERNATIVE SUMMATION SIZES

The concept of a partition series is not restricted to any one summation size. The binary summation is the smallest possible summation size and it generates the largest partition series, but the binary tree generated by a binary summation is well understood and its behaviour well documented.

The smallest partition series consists of the partition of size 1 and the partition of size n only. Between these two

extremes there exist partitions of constant summation size, for example a ternary summation, and partitions with a summation size that is a function of the size of the matrices, for example log n and square root of n. These generate smaller trees than the binary summation and since the data structures are smaller, the time required to fully search them is reduced. However, the expected number of members of the partition series that must be searched to find two summation values in the partition series that are greater than the partition size must be compared between these methods.

Example 5.6

For n = 1Ø, 3 <= log n <= 4 and 3 <= $n^{1/2}$ <= 4. The ternary and quartary summation partition series have respectively 15 and 13 vertices. Both the depth first and breadth first search of either the ternary or quartary trees of summation values requires 3 calls to the procedure Tree_scan to calculate the product c_{81}.

(end of example)

5.5 ANALYSIS OF COMPLEXITY

The number of operations performed by the algorithm is limited by the tree size generated by the partition series. The trees have n leaves but the number of remaining vertices may vary, depending on the summation size of the partition series of the implementation. However, any tree with n leaves has no more than 2*n vertices.

Building the trees of summation values for each of the n rows or columns in each of the two matrices requires 2*n*treesize operations. Building the tree of summation sizes requires an additional treesize operations. To calculate the product matrix, each of the n^2 elements of C must be calculated. To calculate the value of an element of the product matrix, the procedure Tree_scan must be called at least once. This can then lead to successive calls to the procedure.

The total number of operations is

$$2*n*\text{treesize} + \text{treesize} + n^2*(\text{Number of calls to Tree\_scan})$$

The number of calls to the procedure Tree_scan is limited by the size of the trees. This procedure will be called repeatedly until the test condition

$$\text{tree\_A}_{ik} + \text{tree\_B}_{jk} > \text{tree\_sum}_k \qquad (5.1)$$

is satisfied or until sufficient parent vertices have the value zero (implying all descendents in the subtrees have the value zero). This corresponds to a failure of the condition

$$\text{tree\_A}_{ik} > \emptyset \text{ and } \text{tree\_B}_{jk} > \emptyset \qquad (5.2)$$

The probability of these two relations being true can be used to determine the frequency of their application. The number of calls on Tree_scan is a function of this

probability.

In the best case, the sum of the summation values of the root vertices is greater than n so only the root vertices of the summation trees are tested. Thus,

Number of calls to Tree_scan = O(1)

and so the number of operations required in the best case is

$$2*n*treesize + treesize + O(n^2) = O(n^2)$$

If the probability of a 1 occuring in the elements of each of the matrices is strictly greater than Ø.5 then then the expected value of the sum of the roots of the summation values will be greater then n and only a constant number of vertices in the summation tree need to be searched before the search terminates. So in the best case the expected number of operations is $O(n^2)$.

In the worst case every vertex of the tree must be tested. A depth first search will eliminate subtrees from a search if the summation values are Ø. Since each row and column of the matrices is treated independently, the row and column that cause the worst case can be defined. In the worst case every vertex in the summation tree that is not a leaf will have a non-zero value, but the leaves of the summation values will be Ø for one of the matrices. For every possible value of n there are exactly two possible rows or columns for which the entire tree must be searched. For matrices of size 4 the two

rows and there corresponding columns which cause the worst case performance are illustrated in Figure 5.7.

1Ø1Ø	Ø	ØlØl	1
	1		Ø
	Ø		1
	1		Ø

ROWS AND COLUMNS CAUSING WORST CASE PERFORMANCE **FIG 5.7**

There are 2^n matrices which contain only these rows, and 2^n matrices which contain only these columns, but the worst case performance only occurs when the corresponding pairs are multiplied together.

The exact value of the tree size is dependent on the summation size of the partition series. For example, a binary summation partition series generates a binary tree and it is well documented [Knuth, 1973] that a binary tree with n leaves has exactly 2n-1 vertices.

In general, the total number of vertices in a m-ary tree is determined from the sum of the number of vertices at each level in the tree. A m-ary tree with all possible vertices to the depth of the tree has

$$1 + m + m^2 + m^3 + \ldots + m^{depth} \quad \text{vertices.}$$

If the matrix size is a power of m, then the two tree formats discussed in section 2.2 will produce identical trees with

$$n = m^{depth}$$

so

$$\text{depth} = \log_m n$$

Hence, the number of vertices in the tree is

$$\frac{(m^{\log_m n + 1} - 1)}{m-1} \qquad (5.3)$$

When n is not a power of m the two tree formats will produce trees of different sizes. The number of vertices in a tree with all vertices having the maximum possible number of children is defined by equation 5.3. However, the number of vertices in a tree with n leaves at the same depth is defined by the sum

$$n + \left\lceil \frac{n}{m} \right\rceil + \left\lceil \frac{n}{m^2} \right\rceil + \ldots + \left\lceil \frac{n}{m^{\text{depth}}} \right\rceil$$

The ceiling of the elements of the sum will add a constant value to the sum so that the number of vertices in the tree is

$$\frac{(m^{(\log_m n + 1)} - 1)}{m-1} + c \qquad \text{where } 0 <= c <= \text{depth}$$

A log summation partition series generates a tree with log n children at each vertex. The sum

$$1 + \log n + (\log n)^2 + \ldots + (\log n)^{\log n \,/\, \log\log n}$$

generates a total of

$$\frac{((\log n)^{(\log n / \log \log n + 1)} - 1)}{(\log n - 1)}$$

vertices. The square root of n summation partition series generates a tree of

$$\frac{(n^{3/2} - 1)}{n^{1/2} - 1} \quad \text{vertices.}$$

The value of the tree sizes for varying values of n is compared in Table B.1 in Appendix B.

For any value of m the number of vertices in the m-ary tree

$$\frac{(m^{(\log_m n + 1)} - 1)}{m-1}$$

is $O(n)$. So, in the worst case the number of operations required for any summation partition series is $O(n^3)$.

The number of operations required is in the range $O(n^2)$ to $O(n^3)$ and the expected or average number of operations must lie in this range.

Let p be the probability of a '1' occuring in any matrix element and assume that each element is assigned a '1' with probability p, independently of the elements in the matrix. For varying probability ranges the expected number of operations required to calculate the product can be determined. Then, the expected number of '1's in a row or column of a matrix is n*p.

Consider sums of the tree of the summation values

$$tree\_A_{ik} + tree\_B_{kj} \qquad 1<=k<=treesize$$

Equation 5.1 shows a test performed by the algorithm at any vertex k in the tree of summation values, will succeed if

$$tree\_A_{ik} + tree\_B_{kj} > tree\_sum_{k}$$

so, it will fail, causing further tests to occur if

$$tree\_A_{ik} + tree\_B_{kj} <= tree\_sum_{k}$$

So

Probability(test at a vertex k failing) =

$$Probability(tree\_A_{ik} + tree\_B_{kj} <= tree\_sum_{k})$$

The initial test of the sum of the summation values at the root vertices of the summation trees will always occur.
Thus Probability(testing the root vertex) = 1

This test will fail if $tree\_A_{il} + tree\_B_{lj} <= tree\_sum_{l}$.
$<= n$

Thus,

Probability(test at the root fails) = $1/2 + \delta$

where $\delta \sim O(1/n)$

In general, for any vertex k, if all tests were independent of any preceeding test then

Probability($tree\_A_{ik} + tree\_B_{kj} <= tree\_sum_{k}$) = $1/2 + \delta$

with

Probability(testing any vertex k) =

Probability(testing the previous vertex) *

Probability(test of previous vertex failed)

However, when any vertex k is tested, then the test on its parent vertex must have failed.

For any fixed p, the expected number of elements in the matrix tested before assigning a product the value of '1', will be only a constant for the elementary algorithms for boolean matrix multiplication. For the summation algorithms, this implies only a constant number of leaves of the tree will have to be searched. To search only a constant number of leaves using depth first search, O(depth) tree vertices must be tested by equation 5.1, so, the expected number of calls to Tree_scan will be $O(n^2*depth)$. The breadth first search algorithms will have to search O(treesize-n) vertices to reach the leaves of the tree, but, in fact the expected number of procedure calls will be much less than this, as the best case analysis shows that $O(n^2)$ operations are required when the average density of '1's in the matrices is greater than Ø.5.

When the density of '1's in the matrix is sparse there will be many zero values in the summation tree, Hence, both the depth first search and breadth fiest search algorithms will search only a constant number of vertices from the root vertex to the leaves in the tree. When p is a function of the inverse of n, the matrix will be sparse and the depth of

the tree determine the number of calls, so, the expected number of calls to Tree_scan for a depth first search will be $O(n^2*depth)$.

The depth of the summation tree is at most $\log_2 n$. The expected running time of the algorithm will depend on the expected number of calls to the procedure Tree_scan and will be $O(n^2*\log_2 n)$.

5.6 RESULTS

5.6.1 Implementation and Testing

Boolean matrix multiplication algorithms can be tested on randomly generated boolean matrices with the elements of the matrices created using a random number generator. To test the expected running time of the algorithms, a range of matrix sizes and densities of the occurrance of a '1' in the matrix element were generated. The number of '1's in the random matrices can be distributed over the range $\emptyset <= d <= n^2$ by applying a bias to the output of the random number generator. The matrices generated at either extreme of the density range will be termed respectively the sparse and dense matrices, the terms reflecting the number of '1's in the elements of the matrix. Matrices of size 8, 16, 32, and 64 were generated for various phases of the testing. Not all of these were used for all tests. The characteristics of the test data are summarised in Table B.2 of Appendix B.

The tree summation algorithms have been implemented in Pascal with the Sheffield Pascal Compiler on a Prime 750 at the University of Canterbury. The tree summation algorithms were implemented with a count included at the beginning of the procedure Tree_scan. The number of calls to the procedure Tree_scan were counted for the depth first search and the number of iterations were counted for the breadth first search. In the following discussion, the version of procedure Tree_scan, whether recursive or iterative, will not be distinguished, but the single term 'calls' will be used when the depth first search and breadth first search algorithms are discussed together.

5.5.2 Different Tree Structures.

The two different tree formats discussed in section 5.2 generate trees of slightly different sizes. For the matrix sizes used as test data, Table B.1 in Appendix B lists the tree size of the two formats for binary, log n and square root n summation sizes. The log n summation partition series for matrix sizes 32 and 64 has different tree sizes for the different formats. These different formats have been tested and the results are summarised in Table B.3 in Appendix B. These results show that in general, a reduction of 10% in the number of calls to the Procedure Tree_scan is achieved when the smaller tree (the tree with maximum children) is used. This tree format has been implemented in all the following algorithms.

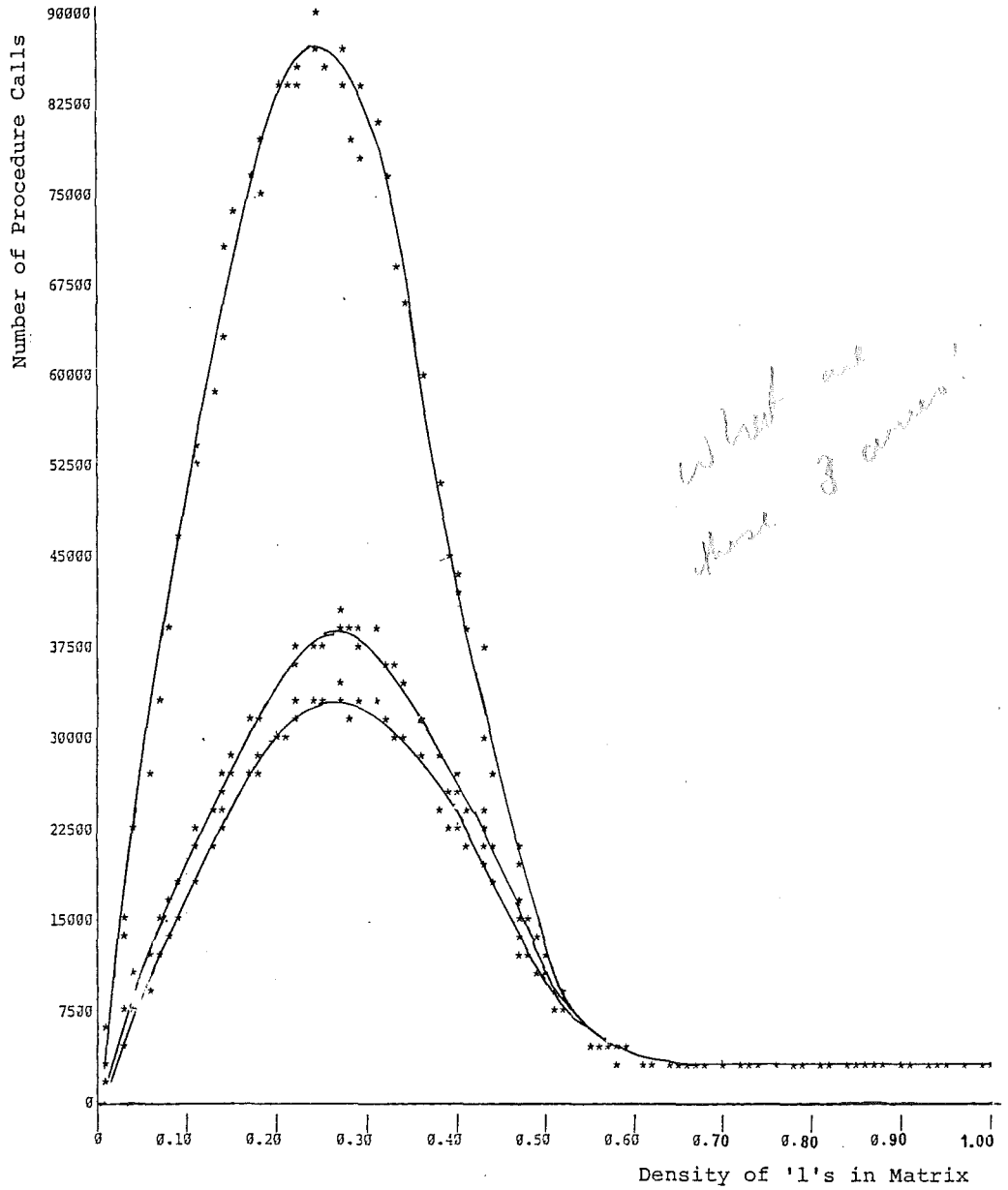
5.5.3 Search Methods and Summation Sizes

Six algorithms have been programmed and tested. These were for three different summation sizes and the two search techniques. The algorithms are:

1. Binary summation with depth first search
2. Binary summation with breadth first search
3. Log n summation with depth first search
4. Log n summation with breadth first search
5. Square Root n summation with depth first search
6. Square Root n summation with breadth first search.

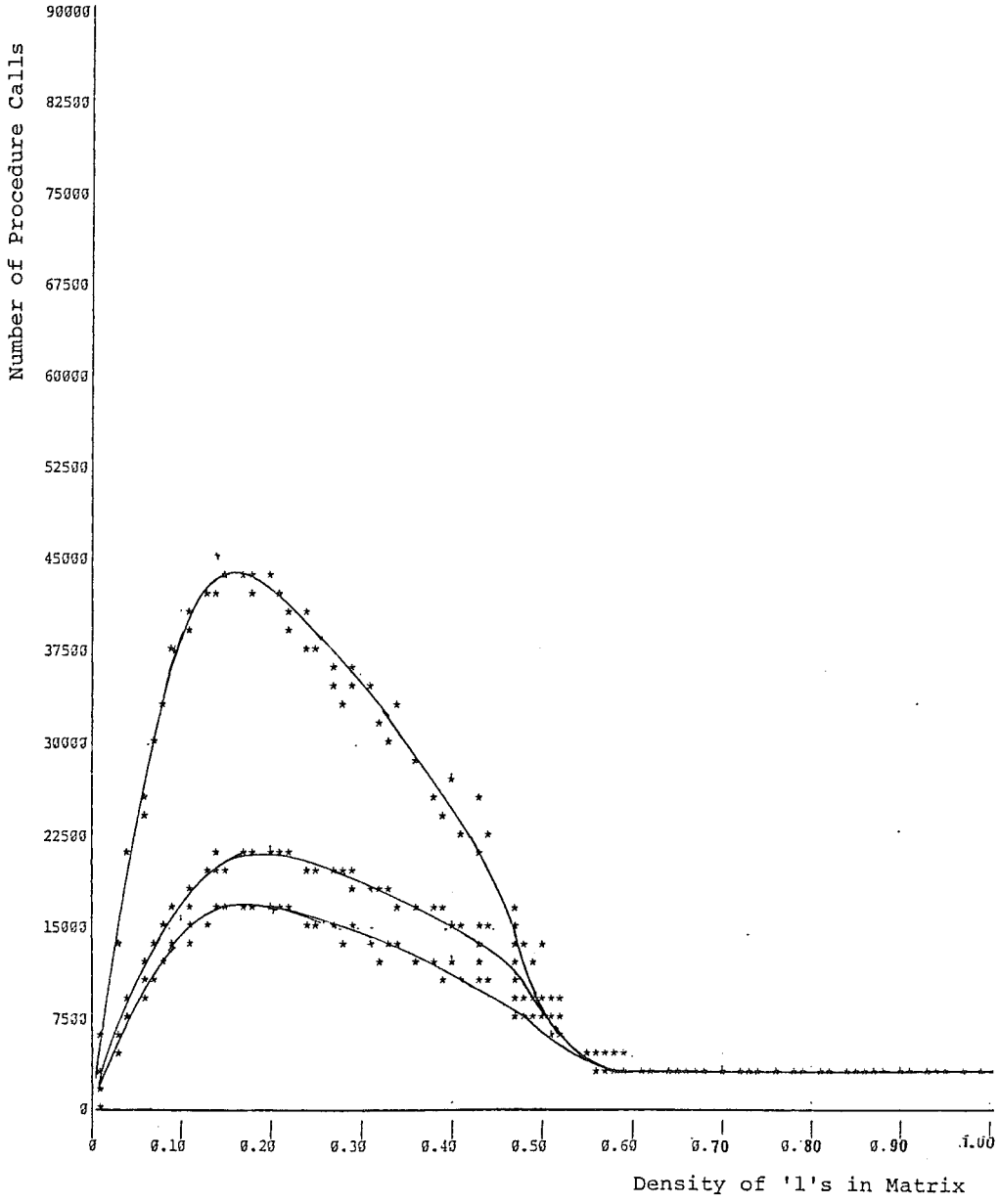
The number of calls to the procedure Tree_scan used by each of the algorithms was counted and the results are summarised in Table B.4 in Appendix B. For all algorithms the minimum and maximum number of calls are recorded and the mean number have been calculated. The density of '1's in test data was evenly distributed over the range Ø <= d <= 1, and so is not representative of the set of all possible matrices, however, it does illustrate the behaviour of the algorithms as the density of the matrices varies.

For the breadth first search algorithms, the complete set of tests results for all three summation sizes have been graphed for the matrices of size 64. Figure 5.8 shows the number of procedure calls versus the density of '1's in the test matrices for the breadth first algorithm.



NUMBER OF PROCEDURE CALLS VERSUS DENSITY OF '1's IN MATRIX FOR BREADTH FIRST SEARCH

FIG 5.8

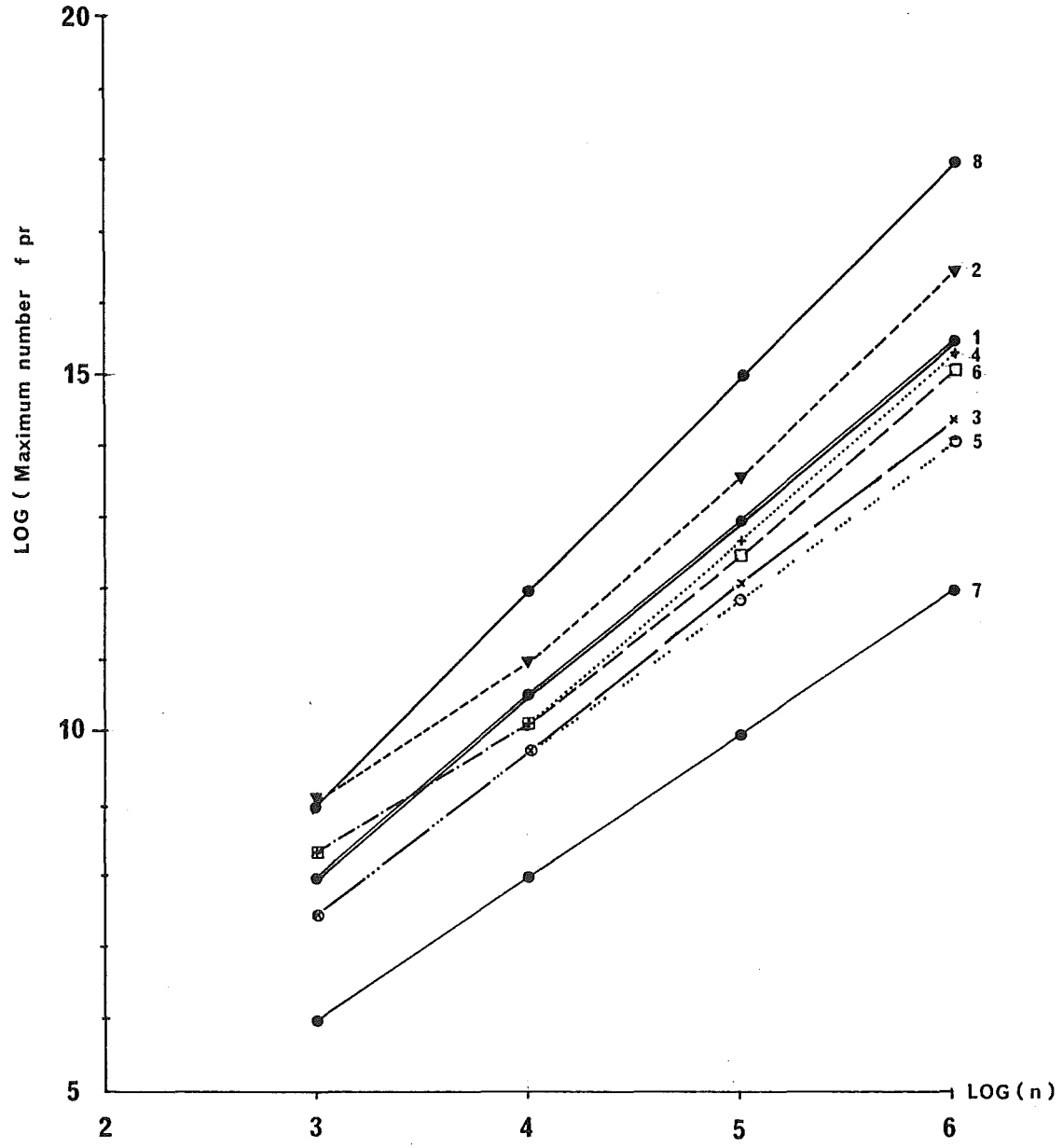


NUMBER OF PROCEDURE CALLS VERSUS DENSITY OF '1'S IN MATRIX FOR DEPTH FIRST SEARCH

FIG 5.9

For the depth first search, the complete set of test results for all three summation sizes have also been graphed for matrices of size 64. Figure 5.9 shows the number of calls to the procedure Tree_scan by depth first search algorithms graphed against the density of '1's in the test matrices. Both graphs show that the number of calls is at a maximum when the density of '1's in the matrices is in the range $1/n <= d <= n^2/2$ and that when $d > n^2/2$ the algorithms require $O(n^2)$ calls to Tree_scan and hence $O(n^2)$ running time.

The number of calls to Tree_scan and hence the running time for the algorithms lie in the range $O(n^2)$ to $O(n^3)$ and Figure 5.10 graphs the maximum number of calls to Tree_scan required by the test data against n for all algorithms. For comparison, the graphs for n^2 and n^3 are also shown in Figure 5.10. For all summation sizes, the algorithms using a depth first search consistently required fewer calls to the procedure Tree_scan and hence a lower running time than the algorithms using breadth first search. Comparing the various summation sizes, the algorithms with smaller maximun tree sizes require fewer calls to Tree_scan than the algorithms with a larger tree size. All algorithms show the number of calls increases faster than n^2, but slower than n^3.



MAXIMUM NUMBER OF PROCEDURE CALL VERSUS NUMBER OF VERTICES FOR ALL ALGORITHMS

FIG. 5.10

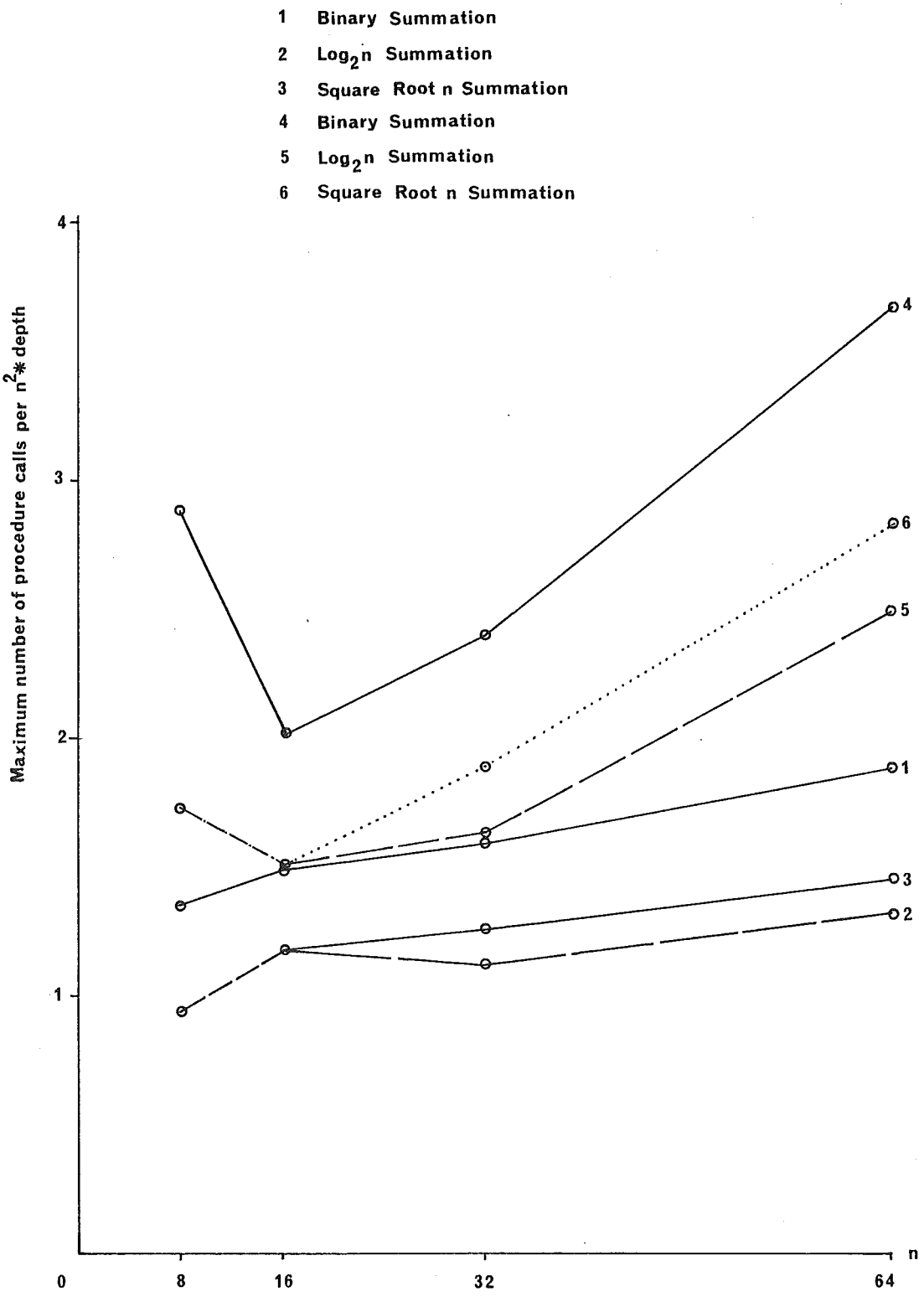
1 Binary Summation - Depth First Search
2 Binary Summation - Breadth First Search
3 $Log_2 n$ Summation - Depth First Search
4 $Log_2 n$ Summation - Breadth First Search
5 Square Root n Summation - Depth First Search
6 Square Root n Summation - Breadth First Search
7 n^2
8 n^3

In the tree of summation values, the number of vertices that must be searched, was shown in section 5.5, to depend on the depth of the tree. Figure 5.11 shows the maximum number of calls to Tree_scan per n^2*depth graphed against n. The depth first search algorithms show a strong consistant relationship between the number of procedure calls per n^2*depth and the size of the matrices. For the breadth first search algorithms, the number of procedure calls per n^2*depth shows an early decline as n increases, but then increases at a much faster rate than for the depth first search algorithms.

The depth first search algorithms require fewer procedure calls than the breadth first search algorithms, so will therefore require less running time. The smaller the tree size, the fewer procedure calls that are necessary while running the algorithm. For any matrix size, the expected number of calls to the procedure Tree_scan is $O(n^2)$ for nearly half of all possible matrices, and for the remaining matrices, the expected number of calls to procedure Tree_scan is $O(n^2*depth)$ or less than $O(n^2*\log_2 n)$.

5.7 COMPARISON WITH OTHER ALGORITHMS

The simplest methods for multiplying matrices over the real field, require $O(n^3)$ multiplication operations [Aho, Hopcroft and Ullman, 1975]. The most notable achievement in the search for more efficient methods was an algorithm by Strassen [1969] which required $O(n^{2.81})$ multiplications.



MAXIMUM NUMBER OF PROCEDURE PER n^2* DEPTH VERSUS n

FIG. 5.11

Winograd [1973] improved the constant factor but not the order of magnitude. Pan [1978] reduced the number of operations required to $O(n^{\log_{64} 111872}) \sim O(n^{2.785})$ and Bini, Capovani and Lotti [1979] reduced the number further to $O(n^{2.7799})$. Pan [1979] combined several previous techniques to derive an $O(n^{2.6654})$ algorithm, although the constant factor was seriously increased.

Boolean matrices are defined on a semiring and do not have an additive inverse. Hence, many fast and efficient methods for multiplying matrices are not applicable to the problem of multiplying boolean matrices. However a variation of Strassen's algorithm was presented by Fisher and Meyer [1971] which confines the multiplication to the ring of integers modulo 2 and normalises the result by replacing all nonzero entries with 1. Using this technique any matrix multiplication algorithm may be applied to boolean matrices without changing the efficiency of the algorithm.

Arlazarov, Dinic, Kronrod and Faradzev [1970] presented an algorithm for boolean matrix multiplication which requires $O(n^2/\log n)$ operations and applies a compression to the elements of the matrix. This algorithm, generally known as the 'Four Russians' algorithm, is described by Aho, Hopcroft and Ullman as being more 'practical' although the algorithm of Fisher and Meyer is asymptotically better.

Takaoka [1979] presented a modification to the Four Russians algorithm that improves the constant factor but not the order of magnitude. The algorithm is called the Data

Compression algorithm and it extends the table lookup feature of the Four Russians algorithm. The algorithm is described in Appendix B.

Many graph problems have been suitable applications for probabilistic algorithms. The term probabilistic has been used in the following two quite different ways. First, for NP-complete problems, where exponential time may be required to produce a correct result, a probabilistic algorithm will produce a result, usually in polynomial time, that is correct, within a probabiltiy, to the actual result. There may be a restriction to the type of graphs for which the probability of a correct solution tends to one. Second, for P-complete problems, a probabilistic algorithm will produce a result in time probably much less than the worst case. The time is usually dependent on the probability of specified events in the input data. Probabilistic algorithms for boolean matrix multiplication imply the second meaning of the term.

O'Neil and O'Neil [1973] presented a probabilistic algorithm for Boolean Matrix Multiplication. The number of operations is directly dependent on the probability of a '1' occurring in the elements of the boolean matrices. The algorithm uses an indexing method and has an expected time complexity of $O(n^2)$. The worst case requires $O(n^3)$ operations but such cases are rare. This algorithm takes advantage of the peculiar feature of taking the inner product in Boolean

arithmetic; when computing a product $c_{ij} = \Sigma_k a_{ik}.b_{kj}$ it is sufficient to find one product $a_{ik}.b_{kj} = 1$ to assign $c_{ij} = 1$ for some k and no other values of k need then be found. This heuristic will be termed 'out of loop' and it can be applied to any boolean matrix multiplication algorithm.

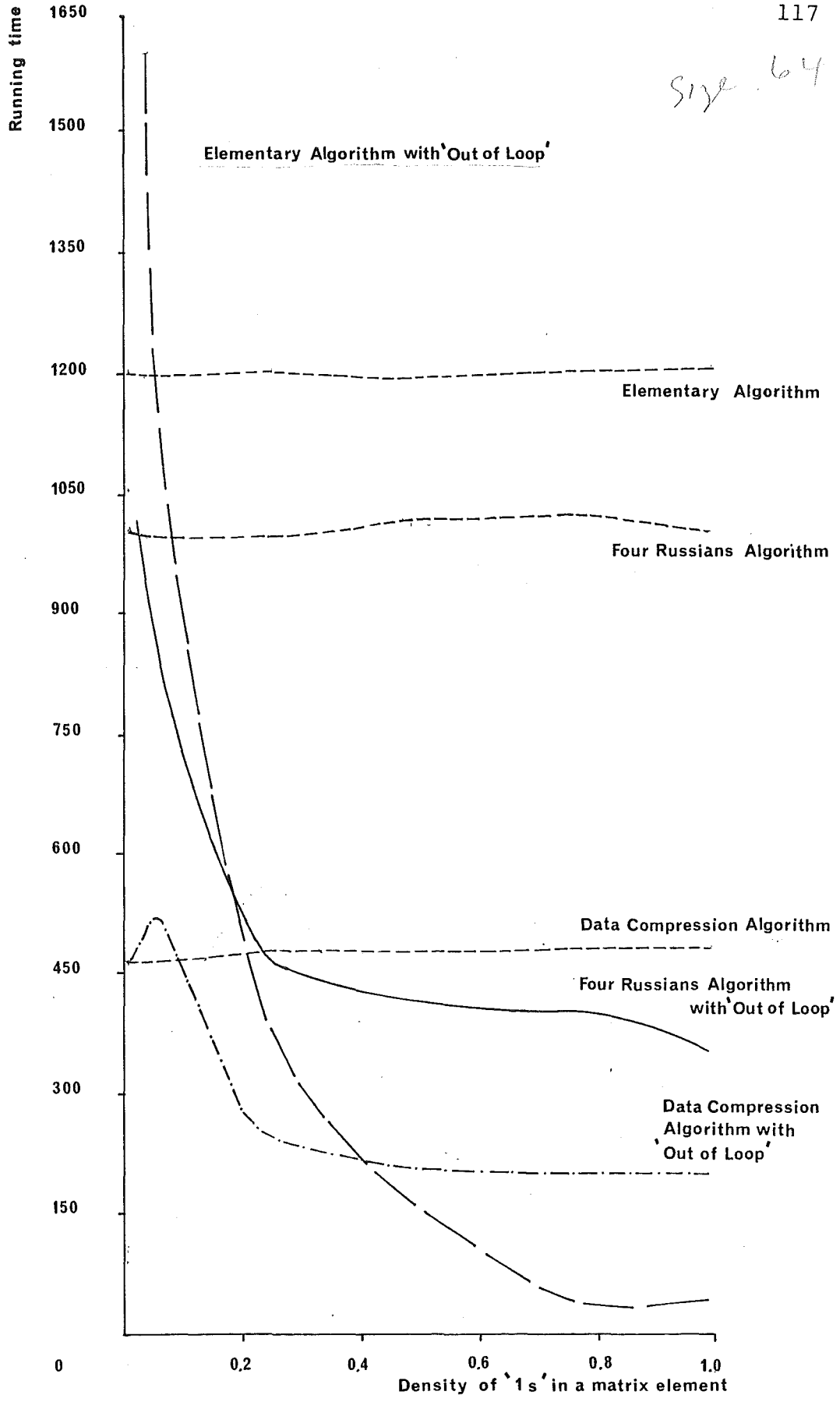
Takaoka [1979] extended the indexing method of O'Neil and O'Neil to a family of algorithms. The average and worst case running time remains the same as O'Neil and O'Neil's original algorithm, but each of the algorithms is most efficient for a different subrange of the probability of a '1' occurring in a matrix element. O'Neil and O'Neil index the rows of the A matrix and lookup the indexed element in the column of the B matrix. The family of algorithms presented by Takaoka index both matrices. The rows of A and the columns of B are indexed in the HAVB algorithm which requires $O(\min(n^3p, n^2/p))$ operations. The columns of A and the rows of B are indexed in the VAHB algorithm and the rows of A and B are indexed in the HAHB algorithm. These algorithms require $O(n^2) + O(n^3p^2)$ operations. These algorithms are described in Appendix B.

The algorithms discussed in this section have been implemented in Burroughs Extended Algol on a B6718 at the University of Canterbury. Boolean matrices were randomly generated for matrix sizes of 8, 16, 32 and 64 for fixed values in the probability range $\emptyset <= p <= 1$. The parameters of the test data are described in Table B.5 in Appendix B.

The 'out of loop' adaptation can be applied to any algorithm for boolean matrix multiplication. If each element of the matrices has, independently, a probability p of being '1' then the expected number of operations required by the elementary algorithm reduces from $O(n^3)$ to $O(n^2/p^2)$ and the Four Russians and the Data Compression algorithms reduce from $O(n^3/\log_2 n)$ to $O(n^3/(p^2*\log_2 n))$. The out of loop variation has been tested for these three algorithms. Table B.6 in Appendix B summarises the running times required by each algorithm and the 'out of loop' version for the matrices of size 64. On average the running time is halved by using the 'out of loop' heuristic. In the worst case the running time is increased by the extra 'out of loop' test added to the algorithm, in the best case the reduction is $O(n)$. The variation is dependent on the density of '1's in the matrices and Figure 5.12 graphs the running time of each algorithm and the out of loop version against the density for matrices of size 64. This shows the effectiveness of the heuristic over the probability range, including the increase in time for very sparse matrices. All algorithms discussed in the remainder of this chapter will include the 'out of loop' heuristic.

The algorithms for boolean matrix multiplication can be divided into two classes. The compression methods of the Four Russians and Data Compression algorithms combine several elements of the matrix to a single element of a smaller matrix. The indexing methods of O'Neil and O'Neil and Takaoka eliminate the zero elements of the matrix by replacing each row or column of the matrix with a list of

Size 64

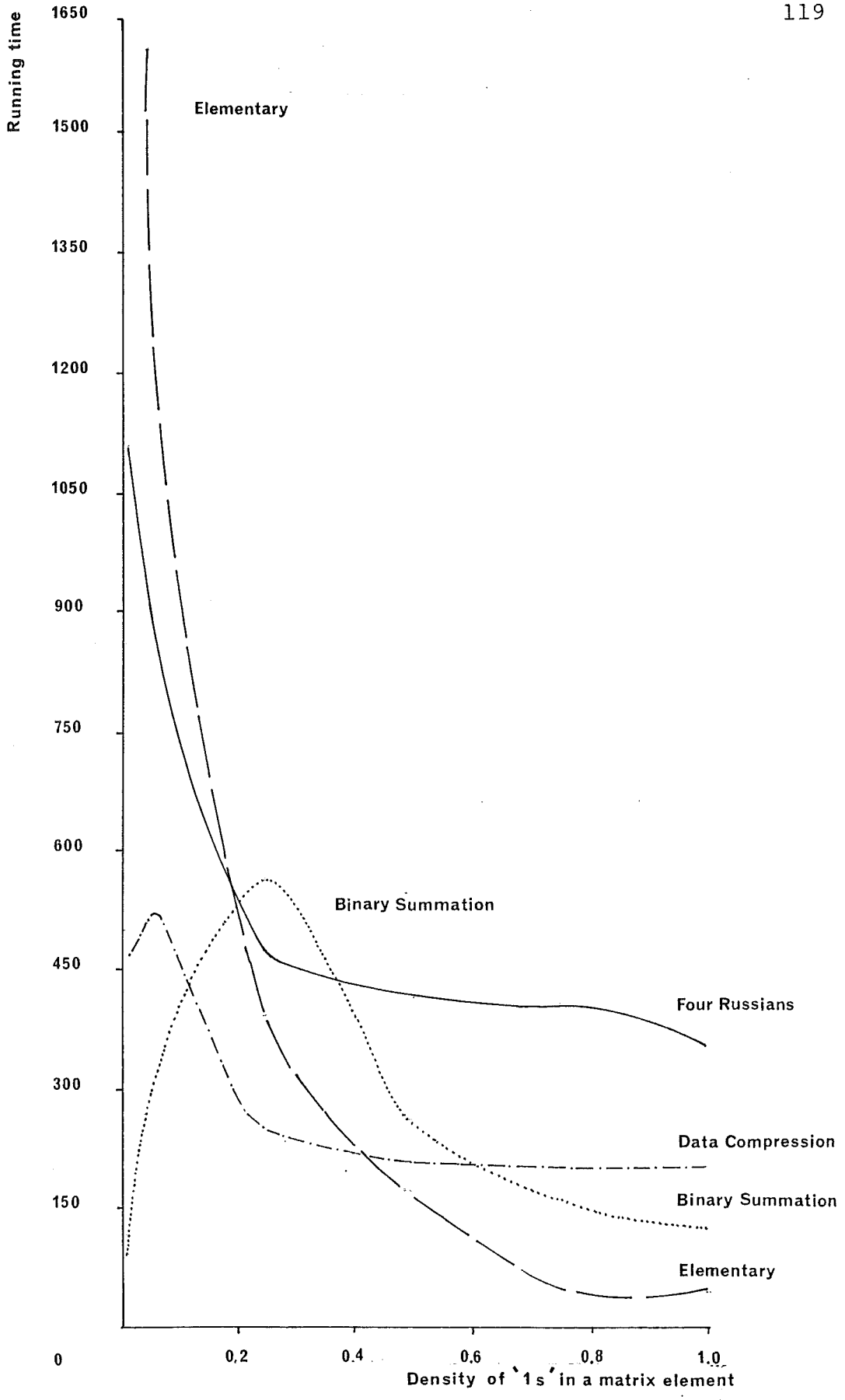


RUNNING TIME IMPROVEMENTS USING THE 'OUT OF LOOP' HEURISTIC

FIG. 5.12

the non-zero elements. The tree summation algorithms compress the elements of the matrix by partitioning a row or column with a partition series. These algorithms have been tested with the matrices described in Table B.5 and the results are summarised in Table B.7 and B.8 in Appendix B.

The results of running the elementary algorithm and the compression algorithms over the test data are summarised in Table B.7. For the smaller matrix sizes tested, namely 8 and 16, the less complex algorithms run faster but, for larger sizes the overhead incurred building the compression does not overwhelm the running times. Figure 5.13 shows the running times for the matrices of size 64. The running time has been graphed against the density of '1's occurring in the matrix and shows the elementary algorithm is more efficient as the density tends to 1. Takaoka's modification to the Four Russians algorithm, the Data Compression algorithm, consistently requires less running time than the Four Russians algorithm, and the Binary Summation algorithm requires less time for half the matrices tested, but the Data Compression algorithm requires less time than the Binary Summation algorithm for the other half of the matrices. The maximum running time is similar for both the Binary Summation and Data Compression algorithms and much less the the Four Russians and Elementary algorithms. For increasing matrix size, the rate of increase of the running time of the Data Compression algorithm is less than the Binary Summation algorithm.

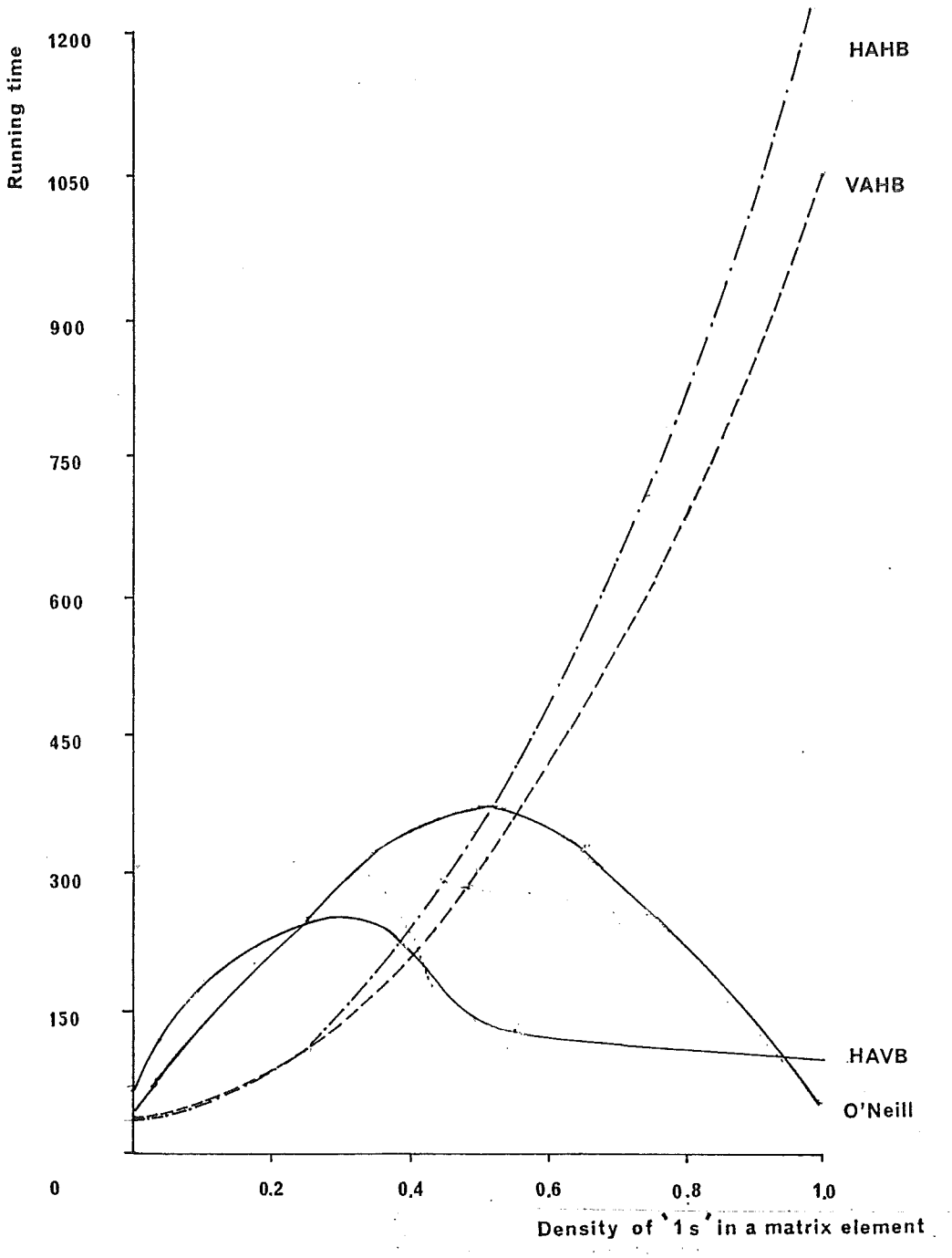


RUNNING TIME FOR ELEMENTARY AND COMPRESSION ALGORITHMS

FIG. 5.13

The results of running the indexing algorithms of O'Neil and O'Neil and Takaoka over the test data are summarised in Table B.8. For the smaller matrix sizes there is less overhead incurred in building the index lists than the compressions of the Four Russians and Data Compression. The VAHB and HAHB algorithms have the lowest minimum running time and the HAVB and O'Neil algorithms have the lowest maximum running time. The algorithms are efficient for different values of the density range. Figure 5.14 shows the running times for the matrices of size 64 graphed against the density of a '1' occurring the the matrix.

Both the HAHB and the VAHB algorithms are efficient for p -> Ø, but the time requirements increase as p -> 1 as these algorithms cannot use the 'out of loop' heuristic. The VAHB algorithm requires slightly less time than the HAHB algorithm, for all except the lowest values of p. Both the O'Neil and O'Neil and the HAVB algorithms are not as efficient for matrices with a density Ø <= d <= Ø.5, but are more efficient for d > Ø.5 when the 'out of loop' heuistic can be applied. Except at the extreme ends of the probability range, where the different time required to preprocess one or two matrices is important, the HAVB algorithm requires less running time than the O'Neil and O'Neil algorithm. As the size of the matrices increases, the HAVB algorithm also shows the slowest increase in the maximum running time than all other algorithms.



RUNNING TIME FOR INDEXING ALGORITHMS

FIG. 5.14

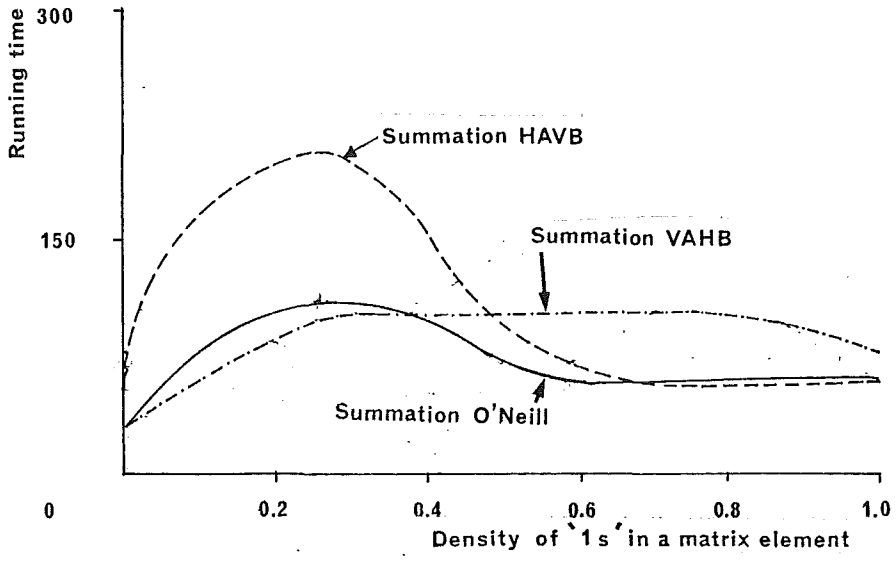
5.8 APPLYING A SUMMATION HEURISTIC TO OTHER ALGORITHMS

The use of the 'out of loop' heuristic has produced improvements in the algorithms for boolean matrix multiplication. The tree summation algorithm described in this chapter also suggests a heuristic that can be used to improve the average case efficiency of other algorithms for boolean matrix multiplication. The Binary Summation algorithm requires $O(n^2)$ processing time to find the product matrix when the density of '1's occurring in an element of the matrices is greater than Ø.5. A single test will determine if the sum of the number of '1's in row i of matrix A and column j of matrix B is greater than n. If this test is successful, then $c_{ij} = 1$.

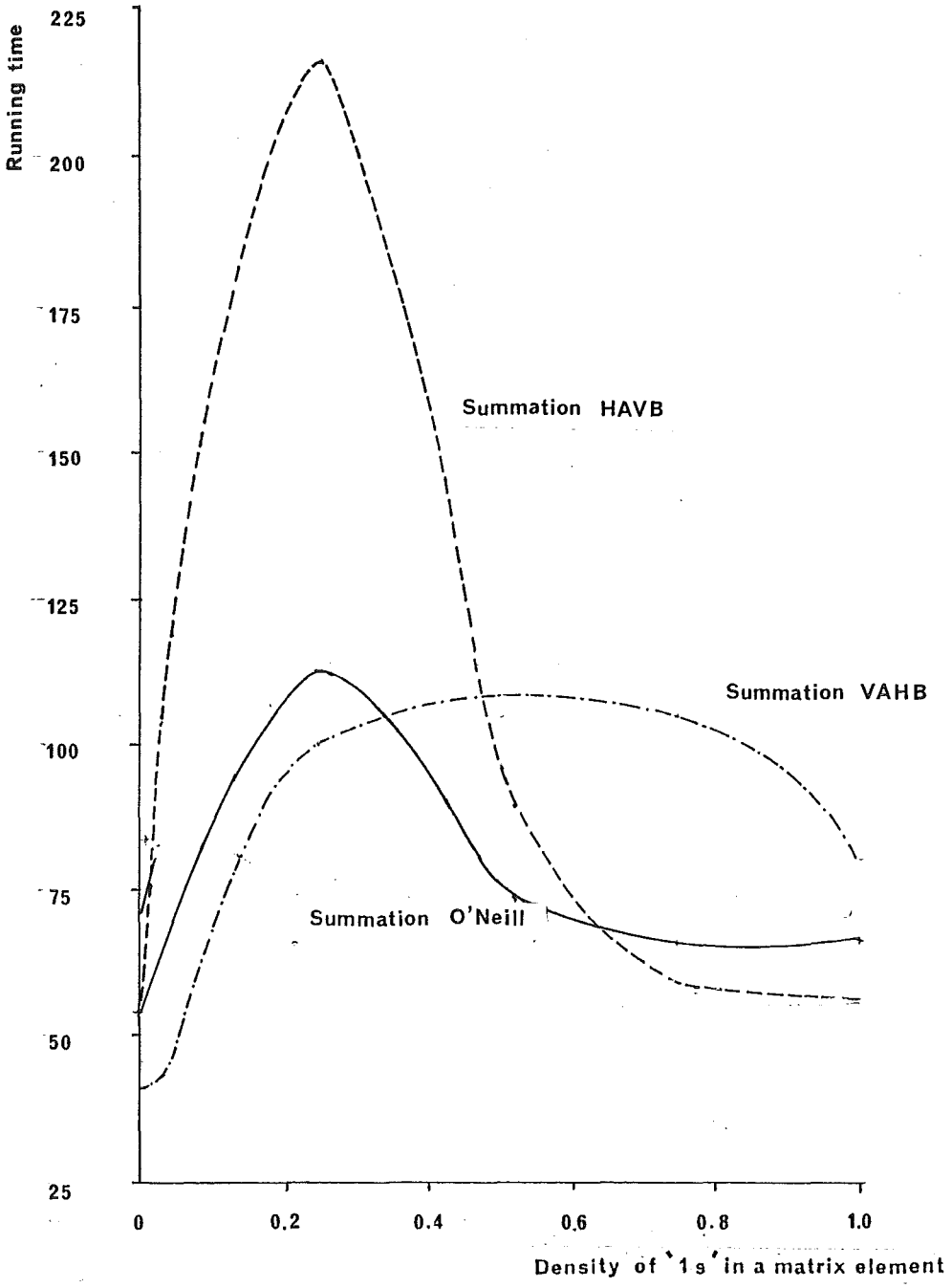
The 'summation' heuristic adds this single test to algorithms for boolean matrix multiplication. A single count of the number of '1's in each row and column respectively of the matrices can be computed in $O(n^2)$ time and stored in a list requiring O(n) space. This process can be added to known algorithms without changing the order of magnitude of their time or space requirements and is particularly applicable to those algorithms which involve a preprocessing section as part of the algorithm. The 'summation' heuristic has higher probability of success as p -> 1.

All algorithms, except the elementary algorithm, discussed in section 5.7 convert the original boolean matrix representation to a compressed matrix or a list structure. From the results obtained testing the algorithms, the

indexing methods of O'Neil and O'Neil and Takaoka require the least running time. The 'summation' heuristic has been added to the algorithm of O'Neil and O'Neil and to the HAVB and VAHB algorithms of Takaoka. The results from testing these algorithms are summarised in Table B.9 in Appendix B. The mean and maximum running times have been halved for the O'Neil and O'Neil algorithm with the minimum increasing slightly as the B matrix must be preprocessed for the 'summation' heuristic. The running times for the HAVB algorithm are reduced slightly, but the most dramatic improvement has been for the VAHB algorithm with all running times reduced by the heuristic and the maximum running time reduced by O(n). Figure 5.15 shows the running time for the matrices of size 64 graphed against the density of '1's occurring in any matrix element on the same scale as the preceeding graphs. This shows the running times relative to the preceeding algorithms. Figure 5.16 redraws the graph of Figure 5.15 on a larger scale.



SCALE REPRESENTATION OF THE RUNNING TIMES FOR ALGORITHMS USING THE 'SUMMATION' HEURISTIC FIG. 5.15

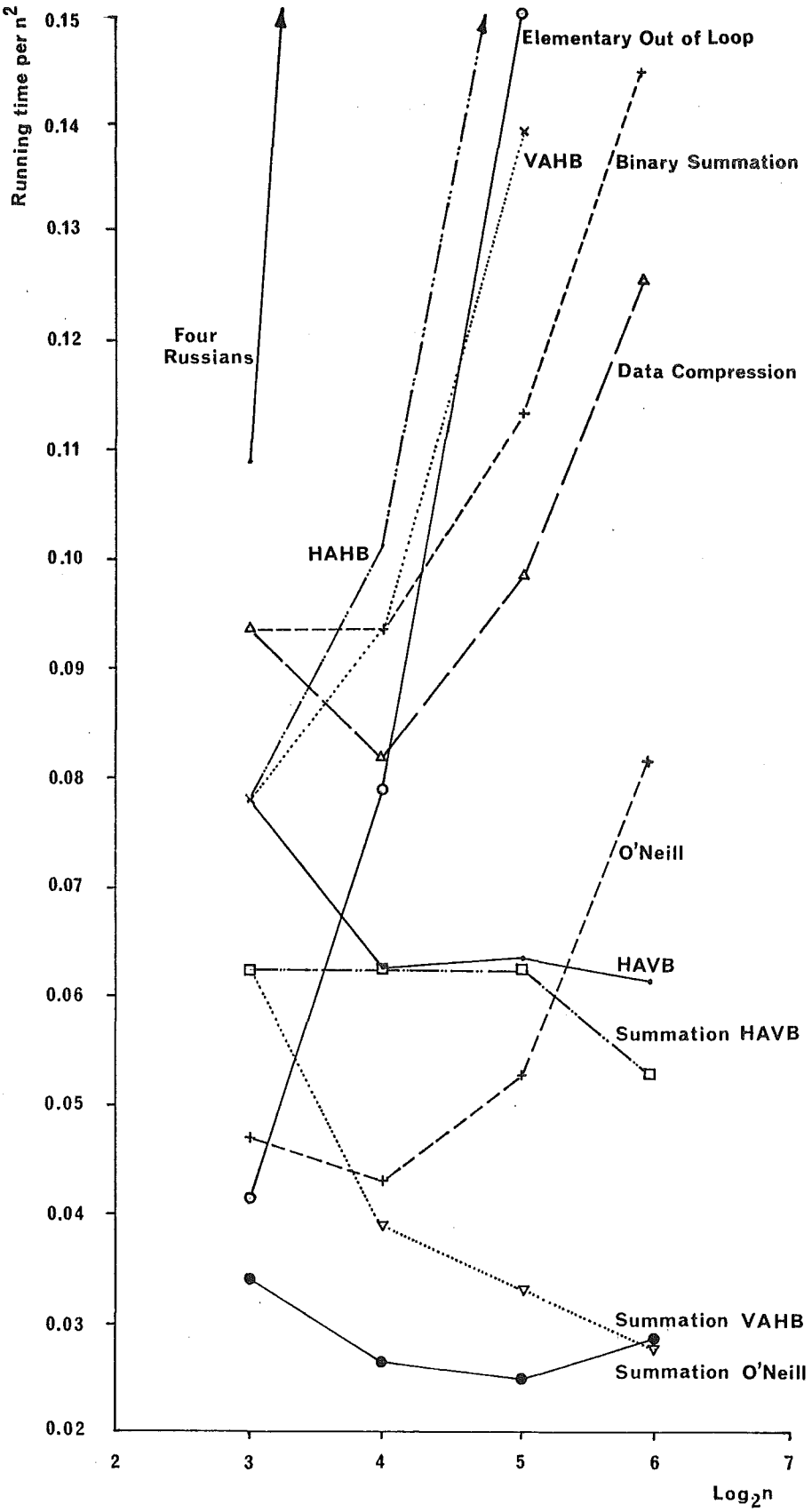


RUNNING TIME FOR ALGORITHMS WITH 'SUMMATION' HEURISTIC

FIG. 5.16

The HAVB algorithm is most efficient as the density tends to 1 and the 'summation' heuristic reduces the running time slightly. The VAHB algorithm is most efficient as the density tends to Ø but as the density tends to 1 it requires more running time than most of the other algorithms tested. The 'summation' heuristic dramatically reduces the running time as the density tends to 1, without increasing the running time where the algorithm is already efficient. The algorithm of O'Neil and O'Neil is most efficient for both sparse and dense matrices. The 'summation' heuristic reduces the running time of average graphs so that the overall result is similar to the VAHB algorithm with the 'summation' heuristic.

The maximum running times per n^2 for increasing n are shown in Figure 5.17. This gives an overview of the relative running times for all the algorithms. The algorithms with the 'summation' heuristic have the minimum running times and minimum increase as n increases. In fact, for the range of matrix sizes, the running times for the VAHB algorithm with 'summation' is decreasing as n increases. The other 'summation' algorithms and the HAVB algorithm show $O(n^2)$ running times while all other algorithms show steeply increasing running times as n increases.



RUNNING TIME PER n^2 VERSUS $LOG_2 n$ FOR ALL ALGORITHMS

FIG. 5.17

5.9 SUMMARY

A family of algorithms for boolean matrix multiplication has been presented. The algorithms apply a compression to the elements of the matrices and then use a tree data structure of the compressed data to represent the matrices. Analysis of the complexity of the algorithms has shown that these algorithm are efficient for the average case, requiring $O(n^2*\log_2 n)$ running time for half of all possible matrices, and $O(n^2)$ running time for the other half of all possible matrices, but in the worse case they require $O(n^3)$ running time. Tests of the algorithms confirm the analysis and have shown that the depth first search methods give a more efficient algorithm, and also that the smaller the tree size used to represent the matrices, the lower the running time required by the algorithm.

While trees have been used to efficiently solve many problems in graph theory they have not previously been applied to boolean matrix multiplication. Compression methods have been applied to boolean matrix multiplication previously and the Binary Summation algorithm has been compared with compression algorithms and also with indexing algorithms for boolean matrix multiplication. The maximum (over the test data) running time of the algorithm as the size of the matrices increase, is increasing at a rate faster than the Data Compression algorithm, but the average running time is increasing at a rate slower than all the compression methods. However, when all the compression

methods are compared against indexing algorithms for boolean matrix multiplication the algorithms of O'Neil and O'Neil and the HAVB algorithm of Takaoka require less running time.

The tree summation algorithms have lead to the introduction of a new heuristic that can be included with the 'out of loop' heuristic in all algorithms for boolean matrix multiplication. The 'summation' heuristic can be used by algorithms which cannot use the 'out of loop' heuistic. The 'summation' heuristic added to the indexing algorithm of O'Neil and O'Neil and the VAHB algorithm of Takaoka has produced the most efficient algorithms of all algorithms tested.

CHAPTER 6

AN EFFICIENT AVERAGE TIME ALGORITHM FOR THE GENERATION OF ALL MAXIMAL INDEPENDENT SETS OF A GRAPH

6.1 INTRODUCTION

The cliques of a graph are equivalent to the maximal independent sets in the complement graph and they are used in many contexts. For example, in graph theory they are used in association with the colouring problem [Halin, 1982], in switching theory for state minimization [Paull and Unger, 1959], in operations research for scheduling [Gorenstein, 1972] and in information systems for representing relationships in the data [Atkin and Casti, 1977]. The cliques of a graph are vertex invariant and uniquely define a graph, which suggests that they may be used to represent a graph. Read and Corneil [1977] report that Knodel [1971] suggested using this to determine graph isomorphism, but for both applications the cliques of a graph are of only restricted use as the problem of finding all the maximal independent sets or cliques of a graph is known to be NP-complete [Garey and Johnson, 1979].

Finding a clique or maximal independent set in a graph requires only polynomial time but the maximum possible number of cliques in a graph was show by Moon and Moser

[1965] to be $O(3^{n/3})$, so the size of the solution set is exponential in the size of the problem description. This is one of several problems in the field of graph theory for which there exist graphs with exponentially large solutions, but the expected size of the solution set is very much smaller and, for specific classes of graphs, the solution is known to be polynomial [Papadimitriou and Yannakakis, 1981], and so can be found in polynomial time.

The only algorithm with a proven bound that is linear in the number of maximal independent sets is the algorithm of Tsukiyama, Ide, Ariyoshi and Shirakawa [1977]. While testing an implementation of this algorithm as part of the study of data representations undertaken in this thesis, it was observed that the ordering of the vertices in the graph affected the running time. Tsukiyama et al state:

> "Let all the vertices in A=r(x) be arranged in an arbitrary order with subscripts 1 through p = |A| [pp 508] . . . . In the order of subscript numbers of y_i e A=$\{y_j\}$ j=1,p . . "[pp 512]

In Chapter 3 the fact that many non-topological data representations impose an ordering relationship that may not exist on the graph has been discussed. In this chapter the effect on the algorithm of Tsukiyama et al of reordering the vertices in the graph is investigated and the algorithms are compared with other efficient algorithms for finding the cliques or maximal independent sets of a graph.

6.2 THE ALGORITHM OF TSUKIYAMA, IDE, ARIYOSHI AND SHIRAKAWA

The algorithm presented by Tsukiyama, Ide, Ariyoshi and Shirakawa builds all possible maximal independent sets by a recursive process that tests, at a seperate level in the recursion, each vertex of the graph for possible inclusion in the maximal independent set. If the algorithm does not include a vertex it will backtrack to test for a maximal independent set that may include that vertex but exclude some vertices previously included in the maximal independent set. Each maximal independent set of a graph will be generated exactly once by the algorithm. The outline in Algorithm 6.1 is from Tsukiyama, Ide, Ariyoshi and Shirakawa.

The algorithm of Tsukiyama et al uses an O(n+m) space adjacency list (Adj) to store the graph and an O(n) space vector (IS) is used to store the partially generated independent set. At each level of recursion, to the limit of n levels, the values in the array IS may be altered. To be able to restore the independent set, the O(n+m) space array BUCKET is used to record any alterations. Thus, the space requirements for the algorithm is O(n+m).

If an element in the array IS has the value zero then the corresponding vertex is potentially a member of the maximal independent set. A non-zero value indicates that the vertex is adjacent to some of the vertices already included in the independent set. The actual value is the number of members of the independent set to which the vertex is adjacent.

Algorithm 6.1

Input: A graph G = (V,E) where n = |V|, represented by the adjacency lists Adj(x) for all vertices x of G.

Output: The Maximal Independent Sets of G.

Initialisation:

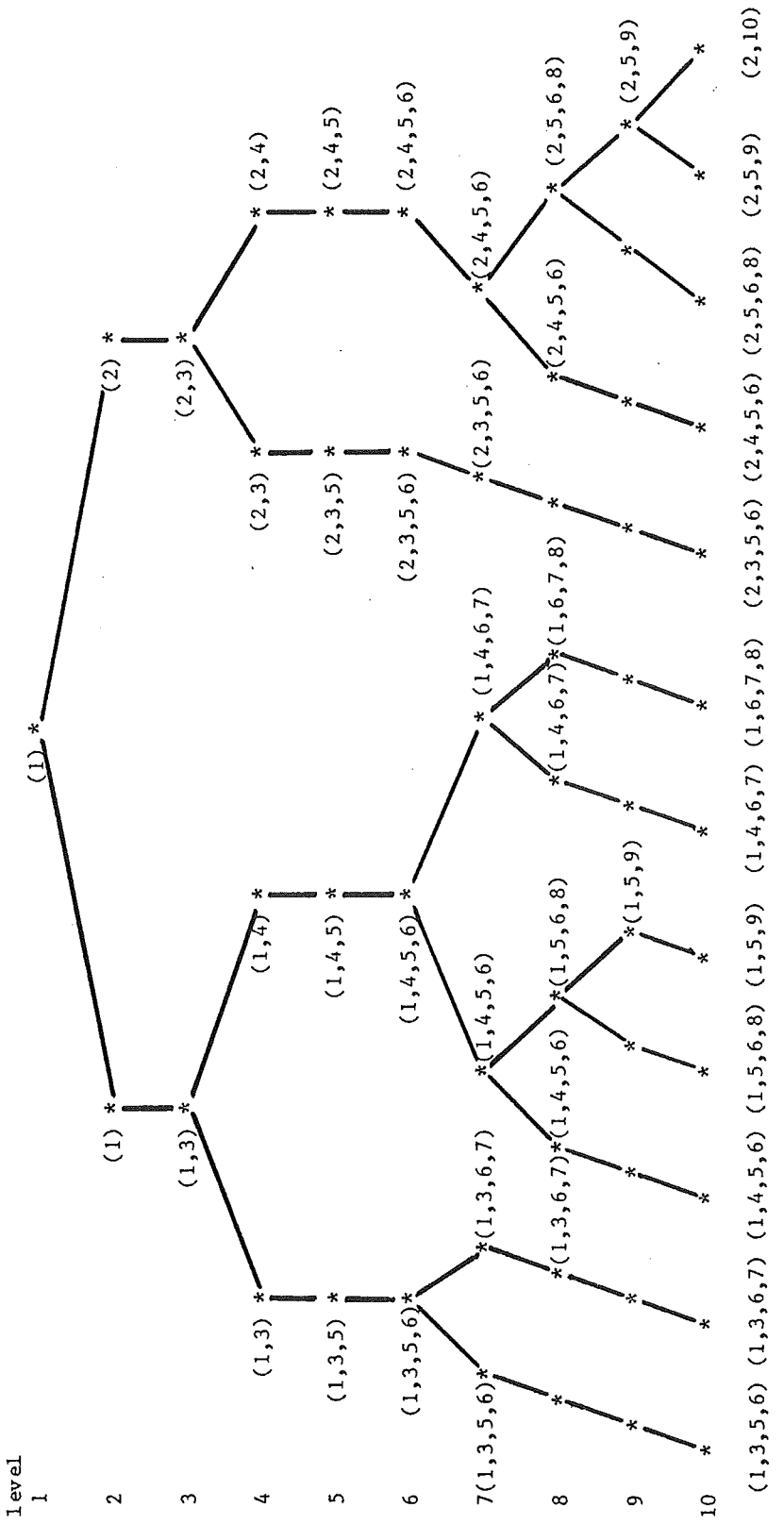
```
for j := 1 to n do begin
  IS(j) :=Ø
  Bucket(j) := 0  end
BACKTRACK(1)
```

Method:

```
procedure BACKTRACK(i)
begin
  if i < n then begin
    x := i + 1
    c := Ø
    for y in Adj(x) such that y <= i do
      if IS(y) = Ø then c := c + 1
    if c = Ø then begin
      for y in Adj(x) such that y <= i do IS(y) := IS(y) + 1
      BACKTRACK(x);
      for y in Adj(x) such that y <= i do IS(y) := IS(y) - 1
      end
    else begin
      IS(x) := c
      BACKTRACK(x)
      IS(x) := Ø
      f := true
      for y in Adj(x) such that 1<=y<=i in increasing order
       do begin
        if IS(y) = Ø then begin
          put y in Bucket(x)
          for z in Adj(y) such that z <= i do begin
            IS(z) := IS(z) - 1
            if IS(z) = Ø then f := false
            end
          end
          IS(y) := IS(y) +1
          end
        if f then BACKTRACK(x)
        for y in Adj(x) such that y <= i do IS(y) := IS(y)-1
        for y in Bucket(x) do begin
          for z in Adj(y) such that z <= i do IS(z):=IS(z)+1
          delete y from Bucket (x)
          end
        end
      end
    else  output new maximal independent set designated
          by IS
    end
```

(end of method)

Example 6.1



GENERATING MAXIMAL INDEPENDENT SETS USING TSUKIYAMA ET AL ALGORITHM

FIG 6.1

The (1,2)-tree in Figure 6.1 illustrates the process of finding all maximal independent sets in the graph of Figure 2.1 using the algorithm of Tsukiyama et al. The maximal independent sets of the graph in Figure 2.1 are shown in Example 2.6.
(end of example)

The procedure starts with all vertices members of an independent set, in particular the vertex labelled 1 is included in the set. The first call to BACKTRACK will then test the vertex labelled 2 for inclusion in the set. If 2 is adjacent to 1 then 2 will be excluded from the set during the first recursive call to BACKTRACK, and it may or may not subsequently be included in the set for a second recursive call to BACKTRACK. A second recursive call is made if and only if no new members of the maximal independent set are added by an examination of the adjacent vertices and this ensures that each maximal independent set is generated only once. When all n vertices of a graph have been tested, those vertices with an adjacency count of zero are members of a maximal independent set.

Each recursive call to the procedure BACKTRACK with i less than n will test the vertex x := i+1 and will follow one of two distinct possible paths. First, if the vertex being tested (x) is not adjacent to any previously tested vertex that is a member of the independent set then it is a member of the independent set. The adjacency counts of any vertices adjacent to the current vertex x are updated and BACKTRACK is called recursively. There will be no second call to

BACKTRACK, so in Figure 6.1 the recursion tree shows the path from vertex i to vertex x (= i + 1) as a vertical line and the call to BACKTRACK will be called a type 1 recursive call or more simply a type 1 call. Second, if the vertex (x) is adjacent to any previously tested vertex that is a member of the independent set, then x is not a member so the algorithm must test two possible independent sets. The first will not include the vertex x so the adjacency count of vertex x is updated and BACKTRACK is called recursively. The call to BACKTRACK will be called a type 2 recursive call or simply a type 2 call. To show that there are potentially two possible sets that may be tested, Figure 6.1 shows the recursion tree with vertex x to the left of vertex i and the path from vertex i to vertex x as an angled line sloping down to the left. The second independent set that may be tested includes vertex x but excludes any previous members to which it is adjacent. This will be termed a type 3 recursive call or simply a type 3 call. To ensure that each maximal independent set is generated only once the process of adjusting adjacency counts when setting up this second independent set must not add any new members to the set. If it does, the recursive call of type 3 will not proceed. In Figure 6.1 when a second or type 3 recursive call occurs, the vertex x is shown to the right of the vertex i and the path from i to x is shown as an angled line sloping down to the right.

Tsukiyama et al show that each maximal independent set of a graph G=(V,E), where n=|V|, m=|E| and c is the number of cliques in the graph, will be generated exactly once by the

algorithm and that each call to the procedure BACKTRACK requires time dependent on the degree of the vertex being tested. All the n vertices of the graph must be tested, so the time required to find a maximal independent set is O(n*m) and the total processing time is O(n*m*c).

6.3 MODIFICATIONS TO THE ALGORITHM

The running time of a recursive algorithm accrues from the number of times the procedure is invoked as well as from the running time of the procedure, so both these aspects must be considered when trying to improve the efficiency of the algorithm.

Consider first the running time of each call to the procedure BACKTRACK(i). Every call will search the adjacency list of vertex i+1 to count the number of vertices in the current independent set to which i+1 is adjacent. This will require at most O(d(i+1)) time where d(i+1) is the degree of the vertex i+1. This count forms the basis for deciding whether the next call to BACKTRACK(i+1) will be a type 1 call or a type 2 call.

If a type 1 call occurs, the adjacency count of the previously tested vertices must be updated, the recursive call is made and, on the return from the call the adjacency count must be restored. So a type 1 call will require at most O(d(i+1)) running time.

If a type 2 call occurs, vertex i+1 is excluded from the independent set by a single assignment, the recursive call is made amd then the possible independent set which includes vertex i+1 must be investigated. All vertices adjacent to vertex i+1 must have their adjacency counts updated by searching their adjacency list. This will require $\Sigma d(y)$ for all y in Adj(x) or on average $O(d(v)^2)$ where d(v) is the average degree of the vertices. If no new vertices are added to the independent set a type 3 recursive call to procedure BACKTRACK will be made and on return from this call or if no call is made the adjacency count must be restored, again requiring $O(d(v)^2)$ running time.

A type 3 call to BACKTRACK will only occur after a type 2 call and all the updating and restoring in preparation for a type 3 call will be performed whether or not the call occurs.

Thus the running time for each possible call to the procedure BACKTRACK is

$O(d(v))$ for a type 1 call

$O(d(v)^2)$ for a type 2 call

and $O(1)$ for a type 3 call

The number of calls made to the procedure BACKTRACK can be discussed in terms of the recursion tree in which each recursive call to the procedure is represented by a vertex. There will always be exactly n levels of recursion to find each maximal independent set in the graph and, as there are c maximal independent sets, there will be exactly c leaves

in the recursion tree. Each internal vertex in the tree will have either one or two children. A (1,2)-tree of height n with exactly c leaves will have c-1 binary vertices and the remaining vertices in the tree will be unary vertices. The binary vertices in the recursion tree represent the occurrances of type 3 calls to BACKTRACK and there will therefore be only c-1 type 3 calls made by the algorithm for any graph with c maximal independent sets. The unary vertices in the recursion tree may be either type 1 calls to BACKTRACK or type 2 calls that are not followed by a type 3 call. The total number of vertices in the recursion tree will depend on where, in the recursion, the type 3 vertices occur. The maximum number of levels on which a type 3 vertex may occur is min(c-1,n) and the minimum number of levels of the tree the type 3 vertices can occupy is $|^{-}\log_2(c-1)^{-}|$.

Example 6.2

The (1,2)-tree of Figure 6.1 has 12 leaves, one for each of the 12 maximal independent sets generated for the graph shown in Figure 2.1. There are exactly 11 binary vertices in the tree and, of the remaining 44 vertices in the tree, 14 vertices represent type 1 calls to procedure BACKTRACK and 30 vertices represent type 2 calls to procedure BACKTRACK. Type 3 recursive calls to procedure BACKTRACK occur at 6 levels in the tree.

(end of example)

Without loss of generality the following analysis assumes that the number of maximal independent sets in a graph is $c = 2^k$ for any k.

Clustering the type 3 vertices together near the leaves of the tree will produce the tree with the minimum possible number of vertices. There will always be c leaves and they will terminate a binary tree with 2*c-1 vertices. This binary tree will have a height of $\log_2 c$ so the remaining $n-\log_2 c$ levels in the recursion tree will have exactly one unary vertex. This tree will have a total of

$$(2*c-1) + 1*(n - (\lceil \log_2 (c-1) \rceil))$$
$$< \quad n + 2*c \quad \text{vertices.}$$

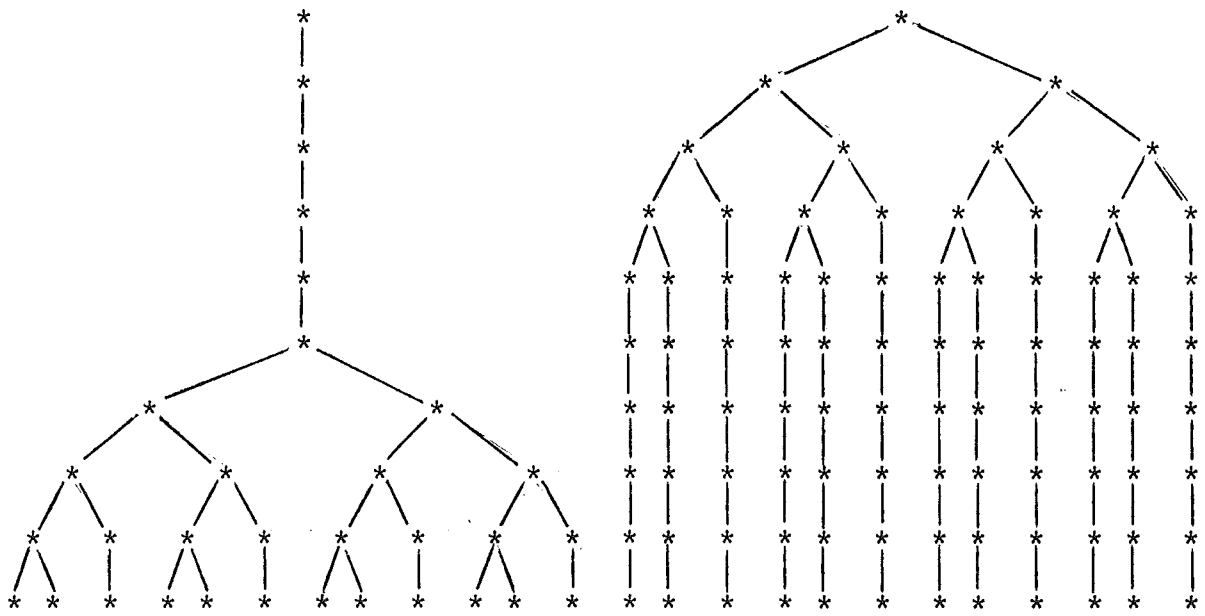
Clustering the type 3 vertices together near the root of the tree will produce the tree with the maximum possible number of vertices. A binary tree with 2*c-1 vertices will occur in the first $\log_2 c$ levels of the tree and the remaining $n-\log_2 c$ levels of the tree will have c unary vertices at each level. A total of

$$(2*c-1) + c * (n - \lfloor \log_2 (c-1) \rfloor)$$
$$< \quad c*n + 2*c \quad \text{vertices.}$$

Example 6.3

Figure 6.2 shows the two extreme forms that can occur for a graph on 10 vertices with 12 maximal independent sets. Figure 6.2 (a) shows all binary vertices clustered at the leaves of the tree which gives a total of 32 vertices in the

tree. Figure 6.2 (b) shows the binary vertices clustered at the root of the tree which gives 87 vertices in total. The actual recursion tree shown in Example 6.1 had 55 vertices.



(a) (b)

POSSIBLE RECUSION TREES **FIG 6.2**

(end of example)

In general, the algorithm will not function at the above minimum or maximum values and to alter the processing to cause the minimum number of recursive calls would require knowing the members of the maximal independent sets. There is, unfortunately, no way of determining this prior to the completion of the algorithm. However, each level in the recursion tree represents a vertex tested by the procedure BACKTRACK and the two extremes of the recursion tree both show that vertices which are tested at lower levels in the

tree will be tested more frequently than vertices tested at higher levels in the tree. For example, at the leaves of the tree the maximal independent sets have been generated and no vertices are tested, but at the level of the tree immediately prior to the leaves, the n^{th} vertex in the graph will be tested O(c) times.

As discussed earlier, each procedure call will require O(d(v)) running time if it is a type 1 call and $O(d(v)^2)$ if it is a type 2 call. The total running time of the algorithm may be reduced by having type 2 calls occur at the levels of the tree nearer the root, and by having the type 1 calls occur at the levels of the tree near the leaves. There are fewer vertices in each level of the tree nearer the root so the type 2 calls will occur less frequently and the type 1 calls near the leaves of the tree will occur more often. This may not reduce the size of the tree, but if the overall number of type 2 calls in the recursion is reduced by 1, then the running time will be reduced by $O(d(v)^2)$. Hence, the total number of type 1 calls may increase by O(d(v)) without increasing the running time of the algorithm, although some extra time will be necessary to invoke the extra procedure calls.

A type 1 call occurs if the vertex is not adjacent to current members of the independent set, and since the probability that a vertex is not adjacent to other vertices increases as the degree of the vertex decreases, the type 1 calls will occur more frequently for the vertices of lower degree in the graph. Conversely, the vertices of higher

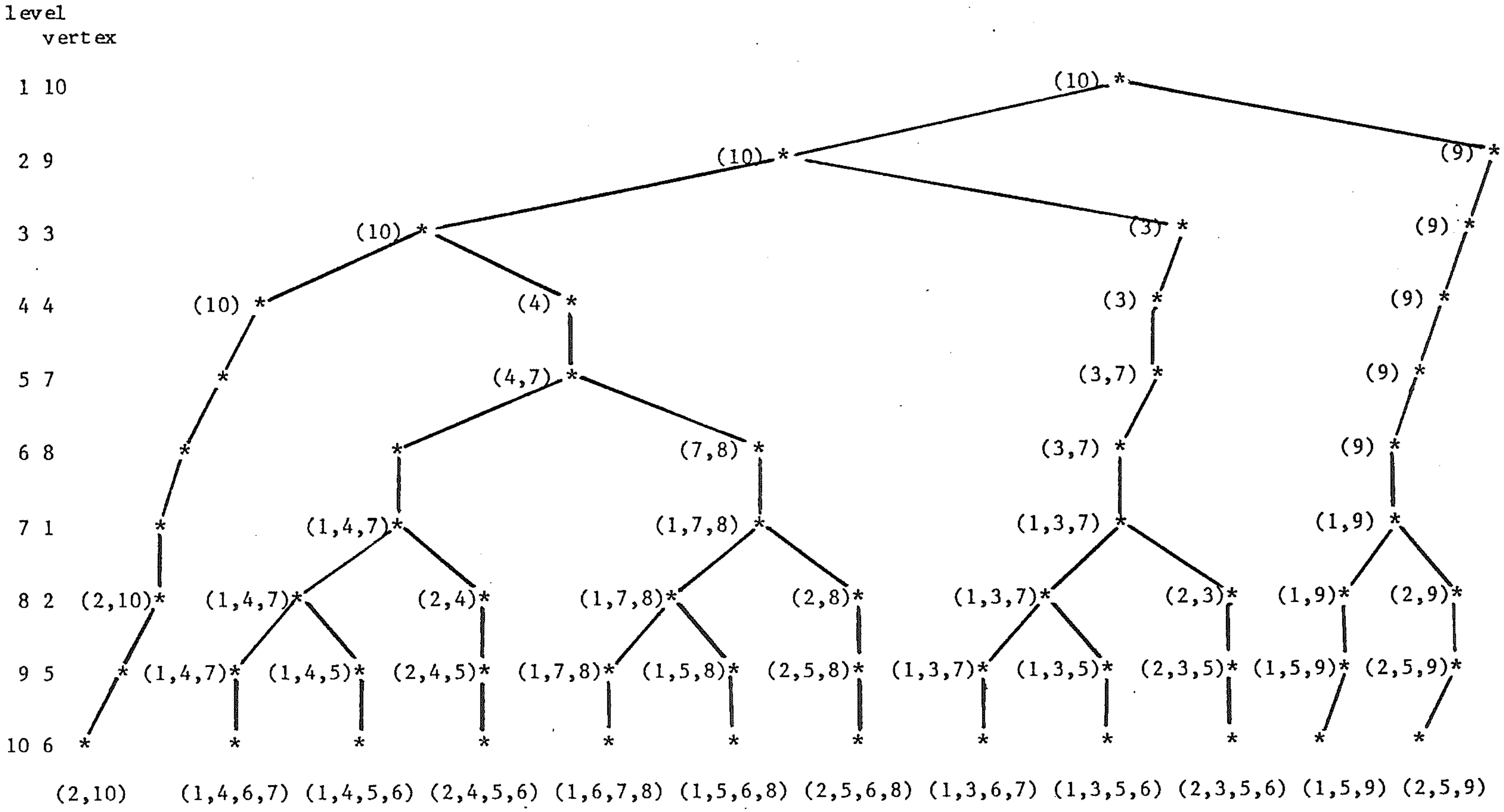
degree will require more procesing time during the procedure whether they occur as type 1 or as type 2 vertices, so the less frequently they are invoked by a procedure call the less time they will add to the overall processing time of the algorithm.

The analysis above suggests that reordering the vertices of the graph by descending degree will provide a practical technique for improving the average case running time of the algorithm. Initial testing confirmed that ordering the vertices by decreasing degree gave, on average, a better response than ordering the vertices by increasing degree.

Example 6.4

Figure 6.3 shows the recursion tree generated by the modified algorithm with the vertices of the graph of Figure 2.1 reordered by decreasing degree. The total number of vertices in the tree has increased to 57 from the 55 vertices in Figure 6.1, but the number of type 2 vertices has decreased to 24, a reduction of 9 vertices and the number of type 1 vertices has increased by 11 to 21. Extensive testing of the modification to the algorithm is reported in section 6.4. The graph of Figure 2.1 was one of those included in the test data. When the original algorithm was used it required 51 milliseconds to generate the maximal independent sets. When the modified algorithm was used it required 43 millisecond.

(end of example)



GENERATING MAXIMAL INDEPENDENT SETS USING THE MODIFIED ALGORITHM

FIG 6.3

Algorithm 6.2 shows the modified algorithm of Tskuiyama et al. A simple $O(n^2)$ process has been used to reorder the vertices of the graph as this has less overhead for the graphs of smaller sizes than more complex methods which have a lower bound on their running time.

Algorithm 6.2

Modified Algorithm of Tsukiyama, Ide, Ariyoshi and Shirakawa for Finding all Maximal Independent Sets in a graph.

Input: A graph G = (V,E) where n = |V|, represented by the adjacency lists Adj(x) for all vertices x of G.

Output: The Maximal Independent Sets of G.

Initialisation:

```
sort the vertices of G into descending order
x := 1
for j := n downto Ø do
  for i := 1 to n do
    if last_adj[i] = j then begin
      label[x] := i
      position[i] : = x
      end
for j := 1 to n do begin
  IS(j) :=Ø
  Bucket(j) := 0  end
BACKTRACK(1)
```

Method:

```
procedure BACKTRACK(i)
begin
  if i < n then begin
    x := label[i + 1]
    c := ∅
    for y in Adj(x) such that y <= i do
      if IS(y) = ∅ then c := c + 1
    if c = ∅ then begin
      for y in Adj(x) such that position[y] <= i do
                              IS(y) := IS(y) + 1
      BACKTRACK(x);
      for y in Adj(x) such that position[y] <= i do
                              IS(y) := IS(y) - 1
      end
    else begin
      IS(x) := c
      BACKTRACK(x)
      IS(x) := ∅
      f := true
      for y in Adj(x) such that 1 <= position[y] <= i
                                        in increasing order do
        begin
        if IS(y) = ∅ then begin
          put y in Bucket(x)
          for z in Adj(y) such that position[z]<=i do begin
            IS(z) := IS(z) - 1
            if IS(z) = ∅ then f := false
            end
          end
          IS(y) := IS(y) +1
          end
        if f then BACKTRACK(x)
        for y in Adj(x) such that position[y] <= i do
                                    IS(y) := IS(y) - 1
        for y in Bucket(x) do begin
          for z in Adj(y) such that position[z] <= i do
                                    IS(z):=IS(z) + 1
          delete y from Bucket (x)
          end
        end
      end
    else  output new maximal independent set designated
          by IS
    end

(end of method)
```

6.4 RESULTS

The algorithms 6.1 and 6.2 were tested on two distinct sets of graphs that form the worst case and average case test data for the algorithms. While the modification is designed to improve the average running time, its effect on the worst case is also of interest.

The graphs which form the worst case test data for any maximal independent set or clique determination algorithm are well defined. In Chapter 2 the graphs defined by Moon and Moser [1965] which have the maximum number of cliques were discussed. The complement of these graphs have the maximum number of maximal independent sets, namely $O(3^{n/3})$. To test the worst case behaviour of the algorithms, the Moon-Moser graphs whose number of vertices is a multiple of 3, ranging from 3 to 30, have been used. The worst case graphs are regular graphs, that is, each vertex in the graph has the same degree, so the modification suggested in section 6.3 will not alter the ordering of the vertices and the preprocessing will slightly increase the overall running time of the modified algorithm. The Moon-Moser graphs used to test the worst case behaviour are described in Table C.1.

The Moon-Moser graphs have n increasing linearly while the number of maximal independent sets increases exponentially with a corresponding exponential increase in the time required to process the graphs. Table C.2 gives the results from testing the worst case data. The timing results are from two test runs for each of the algorithms and show some

variation between runs. The vertices of the Moon-Moser graphs have degree of exactly 2. Therefore, the modification makes no alteration to the order of processing the vertices for these graphs and consequently no improvement is achieved. In fact, the modification increases the time required by the small overhead of preprocessing, but this increase is not greater than the variation that will occur during different runs on a multi-tasking computer. The results for the two different test runs in Table C.2 show that the faster running time achieved by the modified algorithm is less than the slower running time of the original algorithm.

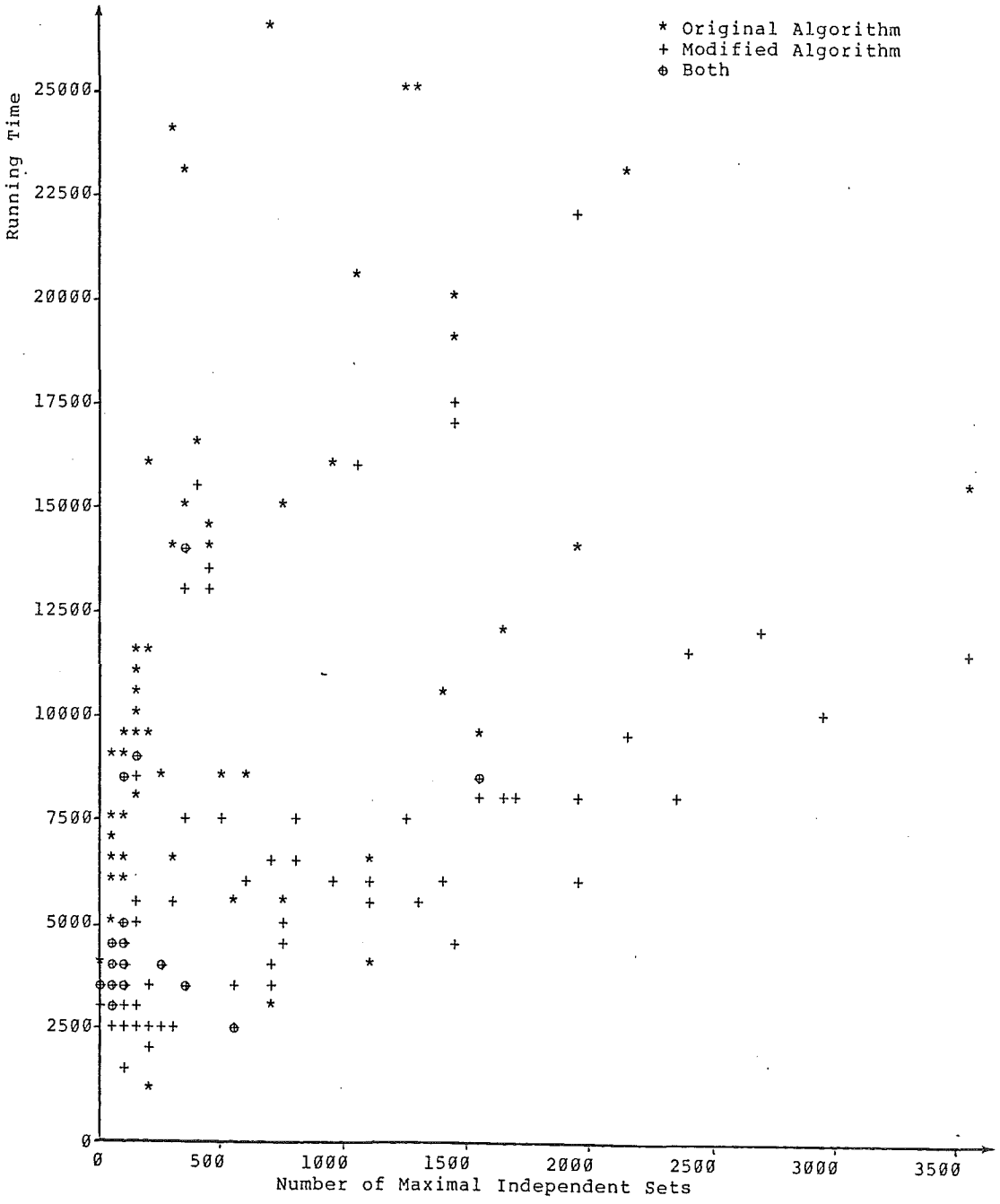
For each of the algorithms, the time required relative to the number of maximal independent sets in the graph has been calculated and is shown in Table C.3. All graphs have a nearly constant time requirement relative to the number of maximal independent sets and as n increases in size this is becoming stable and similar for both the original algorithm and the modified algorithm. The average time per maximal independent set is from 1.40 to 1.51 milliseconds for the original algorithm and from 1.38 to 1.50 milliseconds for the modified algorithm.

The pseudo-random process for the generation of the average case graphs was based on the random graph generator of Kuhn [1972]. A sample of graphs of size 10, 20, 30, 40 and 50 were generated for testing the algorithms. Including complements, 100 graphs were generated for the graphs having 10, 20, 30 and 40 nodes. For graphs of 50 nodes, 10 graphs

were generated. The complement graphs were included in the test sample as other algorithms solve the problem by finding the cliques of a graph and these are later compared with the algorithms for finding the maximal independent sets. The distribution of the graphs generated is summarized in Table C.4. The inclusion of the complements of all graphs in the test data leads to the value of Ø.5 always occurring as the mean value of the density of edges in the graphs.

Tsukiyama et al showed that the running time of their algorithm was dependent on the number of edges as well as on the number of vertices and number of cliques in the graph. Hence, the test graphs have been generated for varying number of edges within the range Ø <= m <= n*(n-1)/2. For any value m in this range, the number of graphs with m edges is $\binom{n*(n-1)/2}{m}$ so, the number of possible graphs increases sharply as m tends to (1/2)*(n*(n-1)/2) from both Ø and n*(n-1)/2. The test graphs were generated to give a distribution over the range of the number of edges, and do not follow the distribution of all possible graphs.

The maximum number of maximal independent sets occurs in the Moon-Moser graphs with m = n, whereas the expected number of maximal independent sets is determined over all possible graphs. Therefore, as the test graphs have a higher frequency of graphs with m = O(n), there will be a higher number of maximal independent sets in the test data than expected for all possible graphs. The number of maximal independent sets in the generated test graphs and the maximum and expected number of maximal independent sets in



RUNNING TIME FOR ORIGINAL AND MODIFIED ALGORITHMS VERSUS NUMBER OF MAXIMAL INDEPENDENT SETS FIG 6.4

all possible graphs is listed in Table C.5.

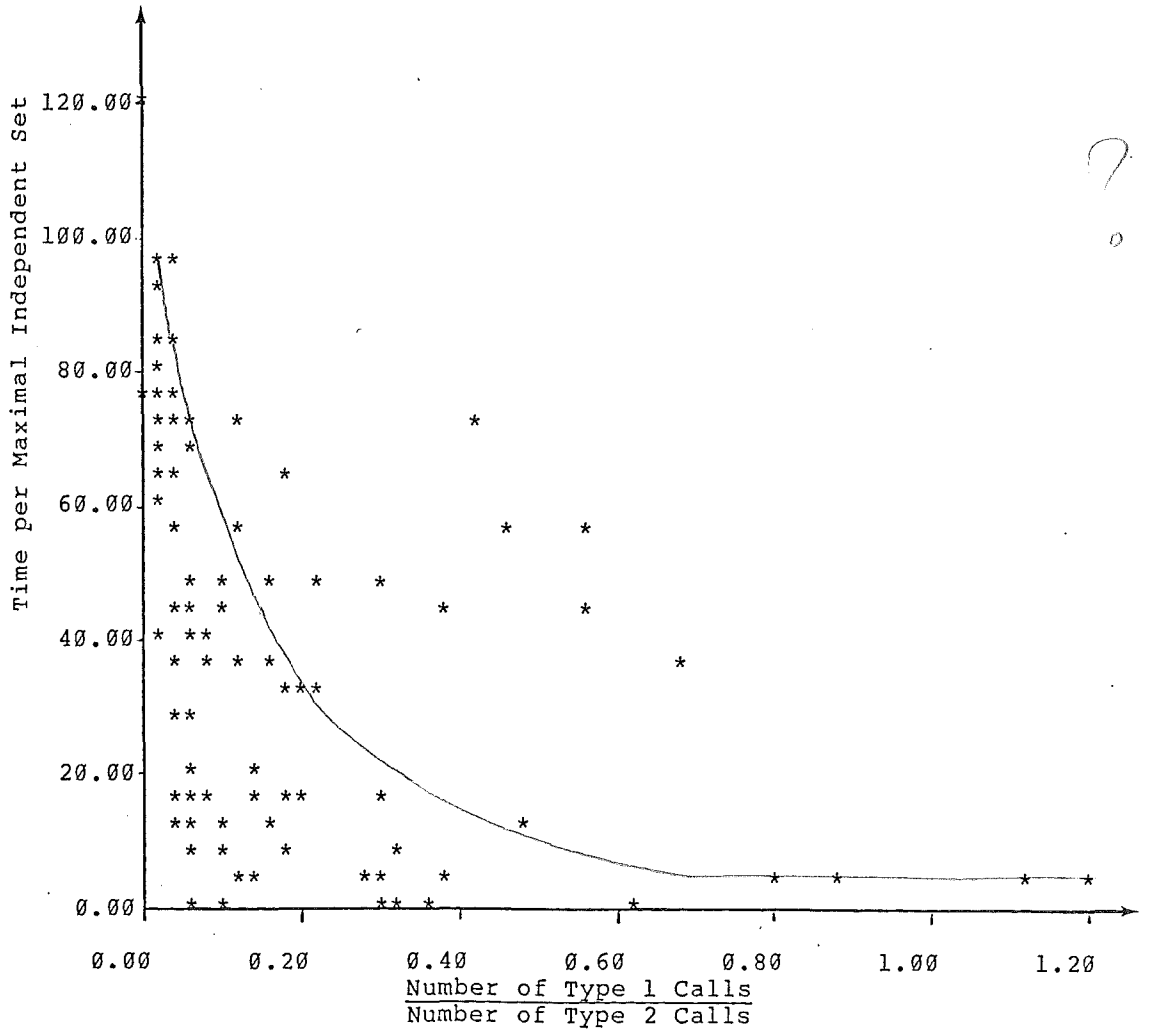
The worst case test graphs are regular graphs so they are not affected by the modification, but in general graphs are not regular and ordering the vertices of a graph into descending order will change the process of searching for the maximal independent sets. Table C.6 summarises the running time required for finding the maximal independent sets in the average case test data using the two algorithms and Figure 6.4 shows the running time of both algorithms for the test graphs with 40 vertices. Where several graphs gave running times to within the accuracy shown in Figure 6.4, only one plot occurs. The running time for both algorithms is O(n*m*c) and the scattered effect in Figure 6.4 reflects the dependence of the algorithms on the number of edges (m) in the graphs when n = 40 is fixed and the number of maximal independent sets (c) is shown on the x-axis. The modification has a varying effect depending on the ordering of the vertices in the original graph. For those graphs already sorted into descending order, the time requirements increase slightly, but the average running time decreases. The improvement obtained by the modified algorithm is summarised in Table C.7. The time required by the modified algorithm relative to the original algorithm shows a percentage decrease as n increases from 10 to 40 vertices. The smaller sample of test graphs generated on 50 vertices gives slightly less improvement.

In section 6.3 the calls of type 2 to procedure BACKTRACK were identified as requiring more processing time than other

calls and the modification is designed to reduce the number of type 2 calls. Table C.9 shows the number of procedure calls for both the original and modified algorithms for the average case test data. This summary shows that only for the graphs of size 50 have the maximum and average number of total procedure calls decreased for the modified algorithm although the minimum number of total calls has increased. For the graphs of other sizes the total number of calls has not been reduced. However, the number of type 2 calls has been reduced by the modification and the mean difference is increasing as a percentage of the number of type 2 calls in the original algorithm as the number of vertices in the graphs increases. The effect of the modification on the number of procedure calls and especially the decrease in the number of type 2 procedure calls will appear in the running time of the two algorithms. A summary of running time improvement achieved by the modification is given in Table C.7. For the smaller test graphs the mean running time of the modified algorithm is greater than the mean running time of the original algorithm, but as the number of vertices in the graph increases, the modified algorithm requires, on average, less time than the original algorithm.

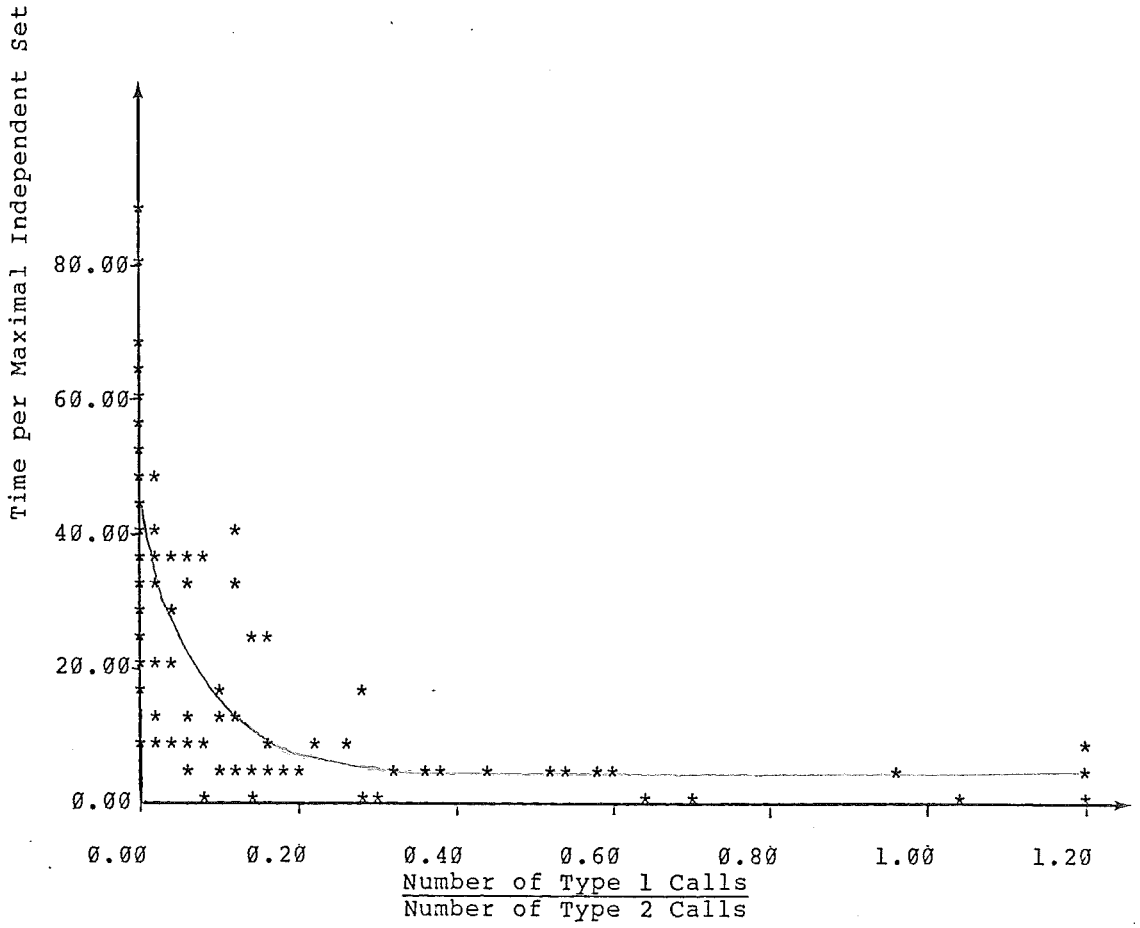
The running time of the algorithm is dependent on the number of maximal independent sets as discussed in section 6.2. The time required per maximal independent set has been calculated for the average case test data and the results are summarised in Table C.8. The aim of the modification was to reduce the number of type 2 procedure calls although the number of type 1 procedure calls may increase, but overall

to make the algorithm more efficient. As the number of type 1 calls increases relative to the number of type 2 calls, the time required to find each maximal independent set should decrease, and if the number of type 2 calls made by the algorithm is much greater than the number of type 1 calls, then the time required to find each maximal independent set will be correspondingly greater. The relationship in both the original and modified algorithms of



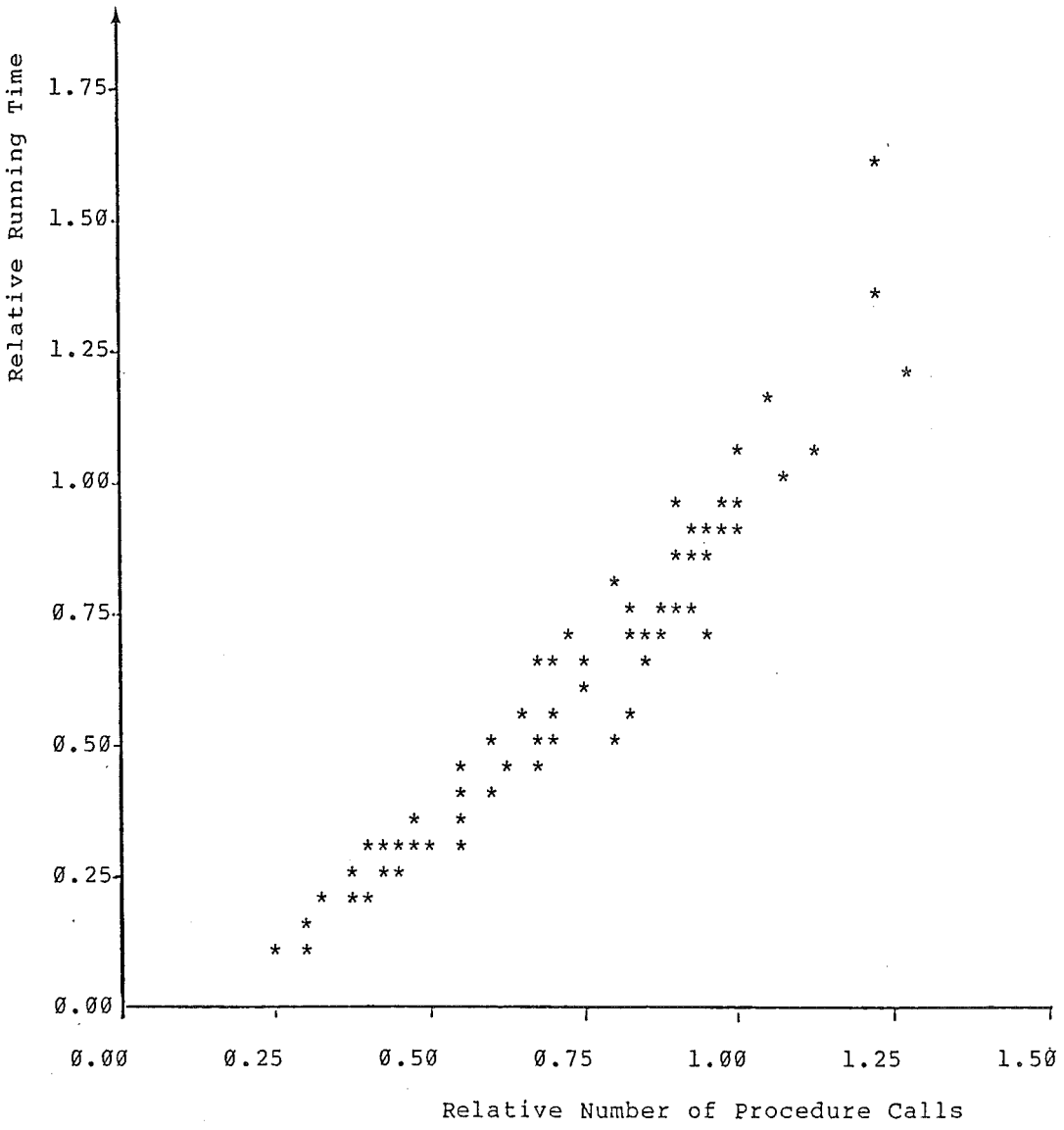
TIME PER MAXIMAL INDEPENDENT SET VERSUS RATIO OF PROCEDURE CALLS FOR THE ORIGINAL ALGORITHM **FIG 6.5**

the time per maximal independent set has been plotted against the ratio of type 1 to type 2 procedure calls for graphs having 40 vertices. Figure 6.5 shows the results for the original algorithm and for several graphs the original algorithm requires much greater processing time for the ratio of calls than for most graphs which require less time than the relationship represented by the curve. In Figure 6.6 the results for the modified algorithm show a very strong relationship between the time per maximal independent set and the number of type 1 calls relative to the number of type 2 calls.



TIME PER MAXIMAL INDEPENDENT VERSUS RATIO OF PROCEDURE CALLS FOR THE MODIFIED ALGORITHM **FIG 6.6**

The change in running time between the original and modified algorithm is dependent on the change in the number of procedure calls made by the algorithms, but is more strongly dependent on the relationship between the number of type 1



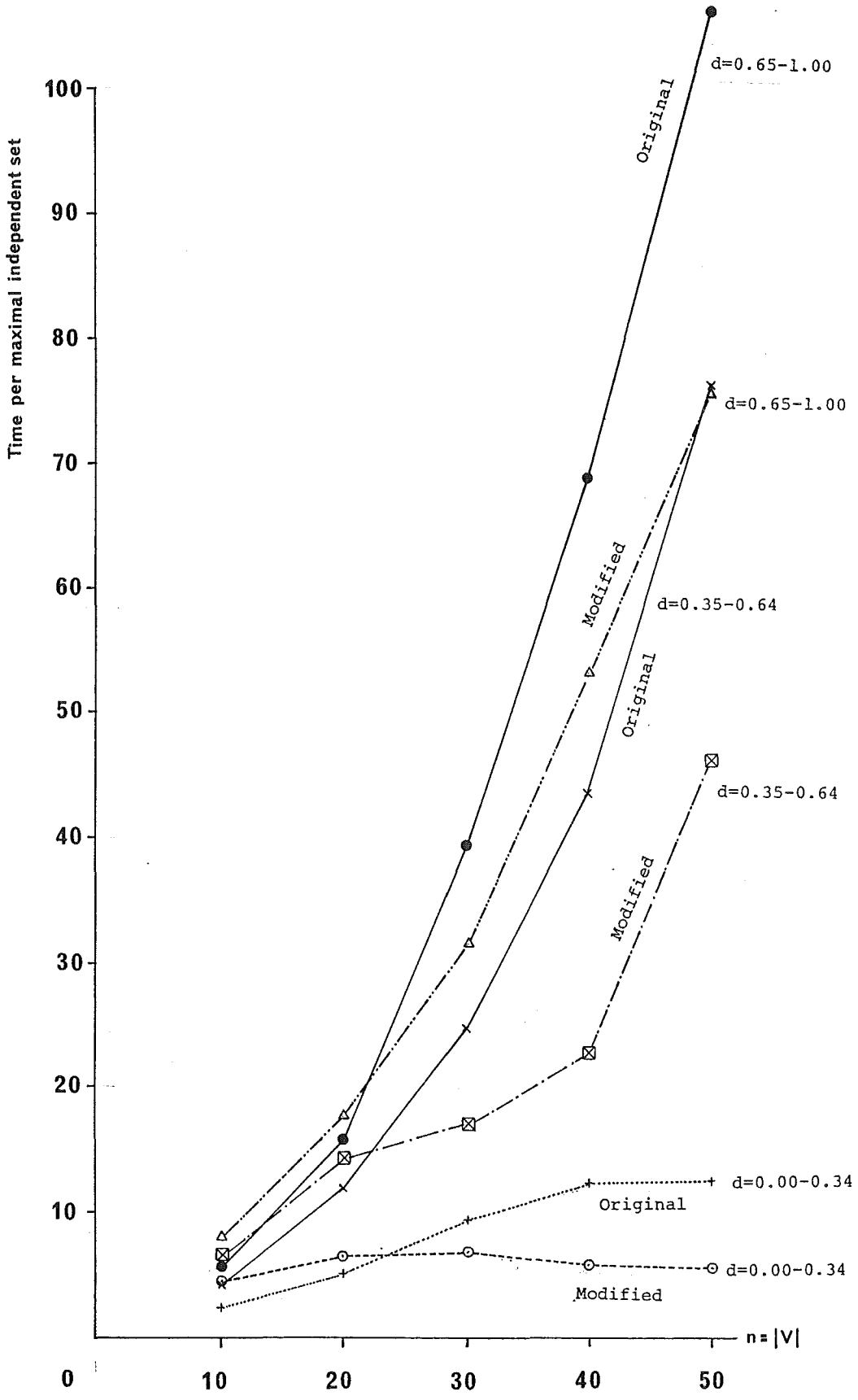
CHANGE IN RUNNING TIME VERSUS CHANGE IN THE RELATIVE NUMBER OF TYPE 1 AND TYPE 2 CALLS **FIG 6.7**

and type 2 calls. The analysis in section 6.3 showed the running time requirements for type 1 and type 2 procedure calls to be $O(d(v))$ and $O(d(v)^2)$ respectively and in Figure 6.7 the relative running time of the algorithm has been plotted against the relative requirements of the number of procedure calls. The relative running time has been calculated as the running time for the modified algorithm divided by the running time for the original algorithm. The average degree of the vertices has been calculated as 2*m/4Ø, so the requirements based on the number of procedure calls have been calculated as

$$\frac{\#(\text{type 1 calls}) + (\#(\text{type 2 calls}) * d(v)) \text{ modified}}{\#(\text{type 1 calls}) + (\#(\text{type 2 calls}) * d(v)) \text{ original}}.$$

This relationship is much stronger than that between the running time and the total number of procedure calls. Therefore, using the relationship between the running time and the number of type 1 and type 2 procedure calls as the basis for the modification to the original algorithm was correct, but as not all points in Figure 6.7 are in the range from Ø to 1, reordering the vertices of the graph by decreasing degree does not always reduce the number of procedure calls and so does not always reduce the running time of the algorithm.

Tsukiyama et al showed the dependence of their algorithm on the number of vertices and the number of edges in the graph. Their Figure 5 showed the time per maximal independent set for increasing number of vertices for three different sets of data; the number of edges occuring in the graph having probability values of Ø.25, Ø.5 and Ø.75. The test data

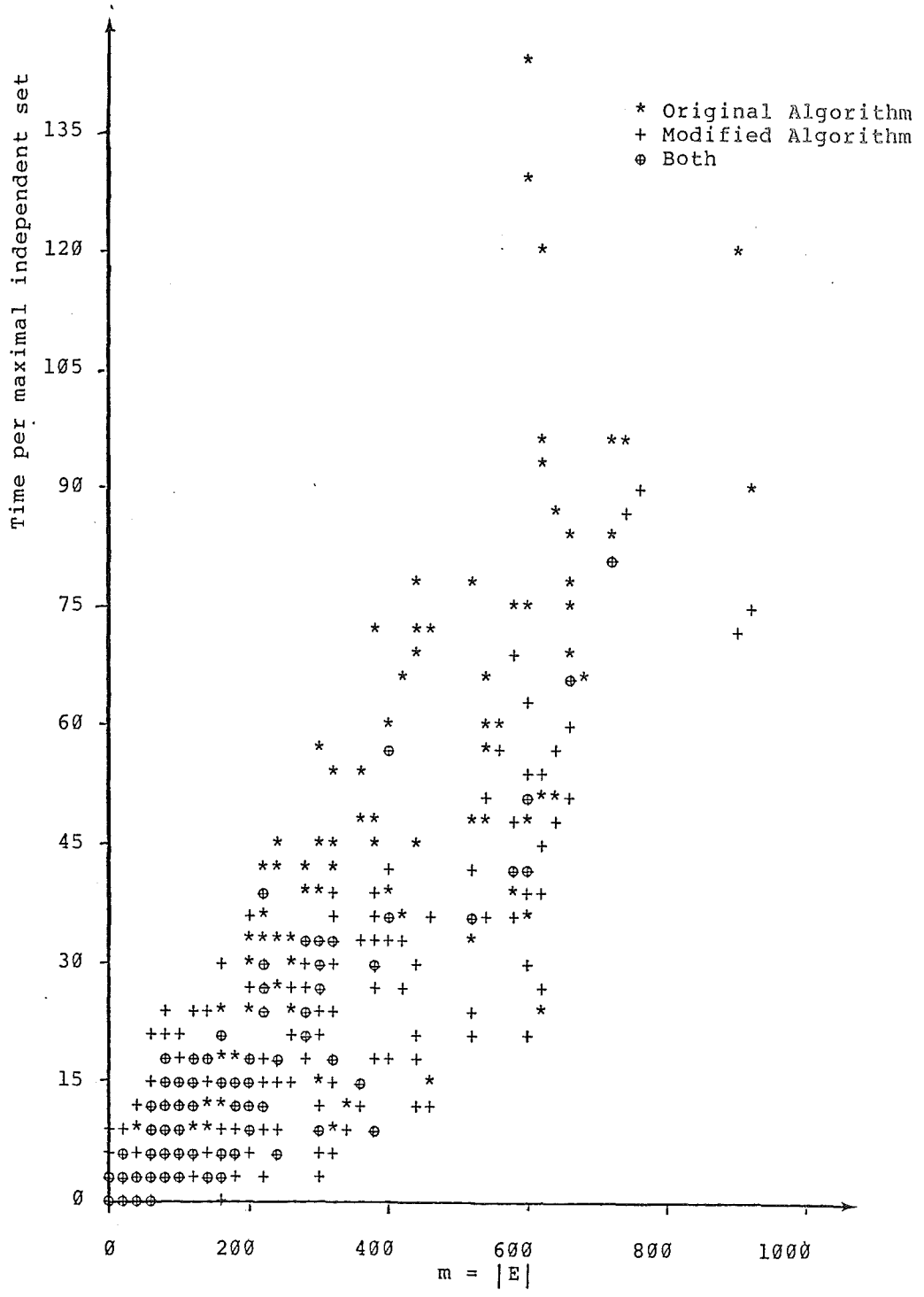


TIME PER MAXIMAL INDEPENDENT SET VERSUS NUMBER OF VERTICES FOR ORIGINAL AND MODIFIED ALGORITHMS

FIG. 6.8

generated to test the modification of the algorithm was over a much wider range of the density of the edges, but these results have been separated into three frequency ranges, Ø to Ø.349, Ø.35 to Ø.649, and Ø.65 to 1.Ø and the results over all test data for increasing n are shown in Figure 6.8. These results are similar to the results obtained by Tsukiyama et al. The improvement obtained by the modified algorithm reduces the running time for each of the probability ranges shown. In fact for graphs with few edges, the time per maximal independent set decreases with n for the modified algorithm.

Tsukiyama et al show a much closer correlation of the running time of the algorithm with the number of edges in the graph than the number of vertices. Their Figure 6 showed the same data grouping as for Figure 5, plotted against the number of edges in the graph. In Figure 6.9 the time per maximal independent set for all graphs tested has been plotted against the number of edges in the graph. There is a wide range of values in this relationship but in general the time per maximal independent set for the modified algorithm is less than for the original algorithm, the greatest value of the time per maximal independent set has been reduced and the rate of increase is also less for the modified algorithm.



TIME PER MAXIMAL INDEPENDENT SET VERSUS NUMBER OF EDGES

FIG 6.9

6.5 COMPARISON WITH OTHER ALGORITHMS

Early algorithms for finding all maximal independent sets or all cliques of a graph required exponential space and time in the worst case. A number of algorithms have since been given which require polynomial space, thus extending the solution to comparatively large graphs. The first polynomial space algorithm, attributed to Bierstone [1970], was presented and corrected by Augustson and Minker [1970], and further corrected by Mulligan and Corneil [1972].

Following Bierstone's algorithm a number of others using depth first search were published. Akkoyunlu [1973] found all the maximal cliques of a graph in time O(c**2) in the worst case (where c is the number of cliques in a graph), but the method has the disadvantage of manipulating symbolic expressions. Das [1973], and Osteen and Tou [1973] also published depth first search algorithms, but these store cliques as they progress, thus requiring exponential space.

A backtracking method was used by Ambler, Barrow, Brown, Burstall and Popplestone [1973] and for the two algorithms presented by Bron and Kerbosch [1973]. The more efficient second algorithm of Bron and Kerbosch has since proved most popular and practical. These backtracking algorithms, use a branch and bound technique to cut off branches that cannot lead to a clique. Johnson [1976] presented a similar algorithm but gave no evaluation of a bound on its running time.

The worst case performance of the Bron-Kerbosch algorithms is unknown, but Tarjan (reported in Reingold, Neivergelt and Deo [1977]) has shown that the simpler algorithm is not polynomial in the number of cliques. The formulation of the Bron-Kerbosch algorithm by Watanabe presented in Reingold, Neivergelt and Deo is claimed to be competitive with known polynomialtime algorithms. Mulligan [1972] showed that the Bron-Kerbosch algorithm is superior to the Bierstone algorithm and Johnston [1976] established that the algorithm of Ambler et al and its later refinements were fundamentally similiar to the Bron-Kerbosch algorithm, but were no more efficient than the simplified Bron-Kerbosch algorithm.

Johnston [1976] extends the Bron-Kerbosch algorithm to a family of algorithms that improve the average running time for varying distributions of edge probability. The results show the relative efficiency of the algorithms and, although this is analysed, no overall measure of the running time is given. Gerhards and Lindenberg [1979] make use of special tree search algorithms to improve the Bron-Kerbosch algorithm for classes of graphs having a highly symmetric structure or sparse adjacency matrices, but their adaptation is less efficient for other classes of graphs.

Sen Gupta and Palit [1979] present an algorithm based on earlier methods using boolean functions which they refined by eliminating redundancy. They claim that their algorithm has a time bound that is linear in the number of cliques in a graph, but they gave no results or analysis to prove this claim.

Loukakis and Tsouris [1981] present a depth first search algorithm which they claim will run considerably faster than the algorithms of Bron-Kerbosch and Tsukiyama et al. No analysis of the algorithm is given. Instead they present a comprehensive set of results comparing the three algorithms for a variety of graphs.

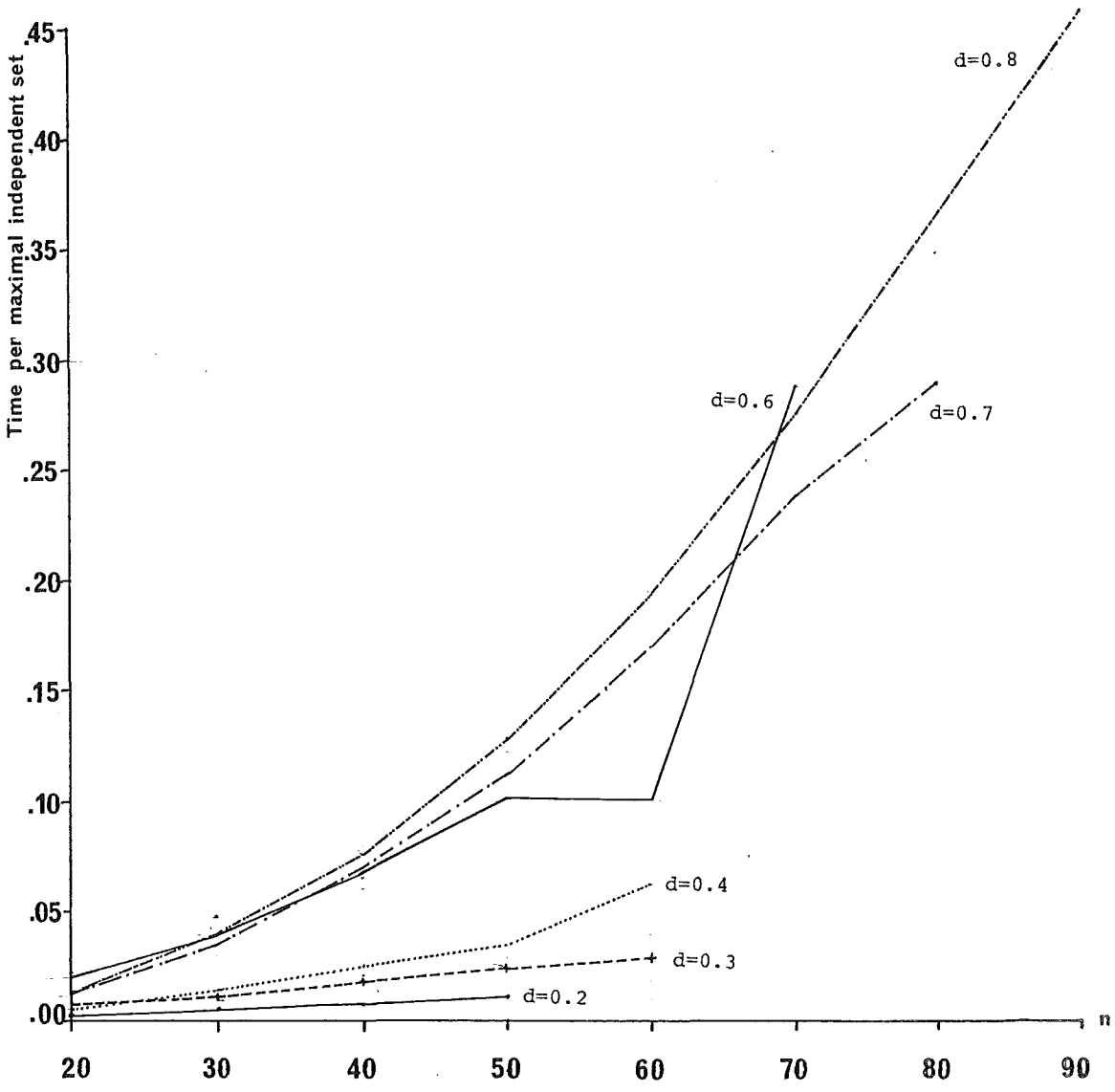
To establish how the original and modified algorithms of Tsukiyama et al compare with other efficient algorithms, two others have been implemented and tested on the same set of test data. Reingold, Nievengelt and Deo presented the Bron-Kerbosch algorithm as the most practical and efficient algorithm for solving the clique problem. The algorithms of Johnston, and Gerhards and Lindenberg are variations of the Bron-Kerbosch algorithm and give improved running time for certain classes of graphs only. Therefore, the Bron-Kerbosch algorithm is the most general in this class and has been implemented. The most recently published algorithm, the Loukakis-Tsouris algorithm for finding the maximal independent sets of a graph was also implemented. A listing of the program used by Loukakis and Tsouris was printed with the paper and, with modifications for different bit manipulation methods has been implemented.

The Bron-Kerbosch algorithm was implemented in Pascal and the results may be compared directly with the results for the Tsukiyama et al algorithm and its modification. The listing of the Loukakais-Tsouris algorithm was in Fortran and has been directly implemented. Because different

languages were used to implement the algorithms, the analysis of the results will compare the change in running time as the size of the graphs increase rather than a direct comparison of the results. This type of analysis was not attempted by Loukakis and Tsouris, but as they have listed the results they obtained, a similar analysis can be undertaken and compared with the results reported in Appendix C.

The running time requirements for these algorithms when finding the maximal independent sets or cliques in the worst case graphs is given in Table C.1Ø. and the time per maximal independent set or clique in Table C.11. For all the algorithms tested, the time per maximal independent set becomes linear as the size of the graphs increase. In comparison, the results presented by Loukakais and Tsouris for testing the worst case running times of these algorithms show the time per maximal independent set for the Bron-Kerbosch algorithm increases as n increases although it is linear for the other two algorithms.

For the results obtained from testing the average case graphs are summarised in Table C.12, Table C.13 summarises the running time per maximal independent set. No direct comparision of the running time per maximal independent set is given by Loukakis and Tsouris, but this information can be obtained from their published results. Figure 6.8 in section 6.4 of this chapter showed the time per maximal independent set plotted against the size of the graph for the original and modified algorithms of Tsukiyama et al.

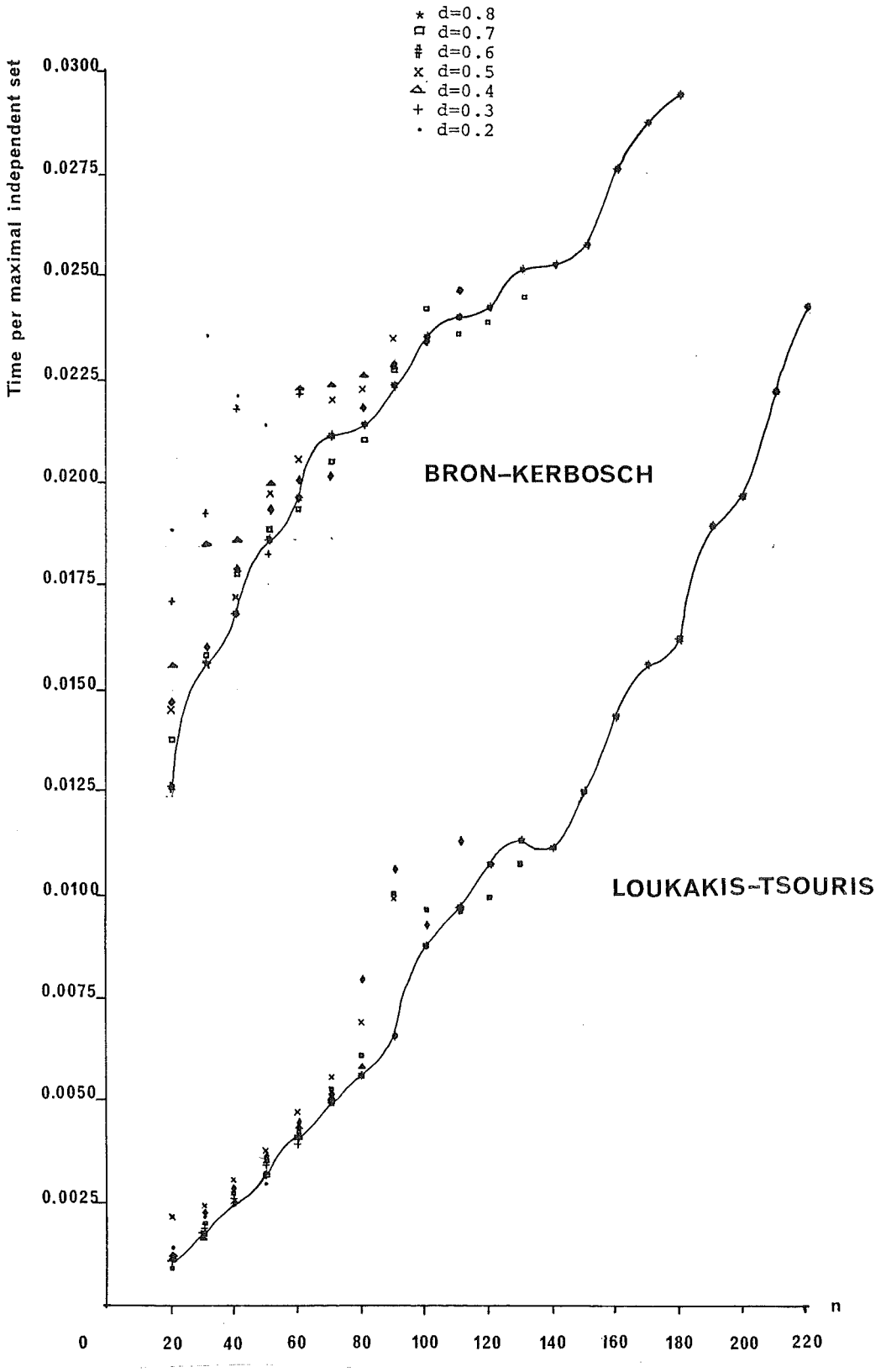


TIME PER MAXIMAL INDEPENDENT SET VERSUS NUMBER OF VERTICES FOR TSUKIYAMA ET AL ALGORITHM REPORTED BY LOUKAKIS AND TSOURIS

FIG. 6.10

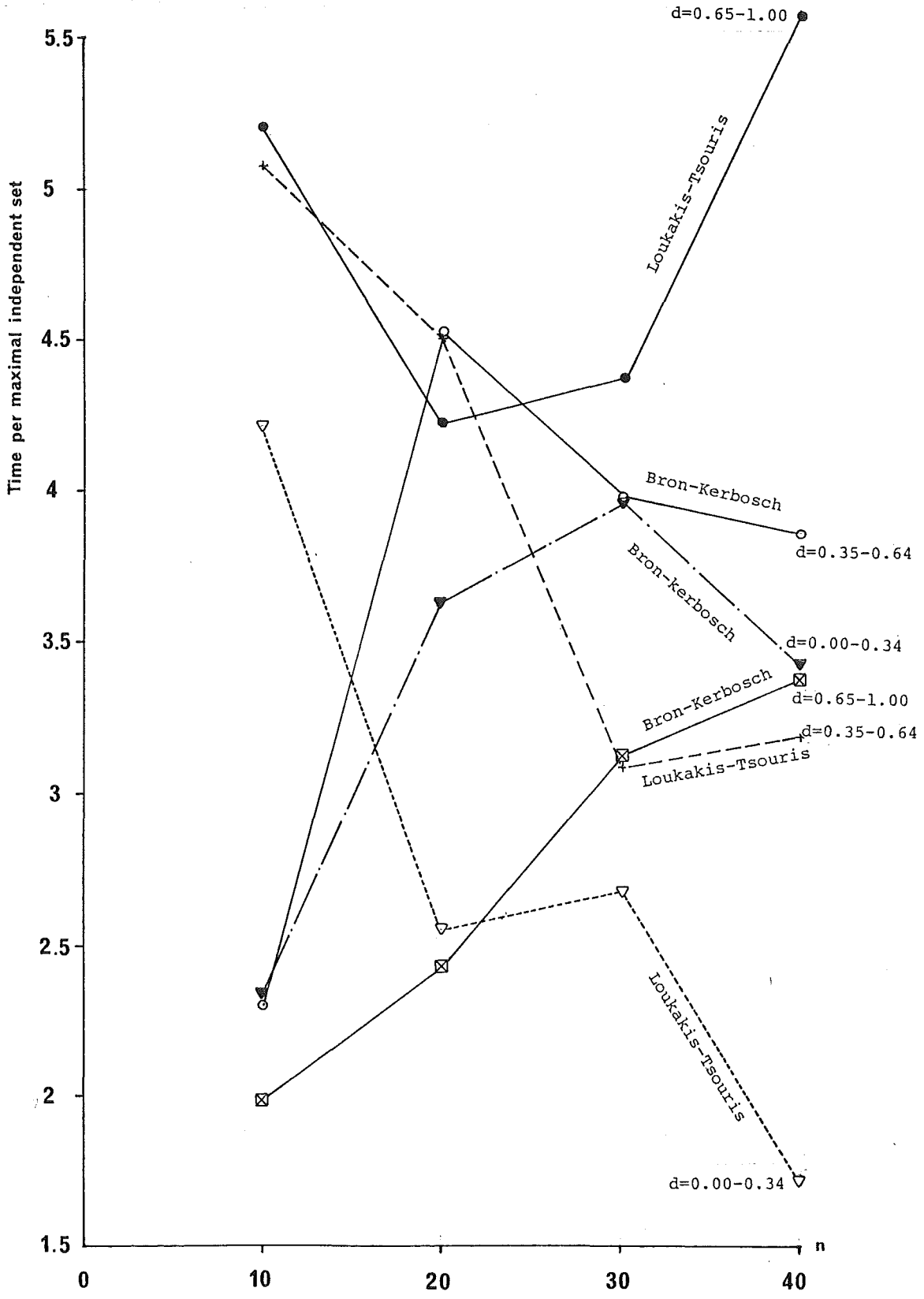
Figure 6.10 shows the same information from the results reported by Loukakais ans Tsouris for the algorithm of Tsukiyama et al. The breakdown of the density of edges in the graphs is finer, but the results show the same trends as the results shown in Figure 6.8.

Similarly, for the results of the Bron-Kerbosch algorithm and the Loukakis-Tsouris algorithm published by Loukakis and Tsouris [1981], the running time per maximal independent set (or clique) is shown plotted against increasing size of the graphs in Figure 6.11. For each algorithm, as the size of the graph increases, there is a strong trend emerging of increasing time per maximal independent set as n increases, but for the Bron-Kerbosch algorithm the rate of increase is less than for the Loukakis-Tsouris algorithm. Both algorithms show a much slower rate of increase than the Tsukiyama et al algorithm. Figure 6.12 shows the same information from the results reported in Table C.13. As in the Loukakis and Tsouris results, the Bron-Kerbosch algorithm shows the rate of increase in the time per maximal independent set decreases as n increases although for dense graphs this is smaller. The Loukakis-Tsouris algorithm shows a decrease in the running time per maximal independent set for sparse graphs, but for dense graphs the time per maximal independent set is increasing. For sparse graphs these results are better than those obtained by Loukakis and Tsouris, but the range of graphs sizes is not as wide.



TIME PER MAXIMAL INDEPENDENT SET VERSUS NUMBER OF VERTICES FOR BRON-KERBOSCH AND LOUKAKIS-TSOURIS ALGORITHMS REPORTED BY LOUKAKIS AND TSOURIS

FIG. 6.11



TIME PER MAXIMAL INDEPENDENT SET VERSUS NUMBER OF VERTICES FOR BRON-KERBOSCH AND LOUKAKIS-TSOURIS ALGORITHMS

FIG. 6.12

Loukakis and Tsouris claim that their algorithm is

> "(a) two to fifteen times faster than the Bron and Kerbosch algorithm and (b) at least three times faster than the algorithm of Tsukiyama et al and becomes increasingly more efficient as both the density and size of the graph increase."

This claim is based on the direct comparison of the running time results. The analysis of the running time as n increases shows a much closer relationship to the claims of Reingold, Nievengelt and Deo that:

> "extensive testing has indicated that it (the Bron-Kerbosch algorithm) is competitive with a clever implementation of the algorithm in . . . (Tsukiyama, Ide, Ariyoshi and Shirakawa)"

In both the results of Loukakis and Tsouris and the results reported in Appendix C, the algorithm of Bron-Kerbosch becomes more efficient than the Tsukiyama et al algorithm as the size of the graph increases, and for the Pascal implementations used in this thesis, a direct comparison shows the Bron-Kerbosch algorithm to be more efficient. Loukakis and Tsouris's results show the running time for their algorithm is increasing at a greater rate than the Bron-Kerbosch algorithm as n increases. For dense graphs, the same trend is observed in the results reported in Appendix C and graphed in Figure 6.12. For sparse graphs, the Loukakis-Tsouris algorithm is more efficient than the Bron-Kerbosch algorithm for the range of graph sizes tested. Both algorithms show the Time per Maximal Independent Set is decreasing for sparse graphs as n increases.

The difference between the results reported in Appendix C and the results presented by Loukakis and Tsouris would appear to be caused by implementation factors. Loukakis and Tsouris report that they implemented all the algorithms in Fortran V on a Univac 1106 and so, the programs would be iterative versions of the recursive algorithms. The relative running times of the algorithms when compared directly, give the differences observed by Loukakis and Tsouris, but when the changes in the running times of the algorithms as the size of the graphs increase, are compared, the results are similar to the results obtained in this thesis.

6.6 SUMMARY

The algorithm of Tsukiyama et al has been investigated and it is made more efficient by reducing the relative number of type 2 procedure calls. The modification to the algorithm attempted to reduce the number of type 2 procedure calls by ordering the vertices of the graphs by decreasing degree. This does not always generate fewer procedure calls, but does give an average running time improvement for the modified algorithm.

Results obtained from a comparison of the algorithm of Tsukiyama et al and the modification with other algorithms show a conflict with the interpretation of other published results. This suggests that implementation factors play a very important role for algorithms to find all the maximal independent sets or cliques of a graph and some initial

testing with various implementations confirmed this. However, when an 'order of magnitide' analysis is performed on the reported results, this conflict disappears. The conclusions reached by Reingold, Neivergelt and Deo stand; namely, an efficient implementation of the Bron-Kerbosch algorithm gives the best running time results. However, this algorithm does not have a proven bound that is linear in the number of cliques in the graph, so it may be subject to extreme behaviour for pathological cases, although testing of the algorithm has not generated such a case.

CHAPTER 7

CONCLUSIONS

The data structures used to represent a graph are as important as the techniques used in the design of algorithms for graph problems. Current data structures can provide efficient graph representations for some graph problems, but the increasing size and complexity of problems to be solved and the subsequent high processing requirements to search or otherwise manipulate a graph indicates a need for more succinct representations. Representing a graph in a compressed form will reduce the size of data structures and increase algorithm efficiency. A compression of a graph to one or more smaller graphs is generated by decomposition techniques, but many of the decompositions developed in graph theory have not been applied to graph algorithms, as many are based on complex graph structures or the representations generated by the decompositions are not appropriate for the operations that have to be performed in solving the graph problem.

Tutte's definition of the nodal three-connected graphs provides a suitable basis for the definition and development of a decomposition of a network that can be applied, in particular, to path algorithms. The decomposition developed in this research has been applied to public transport

simulation models and the results discussed in Chapter 4, show that compression of the networks converts the simulation models into a viable tool for 'every day' use. When implementing algorithms for tranport simulation models, both time and space are important as these transportation networks are generally very large and compression will reduce the processing requirements. For a well defined class of networks the decomposition generates a compressed network smaller than the original network in O(m) time and the shortest path between any pair of vertices in the original network can be found in linear time from the decomposition and the shortest path matrix for the compressed network.

Another field of study closely related to graph algorithms and where compression methods have been applied is the area of boolean matrix multiplication. A new compression technique for boolean matrices and a method for applying the compression to boolean matrix multiplication has been developed. A summation compression applied to the rows or columns of a boolean matrix is formed into a tree structured partition series to give a family of algorithms for boolean matrix multiplication which require $O(n^2*\log_2 n)$ expected running time for half of all possible matrices and $O(n^2)$ expected running time for the remaining half of all possible matrices. Testing the algorithms and comparing them with other algorithms for boolean matrix multiplication shows that the expected average running time of these algorithms is more efficient than other compression methods which require $O(n^3/(p^2*\log_2 n))$ expected running time, however, algorithms which use a compression based on an indexing

technique are shown to be generally more efficient than the other compression methods.

The summation compression method developed during this research has been the basis for the development of a new and simple heuristic that may easily be added to all boolean matrix multiplication algorithms. Algorithms incorporating the 'summation' heuristic have an average improvement in running time and give the best running time performance of all the algorithms tested. The 'summation' heuristic may be applied to algorithms to which the 'out of loop' heuristic cannot be applied and produce an O(n) improvement in running time in the best case. The results obtained suggest that the application of the 'summation' heuristic to other boolean matrix multiplication algorithms will give an improvement to the average running time of the algorithms.

The simplical decomposition of graphs is based on the cliques of graphs, but as the problem of finding all the cliques or maximal independent sets of a graph is known to be NP-complete, an investigation of this decomposition method included the known algorithms for finding all the cliques or maximal independent sets of a graph. An average constant factor running time improvement for the algorithm of Tsukiyama, Ide, Ariyoshi and Shirakawa for finding all maximal independent sets of a graph can be achieved by reordering the vertices of a graph. The algorithm of Tsukiyama et al is the only algorithm for finding all maximal independent sets or cliques of a graph with a proven time bound linear in the number of maximal independent sets

or cliques. However, testing the Bron-Kerbosch and Loukakis-Tsouris algorithms showed that the rate of increase in the running time as the size of the graphs increases was less than the rate of increase for the Tsukiyama et al algorithms, although these algorithms do not have a proven time bound and so may not always be more efficient.

The claim by Loukakis and Tsouris [1981] that their algorithm is faster than either the Bron-Kerbosch or Tsukiyama et al algorithms is based on a direct comparison of running time results. The order of magnitude analysis performed on the results published by Loukakis and Tsouris is presented in Chapter 6 and shows the rate of increase in the running time of the Bron-Kerbosch algorithm is less than the rate of increase for the Loukakis-Tsouris algorithm, although the actual running time for the Bron-Kerbosch algorithm is higher. The claim by Loukakis and Tsouris that their algorithm 'becomes increasingly more efficient as both the density and size of the graph increase' is valid for the Tsukiyama et al algorithm, but not correct for the Bron-Kerbosch algorithm. The results obtained by Loukakis and Tsouris and the results presented in this thesis, show similar order of magnitude growth for all algorithms, although the constant factors show a wide variation, which can be concluded to result from implementation factors. Initial investigation of variations in implementation factors has shown that these algorithms produce sufficient changes in running times for minor changes in implemenations to account for the constant factor variations. A study of the algorithms using the techniques of computer

implementation technology [Glover and Klingman, 1982] may establish which implementation features are most suitable for processing these algorithms.

REFERENCES

Aho A.V., J.E. Hopcroft and J.D. Ullman [1974] The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974

Akkoyunlu E.A. [1973]. The Enumeration of Maximal Cliques of Large Graphs. In: SIAM Journal of Computing 2 (1973) 1-6.

Arlazarov V.L., E.A. Dinic, M.A. Kronrod and I.A. Faradzev [1970] On Economical Construction of the Transitive Closure of a Directed Graph. In: Siviet Math Dokl 11 5 1209-1210

Ambler A.P., H.G. Barrow, C.M. Brown, R.M. Burstall, and R.J. Popplestone [1973]. A versitile computer controlled assembly system. In: Third International Joint Conference on Artificial Intelligence: Advanced Papers (1973) 298-303

Atkin R.H. [1972] From Cohomology in Physics to q-connectivity in Social Sciences. In: Journal of Man-Machine Studies 4 (1972) 139-167

Atkin R.H. and J. Casti [1977] Polyhedral Dynamics and the Geometry of Systems. Research Report RR-77-8, International Institute for Applied Systems Analysis, Austria 1977

Augustson J.G. and J. Minker [1970] An Analysis of Some Graph Theoretical Cluster Techniques. In: Journal of the ACM 17 (1970) 571-588

Bierstone [1970]. Unpublished Report. Reported In: Augustson and Minker, [1970].

Bini O., M. Capovani, F. Romani and G. Lotti [1979] $O(n^{2.7799})$ Complexity for nxn Approximate Matrix Multiplication. In: Information Processing Letters 8 5 (1979) 234-235

Bollobas B. and P. Erdos [1976]. Cliques in Random Graphs. In: Math Proc Camb Phil Soc 80 (1976) 419-427

Bron C. and J. Kerbosch [1973]. Algorithm 457: Finding all Cliques of an Undirected Graph, In: Communications of the ACM., 16 (1973), 575-577

Cardon A. and M. Crochemore [1982] Partitioning a Graph in $O(|A|\log_2|v|)$. In: Theoretical Computer Science 19 (1982) 85-98

Carre B. [1979] Graphs and Networks. Oxford University Press 1979

Cunningham W.H. and J. Edmonds [1980] A Combinatorial Decomposition Theory. In: Canadian Journal of Mathematics 32 3 (1980) 734-765

Cunningham W.H. [1982] Decomposition of Directed Graphs. In: SIAM Journal of Algebra and Discrete Applied Mathematics 13 2 (1982)

Das S.R. [1973]. On a new approach for finding all the modified cut-sets in an incompatibility graph. In: IEEE Trans. Comput. C-22 (1973) 187-193

Dubois D., G. Bel and M. Llibre [1979] A Set of Methods In Transportation Network Synthesis and Analysis. In: Journal of the Operations Research Society 30 9 (1979) 797-808

Ebert J. and Perl J. [1981] Reachability Homomorphisms on Nets. In: Graphtheoretic Concepts in Computer Science. In: Lecture Notes in Computer Science 100, (ed) H Noltemeier (1981) 326-334

Eftimie M. and R. Eftimie [1977] A Decomposition Property of Basic Acyclic Graphs. In: Discrete Mathematics 17 (1977) 271-279

Erdos P. and A. Renyi [1959] On the Evolution of Random Graphs. In: Matematikai Kutato Intezet Kozlemenyei, V.A/1-2, (1959), 17-61

Even S. [1979] Graph Algorithms. Computer Science Press Inc. (1979)

Farley B.A., A.H. Land and J.D. Murchland [1967] The Cascade Algorithm For Finding All Shortest Distances In A Directed Graph. In: Management Science 14 1 (1967) 19-28

Fisher M.J. and A.R. Meyer [1971] Boolean Matrix Multiplication and Transitive Closure. In: Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory (1971) 129-131

Fredman M.L. and R.E. Tarjan [1984] Fibonacci Heaps and Their Uses In Improved Network Optimisation Algorithms. In: Proceedings of the 25th Annual Symposium on the Foundations of Computer Science (1984) 338-346

Frucht R. [197Ø] How to Describe A Graph. In: Annuals of the New York Academy of Sciences 175 1 (197Ø) 156-167

Gabites, Porter and Partners [1982] Christchurch Transport Board Review of Services: Summary. (1982)

Gabow H.N., Z. Galil and T.H. Spencer [1984] Efficient Implementation of Graph Algorithms Using Contractions. In: Proceedings of the 25th Annual Symposium on the Foundations of Computer Science (1984) 347-357

Garey M.R. and D.S. Johnson [1979]. Computers and Intractability: A Guide to the Theory of NP-Completeness. WH Freeman and Co., San Francisco, (1979)

Gerhards L. and W. Lindenberg [1979]. Clique Detection for Nondirected Graphs: Two New Algorithms. In: Computing 21 (1979) 295-332

Glover F., D. Klingman and A. Napier [1973] A Note On Finding All Shortest Paths. In: Trans. Sci. (1973) 3-12

Glover F. and D. Klingman [1982] Recent Developments In Computer Implementation Technology For Network Flow Algorithms. In: INFOR 2Ø 4 (1982) 433-452

Gorenstein S. [1972] An Algorithm For Project (Job) Sequencing With Resource Constraits. In: Operations Research 2Ø 4 (1972)

Habib M. and M.C. Maurer [1979] On the X-join Decomposition For Undirected Graphs. In: Discrete Applied Mathematics 1 (1979) 2Ø1-2Ø7

Halin R. [1978] Simplical Decompositions of Infinite Graphs. In: Advances in Graph Theory, Annals of Discrete Mathematics 3 (ed) B. Bollobas (1978) 93-1Ø9

Halin R. [1982] Simplical Decompositions: Some New Aspects and Applications. In: Annuals of Discrete Mathematics 13 (1982) 1Ø1-11Ø

Hestenes M. [1973] On the Use of Graphs in Group Theory. In: Theory of Graphs. (ed) F. Harary (1973) 97-128

Hopcroft J.E. [1971] An n log n Algorithm For Mininising States in a Finite Automaton. In: Theory of Machines and Computations (ed) Z. Kohavi and A. Paz, Academic Press N.Y. 1971

Hopcroft J.E. and R.E. Tarjan [1973] Dividing a Graph into Triconnected Components. In: SIAM Journal of Computing 2 3 (1973) 135-158

Hopcroft J. and R.E. Tarjan [1974] Efficient Planarity Testing. In: Journal of the ACM 21 4 (1974) 549-568

Hu T.C. [1968] A Decomposition Algorithm for Shortest Paths in a Network. In: Journal of ORSA, 1 16 (1968) 91-1Ø2

Hu T.C. and W.T. Torres A Shortcut in the Decomposition Algorithm for Shortest Paths in a Network. MRC Technical Summary Report #882, Mathematics Research Centre (1968)

James L.D., R.G. Stanton and D.D. Cowan [1972] Graph Decompositton for Undirected Graphs. In: Proceedings of the 3rd South-Eastern Conference on Combinatorics, Graph Theory, and Computing (1972) 281-29Ø

Jarvis J.J. and S. Tufekci [1982] A Decomposition Algorithm for Locating A Shortest Path Between Two Nodes in a Network. In: Networks 12 (1982) 161-172

Johnson L.F. [1976]. Determining Cliques in a Graph. In: Proc Fifth Manitoba Conference on Numerical Mathematics and Computing, Utilitas Math, 429-437

Johnston H.C. [1976]. Cliques of a Graph - Variations on the Bron-Kerbosch Algorithm. In: International Journal of Computer and Information Sciences 5 3 (1976) 2Ø9-238.

Korfhage R.R. [1974] Discrete Computational Structures. Academic Press (1974)

Knuth D.E. [1973] The Art of Computer Programming Volume 1 Fundamental Algorithms. Addison-Wesley Publishing Co. (1973)

Kuhn W.W. [1972]. A Random Graph Generator. In: Proc of the Third Conf. on Combinatorial (1972)

Land A.H. and S.W. Stairs [1967] The Extension of the Cascade Algorithm to Large Graphs. In: Management Science 14 1 (1967) 29-33

Lawler E.L. [1976] Graphical Algorithms and Their Complexity. In: Mathematical Centre Tracts 81 (1976) 3-32, Amsterdam. In: Foundations of Computer Science II Part 1, (ed) K.R. Apt and J.W. de Bakker

Loukakis E. and C. Tsouros [1981]. A Depth First Search Algorithm To Generate the Family of Independent Sets of a Graph Lexiographically. In: Computing 27 (1981) 349-366

Mahr B. [1979] Aspects of Graphrepresentation. In: Graphs, Data Structures and Algorithms, (ed) M. Nagl and H-J Scheider, Applied Computer Science 13, Hanser (1979).

Mahr B. [1981] A Birds-Eye View to Path Problems. In: Graphtheoretic Concepts in Compter Science In: Lecture Notes in Computer Science 100, (ed) H. Notlemeier (1981) 335-353

Matula D.W. [1970]. On the Complete Subgraphs of a Random Graph. In: Proc Second Chapel Hill Conf on Combinatorial Maths and its Applications. (1970) 356-369.

Moon J.W. and L. Moser [1965]. On Cliques in Graphs. In: Israel Journal of Mathematics 3 (1965) 23-28.

Mulligan G.D. and D.G. Corneil [1972]. Corrections to Bierstone's Algorithm for Generating Cliques. In: Journal of the ACM 19 (1972) 244-247

Nievergelt J. [1979] Trees As Data and File Structures: In: Graphs, Data Structures and Algorithms. (ed) M. Nagl and H-J. Scheider, Applied Computer Science 13 Hanser (1979)

O'Neil P.E. and E.J. O'Neil [1973] A Fast Expected Time Algorithm for Boolean Matrix Multiplication and Transitive Closure. In: Information and Control 22 (1973) 132-138

Osteen R.E. and J.T. Tou [1973]. A cliques-detection algorithm based on neighbourhoods in graphs. In: International Journal of Computing and Information Sciences 2 (1973) 257-268

Pan V.Y. [1978] Strassen's Algorithm Is Not Optimal: Trilinear Technique of Aggregating, Uniting and Cancelling for Constructing Fast Algorithms for Matrix Operations. In: Proceedings of the 19th Annual Symposium on the Foundations of Computer Science, IEEE, (1978) 166-176

Pan V.Y. [1979] Field Extension and Trilinear Aggregating, Uniting and Cancelling For The Acceleration of Matrix Multiplication. In: Proceedings of the 20th Annual Symposium on the Foundations of Computer Science, IEEE, (1979)

Papadimitriou C.H. and Yannakais M. [1981] The Clique Problem for Planar Graphs. In: Information Processing Letters 13 4 (1981) 131-133

Paull M.C. and S.H. Unger [1959]. Minimizing the Number of states in incompletely specified sequential switching functions. In: IRE Trans Electronic Computers, EC-8 (1959) 356-367

Perl J. [1980] Reachability-Homomorphisms on Graphs. In: Discrete Structures and Algorithms, (ed) U Pape. Hanser-Verlag, Munchen-Wein 1980 105-113

Pfaltz J.L. [1972] Graph Structures. In: Journal of the ACM 3 19 (1972) 411-422

Read R.C. [1969] Teaching Graph Theory to a Computer. In: Recent Progress in Combinatorics; Proceedings of the 3rd Waterloo Conference on Combinatorics 1968. (ed) W.T. Tutte Academic Press N.Y. (1969) 161-173

Read R.C. and Corneil D.G. [1977] The Graph Isomorphism Disease. In: Journal of Graph Theory 1 4 (1977) 339-363

Reingold E.M., J. Nievergelt and N. Deo [1977]. Combinatorial Algorithms: Theory and Practice. Prentice-Hall Inc., Englewood Cliffs NJ, 1977, 390.

Robinson [1981] Private Communications.

Rotem D. and J. Urrutia [1981] Finding Maximum Cliques in Circle Graphs. In: Networks 11 (1981) 269-278

Sabudissi G. [1961] Graph Derivatives. In: Mathematic Zeitschr 76 (1961) 385-401

Sen Gupta A. and A. Palit [1979]. On Clique Generation Using Boolean Equations. In: Proc. of the IEEE 67 1 (1979) 178-180

Sheir D.R. [1973] A Decomposition Algorithm For Optimality Problems In Tree-Structures Networks. In: Discrete Mathematics 6 (1973) 175-189

Strassen V. [1969] Gaussian Elimination is Not Optimal. In: Numerische Mathematik 13 (1969) 354-356

Takaoka T. [1979]. Private Communications.

Tarjan R.R. [1972] Depth First Search and Linear Graph Algorithms. In: SIAM Journal of Computing 1 2 (1972) 146-160

Tarjan R.E. [1973] Testing Flow Graph Reducibility. In: Proceedings of the 5th Annual ACM Symposium of Computing (1973) 96-107

Tarjan R.E. [1974] Finding Dominators in Directed Graphs. In: SIAM Journal of Computing 1 3 (1974)

Tsukiyama S., M. Ide, H. Ariyoshi, and I. Shirakawa [1977]. A New Algorithm for the Generating All the Maximal Independent Sets. In: SIAM J Comput. 6 (1977) 3 505-517.

Tutte W.T. [1966] Connectivity in Graphs. University of Toronto Press 1966

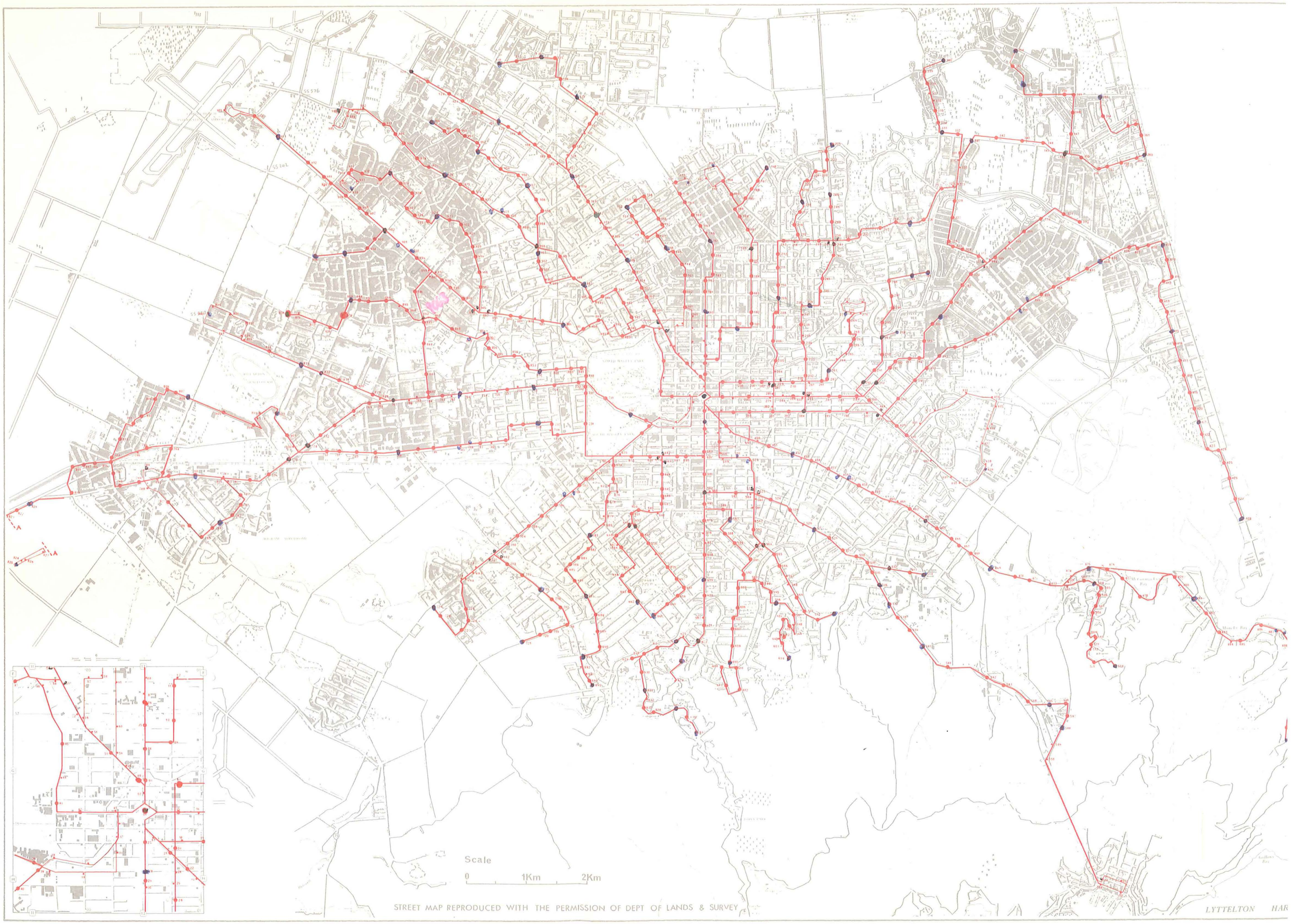
Winograd S. [1973] Some Remarks On Fast Multiplication of Polynomials. In: Complexity of Sequential and Parallel Numerical Algorithms, (ed) J.F. Traub, Academic Press, N.Y. 1973

Yen J.Y. [1971] On Hu's Decomposition Algorithm for Shortest Paths In A Network. In: Operations Research 19 (1971) 91-102

APPENDIX A

SHORTEST PATHS

The decomposition algorithm of Chapter 4 has been applied to the bus network of the Christchurch Transport Board. The full network is shown overleaf. The examples used in Chapter 4 are taken from a section of the network shown at the extreme left of the network (Sockburn-Hornby).



STREET MAP REPRODUCED WITH THE PERMISSION OF DEPT OF LANDS & SURVEY

LYTTELTON HAR

CHRISTCHURCH BUS ROUTES

STUDY BASE
Existing 1979 Network

KEY

- Bus stops on both sides of street
- Bus stops on one side of street only

Scale

0 1Km 2Km

LYTTELTON HARBOUR

Christchurch Transport Board Review of Services.

CHRISTCHURCH TRANSPORT BOARD
GABITES ALINGTON AND EDMONDSON

Figure 5

APPENDIX B

BOOLEAN MATRIX MULTIPLICATION

B.1 RESULTS

Two types of tree discussed in Chapter 5.2. These are (a) a tree with all leaves at the same depth in the tree and (b) a tree with all parent vertices in the tree (with the possible exception of the parent of the last vertex) having the maximum number of children.

The number of vertices that occur in the two tree types are compared in Table B.1. For increasing matrix size n the trees generated by the different summation sizes of binary, log n and square root n are compared. For binary trees the two tree types are identical. Table B.1 shows the summation size or maximum number of children and the number of vertices in the two types of summation trees.

n	Binary Tree (a&b)		Log n Tree (a)		(b)	Square Root n Tree (a)		(b)
4	2	7	2	7	7	2	7	7
8	2	15	3	12	12	3	12	12
16	2	31	4	21	21	4	21	21
32	2	63	5	42	40	6	39	39
64	2	127	6	78	77	8	73	73
128	2	255	7	151	150	12	141	140

COMPARISON OF TREE SIZES **TABLE B.1**

To test and compare the running times for the members of the family of tree summation algorithms, boolean matrices were generated for one hundred tests on each of the matrix sizes 8, 16, 32, and 64. The performance of the algorithms will vary with the density of 1's in the matrices being multiplied, so the test data was generated to span the probability range Ø.Ø1 <= p <= 1, with the same probability for each pair of matrices multiplied. Therefore, the test data spans the probability range, but is not representative of the distribution of graphs in the set of all possible graphs. Table B.2 summarises the test data that was generated.

Size	Number of 1's in matrix	Number of 1's in product	Density of 1's in matrix
n = 8			
Mean	32.Ø5	45.86	Ø.ØØØ8
Minimum	Ø.ØØ	Ø.ØØ	Ø.ØØØØ
Maximum	64.ØØ	64.ØØ	1.ØØØØ
n = 16			
Mean	129.61	2Ø4.83	Ø.5Ø63
Minimum	3.ØØ	3.ØØ	Ø.Ø117
Maximum	256.ØØ	256.ØØ	1.ØØØ
n = 32			
Mean	519.23	873.Ø1	Ø.5Ø71
Minimum	1Ø.ØØ	1Ø.ØØ	Ø.ØØ98
Maximum	1Ø24.ØØ	1Ø24.ØØ	1.ØØØØ
n = 64			
Mean	2Ø67.74	3666.75	Ø.5Ø48
Minimum	43.ØØ	59.ØØ	Ø.Ø1Ø5
Maximum	4Ø96.ØØ	4Ø96.ØØ	1.ØØØØ

PARAMETERS OF THE AVERAGE CASE TEST DATA **TABLE B.2**

The two tree formats discussed in Chapter 5.2 cause differences in the processing results. The log n summation partition series with both tree formats have been tested on the test data for the depth first search and are compared in Table B.3. Matrices of size 32 and 64 are compared.

Size	Number of Procedure Calls Depth First Search		
	Trees with all leaves at one level	Trees with maximum children	Difference
n = 32			
Mean	2498	2238	260
Minimum	170	103	-106
Maximum	5271	4603	1284
n = 64			
Mean	11017	10434	583
Minimum	2146	1782	-1692
Maximum	24012	21597	4324

RESULTS OF THE USE OF DIFFERENT TREE FORMATS FOR LOG N SUMMATION ALGORITHMS **TABLE B.3**

The trees with maximum number of children generally required fewer procedure calls than the trees with all leaves at the same level. On average the difference is 10% of the total number of calls, so for all subsequent testing, the trees with maximum children were used.

The number of calls to the procedure Tree_scan used by each of the algorithms tested has been counted by running each of the programs over the test data. The results are summarised

and compared in Table B.4.

Size	Number of Procedure Calls					
	Binary Tree		Log Tree		Square Root	
	Depth First	Breadth First	Depth First	Breadth First	Depth First	Breadth First
n = 8						
Mean	101	187	85	123	85	123
Minimum	0	0	0	0	0	0
Maximum	260	555	180	332	180	332
n = 16						
Mean	611	714	427	487	427	487
Minimum	25	58	17	68	17	68
Maximum	1529	2076	900	1157	900	1157
n = 32						
Mean	3340	4440	2238	2704	1948	2494
Minimum	186	194	103	105	94	96
Maximum	8146	12334	4603	6726	3863	5824
n = 64						
Mean	17640	29260	10434	15286	8655	13572
Minimum	2829	2875	1782	1796	1385	1395
Maximum	46317	90681	21597	41172	17866	34995

COMPARISON OF RESULTS FOR THE TREE SUMMATION ALGORITHMS **TABLE B.4**

The binary summation using depth first search algorithm has been compared with other algorithms for boolean matrix multiplication. The expected performance of these algorithms, as the density of 1's in the matrix varies, has been determined. Some are more efficient with sparse graphs and others are more efficient with dense graphs. To test and compare these algorithms, matrices were generated for

probabiltiy values of Ø.ØØ1, Ø.Ø3, Ø.25, Ø.5Ø, Ø.75, Ø.97, and Ø.999. For each of the matrix sizes 8, 16, 32 and 64 a selection of 29 matrices were generated using a random function to generate the elements of the matrices. A summary of the parameters of the sample matrices is given in Table B.5.

	A Matrix		B Matrix	
	# of 1's	Probability	# of 1's	Probability
n = 8				
Mean	32	Ø.5Ø22	33	Ø.5141
Minimum	Ø	Ø.ØØØØ	Ø	Ø.ØØØØ
Maximum	64	1.ØØØØ	64	1.ØØØØ
n = 16				
Mean	13Ø	Ø.5Ø78	128	Ø.5Ø11
Minimum	Ø	Ø.ØØØØ	Ø	Ø.ØØØØ
Maximum	256	1.ØØØØ	256	1.ØØØØ
n = 32				
Mean	518	Ø.5Ø56	511	Ø.4995
Minimum	1	Ø.ØØ1Ø	Ø	Ø.ØØØØ
Maximum	1Ø24	1.ØØØØ	1Ø24	1.ØØØØ
n = 64				
Mean	2Ø54	Ø.5Ø15	2Ø44	Ø.499Ø
Minimum	4	Ø.ØØ1Ø	Ø	Ø.ØØØØ
Maximum	4Ø95	Ø.9998	4Ø93	Ø.9993

PARAMETERS OF TEST DATA FOR COMPARISON OF BOOLEAN MATRIX MULTIPLICATION ALGORITHMS **TABLE B.5**

The algorithms have been implemented on a Burroughs B6718 computer at the University of Canterbury, using Burroughs Extended Algol. The time required to process each matrix multiplication was recorded in 60^{ths} of a second, and a

summary of the results is given in Tables B.6, B.7 and B.8. The 'out of loop' heuristic was applied to all the algorithms. Table B.6 compares the summarised results obtained from the original algorithms and with 'out of loop' included for the Elementary, Four Russians and Data Compression algorithms. Only the matrices of size 64 are listed. The improvement gained from using the heuristic is also given in Table B.6. For very sparse graphs the 'out of loop' heuistic adds a small portion to the running times of the algorithms, but there is a major reduction in the running times for dense matrices. For all subsequent comparisons, the 'out of loop' algorithms will be used.

	Elementary		Four Russians		Data Compression	
		out of loop		out of loop		out of loop
n = 64						
Mean	12Ø4	481	993	576	47Ø	285
Minimum	1189	45	979	385	433	191
Maximum	1222	1617	1Ø24	1155	495	517
	Improvement					
n = 64						
Mean	Ø.481		Ø.576		Ø.284	
Minimum	Ø.354		Ø.333		1.1Ø1	
Maximum	1.617		1.155		Ø.517	

EFFECTIVENESS OF THE 'OUT OF LOOP' HEURISTIC **TABLE B.6**

Table B.7 summarises the running times obtained from testing the Elementary, Four Russians, Data Compression and Binary Summation Algorithms.

	Elementary	Four Russians	Data Compression	Binary Summation
n = 8				
Mean	1	5	5	4
Minimum	Ø	4	4	2
Maximum	3	7	6	6
n = 16				
Mean	9	23	16	12
Minimum	2	16	13	7
Maximum	2Ø	81	21	24
n = 32				
Mean	57	1Ø6	67	46
Minimum	7	75	51	22
Maximum	154	179	1Ø1	116
n = 64				
Mean	481	576	285	22Ø
Minimum	45	385	191	83
Maximum	1617	1155	517	594

RESULTS FOR ELEMENTARY AND COMPRESSION ALGORITHMS **TABLE B.7**

Table B.8 summarises the results from testing the indexing methods of O'Neill and O'Neill and the VAHB, HAVB and HAHB algorithms of Takaoka.

	VAHB	HAVB	HAHB	O'Neill & O'Neill
n = 8				
Mean	3.2	3.5	3.2	2.1
Minimum	2	2	2	1
Maximum	5	5	5	3
n = 16				
Mean	12.3	9.9	13.3	6.8
Minimum	5	6	4	4
Maximum	24	16	26	11
n = 32				
Mean	65.9	34.3	76.5	29.6
Minimum	13	19	11	12
Maximum	143	65	172	54
n = 64				
Mean	449.5	122.Ø	514.Ø	154.1
Minimum	35.Ø	62.Ø	33.Ø	43
Maximum	1Ø7Ø.Ø	251.Ø	1225.Ø	335

RESULTS FOR INDEXING ALGORITHMS **TABLE B.8**

Table B.9 summarises the results obtained from testing the algorithms with the 'Summation' heuristic included.

	'Summation' Heuristic		
	VAHB	HAVB	O'Neill
n = 8			
Mean	3	3.2	1.9
Minimum	2	2	1
Maximum	4	4	2
n = 16			
Mean	7.4	8.8	5.Ø
Minimum	4	6	4
Maximum	1Ø	16	7
n = 32			
Mean	23.5	29.4	16.5
Minimum	12	19	13
Maximum	34	64	26
n = 64			
Mean	79.2	93.Ø	72.4
Minimum	32.Ø	56.Ø	53
Maximum	114.Ø	216.Ø	118

RESULTS FOR ALGORITHMS WITH 'SUMMATION' HEURISTIC **TABLE B.9**

B.2 TAKAOKA'S ALGORITHMS FOR BOOLEAN MATRIX MULTIPLICATION

Takaoka [1979] gave a modified version of the Four Russians algorithm for boolean matrix multiplication. The algorithms is outlined in Algorithm B.1.

Algorithm B.1

Takaoka Data Compression Complexity: $O(n^2/(p^2*\log_2 n))$

Input: A and B two n*n boolean matrices

Output: C = A.B and n*n boolean matrix

Initialisation:

```
Procedure Table_set_up(n)
  if n=1 then TABLE[Ø,Ø] := Ø
  else begin
    k := n/2
    Table_set_up(n)
    for i:=k to n-1 do
      for j:=Ø to k-1 do
        TABLE[j,i] := TABLE[j,i-k]
        TABLE[i,j] := TABLE[i-k,j]
        TABLE[i,j+k] := 1
    end {of Table_set_up}
```

Method:

```
  Table_set_up(n)

  for i:=1 to n/log2n

    for j:=1 to n
      for k:=1 to log2n-1 do begin

        size := (i-1) * log2n + k + 1

        daji := daji + A[j,size] * 2^k

        dbij := dbij + B[size,j] * 2^k

        end

  for i:=1 to n
    for j:=1 to n
      cij := Ø

      k:=1
      while cij = Ø and k <= n/log2n

         cij := TABLE[daik,dbkj]
```

(end of method)

Takaoka [1979] gave a family of indexing algorithms for Boolean Matrix Multiplication. The matrices to be multiplied are first preprocessed into list structures. The list records the index at which a '1' occurred in the original matrix. The elements of the matrices can be entered into the list either evaluated by row or by column. HA or HB is used if the matrix is listed by rows, and VA or VB when the matrix is listed by columns. The combination of these lists gives the several variations of Takaoka's Indexing Methods, and the names of the methods.

Algorithm B.2 initialises the lists HA, HB, VA, and VB for the matrices A and B. The algorithm requires $O(n^2)$ running time. Only one list for each matrix will be required to calculate the product matrix as described in Algorithms B.3, B.4, and B.5. The HAVB algorithm described in Algorithm B.3 requires $O(\min(n^3p, n^2/p))$ running time. The VAHB algorithm described in Algorithm B.4, and the HAHB algorithm described in Algorithm B.5 both require $O(n^2 + n^3*p^2)$ running time. All the algorithms need the product matrix C initialised to zero. This will require $O(n^2)$ time and can be included in the initialisation of the lists.

Algorithm B.2

Initialisation of Lists for Takaoka's Indexing Methods

Input: A and B two n*n boolean matrices

Output: VA, VB, HA, HB lists

Method:

```
For i:=1 to n do begin
  k:=1; k':=1;
  h:=1; h':=1;
  for j:=1 to n do begin
    C_ij := ∅
    if a_ij=1 then begin
      HA_ik:=j;
      k:=k+1;
      end;
    if a_ji=1 then begin
      VA_ik':=j;
      k':=k'+1;
      end;
    if b_ij=1 then begin
      HB_ih:=j;
      h:=h+1;
      end;
    if b_ji=1 then begin
      VB_ih':=j;
      h':=h'+1;
      end;
    end;
  end;
```

(end of method)

Algorithm B.3

Takaoka's HAVB Algorithm

Input: HA and VB lists

Output: C = A.B an n*n boolean matrix

Method:

```
For i:=1 to n do
  for j:=1 to n do
    if C_ij=Ø then begin
      h:=1;
      k:=1;
      while HA_ik>Ø and VB_ij>Ø and k<=n and h<=n and C_ij=Ø
        do begin
        if HA_ik = VB_jh then C_ij:=1;
        if HA_ik < VB_jh then k:=k+1;
        if HA_ik > VB_jh then h:=h+1;
        end;
      end;
```

(end of method)

Algorithm B.4

Takaoka's VAHB Algorithm

Input: VA and HB lists

Output: C = A.B an n*n boolean matrix

Method:

```
For k:=1 to n do begin
  i:=1;
  while i<=n and VA_ik>Ø do begin
    h:=1;
    while h<=n and HB_kh>Ø do begin
      C_VA_ik,HB_kh :=1;
      h:=h+1;
      end;
    i:=i+1;
    end;
  end;
```

(end of method)

Algorithm B.5

Takaoka's HAHB algorithm

Input: HA and HB lists

Output: C = A.B an n*n boolean matrix

Method:

```
For i:=1 to n do begin
  k:=1;
  while HA_ik>Ø and k<=n do begin
    j:=1;
    while HB_HA_ik,j>Ø and j<=n do begin
      c_i,HB_HA_ik,j :=1;
      j:=j+1;
      end;
    k:=k+1;
    end;
  end;
```

(end of method)

APPENDIX C

MAXIMAL INDEPENDENT SETS

The algorithm of Tsukiyama et al and the modification described in Chapter 6 have been implemented in Pascal with the Sheffield Pascal Compiler on a Prime 750 at the University of Canterbury. For all programs the number and type of each recursive call or iteration were counted and the time was recorded using the timing functions provided in Sheffield Pascal, namely the mill function. For the modified algorithm, timing results include the time required to reorder the vertices of the graph. A simple $O(n^2)$ process was used to sort the vertices of the graph.

The Moon-Moser graphs were used to test the worst case running time of both the original and modified algorithms and these are described in Table C.2. The running time required by both algorithms to find the maximal independent sets is shown in Table C.2. The table shows the results from two test runs for each algorithm. The running time increases exponentially with the number of maximal independent sets in the graph. Table C.3 shows the time required per maximal independent set becomes stable and constant, to within the bounds possible on a multi-tasking computer, as n increases. The average time per maximal independent set shown in Table C.3 is weighted by the number of maximal independent sets

found for each graph size in the Table.

The Moon-Moser Graphs			
n	e	c = $3^{n/3}$	
3	3	3	3^1
6	6	9	3^2
9	9	27	3^3
12	12	81	3^4
15	15	243	3^5
18	18	729	3^6
21	21	2187	3^7
24	24	6561	3^8
27	27	19683	3^9
30	30	59049	3^{10}

WORST CASE TEST DATA **TABLE C.1**

	Running Time for Worst Case Test Data			
n	original		modified	
3	18	3	9	6
6	19	15	18	22
9	45	43	57	48
12	118	115	127	127
15	321	342	343	363
18	942	1033	967	1039
21	2766	3094	2810	3072
24	8266	9433	8479	9333
27	24776	28740	25254	28009
30	86428	90849	83839	91151

TEST RESULTS FOR WORST CASE **TABLE C.2**

	Time per Maximal Independent Set			
n	original		modified	
3	6.00	1.00	3.00	2.00
6	2.11	1.67	2.00	2.44
9	1.67	1.59	2.11	1.78
12	1.46	1.42	1.57	1.57
15	1.32	1.41	1.41	1.49
18	1.29	1.42	1.33	1.43
21	1.26	1.41	1.28	1.40
24	1.26	1.44	1.29	1.42
27	1.26	1.46	1.28	1.42
30	1.46	1.54	1.42	1.54
Average	1.40	1.51	1.38	1.50

TIME PER MAXIMAL INDEPENDENT SET FOR WORST CASE DATA **TABLE C.3**

To test the average running time of both algorithms a total of 410 graphs were randomly generated and they are described in Table C.3. The random graph generator based on Kuhn [1972] accepts the degree of each vertex in a graph and will attempt, by randomly adding edges to the graph, to generate a graph with each vertex having exactly that degree. The degree of the vertices of the graphs were randomly generated and included graphs with fixed degree, increasing degree order and decreasing degree order. Fifty graphs for each of the sizes 10, 20, 30 and 40 were generated and the complement graphs were included in the sample test data. Ten graphs on 50 vertices were generated. The random graph generator works on the number of edges in the graph, so unlike the random matrix generator used to generate the boolean matrices in Chapter 5, this generator produces undirected graphs. The number of edges in a graphs is in the range 0 <= m <= n*(n-1)/2. The algorithm of Tsukiyama et al

had a running time bound of O(n*m*c), so the graphs were generated to give a range over the number of edges, rather than being representative of all possible graphs.

Number of Vertices	Number of Edges			Density 2*m/n*(n-1)		
	minimum	mean	maximum	minimum	mean	maximum
10	5	22.5	40	0.1111	0.5	0.8889
20	15	95.0	175	0.0789	0.5	0.9210
30	25	217.5	410	0.0575	0.5	0.9430
40	40	390.0	740	0.0512	0.5	0.9487
50	305	613.0	920	0.2490	0.5	0.7510

Density	n = 40 Number of Graphs	Number of Edges		
		minimum	mean	maximum
0.00-0.349	35	40	161	254
0.35-0.659	30	310	390	470
0.65-1.000	35	526	619	740

AVERAGE CASE TEST DATA **TABLE C.4**

For several sections of the analysis the graphs are divided into three groups based on the density of the edges. The sparse graphs are those whose edge density is in the range 0 to 0.349, the middle graphs are those whose edge density is 0.35 to 0.649, and the dense graphs are those with edge density from 0.65 to 1.00. The graphs of 40 vertices, which are also used during the analysis, had a distribution of graphs into these groups as shown in Table C.4.

The number of maximal independent sets that will be in any particular graph cannot be predetermined. Those that did occur in the test data are summarised in Table C.5, with the expected and maximum number of maximal independent sets in all possible graphs for comparison. The expected number of maximal independent sets has been calculated from the expected number of cliques in a random graph given by Bollobas and Erdos [1975] and described in Chapter 2. The number of maximal independent sets varies with the density of the graphs and for the graphs of size 40, this variation is shown in Table C.5.

Number of Vertices	Number of Maximal Independent Sets				
	All Possible Graphs		Sample Test Data		
	expected	maximum	minimum	mean	maximum
10	8	39	5	9.6	32
20	36	1516	8	35.0	258
30	155	59049	20	165.5	4032
40	287	2299412	38	1820.3	50500
50	772	89540788	54	2061.0	10583
Density			n = 40		
			minimum	mean	maximum
0.00 - 0.349			106	4820	50500
0.35 - 0.659			62	334	1066
0.65 - 1.000			38	94	199

DISTRIBUTION OF MAXIMAL INDEPENDENT SETS IN AVERAGE CASE TEST DATA **TABLE C.5**

The average number of maximal independent sets in the test data is similiar or greater than the expected number of maximal independent sets. For the graphs of size 40 and 50, the test data have a higher average number of maximal independent sets than the average for all graphs of the corresponding size. Hence, the average processing time requirements will also be higher. An examination of the test data for graphs of size 40, shows the graph with 40 edges has 48480 maximal independent sets, the graph with 44 edges has 50500 maximal independent sets and the graph with 50 edges has 14432 maximal independent sets compared to the average of 707.4 maximal independent sets for the remaining 97 graphs in the test data. For graphs of size 50, one graph has 10583 maximal independent sets compared to an average of 1114 maximal independent sets for the remaining test data. These graphs have a disproportionate effect on the average number of maximal independent sets but they give an important indication of the efficiency of the algorithms, and so the results given below are for the complete set of test data.

The running time required by both algorithms to find the maximal independent sets in the average case graphs is summarised in Table C.6. The time has been measured using the mill function in the Pascal compiler and is given in milliseconds.

n	Processing Time Requirements					
	Original			Modified		
	Minimum	Mean	Maximum	Minimum	Mean	Maximum
10	15	36	67	39	55	82
20	110	298	637	169	337	663
30	267	2258	7204	412	1489	7685
40	1376	13722	82161	1727	8604	88630
50	4103	42343	161313	3488	25109	57288

TEST RESULTS FOR AVERAGE CASE DATA **TABLE C.6**

The running time requirements for the modified algorithm is, on average, decreasing as the size of the graph increases. The improvement achieved by the modified algorithm is shown in Table C.7.

n	Processing Time Improvement Modified / Original		
	Minimum (No Improvement)	Mean	Maximum
10	2.60	1.59	0.96
20	1.68	1.19	0.66
30	1.74	0.79	0.25
40	1.81	0.71	0.13
50	1.06	0.73	0.15

IMPROVEMENT IN RUNNING TIME ACHIEVED BY MODIFIED ALGORITHM **TABLE C.7**

This improvement is also reflected by the lower time per maximal independent set required by the modified algorithm as n increases. The summary of these values is shown in Table C.8.

n	Time per Maximal Independent Set					
	Original			Modified		
	Minimum	Mean	Maximum	Minimum	Mean	Maximum
10	1.13	4.13	10.20	1.44	6.43	14.00
20	1.05	11.56	25.27	1.58	13.62	30.07
30	1.79	24.74	62.76	1.91	18.16	59.16
40	1.41	41.61	121.60	1.75	27.53	91.08
50	10.17	69.65	145.13	5.41	44.10	77.57

TIME PER MAXIMAL INDEPENDENT SET **TABLE C.8**

The modification was designed to reduce the running time af the original algorithm by reducing the number of type 2 procedure calls made while processing. Table C.9 summarises the number of calls of type 1 and type 2 for both algorithms for the average case test data.

	Number of Each Type of Procedure Call					
	Original Method			Modified Method		
n = 50	total	type 1	type 2	total	type 1	type 2
Mean	17660	2090	13509	13630	2851	8718
Minimum	936	21	861	1488	128	768
Maximum	85351	10725	67625	56531	13066	32882
n = 40						
Mean	9512	482	6210	9976	3518	4637
Minimum	684	14	628	654	16	490
Maximum	192548	55141	86907	203666	71812	88363
n = 30						
Mean	1143	186	792	1143	357	621
Minimum	174	7	125	294	14	129
Maximum	16576	5271	7273	16576	5271	7273
n = 20						
Mean	226	34	157	254	68	151
Minimum	75	5	55	129	10	60
Maximum	733	144	426	1026	320	418
n = 10						
Mean	43	8	25	48	12	26
Minimum	24	1	11	33	3	14
Maximum	83	21	45	83	23	45

SUMMARY OF THE NUMBER OF RECURSIVE CALLS FOR AVERAGE CASE TEST DATA **TABLE C.9**

The algorithms of Bron-Kerbosch and Loukakis-Tsouris were also tested on the data generated to test the original and modified algorithms of Tsukiyama et al, although not as extensively. For convenience, these results are listed together, but the algorithms were implemented in different languages. The running time on the worst case test data for these algorithms is summarised in Table C.1Ø.

	Running Time for Worst Case Test Data	
n	Bron-Kerbosch	Loukakis-Tsouris
3	25	6
6	25	9
9	33	12
12	81	3Ø
15	221	82
18	648	252
21	193Ø	773
24	5794	2479

TEST RESULTS FOR WORST CASE **TABLE C.1Ø**

The time required per maximal independent set by the Loukakis-Tsouris algorithm and the time required per clique by the Bron-Kerbosch algorithm has been calculated and is given in Table C.11. For both algorithms the time is dependent on the number of cliques or maximal independent sets in the graph, and as with the results in Table C.3, this time becomes stable as the size of the graph increases (to within the limits possible on a multitasking computer).

	Time per Maximal Independent Set or Clique	
n	Bron-Kerbosch	Loukakis-Tsouris
3	8.33	2.00
6	2.78	1.00
9	1.22	0.44
12	1.00	0.37
15	0.91	0.34
18	0.89	0.35
21	0.89	0.35
24	0.88	0.38
Average	0.89	0.41

TIME PER MAXIMAL INDEPENDENT SET FOR WORST CASE DATA **TABLE C.11**

n	Processing Time Requirements					
	Bron-Kerbosch			Loukakis-Tsouris		
	Minimum	Mean	Maximum	Minimum	Mean	Maximum
10	9	20	48	37	44	61
20	45	109	712	72	107	230
30	106	495	11261	134	270	1133
40	173	2953	46037	356	2728	72988
n	Tsukiyama et al Original			Tsukiyama et al Modified		
	Minimum	Mean	Maximum	Minimum	Mean	Maximum
10	17	34	55	39	55	76
20	110	300	524	169	343	603
30	267	2313	5194	412	1487	4218
40	1376	13722	82161	1727	8604	88630

TEST RESULTS FOR QAVERAGE CASE DATA FOR ALL ALGORITHMS **TABLE C.12**

The average case running times for the Bron-Kerbosch and Loukakis-Tsouris algorithms is summarised in Table C.12. Table C.12 also gives the average running times of the original and modified Tsukiyama et al algorithms for the graphs that were tested. The time per maximal independent set or clique is summarised in Table C.13. These results have been separated into the three categories based on the density of edges in the graphs.

n	Time per Maximal Independent Set or Clique					
	Bron-Kerbosch			Loukakis-Tsouris		
	Ø-Ø.349	Ø.35-Ø.649	Ø.65-1	Ø-Ø.349	Ø.35-Ø.649	Ø.65-1
1Ø	2.33	2.3Ø	1.98	4.22	5.Ø8	5.2Ø
2Ø	3.63	4.53	2.45	2.56	4.51	4.23
3Ø	3.97	3.99	3.13	2.69	3.1Ø	4.37
4Ø	3.34	3.86	3.37	1.72	3.19	5.58
n	Tsukiyama et al Original			Tsukiyama et al Modified		
	Ø-Ø.349	Ø.35-Ø.649	Ø.65-1	Ø-Ø.349	Ø.35-Ø.649	Ø.65-1
1Ø	2.18	3.92	5.Ø2	4.4Ø	6.26	7.67
2Ø	5.1Ø	11.69	15.75	6.38	14.13	17.57
3Ø	9.85	26.13	38.58	8.7Ø	17.54	31.27
4Ø	12.44	43.75	68.97	5.92	22.85	53.14

TIME PER MAXIMAL INDEPENDENT SET OF CLIQUE FOR AVERAGE CASE DATA FOR ALL ALGORITHMS **TABLE C.13**