THE MANIPULATION OF TREES AND LINEAR GRAPHS WITHIN A COMPUTER AND SOME APPLICATIONS

ALEX KAREL OBRUČA

Thesis submitted for the degree
of Doctor of Philosophy at the
University of Newcastle-upon-Tyne
on June 1966

P R E F A C E .

The ideas for this thesis developed during and shortly after my work for the Master of Science degree. A few concepts and definitions of this thesis were introduced in the dissertation but are re-introduced for completeness . Most of the thesis is original and where it is not, the parts are indicated as such. The last part of the matrix bandwidth minimisation is the result of a suggestion from and collaboration with Dr. H.I.Scoins. One of my main problems has been terminology. Every one who has published seems to make it a point of principle to use different names for the same concepts. I am not blameless, however, I try to use a combination of Berge's and Harary's definitions and terminology as a basis, with a few additions of my own.

The thesis is divided into two parts : Chapters and Appendices. Chapters 1 to 6 set out to define the terms used and the theory behind the applications. The Appendices contain the relevant programs and other practical work.

I would like to take this opportunity of thanking the Science Research Council (or the then Department of Scientific and Industrial Research) for their finan-

cial help without which this thesis would not have been possible. I would like to also thank the members of the Computing laboratory for their patience and especially Dr. H.I.Scoins for his helpful supervision.

Last but not least, I would like to record my appreciation to my wife who has put up with a lot of bad temperedness on my part as a result of this thesis and without whose help this thesis would not have appeared at all.

A. Obrača

A B S T R A C T .

A digraph of z points and br arcs can be represented by its adjacency matrix. Within a computer this means a storage of z^2 elements. By suppressing obvious information, a reduction can be made in the storage required. The branches list representation stores the non-zero elements of the adjacency matrix and requires only (br + z) elements.

Any trees required for computer manipulation are rooted and ordered. They can be represented in the two arrays below[j] and posnbr[j], where below[j] stores the below of a point j and posnbr[j] its positive neighbour. However, this representation is very inconvenient for 'going up' the tree. Thus another representation called the rd, lu representation is defined such that it is nearly as easy to go up the tree as to go down it. A few procedures were written which enabled an ordered-rooted tree to be divided into two parts and rejoined together at different points. This technique forms a basis for Toptree and Transportree. A succesful investigation was also carried out to find a relationship between labelled ordered-rooted trees and labelled binary pendant trees.

Toptree is a heuristic method of obtaining a good solution in a relatively short time to the Travelling Salesman Problem . It is based on the observation that the majority of lines of a minimal solution (to the problem) appear in the minimal spanning tree (for that same graph). The technique is to reduce multi-membered stars of the minimal spanning tree so as to have all points incident to at most two lines. This seems to give very good results on both random data and published examples.

The problem of minimising the bandwidth of a matrix was also examined . The problem was re-stated as that of having to label the points of a large graph so that the maximum difference between the labels of adjacent points is a minimum. The problem of doing this quickly was not solved but here again , techniques based on the spanning tree for that graph were evolved which reduced the initial bandwidth considerably . An algorithm was written which did find the minimum bandwidth labelling by going through the permutation list. But due to the size of the list this was slow and impractical for graphs with z greater than 20.

The nature of this work was such that it was suitable to tackle the Shortest Paths (through a digraph) Problem. The tree spanning technique was developed so that for large, highly sparse digraphs (or networks) , it was found to be more efficient than the Cascade method, one of the better matrix type methods.

Finally H.I.Scoins method of solving the Transportation Problem was refined (and called Transportree) so that the tree was not kept in the below array (i.e. as a rooted tree) but in the rd, lu representation. This results in the time spent list processing in order to go up the tree being drasticaly reduced . This last section was merely an exercise in showing how ordered-rooted trees and their manipulation are of use in a wide array of problems.

I N D E X .

I INTRODUCTION.

1.1 General Discussion.

The underlying theme behind this work, as the title may suggest, has been the study of trees and how they may be of use in the solution of some types of problems. A little time is spent describing graphs but merely for completeness sake.

A lot of work has been done in graph theory many theorems have been proved or disproved, but unfortunately little of it has been applied . Concepts like Grundy functions , chromatic or ordinal numbering of a graph or various operations such as conjunctive products of and compositions on two graphs, seem to be merely the toys of pure mathematicians. The thesis goes only a little way in using some of these ideas. The main building block behind the applications in Chapters 3 to 6 has been the spanning tree. In Chapter 3, the minimal cost spanning tree for a given graph is used to find a good solution to the Travelling Salesman Problem . In Chapter 4, the spanning tree is used to find a good permutation matrix which yields a reduction in the bandwidth of a symmetric matrix . In Chapter 5, we use the spanning tree technique to obtain the shortest distance between pairs of points in a digraph . In Chapter 6,

we could have used the minimal cost spanning tree to obtain an initial solution to the Transportation Problem. However as the basic solution is a spanning tree , we use the manipulation of Chapter 3 to rearrange the configuration of the tree into one which has optimal solution.

Working knowledge of Algol is assumed when discussing the programming. However, slight traces of Algol terminology do appear in places throughout the thesis. The definitions that follow will be primarily concerned with the terms used in the following chapters . Further definitions and reading can be found in [1, 2, 5, 7, 45].

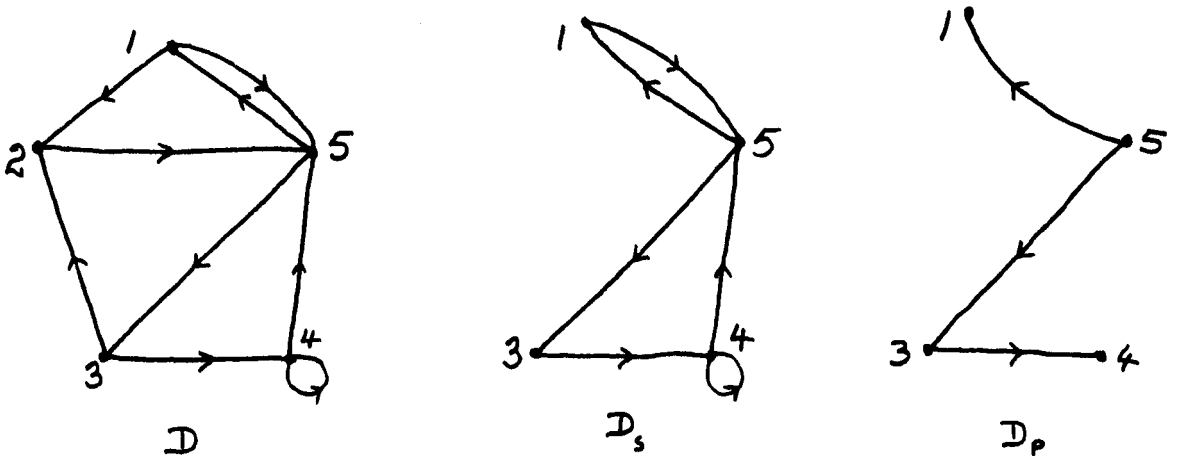
1.2 Definitions.

1.21 Digraphs.

We define a <u>set</u> as a collection of objects (which themselves will be referred to as <u>elements</u>). A set will be represented by capital letters and its elements by small ones, e.g. $V = \{a, b, \ldots, v, w\}$. Let $\mathcal{L}$ be a one-to-many function mapping a set V into itself . Then we define a <u>digraph</u> (<u>directed graph</u>) to be the pair $(V, \mathcal{L})$ for some V and some function $\mathcal{L}$. A pictorial representation can be obtained if the set V is represented by points in a space and if $y \in \mathcal{L}(x)$, where $x, y \in V$, then in the space there will be a continuous line joining x to y. To distinguish between a line joining y to x and one joining x to y (i.e. $x \in \mathcal{L}(y)$ and $y \in \mathcal{L}(x)$), we insert an arrowhead in the appropriate direction. Within the pictorial representation, the elements v_i of V will be referred to as <u>points</u> (<u>vertices</u> or <u>nodes</u>). The pair (x,y) where $y \in \mathcal{L}(x)$, will be called an <u>arc</u> of the digraph. The set of arcs of a digraph will be denoted by U .

Having defined a digraph in the abstract, we now proceed to make further definitions in terms of the pictorial representation, rather than in terms of sets and function mappings. A <u>sub-digraph</u> of a digraph D, is defined

to be a digraph whose points are a subset of those of D and consisting of all the arcs of D joining these points. A <u>partial digraph</u> of a digraph D, is any digraph whose points and arcs are a subset of those in D. In fig. 1.211 we have an example of a digraph D, a subdigraph D_s of D and a partial digraph D_p ,also of D. D_s and D_p contain all the points of D except for v_2 .



(fig. 1.211)

Two points x,y within D are said to be <u>adjacent</u> if 1/. they are distinct and

2/. there exists an arc going from x to y or from y to x.

If there is an arc u going from x to y, then we say that x is the <u>initial</u> and y the <u>terminal</u> points of that arc. We may also say that u is <u>incident from</u> x and <u>incident to</u> y.

Similarly two arcs u,v are said to be <u>adjacent</u> if

1/. they are distinct and

2/. they have a point in common.

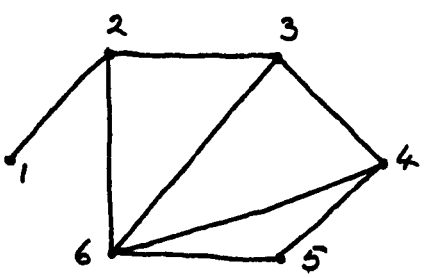
A <u>loop</u> of a digraph is an arc whose terminal and initial points are identical. We say that a digraph contains <u>parallel lines</u> if there is more than one arc in the same direction joining any two points. In the example of fig. 1.2111, points v_1 and v_5 are adjacent in all three figures, but v_4 and v_5 are adjacent only in D and D_3. There also is a loop in both D and D_3 centred on v_4.

We go on to define a <u>path</u> as a sequence of arcs $(u_1, u_2, u_3, \ldots, u_3)$ of a digraph such that the terminal point of each arc coincides with the initial point of the succeeding arc. A path is <u>simple</u> if it does not use the same arc twice and <u>composite</u> otherwise. If a path does not use the same point twice, it is said to be <u>elementary</u> . If a path meets in turn the points $\{x_1, x_2, x_3, \ldots, x_k\}$ we can represent it by $[x_1, x_2, x_3, \ldots, x_k]$, or if there is no ambiguity by $\mu[x_1, x_k]$. The <u>length</u> of a path $(=(u_1, u_2, \ldots, u_k)$ say), is the number of arcs in the sequence, $l(\mu)=k$. In D of fig. 1.211 , if $\mu=[v_1, v_2, v_5, v_3, v_4]$ then $l(\mu) = 4$.

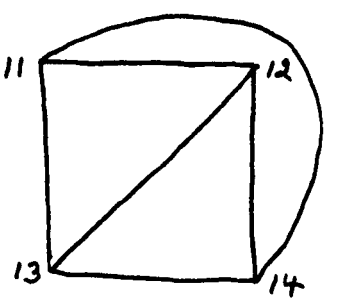
1.22 Graphs.

A digraph is said to be a graph if for every $(x,y) \in U \Rightarrow (y,x) \in U$ where U is a digraph. Thus every pair of adjacent points are connected by two oppositely directed arcs. To simplify the representation of such arcs , we shall set up a rule that two adjacent points (of a graph) will be joined by a single continuous line (to be called a line or segment or edge) which carries no arrowhead. A graph is connected if for every pair of distinct points , there is a path going from one to the other. If the graph is disconnected, then each of its connected subgraphs is called a component. A graph is said to be complete (or maximally connected) if every point is adjacent to every other.

A circuit is a path $(u_1, u_2, \ldots, u_k)$ where u_k is adjacent to u_1. A simple circuit is one where the path is elementary. The degree of a point v_i , denoted by $d(v_i)$, is equal to the number of points adjacent to v_i.



a connected graph



a complete graph

(fig. 1.221)

In the graphs of fig. 1.221, we have an example of a connected graph and one of a complete graph. Either of $[v_2, v_3, v_4, v_5, v_6]$ or $[v_{11}, v_{12}, v_{14}]$ could be taken as an example of a simple circuit. We also have $d(v_1) = 1$, $d(v_6) = 4$ and $d(v_{13}) = 3$.

A partial graph is said to <u>cover</u> a graph if the partial graph contains all the points in the graph. A <u>labelled graph</u> is one where each point is associated with a unique positive integer. The graphs considered in fig.1.221 were labelled graphs. It is usual when labelling graphs to number the points from one upwards. The point which is labelled j will be referred to as v_j. Consider a labelled graph G and suppose we wish to permutate the labels of G. We denote the new labelling of the points by means of a superscript. Thus if the point originally labelled 11, i.e. v_{11}, was to be relabelled 27, we would refer to it by v_{11}, in terms of its old labelling or by v'_{27}, in terms of the new. If this point was to be relabelled, 6 say we could refer to it by v^2_6. We can thus write $v_{11} \equiv v'_{27} \equiv v^2_6$. It may also be desirable when relabelling a graph, to discuss the new label of a point. We denote the new label of a point j by lab(j). In the example just described, we can thus write either $v_{11} \equiv v'_{27}$ or lab(11) = 27. We may have within a labelled graph, labelled lines u_1 to u_{br}, where

br is the number of lines within the graph . If we have not labelled the lines but we wish to refer to one in particular, we can do so by means of its two end points v_i, v_j, i.e. by $(i - j)$. Suppose we wish to refer to a partial graph within a labelled graph and that this partial graph contains the points $v_1, v_2, v_3, \ldots, v_s$. We would do so by means of $\{v_1, v_2, v_3, \ldots, v_s\}$.

An <u>associated</u> <u>cost</u> graph is one where every line u_i (or $(j - k)$), has associated with itself a cost parameter $c(i)$ (or $c(j,k)$). Thus we can thus refer to the cost of the total graph which is equal to $\sum_{i=1}^{br} c(i)$ or $\sum_{j=1}^{z} \sum_{k=1}^{z} c(j,k)$. We may also refer to the cost of a partial graph or in particular, to the cost of a path $[u_1, u_2, u_3, \ldots, u_k]$ which will be equal to $\sum_{j=1}^{k} c(j)$.

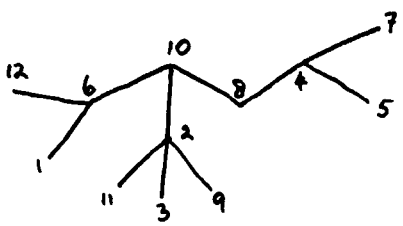
1.23 Trees.

A tree T , is a connected graph which has no circuits. A rooted tree or r-tree is one which has one and only one of its points designated as a root point (or simply root). An ordered rooted tree is defined within a halfplane such that all its points lie on one side of a cut , which meets the tree at the root . An ordered rooted tree can be abbreviated to an o-r tree, or with a slight change in the juxtaposition of its first two letters , a r-o tree which , phonetically, becomes a e-tree. A star (at a point) is the set of all lines incident to that point. Two stars are said to be adjacent if they have a line in common. A star will be multi-membered if it contains more than two lines and a tree with no multi-membered stars will be called a chain .

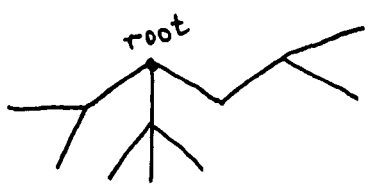
Every point of a r-tree has associated with itself a generation number which has the value equal to the length of the path between it and the root (i.e. if gen[j] stands for the generation number of the point j , then we have $gen[j] = l(\mu[v_j, root])$). A point adjacent to only one other point can be referred to as an end point (of the tree). We define a partial ordering on the points of the r-tree by means of the symbols > , < . We write $v_i > v_j$ i.e. v_j precedes v_i , if $gen[j] < gen[k]$ and $\mu[v_i, v_j]$ does not include the root point. We can thus define a sub-

<u>tree</u> at v_j , S_{v_j} , of T as being that subgraph of T which includes v_j and all points that <u>succeed</u> it. v_j is designated as the root of S_{v_j} .

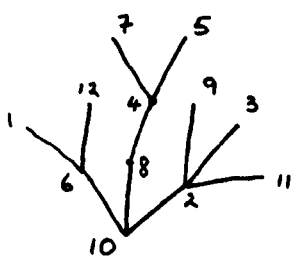
For any point v_k , the set of points adjacent to it and succeeding it, is called a <u>package</u> of which it, v_k , is called the (<u>package</u>) <u>head</u> . If v_k is the head of a package consisting of $v_\ell, v_m, \ldots, v_p$ then $S_{v_\ell}, S_{v_m}, \ldots, S_{v_p}$ are all <u>branches</u> of the subtree at v_k .



tree , T

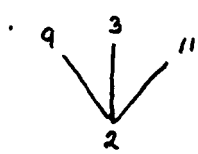


r-tree , T_r



ρ-tree , T_ρ

S_{v_8}



a package (head at v_2)

(fig. 1.231)

Within the cut plane wherein the ρ-tree lies, we can define a sign convention where <u>positive sense</u> (+ve sense) is a clockwise motion around a point . Thus within a package, a point v_p has a +ve and/or -ve neighbour or neither. v_q is the <u>+ve neighbour</u> if on traversing from v_p in the +ve sense around the package head , the next point within the package is v_q . A <u>-ve neighbour</u> is similarly defined . A <u>ρ-chain</u> is a chain which is ordered and rooted at an end point.

A <u>binary ρ-tree</u> is one whose every package contains two elements. An example of a binary ρ-tree is the monkey puzzle tree sort. A <u>bifurcating ρ-tree</u> is one whose packages contain at most two elements or lines: similarly for trifurcatingρ-trees and so on. A <u>labelled tree</u> is a labelled graph which is a tree. We shall assume in general , that all ρ-trees are labelled ones unless otherwise stated. A <u>pendant ρ-tree</u> is a ρ-tree whose end points (or pendants) only are labelled. A practical example of one is a list processing tree list.

Let us finally define a <u>directed r-tree</u> , T_D on a digraph D , to be a partial digraph of D where every point of T lies on an elementary path whose initial vertex is unique and is called the root.

1.23.1 Some Properties of Trees and Graphs.

Any tree, T, will have a diameter denoted by diam (T) , which is the length of the longest path in T . Similarly a ρ-tree T_ρ , will have a height , hght(T_ρ) which is equal to Max l (μ[root,v_i]) , where v_i is any end point of T_ρ . This can be defined recursively as

$$\text{hght}\ (T_\rho) = \text{Max}\left\{\text{hght}\ (S_{v_j})\right\} + 1 ,$$

where v_j is adjacent to the root of T_ρ .

Within a connected graph, there will always be at least one path between any two points . Define $\mu_m[v_i,v_j]$ as ~~the~~ a shortest path between v_i and v_j. Then if we consider all μ_m between all pairs of points, there will be a longest one which will be called the maxmin path. It can also be described as that path which yields

$$\text{Max}\left\{l\ (\mu_m\ [v_i,v_j])\right\} \quad \text{over all pairs } v_i, v_j \text{ in the graph.}$$

A spanning tree of a graph is any tree that covers the graph. A minimal spanning tree of a cost associated graph is that spanning tree whose cost is a minimum. A mushrooming r-tree , T_m , of a graph is a spanning r-tree and derives its name from the method of its construction (Chpt. 4.41) . For a given root point, v_r , the path length l (μ[v_r,v_i]) ,for any v_i of the mushrooming r-tree, is equal to the maxmin path length within the graph

i.e. for all mushrooming r-trees whose root is v_r we have

$$l\ (\mu[v_r, v_i]) = \mu_m[v_r, v_i] \quad \text{within the graph .}$$

A <u>mushrooming r-tree with maximum height</u> , T_H , is that mushrooming r-tree of the graph which yields the maximum value for hght (T_m) .

A <u>maximal spanning directed r-tree</u> is a directed r-tree which contains every point reachable from the root by means of a path in the digraph.

Lastly, given a graph G and a spanning tree on it , T_s , we define the <u>cotree</u> , C , as being that set of lines in G which do not appear in T_s . We thus have $T_s \cup C = G$. (We can, if necessary, extend this definition so as to include the cotree within a digraph of a maximal spanning directed r-tree.)

1.23.2 Functions on Trees.

For a given ρ-tree, T_ρ, and its set of points V, we can associate certain further one-to-one functions mapping V into itself. These functions will be dependent upon the configuration of T_ρ. As a consequence of the definition of precedence and packages, we can define a function below(x), where $x \in V$, as having the value of that package head within which x lies. This is not quite complete as we have not mentioned the predecessor of the root. Thus the definition of below(x) is completed by making below(root) equal root. From the definition of +ve and -ve neighbours, we can define the functions pos nbr(x) and neg nbr(x) as we did for below(x). If a point v_j has no positive neighbour, we put posnbr(j) = 0 and also make posnbr(root) = 0.

It is easy to see that if we were given only the values of below(x), for all x, we would be able to reconstruct the r-tree. And with the addition of the values of posnbr(x) or negnbr(x), again for all x, we would be able to reconstruct the corresponding ρ-tree. In the next chapter we use these functions in order to discuss the best ways of storing (the configuration of) a ρ-tree within a computer.

Combining the definitions of below(x) and

posnbr(x) , we can derive the following new function , poss succ (x), standing for the positive successor of x , and it is equal to

1/. posnbr(x) , if posnbr(x) $\neq$ 0 , else

2/. pos succ(below(x)) , if below(x) $\neq$ x , else

3/. x (x being the root) .

Conversely neg succ(x) can be similarly defined . Another function which can be derived from below(x) and posnbr(x) is rd(x) , the 'right or down' function . This is equal to

1/. posnbr(x) , if posnbr(x) $\neq$ x , else

2/. x , if x is the root , else

3/. - below(x) .

The rd function is slightly different from previously defined ones for two reasons . Firstly , it carries more information by the inclusion of the +ve or -ve sign. Secondly , the rd function could not be used , as it stands, repeatedly. That is , below^n(x) has some meaning (where n is a positive integer) , whereas we could not always be sure as to the existence of rd^n (x) , for we have not defined rd(y) , where y is a -ve integer .

The above definitions were of terms or functions which , one could think [redacted] , moved across or down the ϱ-tree . That is , repeated applications (with suitable changes of sign in the case of the rd function)

would yield the root or the right hand side of a package . We now define a function , above(x) , which is, in a sense, the opposite to below(x) , and it is equal to

1/. 0 , if x is an end point of the ϱ-tree, else

2/. y , where negnbr(y) = 0 and below(y) = x .

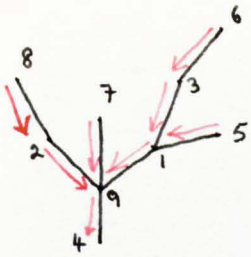
We can now combine above(x) and negsucc(x) into another function called the 'left and up' function or lu(x) , which will be equal to

1/. x , if above(x) = 0 and negsucc(x) = root, or

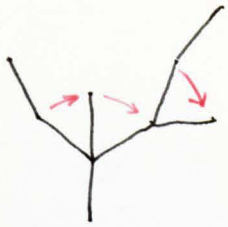
2/. - below(negsucc(x)) , if above(x) = 0 , else

3/. above(x) .

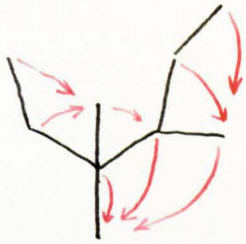
lu(x) is similar to rd(x) , in that an application of lu(x) will yield +ve or -ve numbers . As lu(x) is dependent upon above(x) , we see that repeated applications of this function will move us up and leftwards across the ϱ-tree . As an example of the above described functions, consider the ϱ-tree in fig.1.23.21. The ϱ-tree is represented by black lines and the result of the application of each of the functions on a point is indicated by red lines . Where there is a minus sign on the diagram , this means that the result of the application of the function on that particular point was minus the point indicated . Table 1.23.22 shows the result in applying all the mentioned functions on the ϱ-tree of fig. 1.23.21 .



below(x)

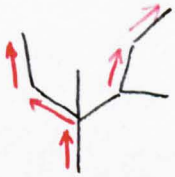


pos nbr(x)

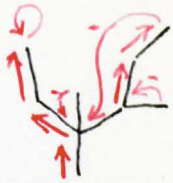


pos suc(x)

rd(x)



above(x)



lu(x)

(figs. 1.23.21)

j	gen[j]	below(v_j)	pos nbr(v_j)	poss succ(v_j)	rd(v_j)	neg nbr(v_j)	neg succ(v_j)	above(v_j)	lu(v_j)
1	3	9	0	4	-9	7	7	3	3
2	3	9	7	7	7	0	4	8	8
3	4	1	5	5	5	0	7	6	6
4	1	4	0	4	4	0	4	9	9
5	4	1	0	4	-1	3	3	0	-1
6	5	3	0	5	-3	0	7	0	-9
7	3	9	1	1	1	2	2	0	-9
8	4	2	0	7	-2	0	4	0	8
9	2	4	0	4	-4	0	4	2	2

(table 1.23.22)

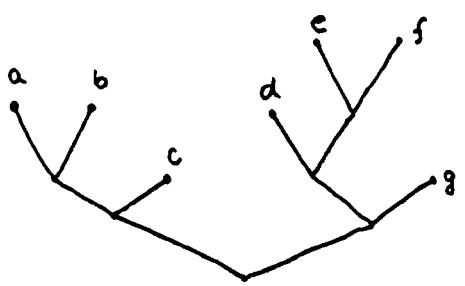
1.3 A Short Note.

Early in the work, an interesting relationship was found between binary pendant ρ-trees and multifurcating -trees . Consider a set $A = \{a,b,c, \ldots ,t\}$ and an operation on A which is neither commutative nor associative. Let us denote this operation by $\odot$. In algebra , we call the result of a certain number of operations a <u>monoid</u> . Define a <u>simple monoid</u> to be an element of A . If u,v and w are monoids , let $w = (u \odot v)$, where

$$u = [(a \odot b) \odot c] \quad \text{and}$$

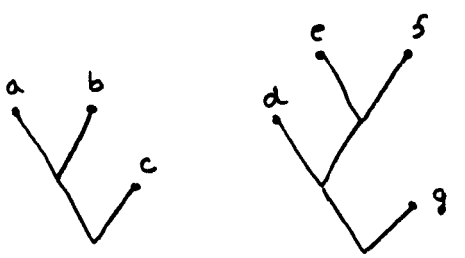
$$v = \{[d \odot (e \odot f)] \odot g\} ,$$

where a,b,c,d,e,f and g are elements of A . With the monoid w , we can associate the binary pendant ρ-tree in fig. 1.31 and the monoids u and v have representation as in fig. 1.32 [1 ,page 161] .



$w = (u \odot v)$

(fig. 1.31)



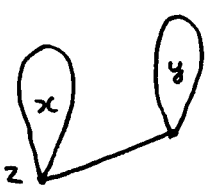
u v

(fig. 1.32)

It can be seen that the operation $\odot$ represents, in the ρ-tree , a join of the two structures corresponding to the two symbols on either side of it . The join is accomplished by inserting a new root which is adjacent to the two other roots , taking into account the sense of rotation .

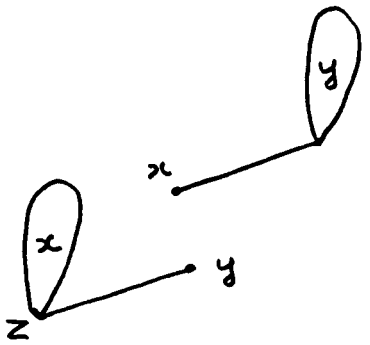
The counting series for the number of binary pendant ρ-trees with n pendants (or labelled end points) is given by $\frac{1}{2n+1}\binom{2n+1}{n}$. The point of interest is that this series also represents the number of multifurcating ρ-trees with n points. An investigation was carried out to find the relationship , if any , between these two sorts of ρ-trees .

It was discovered that the monoids could be interpreted to represent multifurcating ρ-trees. This representation is unique . Suppose x and y are two monoids (simple or otherwise) and let z be the first element to appear in x , when reading from left to right . In the case where x is simple we have x = z . Further let x and y be represented by  and  , these being their ρ-tree representation. Then the pictorial representation of the ρ-tree ($x \odot y$) can be defined uniquely as :



Consider the case when :

x is simple (i.e. $x \equiv z$) , we obtain



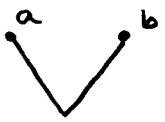
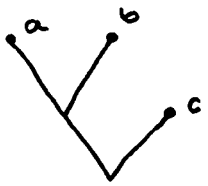
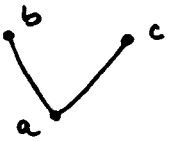
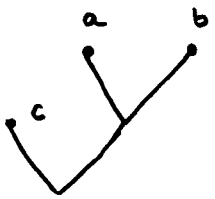
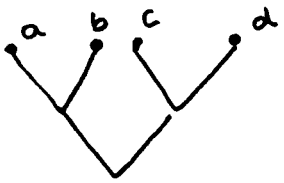
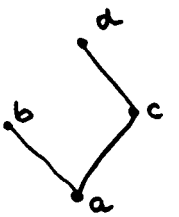
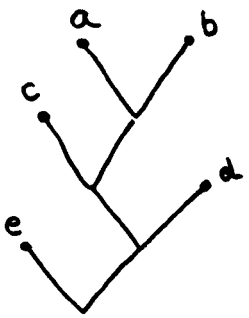
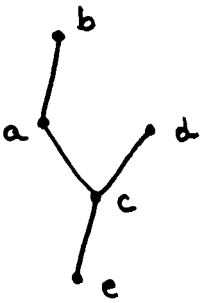
y is simple , and we obtain

both x and y are simple, we have

In other words, the multifurcating ρ-tree ($x \odot y$) is created by joining the root point of y , to the root point of x by means of an extra line , placed in the most positive position of the package of which z is the head . Similarly we can , for any multifurcating ρ-tree produce its corresponding monoidal expression .

Hence given any monoid we can associate with it a unique ρ-tree and vice-versa . Thus any binary pendant ρ-tree can be associated with a unique multifurcating ρ-tree and vice-versa. The author managed to find algorithms which, given a ρ-tree in the one representation, evaluated the corresponding ρ-tree in the other representation. Initially, this was accomplished by reference to their common monoidal expression, but later a direct transformation was achieved. Examples of some corresponding multifurcating and binary pendant ρ-trees appear in fig. 1.33 .

(When writing the monoidal expression, the operator $\odot$ may be omitted without any loss of clarity,e.g. (a(b c)).)

pendant binary ρ-trees	monoidal expression	multifurcating ρ-trees
	(a b)	
	((a b)c)	
	(c(a b))	
	((a b)(c d))	
	(e((c(a b))d)	

(table 1.33)

II REPRESENTATION AND MANIPULATION .

2.1 Representation.

The type of representation of a graph (or tree) within a computer is going to be governed, to a certain extent , by the use it is put to in any given problem. We may be interested in a graph only for its existence. That is , it may be necessary to enumerate the graphs or examine them sequentially , testing whether they have some property or not . The representation in this case can be made extremely compact . On the other hand , we may require the graph to store information , where the configuration of the graph determines the relationship between pieces of information , it may be desirable to manipulate this graph into another configuration or vary the pieces of information attached to each node of the graph . In this case, the representation has not only to be reasonably compact , but has to have the virtue of being easily manipulated and each node and its corresponding piece of information has to be easily accessable.

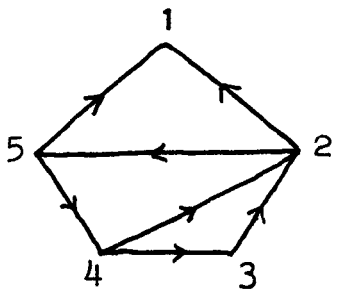
2.11 Theoretic Representation.

2.11.1 Graphs and Digraphs.

The instinctive and most obvious way to represent a graph or digraph , is to indicate the presence or absence of a line or arc , between all pairs of points. This is achieved in the adjacency (or associated) matrix , A . If we have a z-point digraph , D , then we associate with it , the zxz matrix , A , where a_{ij} ($\in$ A) = 1 , if there is an arc going from v_i to v_j and a_{ij} = 0 otherwise. For a graph, A , will be symmetrical. Another common representation is the incidence matrix , M . In this matrix , rows represent points and columns are to be associated with arcs. Thus we have

m_{ij} = 1 , if v_i is the initial point of u_j ,
= -1 , " " " " terminal " " " ,
= 0 , if there is no incidence between them.

It can be seen that the second representation is more space consuming. Examples of the two representation will be found in fig. 2.11.11 .



digraph D

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

adjacency matrix A

(fig. 2.11.11)

$$\begin{bmatrix} -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

incidence matrix M

We note as before , that if the digraph has z points and br arcs where br > z , then the number of elements in M exceeds those in A . We see that A , even if it is a more compact representation than M , is still very wasteful in so much as that unless the digraph is complete,

there is redundant information . A more compact representation would result , if we were to indicate either the presence or absence of an arc , but not both . This leads to the listed pairs representation . Here we represent the digraph as a list of pairs of numbers :-

(a_1,b_1) , (a_2,b_2) , (a_3,b_3) , ... ,(a_{br},b_{br}) where (a_i,b_i) denotes the presence of an arc in the digraph joining v_{a_i} to v_{b_i} . We note that if the digraph is a graph, every arc will occur twice (i.e. there will occur both (x,y) and (y,x)) . Thus the list will be halved ,if we suppress one of the pairs (of arcs for the graph) . This may be done by stipulating that for all pairs (a_i,b_i) , we have $a_i < b_i$. Let us assess the amount of space required to store a graph uniquely . The incidence matrix requires $z \times br$ elements. The adjacency matrix requires $z \times z$ elements . The listed pairs uses only $4 \times br$ elements and this is reduced to $2 \times br$ elements , when the redundant pairs are suppressed.

We must inevitably sacrifice something in making the representation smaller and smaller : in this case it is clarity . Within A and M , we could immediately not only visualize the digraph, but very quickly find out , say, the number of arcs eminating from any point. This cannot be said for the listed pairs representation.

We can make one more reduction in the representation which requires $4 \times br$ elements, to one which requires only $(br + z)$ elements. This is achieved by shuffling the list about and suppressing some further redundant information. Let us ~~suppose we~~ stipulate a further ordering on the initial elements in the pairs. That is, we order the list of pairs lexicographically with respect to a_1, the first elements in all the pairs. This ordered list will contain sublists, all of which contain the same initial number.

Consider each sublist seperately e.g. (a_k, b_{k_1}), (a_k, b_{k_2}), (a_k, b_{k_3}), . . . , (a_k, b_{k_r}). There are r pairs in this sublist, which means that v_{a_k} is connected to r other points. This sublist could be shortened to

a_k ; r ; b_{k_1} ; b_{k_2}; b_{k_3}; . . . ; b_{k_r}. The first element is the first number in all the pairs, the second indicates the number of lines incident from it, followed by a further r numbers, which are the labels of those points to which v_{a_k} is adjacent. We come now to the only slightly tricky part. If we now join all these sublists up, beginning to end, in their original order and also suppress a_k and r of each sublist, we are left with a string of numbers. We can read sense into this apparent chaos by defining another list $st_1, st_2, st_3, \ldots, st_z$. This list indicates

the starting position , st_α , in the main list , where we look for the labels of those points adjacent from any point v_α . For example , if we wish to find all points adjacent from point v_p , we note the value of st_p , and look in the main list at all the elements in the st_p^{th} position to the $(st_{p+1}^{th} - 1)$ position . Consider the example in fig.2.11.11. The fully listed pair representation would be

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

Rearranging this we get

<u>(2,1), (2,5),</u> <u>(3,2),</u> <u>(4,2), (4,3),</u> <u>(5,1), (5,4)</u>

where the sublists are underlined. In the last step this becomes 1, 5, 2, 2, 3, 1, 4 ;

and 1, 1, 3, 4, 6 ;

to be called the branches list representation.

Let us call the main list , the branch list and the subsidary list , the rowstart list . Thus in the previous example, if we wish to find all the points adjacent to v_4 ,we note the values

branches[rowstart[4]] to branches[rowstart[5] - 1]

which is branches[4] to branches[5] , giving us the values 2 and 3 . We note that $st_1 = st_2$. We thus have to always check for the possibility of no outward directed arcs in the case of a digraph . In connected graphs this problem naturally does not occur.

2.11.2 Unlabelled Trees.

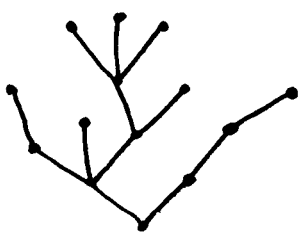
There is as yet no known method of representing uniquely and compactly a (unlabelled) tree . However ρ-trees were represented by the author as a string of z numbers, the ρ-tree having z points . Recapulating from the authors M.Sc. dissertation, [12] , another ordering is defined on all the points of the ρ-tree such that

a) a point with lower generation number precedes a point with a higher one and

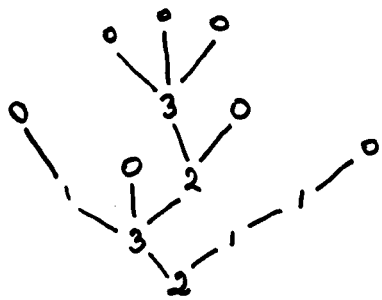
b) within a package , the more +ve of two points has precedence and finally

c) of two packages with the same generation number , the points within the one which has a more +ve package head have precedence over the others.

Now consider each point in turn starting from the root (by means of the above ordering) and write down the number of lines , within the package of which it is the head . This will then represent , by definition , a ρ-tree uniquely and can be translated back into the pictorial representation very simply .



(fig. 2.11.21)



As an example, consider the unlabelled ρ-tree in fig. 2.11.21 , the root point being the lowest point in the configuration. We can within the same configuration insert at each point of the ρ-tree, the number of lines in the package of which the point is the head . This is indicated in fig. 2.11.21 next to the ρ-tree . If we now read the numbers at each node by generations , starting from the root and from left to right , we obtain the following sequence : 2, 3, 1, 1, 0, 2, 1, 0, 3, 0, 0, 0, 0, 0 ;
We can shorten the string, without any ambiguity resulting, by suppressing the last 5 zeros and the string becomes :

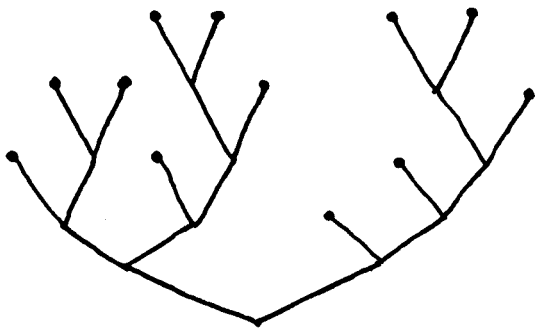
2, 3, 1, 1, 0, 2, 1, 0, 3 ;

Consider an unlabelled binary pendant ρ-tree and its monoid representation . As it is unlabelled the monoid will comprise of opening and closing brackets and some symbol, an asterisk say , to denote an end or pendant point . As an example consider the pendant ρ-tree of fig. 2.11.22 . Suppose we let the integer one denote open brackets and a zero stand for the asterisks . We can now represent the monoid as a string of zeros and ones, having suppressed the closing brackets (as they are redundant). Thus we have a binary representation for pendant ρ-trees.

(((*(**))(*((**)*)))(*(*((**)*))))

1110100 101100 0 10101100 0

= 111010010110001010110 00

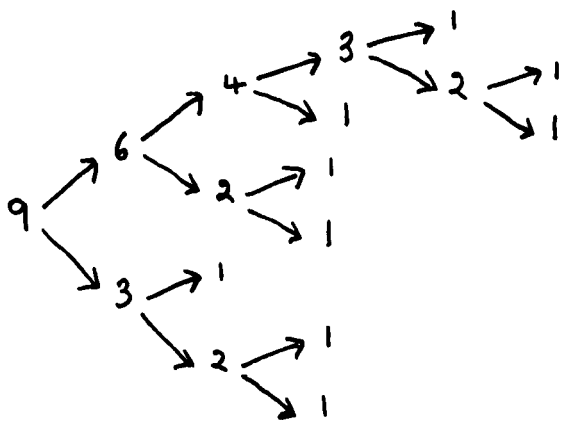


(fig. 2.11.22)

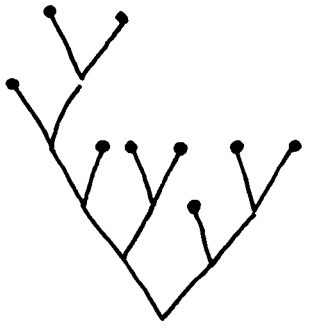
This results in a much bulkier representation than the earlier one (for the same multifurcating ρ-tree) but it can be easily extended to include labelled pendant ρ-trees and hence labelled multifurcating ρ-trees, by simply inserting the label of the point instead of a zero.

It is interesting to note that in the above mentioned binary representation, we have one more zero than ones . Thus if we write down randomly , zeroes and ones in succession (noting only that sequence which starts with a one) and stop as soon as we have more zeroes than ones, we shall have generated a random pendant or multifurcating ρ-tree. However this is not much use as we cannot guarantee the size of the tree, i.e. we cannot use this method as it stands, to generate random ρ-trees of z points.

H.I.Scoins solved this difficulty by developing a method whereby random pendant ρ-trees of z pendants are randomly generated , from which of course multifurcating ρ-trees can be obtained. The method is a partitioning type one. We randomly partition z into two parts z_1 and z_2 . These two parts are further randomly partitioned into four parts and so on . At each partition , we must obtain two nonzero parts. The connection will be seen immediately from the following example. Suppose z = 9 and we partition this into 3 and 6 . Three is further partitioned into 2 and 1, say , and six into 2 and 4 . Diagramatically we may get



This immediately becomes the pendant ρ-tree as in fig. 2.11.23. whose monoidal representation is ((((*(**))*)(**))(*(**))) .
And hence we have obtained a random ρ-tree of 9 points.



(fig. 2.11.23)

2.11.3 ρ- Trees.

It can be seen that any ρ-tree will be uniquely determined , if we specify below(j) and posnbr(j) for all v_j within the ρ-tree . Other pairs of functions which will determine a ρ-tree uniquely are:- below(j) and negnbr(j), below(j) and pos succ(j). The last pair of functions being that pair which was used to represent a ρ-tree in [12].

The only other representation used within this thesis was the pair of functions rd(x) and lu(x). That is a ρ-tree was represented by the values rd(j) and lu(j) for all v_j in the ρ-tree. It is true that rd(j) on its own would have been sufficient to represent a ρ-tree uniquely, but in order to facilitate the searching and manipulation of the ρ-tree , the added function lu(j) was used.

2.1.2 Computer Representation.

In general the graphical representation for any given problem will indicate the relationship between various pieces of information. The information may be associated with the points or it may be the function of pairs of points , in which case it is assigned to the corresponding lines. In both cases the lines of the graph merely indicate the presence or absence of some relationship between pairs of points. A final solution to the problem will consist of some subset of the lines connecting some, if not all, of the points. Thus in manipulating the graph, we should be primarily interested in the nodes and the relationship between them. Within the computer this means being able to locate any particular point, and both quickly and easily finding its immediate neighbour. This leads us to choose the adjacency type representation rather than the incidence type. The word type is deliberately used, the author not only uses the adjacency matrix, but also the derived listed pairs and branches list representation.

2.12.1 Digraphs and Graphs.

We shall only deal with the representation of digraphs and ρ-trees , as these are the pictorial representations of the applications in Chapters 3 to 6. We mentioned previously that a graph could be represented by two lists :- a branches list and a rowstart list . In the applications we regard the graph as a digraph , insomuch as that we note both (v_i, v_j) and (v_j, v_i) if v_i is adjacent to v_j . That is, the array (or list) branches comprises 2 × br elements. The two lists are stored in <u>integer arrays</u> branches[1:2×br] and rowstart[1:z]. In order to make it easier to follow the working of the procedures the author introduced a third array nrinrow [1:z] which indicated the number of points to which each point is adjacent.

It would not be out of place to note that if it was a digraph, which we were actually storing and not a graph, it would still be very easy to search by rows but not by columns. That is ,we could easily find out all the points to which v_j is joined, simply by scanning row j . However, to find those points which join v_j would be a different matter altogether. This would necessitate scanning the complete matrix or list.

2.12.2 ρ - Trees.

If we wish to represent a rooted tree, this can be done as mentioned before , in the array below [1:z], where the tree contains z points. We would store in below[j] the value of that point which is adjacent to v_j and precedes it . We make below [root]:= root. However a ρ-tree requires another array, in order to planarize or order the tree, and this is posnbr[1:z] . These two arrays used together are sufficient for any work to be done on trees . H.I. Scoins in his method of solving the transportation problem uses only below[j] as he does not require his tree to be ordered . However the author, in Chapter 5 , improved upon this by altering the representation of the ρ-tree using the rd, lu arrays in order to facilitate the movement up and down the ρ-tree.

As mentioned before, in order to move up the tree, given the below, posnbr representation , would require repeated scans of the list in order to climb up one generation at a time. This is very time consuming. However if we use the rd, lu representation there is no trouble because we can find the upper left of any point (by lu[j]) and then move across the package by successively using rd [j].

2.2 Manipulation within the Computer.

2.21 Graphs.

The detailed manipulation required for each application is described in the appropriate chapters. However , there are a few standard techniques and manipulations common to most of them. One is the input of data. The best method of presenting the data had to be found and then fed into the program . Assuming that we use the branches list representation it was found that the data punched in the following form was most appropriate :-

$$
\begin{array}{lll}
z\ ;\ br\ ; & & \\
r_1\ ; & b_{11}\ ; b_{12}\ ; b_{13}\ ; \ \ldots\ ; b_{1r_1}\ ; & \\
r_2\ ; & b_{21}\ ; b_{22}\ ; b_{23}\ ; \ \ldots\ ; b_{2r_2}\ ; & \\
\cdots & \cdots & \cdots \\
r_z\ ; & b_{z1}\ ; b_{z2}\ ; b_{z3}\ ; \ \ldots\ ; b_{zr_z}\ ; &
\end{array}
$$

The column r_i indicates the number of elements in row[i], followed by the labels of the r_i adjacent points. Output was in a similar manner . There is one drawback in this method of representing the data for a given problem, ready for input into the machine . It is necessary to know the number of lines in the graph and also the degree of every point.

As stated in Chapter 1, the main building block , common to all the applications in this thesis, is the spanning tree (in one form or another). An algorithm for obtaining a spanning tree is as follows :-

Assign one of the points of the (di)graph as root point. This now becomes the below of all points adjacent to it. They in turn become the belows of all points adjacent to them and not encountered before (this is easily verified if we initially make below[j]:= 0 for all j, and then filling them in as each point is encountered. Thus a point which has not been met before will have its below still equal to zero) . At the end the below array will be non-zero,unless a point is not connected by a path to the root point.

Another manipulation encountered in Chpt.4 is the systematic reduction of a graph, by the removal of one point at a time . The problem is not as simple as it looks , as we have to cater for the possibility that the removal of one point results in the previously connected graph becoming disconnected . Within the same chapter , it was required to alter the labelling of a given graph. That is, we are given a labelled graph (in the branches list re-

presentation) and an integer array containing a permutation vector and we wish to obtain the branches list representation of the resulting permutated labelled graph. This and other exercises in graph manipulation are best described in their respective chapters and appendices.

2.22 Trees.

Two types of manipulation occur within this work. One was, given a tree within a computer, to find the most efficient way of obtaining the information attached to any given node or sets of nodes . In this case the interest is in the ability of being able to go up and down the tree. The other type of manipulation was the physical alteraton of the configuration of the tree itself. That is, to alter the relationship between points as denoted by the presence or absence of lines between them. This is usually accomplished by the deletion of the lines of the tree one at a time and other suitable lines added in . As explained in the last subchapter the discussion on the manipulation of graphs and trees is best left to the individual chapters and the accompanying Appendices.

III TOPTREE .

3.1 Introduction.

The Travelling Salesman Problem stated simply is this : given an associated cost graph, find a circuit covering the graph which has its associated cost as a minimum . This circuit , not necessarily unique , will be known as a minimal tour or minimal solution , where a tour or solution to the problem is any circuit which covers the graph. The problem has just been stated in the more general form. Usually the lines obey the triangular inequality law , i.e. no two sides together are smaller than a third , and hence the minimal solution will be a simple circuit . The problem may be considered as a special assignment or even as a transportation problem . However these two approaches are very impractical due to the number of necessary constraints.

In practical terms, the graph may be a road map, obeying the triangular inequality law, where the points represent towns and the lines represent the roads connecting them. The problem may then be defined as that of having to visit each town , keeping the amount of travelling to a minimum (hence the name given to the problem). It follows intuitively, that a salesman would not return to a previously visited town , thus making the tour or circuit simple.

There are two standard methods of finding a minimal solution. One is by Integer Programming [31] and the other by Dynamic Programming [23]. However both methods are very limited by storage space and time . In general , if the number of towns or points is over 13 , neither of the above methods is practicable . In particular , both these methods were used in this work, the programs being written for a KDF9 computer , and the maximum number of towns solvable was 11 .

Thus there is every incentive for finding a method of solving larger problems, and failing this , of obtaining tours whose cost is very close to that of the minimal solution . This is what Toptree does. It is a heuristic method which specifies a small subset of the set of possible solutions , and then searches through this subset in order to find the best possible solution within this subset.

In testing the method on randomly generated data and comparing the Toptree results with that obtained by either Integer or Dynamic Programming, it was found that in over 50 % of the cases , the Toptree solution lay within 2 % of the minimal solution . Another advantage of Toptree is that, while searching through the subset, it notes and outputs the best solution found so far. Thus the process may be terminated at any point.

The method suggested itself when a comparison was made between a route for the minimal solution of the problem and a minimal spanning tree for that same graph. It was noticed that in almost all the cases examined, the minimal solution had at least half the lines of a minimal spanning tree within its route. Hence the idea developed of manipulating the tree into a chain and consequently obtaining a simple circuit (by joining the end points of the chain).

The next few pages are devoted to an explanation of the derivation of a minimal spanning tree, followed by a description of how it can be reduced to a chain and hence a tour. This is followed by a brief discussion on the preparation of data, and how Toptree solutions compare with the minimal solutions. We shall, within this and subsequent chapters, assume a graph (or digraph) to have z points and br lines (or arcs).

3.2 The Minimal Spanning Tree (Mintree).

Given a connected graph with an associated cost matrix , there exists a partial connected graph covering the original graph whose cost is a minimum. With a little thought it can be seen that this subgraph is in fact a tree, hence its name: the minimal spanning tree or mintree for short. Kruskal , in his paper [27], proved this and provided a single algorithm to achieve it . Loberman and Weinberger elaborated further (in [30]) with flow charts and so on , two algorithms based uppn Kruskals ideas. The author translated both ideas into Algol and finding that one of them was quicker and less space consuming , improved upon it and published it as a procedure in [32]. Here again the method of obtaining mintree was itself an exercise in tree manipulation.

3.21 Description of Mintree.

The procedure Mintree obtains the minimal spanning tree and describes it in the below representation (i.e. as a r-tree) . This in no way restricts or hinders the obtaining of the spanning tree. The final result is that we have the minimal spanning tree rooted at some point. The technique is to search through the graph repeatedly, looking for suitable members of this tree. The method is as follows:-

1a) Search for the least cost edge of the graph, G . This now forms a r-subtree J_1 ,say.

1b) Search for the next smallest cost edge (after having excluded the edge of 1a) from further consideration: in the computer this was done by putting its cost equal to $\infty 10$) .

1c) Test if this edge (from 1b) is adjacent to J_1 .
if it is: attach it to J_1
if not : this edge forms a new r-subtree J_2 .

2) Exclude the last edge chosen, from further consideration (i.e. $:= \infty 10$)

3) Search for the next smallest cost edge and test for one of the four following cases :

a) It forms a circuit in one of the r-subtrees. Do nothing.

b) It is adjacent to a line in only <u>one</u> of the r-subtrees. Attach this edge tothe r-subtree.

c) It is adjacent to a line in two r-subtrees. Use this edge to join the two r-subtrees into one.

d) It is adjacent to no r-subtree. The edge then forms a new r-subtree.

4) Go back to 2).

5) The loop 2), 3), 4) is repeated until z-1 edges have been picked , resulting in one connected minimal spanning tree (mintree).

<u>Theorem 3.21.</u> If the lines of G have distinct costs then mintree is the minimal spanning tree of G .

<u>Proof (due to Kruskal):-</u> Let the lines of mintree be called $l_1, l_2, \ldots, l_{z-1}$ in the order they were chosen. From the hypothesis that the lines of G have distinct costs, it is easily seen that the construction of mintree proceeds in a unique manner.

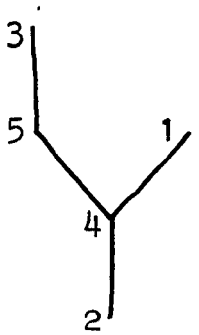
Suppose that mintree is not a minimal spanning tree and let l_i be the first line from mintree which differs from that of a minimal spanning tree , T^*. Then $l_1, l_2, \ldots, l_{i-1}$ are in T^*. $T^* \cup l_i$ must have exactly one loop which contains l_i. This loop must also contain some line e which is not in mintree. Then $T^* \cup l_i - e$ is a tree. But according to the construction cost(e) > cost(l_i). Therefore the cost of $T^* \cup l_i - e$ is less than that of T^*. This contradicts the definition of T and hence indirectly proves that mintree $\equiv T^*$. (If the costs of the lines are not distinct, mintree would not be unique: it will be one of the possibly many spanning trees of G .)

3.3 Theoretical Description of Toptree.

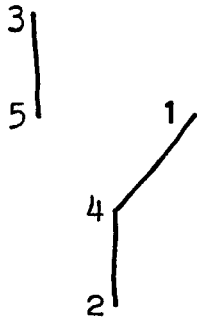
Having produced a mintree, it is now required to manipulate it into a tour. The main criteria is to keep as many lines of mintree as possible in the resulting tour. Thus the assumption is made that the points which have only two lines incident to them within mintree , stand an excellent chance of having the same two lines incident to them within a minimal solution. The problem arises of what action to take, with those points incident to more than two lines. Toptree solves this by reducing these multi-membered stars, by the deletion of their members, one by one, until all points are incident to at most two lines. At each deletion, Toptree adds a new line elsewhere, so as to keep what was mintree connected. The final tree is a chain, whose end points are then joined to form a simple circuit or tour. Here is the method in more detail.

Let T_m denote a mintree, obtained for a given cost associated graph. T_m will consist of a connected set of stars , some of which will contain more than two lines. Each multi-membered star is examined in turn and is reduced by deleting its members one by one, to contain only two lines. When a line has been deleted from a star, T_m will become disconnected. The resulting two subtrees of T_m will

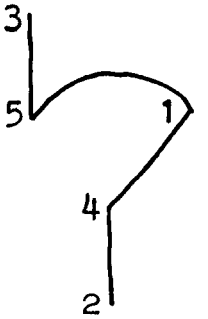
be rejoined by means of a line from the cotree. A proviso is made, to be explained a few paragraphs later, that this new line will be such that the number of members in any star of the reconstructed T_m will not be increased. This can only be accomplished if the line were to join the end point of one subtree to an end point of the other . The final choice being that line which has the least cost. This process is carried out on all the stars until T_m results in a chain . As an example , consider the following tree in fig. 3.0 .



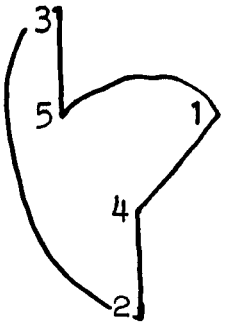
(fig. 3.0)



(fig. 3.1)



(fig.3.2)



(fig. 3.3)

Suppose fig. 3.0 represents a mintree for a given 5 point graph. There is a multi-membered star with its head at v_4 . Thus one of (4-5), (4-2), or (4-1) must be removed , so that v_4 will have only two lines incident to it. Suppose that (4-5) is removed. This results in the

two subtrees $\{3, 5\}$ and $\{2,4,1\}$ of fig. 3.1. They are rejoined by the smallest cost associated line between end points of the two subtrees. The full choice of lines is (3-1), (3-2), (5-1) or (5-2). Suppose (5-1) is chosen. The resulting tree is shown in fig. 3.2. The tree is a chain and so the two end points are joined together, by means of (2-3), to form a tour as in fig. 3.3. Another two tours would have resulted if we had deleted (4-1) or (4-2), instead of (4-5), and repeated the above process.

There is one point to note. Unless mintree is a chain, whereby only one tour is obtained, there will be many ways of reducing T_m to a chain. This will result in a subset of tours, S_m. Thus we have to find a method of ordering the deletions within a star, and between the stars themselves, so as to obtain, for a given mintree, the maximum number of tours possible.

Before we go on to discuss the ordering of the stars and their members, let us briefly study the position when T_m has just been disconnected. We asserted that the join of the two subtrees has to be accomplished by means of a line connecting their end points. Consider the situation where this proviso was relaxed. While reducing a star,

s_1 say, we would be able to increase the members of another star, s_2 say. When we come to reduce s_1, after having reduced s_1 to a duo-membered star, there would be nothing to stop us from increasing s_1 again. Hence this could be repeated endlessly, without either reducing simultaneously to duo-membered stars.

3.31 The Ordering of the Stars.

Consider the mintree of fig. 3.0. It had one multi-membered star at v_4 which contained three lines. We showed that there are three different ways of reducing the star to one which contains only two lines. This in fact was done by deleting each of the three members of the star. Suppose that there had been four lines in the star at v_4. Then there would have been an initial choice of four lines to delete, resulting in a three-membered star at v_4. Thus there will be altogether 4×3 possible ways of reducing this four-membered star to one which contains only two lines. Using a similar argument, we can see that a star containing n members, can be reduced to a duo-membered star in $\frac{n!}{2}$ different ways.

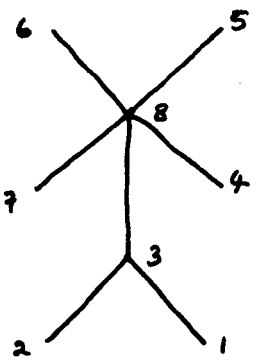
Now if there had been two non-adjacent, (the importance of which will be explained later) multi-membered stars in T_m, containing m, n lines respectively, we can calculate the number of ways of reducing T_m to a chain. Consider the m-membered star. There will be $\frac{m!}{2}$ ways of reducing it to a duo-membered star without affecting the other star in any way. When we come to reduce the n-membered star we find that this can be reduced to a duo-membered star in $\frac{n!}{2}$ different ways. Here there will be

$\frac{m!}{2} \times \frac{n!}{2}$ different ways of reducing T_m to a chain. Similarly if we reduce the n-membered star first, followed by the other , we obtain a further $\frac{m!}{2} \times \frac{n!}{2}$ ways of reducing T_m to a chain. Hence there will be altogether $2 \times (\frac{m!}{2} \times \frac{n!}{2})$ different ways of reducing T_m to a chain.

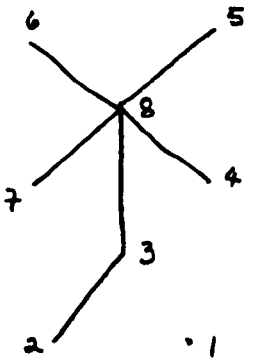
The point is thus illustrated, that not only do we have to take the ordering of the deletions within a package into account , but also the ordering of the packages themselves . For if T_m were to contain three multi-membered stars s_1 , s_2 , s_3 say, there would be no necessity for the subset of tours S_m , obtained by reducing first s_1 , then s_2 and finally s_3 , to be equal to S'_m , the subset obtained by reducing s_2 first, then s_1 and finally s_3 . Suppose that we have a mintree T_m which contains r multi-membered stars. Then there will be r! different ways of ordering these stars . Let us further suppose that each of these r multi-membered stars contains s_i (i = 1, 2, ... , r) lines. Then it follows that the maximum number of different ways of reducing T_m to a chain and hence a tour is given by

$$r! \prod_{i=1}^{r} \left(\frac{s_i!}{2}\right) = \frac{r!}{2^r} \prod_{i=1}^{r} s_i! \qquad \text{---------} \quad (3.3\ A)$$

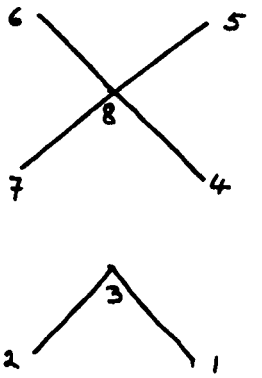
3.31.1 Adjacent Multi-membered Stars.



(fig. 3.4)



(fig. 3.5)



(fig. 3.6)

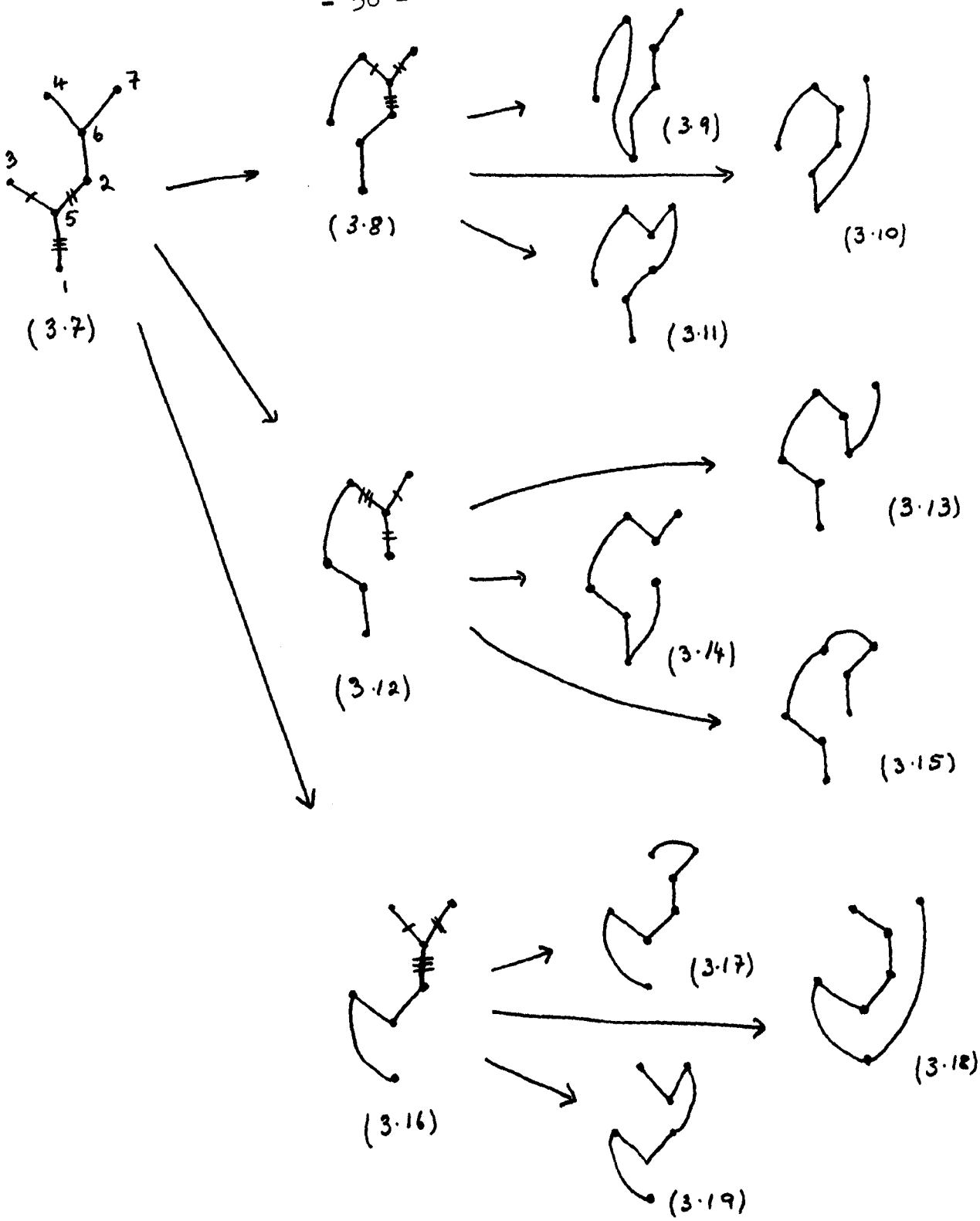
Consider the above mintree in fig. 3.4. T_m contains two adjacent multi-membered stars at v_3 and v_8. Using the theory developed in the last subchapter and (3.3A) we would expect at most $\frac{2}{4} \times 3! \times 5! = 360$ ways of reducing T_m to a chain. However, there are far fewer tours in S_m due to the adjacency of the two multi-membered stars. Consider the reduction of the star at v_8. We can delete either (3-1) or (3-2), and obtain two subtrees similar to fig. 3.5. The resulting reconnected tree will contain a five-membered star at v_8. Hence either deletion leads to $\frac{5!}{2}$ possibly different chains. However, the third possible deletion at v_3 is (3-8), the connecting line between the two

multi-membered stars. When this is deleted, we not only reduce the star at v_3 but also the one at v_8. The two disconnected subtrees are depicted in fig.3.6. When rejoined, the reconstructed tree now contains a four-membered star at v_8. Hence this deletion of (3-8) at v_3 leads to only $\frac{4!}{2}$ further chains. Thus deleting the lines at v_3 first, we obtain altogether $\frac{5!}{2} + \frac{5!}{2} + \frac{4!}{2} = 132$ possibly different chains. (This is to be compared with $3 \times \frac{5!}{2} = 180$ possibly different chains if the stars had not been adjacent). Reducing the star at v_8 first, followed by that at v_3, we obtain by similar argument, 108 possibly different chains. The grand total of possibly different chains in this case is thus 240 .

3.4 Practical Description of Toptree.

Within the context of this thesis, it was necessary to both root and order T_m, the mintree, so that it could be represented and manipulated within a computer. Hence stars become packages (with the addition of the line joining the package head to the point below it) and chains become ρ-chains . A procedure called process was written (see App.1.1), which, when called, yielded all the points of a ρ-tree, one by one . This was used to order the packages of T_m so that they could be sequentially reduced. The method can be better illustrated by discussing a small example.

Consider the mintree of seven points in fig. 3.7. There are two multi-membered packages within T_m, one at v_5 and the other at v_6. Let T_m be rooted at v_1 and ordered as in the figure . Repeated application of the procedure process will yield the points of the ρ-tree in the following sequence:- 1, 5, 3, 2, 6, 4, 7 . The first multi-membered package to be considered, will thus be the one at v_5 . In general, the ordering within a package of which v_p is the head , will be firstly $lu(v_p)$, followed by its positive neighbours, and finally the below of the package head. Thus in the package at v_5 the ordering of the deletions will be (5-3),(5-2) and (5-1): on the figure


4
7
6
3
2
5
1
(3·7)
(3·8)
(3·9)
(3·10)
(3·11)
(3·12)
(3·13)
(3·14)
(3·15)
(3·16)
(3·17)
(3·18)
(3·19)

this is denoted by single, double and treble strokes through the respective lines . Thus (5-3) is the first line to be deleted, resulting in the new ρ-tree of fig. 3.8 , where the line (3-4) has been inserted. We regard this ρ-tree afresh and repeat the above process. That is we sequentially order the points of the r-tree , find the first multi-membered package, and this will be at v_6 . We again order the lines within the package and delete the first which will be at (6-4) . We obtain the ρ-tree of fig. 3.9 . We again repeat the process of ordering the packages within this new ρ-tree and find that it has no multi-membered package. We deduce that it is a ρ-chain , connect its end points together and obtain a tour.

Having obtained a ρ-chain, we go back one step (to the previous ρ-tree in the sequence, fig.3.8) , and re-examine that ρ-tree. We choose the next line in that ρ-tree to delete and in this case it will be (6-7). As a result we obtain the ρ-chain of fig. 3.10. We return, again to the ρ-tree if fig. 3.8 and delete the next and last line, (6-2), to obtain the ρ-chain of fig. 3.11 . When we return again to the ρ-tree in fig. 3.8 , we find that there are no more deletions to make , and hence we go back another step to the ρ-tree in fig. 3.7. As mentioned just now, we search for the next line to delete and in the case of fig. 3.7 it will be (5-2) .

This results in the ρ-tree of fig. 3.12. The same procedure is applied to fig. 3.12 as was to fig. 3.8 and we obtain three further ρ-chains, those of figs. 3.13, 3.14, and 3.15 . However, it is worth noting that the ordering within the package at v_6, is different. Previously (6-4) was the first to be deleted , in this case it is (6-7). Having obtained the ρ-chain of fig. 3.15 we return to fig. 3.12 and again back to fig. 3.7.

This time we delete the third and last line (5-1). We obtain the ρ-tree in fig. 3.16, and from it the three ρ-chains of figs. 3.17 , 3.18 , and 3.19. When we finally return to fig. 3.7 we notice that there are no more lines to delete and so the algorithm stops.

We have obtained nine, possibly different, ρ-chains and hence tours from the T_m of fig. 3.7 . This fits in with the theory of the previous subchapter where for one ordering of the non-adjacent packages we calculated a maximum of $\frac{1}{2}\left(\frac{3!\times 3!}{2}\right) = \frac{36}{4} = 9$ possible ρ-chains If we assigned a different point of T_m as the root (one of v_4 or v_7), we could obtain a firther nine ρ-chains, but these need not necessarily be the same nor indeed different, from the ones just obtained.

While going through this last example, it will have been noticed that the deletion of a line , causes a new ρ-tree to be constructed. All these ρ-trees have to be either stored or easily reconstructed . For when we have exhausted all possible deletions from one ρ-tree, (as in fig. 3.11) we have to go back one step to the previous ρ-tree (as from fig. 3.8 to fig. 3.7). The storage of the ρ-trees is solved by keeping them in a dynamic nesting type stack. That is, we have a stack of ρ-trees where the bottom one is the first to have been examined (in the previous example it corresponds to fig. 3.7) and one of whose lines has been deleted to yield the second from bottom ρ-tree (this corresponds to either fig. 3.8, 3.12 or 3.16) and so on. Thus the ρ-tree at the top of the stack, the current ρ-tree, has been formed by the deletion of one of the lines of the ρ-tree below it.

The current ρ-tree, T_k , is examined to see if there are any multi-membered packages . If it has, we delete the first line in the first package. This will form a new current ρ-tree, which is placed on top of the stack , pushing all the previous ones down one (like the bullets in a cartridge holder). If the current ρ-tree has no multi-membered packages, we form the tour, note both it and its total cost, and then reject the current ρ-tree.

Thus the stack 'pops up' one position and the ρ-tree below becomes the current ρ-tree. While examining T_ℓ, we may note that it has a multi-membered package and yet not have a next line to delete, i.e. we have exhausted all possible lines to delete within T_ℓ (this will correspond to the ρ-tree in fig. 3.8 after having returned to it from the ρ-tree in fig. 3.11). We reject this ρ-tree and 'pop up' the next ρ-tree in the stack and continue the analysis. This corresponds to the mechanism of finding a ρ-chain and rejecting it.

Thus there is a continuous stack of ρ-trees, the top one of which is being examined. The stack increases, if the current ρ-tree yields another by the reduction of one of its multi-membered packages , or decreases if the current ρ-tree is either a ρ-chain or an exhausted ρ-tree. In the example of fig. 3.7 , the first ρ-tree in the stack will be that of fig. 3.7 . The second ρ-tree will be either of figs. 3.8 , 3.12 , or 3.16. The third ρ-tree will be one of the nine chains. The process terminates when the bottom or first tree has been exhausted of all possible deletions.

3.5 The Number of Tours in the Subset S .

As mentioned previously, if T_m has r stars containing s_i (i = 1, 2, ... , r) members, the number of different tours for one ordering of the packages is $\frac{1}{2^r} \prod_{i=1}^{r} s_i!$. (We see that the number of tours in the subset, S_m, depends not only on the number of multi-membered stars but also on their 'density'.) As we shall be concentrating on the ρ-tree representation of T_m, the above formula becomes :

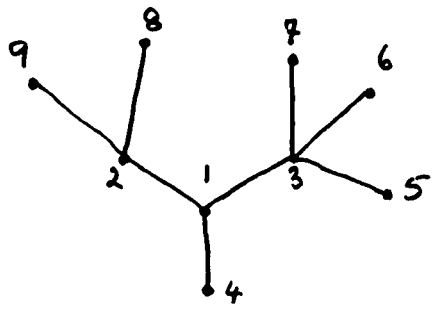
$$n(S_m) = \frac{1}{2^r} \prod_{i=1}^{r} (p_i + 1) \quad \text{-------(3.4 A)}$$

where $n(S_m)$ is the number of possibly different tours in the subset S_m and p_i is the number of members in package (i). We see that p_i is one less than the value of the corresponding s_i. A simple table will illustrate the number of different ways of reducing a package to one containing one line only. If $c(p_n)$ stands for the number of chains (containing only 2 lines) derivable from a package containing n lines, we have $c(p_n) = \frac{(p_n + 1)}{2}$

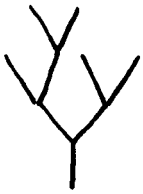
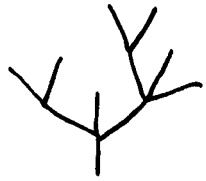
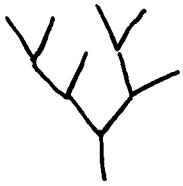
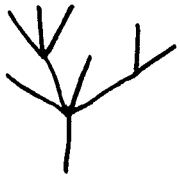
n	1	2	3	4	5	6
type of package						
c(p)	1	3	12	60	360	2520

As mentioned in Ch. 3.3, the formula (3.3A) and hence (3.4A), will not hold for adjacent packages. Thus a configuration of adjacent packages has to be tackled by hand, in order to evaluate its $n(S_m)$. Consider the complex configuration of fig. 3.521.

There are three adjacent packages containing two, two and three lines respectively. If they were non-adjacent we would expect, for one ordering of the packages, $\frac{3}{2}! \times \frac{4}{2}! \times \frac{3}{2}! = 108$ possible reductions. However consider what actually happens in more detail. First let us delete (1-2). This results in a configuration with only one package of three lines : giving twelve ρ-chains. If we delete (1-3), we obtain a configuration with two non-adjacent multi-membered packages, each containing two lines: resulting in nine further ρ-chains. Finally, when we delete (1-4), we obtain a configuration with a 2- and a 3-membered package: yielding 36 ρ-chains. Hence the total number of ρ-chains possible for this configuration is 12 + 9 + 36 = 57, which is quite a reduction from 108 . Using similar techniques we obtain the results in the following table (3.522).



(fig. 3.521)

package configuration	no. of chains	package configuration	no. of chains	package configuration	no. of chains
	7		27		57
	17		15		297
	37		39		117
	91		46		108
	288		537		186

(table 3.522)

3.6 Obtaining the Minimal Solution.

With the published problems, [page 73] there was no trouble in obtaining their minimal solutions as they were usually included, speculative or otherwise. However, it was necessary to obtain the minimal solution, as well as the Toptree one, for the random data problems. Thus a dynamic programming procedure was written to solve the problem . It was based on the paper by Held and Karp [23] . Due to the limitations of store and time, the program, an Algol procedure, could solve problems of up to twelve points (or towns) only. An indication of time taken is as follows:

z	6	7	8	9	10	11	12
time (min.)	15s.	25s.	65s.	5	27	110*	7hrs.*

*estimated.

It was found, that for large z , i.e. >9, this was still taking too long , as there were many sets of data to analyse, so an Integer Programming[1] program based on Gomorys method, was used . This sometimes failed

[1] The program was in KDF9 User Code (i.e. machine code) hence its speed, and was written by J.S. Clowes of this laboratory.

to reach a solution in a reasonably short time , and so the program was terminated and solved by Dynamic Programing.

Even so, the number of problems actually tackled, was limited by the speed of the above two programs. Toptree, on average, took from two to twenty seconds for a graph of up to twelve points.

3.7 Data Preparation and a Short Note.

At first, the data was to be generated by using a random number generator to fill in a symmetric cost matrix . This was rejected as it , the cost matrix, would have no connection with reality : an obvious consequence would have been that the shortest path between any two points, would not always have been the line between them. Hence a slightly longer and more complicated method was used, which by construction, assured the 'planarity' of the cost matrix and hence the corresponding graph. The method was to generate z pairs of (x,y) random coordinates within a fixed area. The distances between the z points were computed and inserted into the (z × z) cost matrix.

Because the method of data preparation placed the z points, randomly within a given area, the author was able to test Hammersley and Handscombs formula of

$$l_A = k_A \times (z \times A)^{1/2} \quad \text{--------(3.7 A)}$$

where, l_A = length or cost of minimal tour,

z = number of points in graph,

A = the area,

for the Travelling Salesman Problem [22].

Their argument for obtaining the above formula (3.7A) was as follows. The length of the tour depends upon the area A and upon the density of towns in this area $\left(\frac{z}{A}\right)$. Let $l_A \propto A^{\alpha}\left(\frac{z}{A}\right)^{\beta}$. Dimensional analysis shows that $\alpha - \beta = \frac{1}{2}$. Now if we multiply the area by c and keep the density constant, we shall be multiplying the tour length by c . Hence $\alpha = 1$ and we obtain

$$l_A = k_A \times (z \times A)^{1/2} .$$

The formula was tested on 463 set of data with z varying from five to eleven, and it was found that $k_A = 0.9$ with standard deviation of 0.139. This seemed to indicate that there might be some validity to the argument and hence (3.7A).

(Assuming a similar argument, we may obtain the formula

$$l_M = k_M \times (z \times A) \quad \text{---------(3.7 B)}$$

where, l_M = length of mintree and

k_M = k for mintree.

It was found that $k_M = 0.635$ with a better standard of deviation of 0.109. It was decided, as a natural extension, to write

$$l_T = k_T \times (z \times A) \quad \text{--------(3.7 C)}$$

for Toptree solutions, and it was found that $k_T = 0.915$ with a standard deviation of 0.140.)

3.8 Analysis of Results.

463 sets of random data were prepared with the number of towns varying from 5 to 11. The first table, table 3.81 , shows how the Toptree solutions were distributed as regards their approximation to the minimal solution. For example, of the 71 problems with 8 points in the graph, 42 Toptree solutions had equal lengths with the minimal solution, five were between 0 and 0.5 % of the minimal solutions and so on. If we look at the last column , this indicates the analysis on all the random data together. We see that the majority of Toptree solutions lie between 0 and 5 % of the minimal. In fact a better picture is given by the table of (3.82), the frequency distribution table. This indicates in percent of the total number of problems solved, the distribution of the percentage difference between the Toptree and minimal solutions, e.g. for the 75 problems containing 9 points , 66 % Toptree solutions were minimal, 73 % were within 2 % of the minimal, 85 % were within 5 % of the minimal and all lay within 10 % of the minimal. All the problems taken together are layed out on the last line.

The Toptree method was also applied to published problems. A table follows (table 3.83), and in the time limit column it will be noted that both Dantzigs

(42 × 42) and the Spic (34 × 34) problems were prematurely terminated.

The Spic problem has not been published before, so the cost matrix is given on the next page. Proctor and Gamble initiated an advertising campaign, in order to sell a soap product of that name. They asked the public to draw the shortest route through 34 towns in England. A solution which shared equal first prize gave the tour length as 1240 miles . The firm in question refuse to disclose any information, hence the tentative assumption of this being the minimal tour length.

Upon studying the Barachet problem, it was noted that had a different root point been chosen, Toptree would have given the minimal solution. This strengthens the suggestions that if the problem is reasonably small, or has an overall simple configuration (and hence yields a small subset of Toptree tours) a greater chance of obtaining the minimal solution is achieved by varying the root point to different end points of T_m .

The lower triangle of the Spic cost matrix is given split into two parts due to the size of the page. The bottom triangle is to be attached to the right of the incomplete triangle above it. The size of the cost matrix is (34 × 34) .

```
 70;
165;105;
120; 85;185;
 25; 61;165; 91;

 30;100;170;150; 45;
 70;145;215;170;105; 50;
 55; 20;120; 90; 45; 80;130;
 80; 40; 85;125; 80; 95;145; 40;
100; 50;155; 35; 70;115;155; 55; 95;


150;125; 50;210;155;160;215;130; 85;190;
130; 55;150; 65; 85;145;185; 70; 95; 30;180;
 35;100;195;135; 50; 40; 45; 75;115;105;185;140;
 50; 40;115;115; 45; 70;115; 25; 30; 80;110; 95; 85;
160;100; 40;160;175;175;225;110; 80;125; 90;110;200;110;

 90; 85;115;165;100; 85;145; 75; 50;130; 85;140;115; 50;120;
 50;110;215;120; 60; 55; 55; 90;130;105;205;130; 20;100;200;135;
 20; 90;180;110; 40; 35; 65; 70;110; 80;180;110; 25; 70;170;105; 30;
140; 85; 25;160;135;155;205; 95; 60;130;155;120;170; 90; 35; 85;190;
 90;160;225;210;120; 60; 60;145;140;180;190;200; 90;120;220;105;110;


 20; 50;145;110; 20; 50;100; 30; 60; 80;140; 95; 55; 30;140; 80; 65;
 75; 50; 95;140; 70; 80;130; 50; 15;100; 85;105;100; 25; 95; 35;125;
 50; 60;170; 70; 25; 80;105; 50; 90; 50;175; 80; 60; 75;150;125; 55;
 40; 80;155;155; 65; 35; 85; 65; 70;115;130;125; 65; 40;150; 50; 80;
 70; 90;200; 75; 50; 85; 90; 80;115; 70;205;100; 55;100;180;155; 40;

115;115;215; 55; 90;135;135;105;145; 75;240;105;110;140;200;190; 80;
115; 75; 60;160;115;120;165; 75; 35;125; 50;130;145; 65; 75; 45;165;
115; 45; 95;110;100;145;190; 60; 65; 75;125; 55;140; 80; 55;115;155;
115; 45; 65;125;120;125;175; 60; 35; 90; 85; 90;140; 55; 55; 90;150;
 65; 25;130; 75; 35; 80;130; 20; 75; 40;150; 50; 90; 45;125;100; 90;
```

80; 75;180; 40; 55;110;130; 65;125; 40;205; 65; 80; 95;165;155; 80;
90; 25;130; 60; 60;115;165; 45; 65; 25;150; 25;115; 65;100;110;110;
150;125; 75;210;150;140;190;125; 85;180; 25;180;180;100;100; 75;200;
230;200;130;285;230;230;275;200;155;250; 80;250;255;180;165;150;275;

160;
95;190;

40;120;110;
90; 70;125; 60;
45;145;140; 40;100;
60;125; 75; 40; 55; 85;
55;175;150; 70;130; 30;105;

110;200;195;105;155; 65;160; 50;
130; 40;150; 95; 40;125; 85;155;185;
125; 70;200; 95; 80;105;120;135;150; 85;
125; 40;175; 90; 50;105;100;135;165; 50; 35;
65;110;145; 40; 70; 40; 80; 70; 90;100; 65; 65;

75;165;170; 65;120; 30;115; 40; 40;145;110;125; 50;
90;100;175; 65; 75; 55; 95; 85;100;100; 50; 65; 25; 55;
170; 65;180;130; 80;175;120;200;235; 50;140; 95;145;205;150;
250;130;250;210;155;255;200;280;310;125;200;160;225;280;225; 80;

→→→

Number of problems with this percentage difference between the Tsoptree + Minimal solutions. (/ ≡ None).

z - number of points	5	6	7	8	9	10	11	ALL
N -number of problems	60	60	60	71	75	57	80	463
0	44	38	28	42	48	23	37	260
0 - 0.5	/	/	/	/	/	/	/	/
0.5 - 1.0	/	1	2	5	2	5	3	18
1.0 - 1.5	2	1	3	3	1	2	5	17
1.5 - 2.0	4	2	2	3	4	3	4	22
2.0 - 2.5	1	2	/	3	2	3	4	15
2.5 - 3.0	2	/	2	2	3	2	5	16
3.0 - 3.5	/	2	1	3	2	2	2	12
3.5 - 4.0	1	/	/	/	1	1	3	6
4.0 - 4.5	/	3	2	2	/	1	1	9
4.5 - 5.0	1	3	4	1	1	3	2	15
5.0 - 6.0	/	/	3	1	/	1	2	7
6.0 - 7.0	2	4	6	/	6	3	2	23
7.0 - 8.0	/	2	5	1	3	5	3	19
8.0 - 9.0	1	1	/	1	/	1	2	6
9.0 -10.0	1	1	2	2	2	1	1	10
10.0-11.0	/	/	/	1	/	1	1	3
11.0-12.0	1	/	/	/	/	/	2	3
12.0-13.0	/	/	/	/	/	/	1	1
13.0-14.0	/	/	/	/	/	/	/	/
14.0-15.0	/	/	/	1	/	/	/	1
15.0- 100	/	/	/	/	/	/	/	/

(table 3.81)

z	N	% difference between Toptree and Minimal soln. 0	1	2	3	4	5	6	7	8	9	10	15
5	60	73	73	83	88	90	92	92	95	95	97	98	100
6	60	63	65	70	73	77	87	87	93	97	98	100	
7	60	47	50	58	62	63	73	78	88	97	97	100	
8	71	59	66	75	82	86	90	92	92	93	94	97	100
9	75	66	67	73	80	84	85	85	93	97	97	100	
10	57	40	49	58	67	72	79	81	86	95	96	98	100
11	80	46	50	61	73	79	83	85	88	91	94	95	100
ALL	463	56	60	68	75	79	84	86	91	95	96	98	100

(table 3.82)

name of Problem	z	Solution distances: Minimal	Toptree	% diff.	Time	Mintree
[19] Dantzig	42	699	729	4.4	2 Hrs.*	591
[23] Held/Karp	25	1711	1783	4.2	3 min.	1240
[17] Croes	20	246	260	5.8	3 min.	154
Spic	34	1240	1280	3.3	90 min.*	1040
[15a] Barachet	10	378	381	0.8	19 sec.	201
Austrian	12	1745	1859	6.5	40 sec.	1284

* prematurely thrown off.

(table 3.83)

3.9 Conclusion.

It was found that Toptree, as programmed in Algol for the KDF9 and using the tree techniques as developed within this thesis, was a highly efficient method for finding a tour, which was reasonably close to the minimal solution. On top of that was the added advantage of terminating the process when a solution which satisfied certain prior conditions had been found (e.g. tour length to be less than a certain quantity, or after a certain time had elapsed). It seems pure coincidence as to whether a minimal solution exists in S_m, the subset of Toptree tours. It was also noted that the lengths of the tours within the subset were not extravagantly large compared to the minimal or best in the subset. In particular it was noted that Toptree did very well on the large, published problems.

In Chpt. 3.7 Hammersley and Handcombs formula was extended to Toptree and Mintree solutions but no conclusion was arrived at by the author. The author had hoped to be able to predict from the values of k_a, k_n, k_T and either Mintree and/or Toptree distances , upper and lower bounds on the minimal solution for any given problem.

IV THE MINIMISATION OF THE BANDWIDTH OF A MATRIX .

4.1 General Discussion.

The problem of minimising the bandwidth of a matrix occurs in the solution of large sets of simultaneous equations . In civil engineering, for example , it is frequently desirable to find the displacements and rotations of the joints of a building frame or bridge trusses under stress. This leads to the solution of large sets of simultaneous equations : $A\,\underline{x} = \underline{b}$ ------ (1) , which expresses the joint equilibrium equations relating the joint displacements to known applied loads . There may be anything up to 50 joints in a frame , leading to 150 or 300 variables, x_i . Each joint is usually connected to perhaps 4 or 8 others and hence the matrix A will have a large number of zero elements . If one were to calculate the density of a matrix as the number of non-zero elements divided by the total number of elements , it will be seen that A will have density of the order of 5 % . Sparse matrices are not peculiar to civil engineering only . In electrical engineering the model might be a network of again the same number of joints and also again each joint (or terminal) may be connected to only 4 to 8 others.

Thus there occur many problems which lead to the solution of simultaneous equations , where the matrix concerned is highly sparse . If the number of variables, z , is small , the problem can be solved by the conventional elimination or iterative techniques. But when z is large, of the order of 100 or over , the standard methods fail owing to storage limitations. However if A is banded (all the non-zero elements lie close to the main diagonal) , other methods have been developed [40/a,42/a] which require storage of only the elements of the band, making the solution of larger order equations possible. The speed of the solution in these methods also depends upon the bandwidth of the matrix : the smaller the bandwidth, the quicker the solution is found. In Livesley[38], the time taken to solve a set of simultaneous equations varies as the square of the bandwidth. Thus given $A\,\underline{x} = \underline{b}$ to solve, where A is highly sparse, it is highly desirable to manipulate A , by pre- and post-multiplication of a suitable permutation matrix P , into a banded form and if possible into a form where the bandwidth is the minimum possible.

The problem of finding P has been split into two parts. The first, Stage 2, is concerned with obtaining a good approximation , i.e. a permutation which reduces the original bandwidth considerably. The second, consisting

of two further parts , named Stages 1 and 3 , finds a permutation which gives the minimum bandwidth . Stage 1 tries to evaluate the minimum bandwidth , failing this it gives a lower bound for it. This is important to Stage 3, which is an algorithm that finds a minimum bandwidth permutation. The more precise the value derived from Stage 1 , the quicker will a solution to Stage 3 be found.

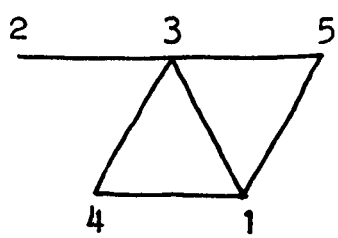
In this chapter, A will be regarded as a $z \times z$ binary symmetric matrix , i.e. the non-zero elements will be replaced by ones . A is usually symmetric due to the nature of the models : actions and reactions are equal and opposite implying symmetry in the resulting derived equations. If π is the permutation $\begin{pmatrix} 1 & 2 & 3 & \cdots\cdots & z \\ p_1 & p_2 & p_3 & \cdots\cdots & p_z \end{pmatrix}$ let us associate with it the $z \times z$ permutation matrix Q where $q_{ij} \in Q$, such that $q_{ij} = 1$, when $j = p_i$ and $q_{ij} = 0$, otherwise.

Let us further shorten $\begin{pmatrix} 1 & 2 & 3 & \cdots\cdots & z \\ p_1 & p_2 & p_3 & \cdots\cdots & p_z \end{pmatrix}$ to $(p_1 \ p_2 \ p_3 \cdots p_z)$. Then the problem of finding a suitable permutation matrix Q , with which to pre- and post-multiply A in order to minimise the bandwidth , can be stated as that of finding the right sequence of integers $(p_1 \ p_2 \ p_3 \cdots\cdots p_z)$ of L(z), the list of all permutations of $(1 \ 2 \ 3 \ \cdots\cdots z)$. Denote the half bandwidth, or hbw , which occurs for a permutation π as $b(\pi)$.

4.2 Relationship between Matrices and Graphs.

Suppose we are given a binary , symmetric $z \times z$ matrix, A . There will exist a corresponding undirected graph of z nodes , labelled from 1 to z , such that if $a_{ij} = 1$, where $a_{ij} \in A$, there is a line joining v_i to v_j (A will be the adjacency matrix representation of the resulting graph G_A) .

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$



matrix A and its graph G_A

(fig. 4.21)

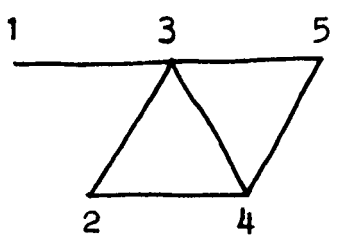
It follows that the problem of minimising b(A) becomes the problem of re-labelling the nodes of G_A, such that the maximum difference between the labels of any two adjacent nodes is a minimum.

Define $dif(i,j)$ as equal to $|i - j|$, if v_i is adjacent to v_j, and equal to zero otherwise. Let $ld(G)$ stand for the maximum or largest difference between the present labels of adjacent points of G, i.e.

$$ld(G) = \text{Max}\left\{ dif(i,j) \right\}, \quad \text{over all } i \text{ and } j .$$

We are interested in the minimum value of $ld(G)$, $= ld_m(G')$ say, for some permutation G' of the labels of G. Thus the problem is to find this quantity $ld_m(G')$ and a labelling, G', that leads to it. In fig. 4.21 we see that $b(A) = 4$ and so is $ld(G_A) = 4$. By definition they will always be the same. Mathematically we define $b(A)$ as equal to $\underset{\forall i}{\text{Min}} \left\{ \frac{l(i)-f(i)-1}{2} \right\}$ where $f(i)$ denotes the position of the first non-zero term in the ith row and $l(i)$ the last. We shall use b when we wish to refer to matrices and ld to graphs. The matrix A can be rearranged by the pre- and post-multiplication of the permutation matrix corresponding to $(4\ 1\ 3\ 2\ 5)$ as in fig. 4.22, to give $ld(G_{A'}) = b(A') = 2$.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$



<u>matrix A'</u> (fig. 4.22) <u>graph $G_{A'}$</u>

4.3 Tree-like Matrices.

The tree is a very special sort of graph and it seems highly unlikely that in an actual problem, the matrix will have representation of this type. However it was thought that if the problem of relabelling the tree was tackled and solved , this might lead to a method of solving the more highly complicated problem of relabelling the graph.

Two different approaches were tried and both were only partially successful , i.e. two algorithms were evolved which labelled trees such that the maximal difference between adjacent labels was minimal , but only if the number of points was less than 50 or so . The author feels that perseverance may have yielded something, but desisted from going on , as a lot of ideas had been thrown up which were of considerable use in the more general problem. For example it was noted that in handworked examples $ld_m(T)$ was found to satify $ld_m(T) = E(T)$, where

$$E(T) = \max_{\text{Subtrees } T'} \left\{ \left[\frac{z' + m' - 2}{m'} \right] \right. \qquad \text{-------(4.3 A)}$$

where for a particular subtree T' of T ,

z' = number of points in T' and

$m' = \text{diam}(T')$.

That is , given a matrix A having a tree-like correspondence T_A , there was always a labelling of T_A which satisfied (4.3 A). That $E(T_A)$ is a lower bound on $ld_m(T_A)$ can be proven as follows :

<u>THEOREM 4.3</u>

$$ld_m(T_A) \geq E(T_A) .$$

<u>Proof :-</u> Let us denote $ld_m(T_A)$ by ld_m . Consider any two points v_i , v_j in T' , one of the subtrees that yields the value $E(T_A)$. Let $l(\mu[v_i, v_j]) = r$, i.e. the path length between v_i and v_j is equal to r. If we try to label T' , we see that $|lab(i) - lab(j)| \leq r \times ld_m$. That is , if we have to label v_i and v_j , the label of v_j cannot have a difference from v_i greater than the number of lines between them times ld_m . If we assume, without loss of generality, that $lab(i) > lab(j)$ then

$$lab(i) \leq lab(j) + r \times ld_m .$$

If we take the worst situation which can occur, i.e. lower bound on the right hand side and upper bound on the other we obtain $z' \leq 1 + r \times ld_m$,

i.e. $$ld_m \geq \frac{z' - 1}{r} .$$

As we are interested in obtaining the lowest bound on ld_m, this will occur when r has greatest value, i.e. when $r = m' = \text{diam}(T')$ or the maxmin path length in T',

thus $$ld_m \geq \frac{z' - 1}{m'}$$

But ld_m is integral, $$\therefore \quad ld_m \geq \left[\frac{z' - 1}{m'} + \frac{m' - 1}{m'} \right]$$

$$\geq \left[\frac{z' + m' - 2}{m'} \right]$$

Another way of putting the above formula is

$$ld_m \geq \text{the least integer} \geq \frac{z' - 1}{m'} .$$

4.4 Stage 1.

The object of this subchapter and associated program was to compute a lower bound for $b(A)$ and if possible , to obtain a value for $b_m(A)$. This will help us decide how long the program for Stage 3 should be run. In the previous subchapter, it was shown that $E(T) \leq ld_m(T)$ and the author feels that the equality sign holds true. There is a similar and more general relationship , which includes that of Thm. 4.41 , for graphs.

Consider a graph G , where G' indicates any connected partial graph of G . Let z' and m' denote , as before , the number of points and the maxmin path length in G' . Within G' there may be many pairs of points which lie on the ends of a maxmin path . There will be a pair v_i , v_j which have the the least number of distinct maxmin paths between them (distinct maxmin paths have no point in common except for v_i and v_j) . Let the least number of distinct maxmin paths be k , we then have $ld_m(G) \geq E(G)$, where

$$E(G) = \min_{\forall G'} \left\{ \left[\frac{z + m + k - 3}{m} \right] \right\} \quad \text{---------(4.4 A)}$$

Theorem 4.4

$$ld_m(G) \geq E(G) .$$

Proof :- We label one end of the maxmin path as 1.

The other end will have a maximum value for its label. If there is only one maxmin path joining the two points , it will be $1 + m' \times ld_m$, where we have shortened $ld_m(G)$ to ld_m . However if there are two distinct maxmin paths between these two end points , the upper bound on the label of one of them is now $ld_m \times m'$. For if we assign to one path the labels $1, ld_m+1, 2\times ld_m+1, 3\times ld_m+1, \ldots , m'\times ld_m+1$, the other cannot have identical labels and hence its labelling must have a line with a difference greater than ld_m . Similarly, if there are k distinct paths between the two points, the maximum value of a label for one of them is

$$m' \times ld_m - (k - 2).$$

$$m\times ld_m-(k-2) \geq z'$$

i.e. $$ld_m \geq \frac{z'+ k - 2}{m'}$$

As in Thm. 4.3 , ld_m is integral and we obtain

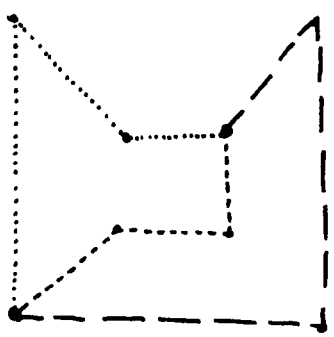
$$ld_m \geq \left[\frac{z'+ m'+ k - 3}{m'}\right]$$

or $ld_m \geq$ the least integer $\geq \dfrac{z'+ k - 2}{m'}$.

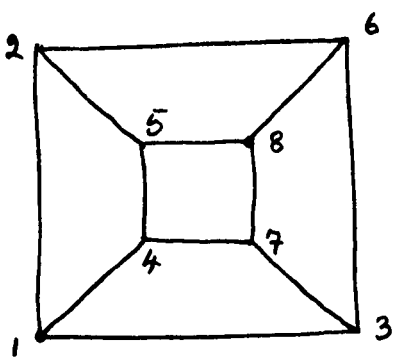
Q.E.D.

If we were willing to investigate all the partial graphs for a given graph, finding for each partial graph its maxmin path length , we would be able to evaluate E(G). For small graphs it is a feasable proposition. However for large z , in the order of 30 and upwards , the method is too time consuming.

The author has met counter examples to the proposition that $ld_m(G) = E(G)$. But the expression does not seem to underestimate $ld_m(G)$ either too much or too often. The author feels that some slight modification to (4.4 A) will lead to the equality sign holding true. As a counter example to the equality sign holding true consider the simple cube as in fig. 4.41 . Considering the graph in toto we obtain $E(G) = 3$. ($z = 8$, $k = 3$ and $m = 3$). Any partial graph will not yield a larger value for $E(G)$ and yet $ld_m(G) = 4$ (this was found by appealing to symmetry and going through the reduced number of possible partial graphs).



(fig. 4.41)



(fig. 4.42)

4.41 Mushrooming r-Trees with Maximum Height.

Given a connected graph , the mushrooming r-tree with maximum height will not be unique . This is easily seen because the r-tree which gives us the maximum height , will have as its highest point that point which itself must be the root of another mushrooming r-tree with exactly the same height.

Mushrooming r-trees, T_m, are built up in the following manner. Consider all points in G adjacent to v_r (the designated root point) . They will become adjacent points to v_r in T_m . Now consider each of these first generation points in turn and connect them to adjacent points within the graph , care being taken to exclude any connections which result in a circuit being formed in T_m . Having exhausted all connections to and from 2nd generation points the process is repeated on 3rd generation points. We notice a mushrooming-out effect from which the process derives its name. The process terminates when no new points can be added to the r-tree, i.e. the r-tree covers the graph. Hence all mushrooming r-trees are also spanning trees for that same graph.

The mushrooming r-tree is constructed for each point of G in turn and the one with maximum height gives us naturally , the mushrooming r-tree with maximum height . This then is a simple method for finding a maxmin path and its distance . We compute the mushrooming r-tree with maximum height which gives us a maxmin path (the path between the highest point of the r-tree and the root) and its length (being the height of the r-tree) .

4.42 Evaluation of E(G).

Thus for a given partial graph we can compute the maxmin path distance. While we were computing this, we would also note the root point of that mushrooming r-tree which yields the maximum height . This root point will form one end point of the maxmin path . The other depends upon which of the highest points of the r-tree yields the least value of k, the number of distinct maxmin paths between the two end points of a maxmin path (see Thm. 4.4) .

We construct all possible connected partial graphs G' of G and their associated mushrooming r-trees of maximum height and compute $E(G')$. We note the best one, i.e. the largest $E(G')$ and this gives us our value for $E(G)$.

For large problems it is both impracticable and impossible to carry out the above operations, due to the length of time spent analysing all possible partial graphs. However , if one could find this important partial graph G'' which yields the largest value for $E(G'')$, then it would be a simple matter . The method described next cannot be proven to find this partial graph . However the author again feels that, if it does not sometimes find this important partial graph , it finds others which give a close value for $E(G)$.

4.43 Finding the Important Partial Graph.

Within a graph define $p(i,j)$ as

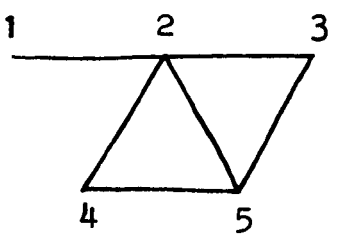
$$p(1,j) = d(j) ,$$

$$p(i,j) = \sum p(i-1,k) , \begin{cases} \text{for } i > 1 \text{ and the sum to be} \\ \text{taken over all } k \text{ where } v_k \text{ is} \\ \text{adjacent to } v_j . \end{cases}$$

Pictorially $p(i,j)$ can be thought of as the number of different paths starting from any point in the graph , having length i and terminating at v_j . Thus as i increases, the values of $p(i,j)$ for varying j give an indication of the relative density of the subgraph around any point j . Consider the binary matrix and its graph of fig. 4.21 and reproduced below :

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

A



G_A

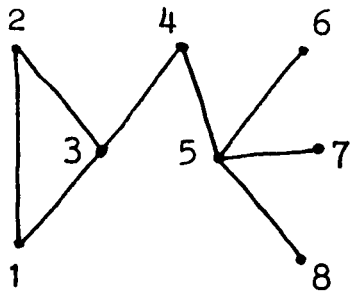
(fig. 4.431)

If $P = p(i,j)$, where P is $m \times z$ (and for fig. 4.431 , $m = 2$, the maxmin path length) we obtain

$$P = \begin{bmatrix} 1 & 4 & 2 & 2 & 3 \\ 4 & 8 & 7 & 7 & 8 \end{bmatrix}$$

Let us examine the mth row of P. The values in this row indicate the number of paths of length m which terminate at each of the z points. Thus a point in the centre of a dense subgraph is going to have a higher value of $p(m,j)$ then, say, one right on the end of a 'long thin arm'. Hence in the graph of fig. 4.431, we have $p(2,2) \geq p(3,j)$ for all j. This bears out what we realize intuitively, that v_1 is much further out and in the centre of a much less dense surrounding then v_2. We say that v_1 is relatively less reachable than any other point and v_2 is one of the most reachable points. It is easy to see that if P was extended by adding more rows then the rth row ($r > m$) would indicate roughly the same reachability for the points.

This then ranks the points of the graph in the order of relative reachability. It may be suggested that there is no need to laboriously compute all the elements of P, but to simply rank the points in order of their degrees i.e. by $d(j)$, and only compute $p(2,j)$ or $p(3,j)$ for those of equal degree. This ranking will lead to a different ordering of the points, as can be seen from the following example.



j	1	2	3	4	5	6	7	8
p(1,j)	2	2	3	2	4	1	1	1
p(2,j)	5	5	6	7	5	4	4	4
p(3,j)	11	11	17	11	19	5	5	5
p(m,j)	28	28	33	36	26	19	19	19

We see that v_4 is of degree 2 and hence initially ranks lower than v_3 and v_5, and yet $p(4,4)$ has the greatest value of all $p(4,j)$.

The author decided that he would use the above concepts in tackling the problem of finding the important partial graph. The graph was reduced by one point at a time while computing $E(G)$ for each successively smaller graph. ($E(G)$ was computed for only the graph in question and not over all its partial graphs. From now on when we discuss the computation of $E(G')$ for any graph G' we shall mean the evaluation of (4.4 A) over the graph G' and not over all its partial graphs.) The point to eliminate was the one with least reachability, i.e. v_j is eliminated if $p(m,j) \leq p(m,i)$, for all i. This gave excellent results as worked out on the computer. Unfortunately for larger values of z ($z \geq 60$), the time taken to do this was still too long and hence the stage was rarely ever completed.

Having decided which point to eliminate (subtract or delete) from the graph , it was usually easy to accomplish the elimination , because the point was usually an end point. But there are examples where the point to be subtracted is a cut point, i.e. upon removing the point, the graph becomes disconnected . This then introduces difficulties of sequentially ordering the labels of points within the components and a little thought has to be put into the programming (see Appendix 3.1) . In neither case will the time taken be affected as there are at most $z - 2$ deletions to make.

4.44 Analysis of Stage 1.

The conclusion is reached that this stage is not as necessary as was first thought. This is because the third Stage , for which this Stage was attempted , was found to be ineffective for large values of z. However it is worth running the program corresponding to this Stage in order to obtain an idea of the relative values of ld given by Stages 2 and 3 . For large values of z where Stage 3 makes no improvement , the results of this Stage are the only yard-stick whereby to judge the value of the labelling given by Stage 3.

4.5 Stage 2.

This stage is concerned with the problem of finding a useful approximation to the minimal permutation. That is , to find a permutation on the original graph which has a bandwidth which we hope is not far from the minimum. Here again the technique used was that of obtaining spanning trees , manipulating them into suitable configurations and then using the final configuration to give us a solution.

In Stage 1 we saw that E(G) is directly related to the maxmin path length. That is if one end of the maxmin path is labelled 1 , we hope to be able to label the graph minimally . Thus the first step to take is to find the end point of a maxmin path as explained in Chpt. 4.441 . We will show now how this helps us.

4.51 Graphical Model.

Suppose we have a graph and let us imagine that we have a string model corresponding to the graph , where each piece of string , which joins any pairs of adjacent points , is of equal length . Further imagine a small weight to be attached to each point (or knot). If we now hold the string model up by one of its points , the rest will hang down and each point will lie in a well defined layer. The first or highest layer would consist of those adjacent to the point being held , the second layer would comprise of those points adjacent to the first layer and so on . Suppose we label the point being held as one . We now label the points in the second layer : two, three and so on, till all the second layer points have been labelled. We turn our attention to the next layer, the third, and continue the process on succeeding layers till all the points have been labelled . Thus all points in the preceding (r-1) layers will have been already labelled when we start labelling the points in the rth layer. If we know the number of points in each layer, we can calculate the maximum possible value of ld(G) resulting from this method of labelling G . The worst possible difference will occur in the two adjacent layers which together have most points between them.

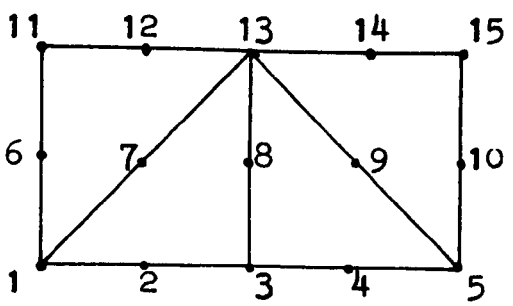
Let us define this quantity as $ld_{max}(G)$ and it will be equal to $\underset{1 \leq j \leq m}{Max} \{ \text{sum layer}[j] + \text{sum layer}[j+1] - 1 \}$,

where layer[j] = value of the layer within which v_j lies and layer[v_r] = 0 , where v_r is the label of the point being held,

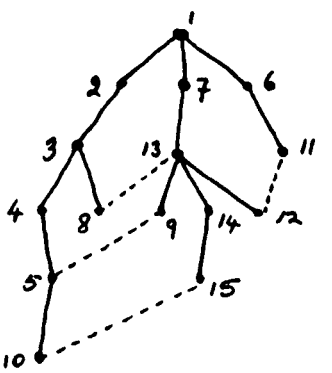
and sum layer[j] = $\sum_{i=1}^{z} \delta_{j,\text{layer}[i]}$

where $\delta_{s,t}$ is defined as the Kroenecker delta.

sum layer[j] , in other words , is equal to the number of points in layer[j] . This can be best illustrated by a simple example , as in fig. 4.511 .



(fig. 4.511)

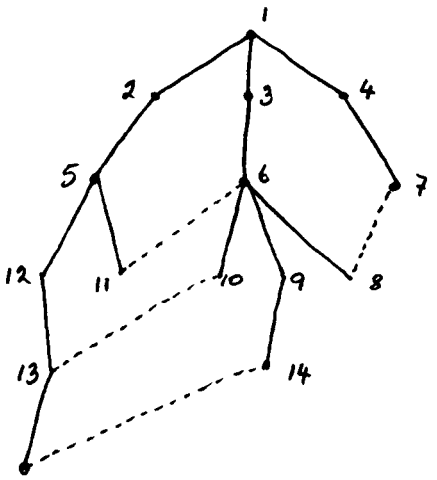


(fig. 4.512)

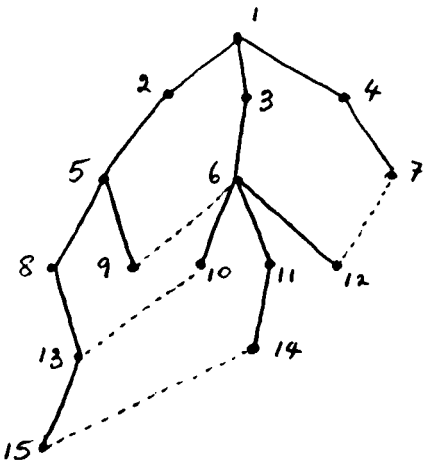
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
layer[i]	1	2	3	4	5	2	2	4	4	6	3	4	3	4	5

(tables 4.513)

i	1	2	3	4	5	6
sumlayer[i]	1	3	3	5	2	1



(fig. 4.514)



(fig. 4.515)

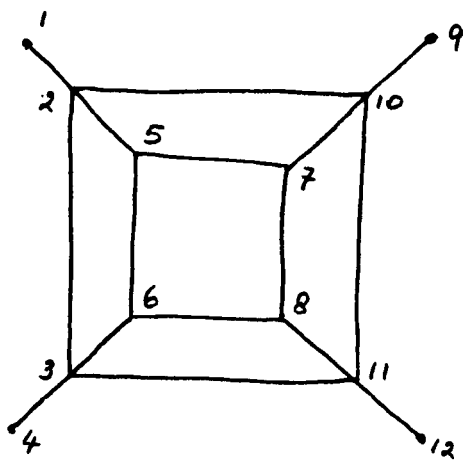
In the example of fig. 4.511 we have a simple metal girder G already labelled (and thus having ld(G) = 6) . If we now hang the corresponding string model at v_1 we obtain the figure of 4.512. We have $ld_{max}(G) = 7$, after examining the sumlayer table of tables 4.513. We can also build a mushrooming r-tree from v_1 and this is indicated in the same figure (fig. 4.512) by heavy lines , its cotree being indicated by dotted lines . We can relabel fig. 4.512 , as indicated a paragraph or two ago and in the worst case obtain the labelling as in fig. 4.514 . This has ld(G') = 7 . But with a little thought we can immediately reduce this to ld(G') = 5 , as given by the labelling in fig. 4.515 .

Hence if we can , firstly, choose the right point from which to hang the model and secondly , manipulate points from one layer to another (i.e. by altering the layer value attached to the points) such that we decrease the numbers within those layers with most points, we have an excellent chance of obtaining a labelling with a smaller ld. Lastly, having got our model in a sausage shaped formation, as opposed to the original multi-triangular configuration, we must try to find a method of labelling which produces the lowest value of ld(G) possible.

4.51.1 Manipulation of the Graph.

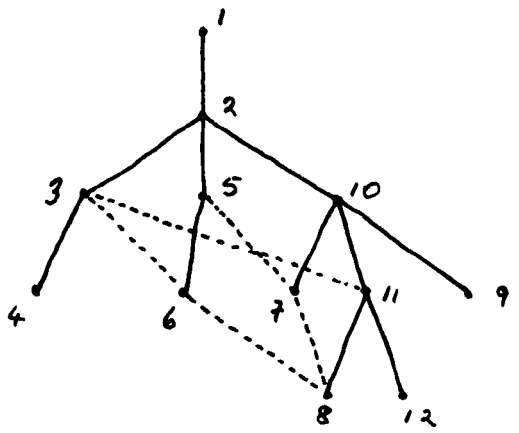
The hanging model and a r-tree have at least one property in common. Adjacent points differ by at most either one generation or one layer. Let us within the graph, costruct a mushrooming r-tree whose root is the point being held. Then the generation number of a point will give an exact indication of the layer within which it lies. Initially layer[i] would be set equal to gen[i] . The problem is that of altering the value of layer[i] for some v_i such that we achieve a smooth, sausage-like hanging graph , i.e. minimise $ld_{max}(G)$. Given that the resulting method is heuristic , it would be to our interest if the mushrooming r-tree was one with maximum height . For the more layers possible the better the chance of decreasing $ld_{max}(G)$. Returning to the sausage analogy, for a given amount of stuffing the sausage is likely to be much thinner if we have a longer skin . Conversely , if the length of skin is decreased the sausage is bound to get thicker. Hence it is desirable that the point being held should be the end point of a maxmin path. That is, the mushrooming r-tree should be one of maximum height.

The general outline of the program is as follows. A search is made along sumlayer[i] for $1 \leq i \leq m$, looking for that layer with most points or two adjacent layers which together have most points . The points of the layer with most points, are examined to see if there is one which can be moved up a layer (i.e. layer[j]:=layer[j]-1;) and yet not create a situation where two adjacent points (of G) have more than one layer between them . Thus if the point to be moved up one layer has adjacent points in the layer below, these also have to be moved up one layer and so on . As an example consider the following graph in fig. 4.51.11 .



graph G

(fig. 4.51.11)



r-tree T_m

(fig. 4.51.12)

The graph G of fig. 4.51.11 represents a simple structure in civil engineering . The shape of the structure is a box or cube sitting on four legs . We find from G that the maxmin path length is 4 and that one of the maxmin paths is given by $[v_1, v_2, v_{10}, v_{11}, v_{12}]$. We can construct a mushrooming r-tree T_m with v_1 as root and obtain the r-tree in fig. 4.51.12. The dotted lines indicate the members of the cotree. The sumlayer table for T_m can be constructed and it is as follows (in table 4.51.13):

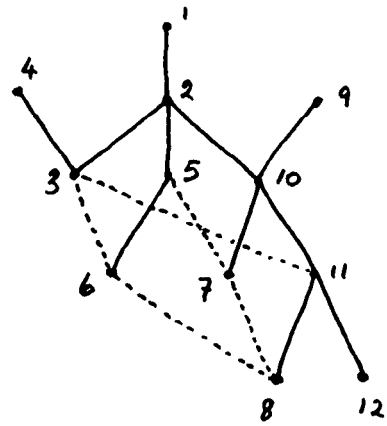
i	1	2	3	4	5
sumlayer[i]	1	1	3	5	2

(table 4.51.13)

We see that sumlayer[4] and sumlayer[3] together have maximum value. We concentrate on sumlayer[4] as that is the larger . We note that both v_4 and v_9 could have their layer numbers decreased without any complications . If we allow a step at a time to the reduction of a points layer value it will take 4 steps to reduce the layer values of v_4 and v_9 each from 4 to 3 , e.g. %

i	1	2	3	4	5
sumlayer[i]	1	1	3	<u>5</u>	2
	1	1	<u>4</u>	4	2
	1	2	3	<u>4</u>	2
	1	2	<u>4</u>	3	2
	1	3	3	3	2

(table 4.51.14)



(fig. 4.51.15)

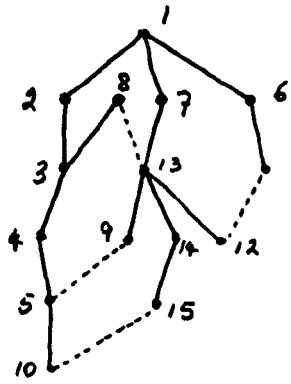
In each case the layer chosen to be reduced is underlined in table 4.51.14 . The last sumlayer values in the table corresponds to the configuration in fig. 4.51.15. The r-tree is still discernable except that its lines do not 'hang down' as in the previous examples but lie at their various levels.

In the last example there was no trouble as to the choice of which point to move up . However it is not always as simple as that in general. The point to be chosen is that one with least degree. This covers the previous example as v_4 and v_9 were both of degree 1 . Thus a search is

made through the points of the particular (maximum) sum-layer and the one with least degree (with respect to the graph) is chosen to move up . If there is more than one point of least degree an arbitrary choice is made between them : in the actual program it was the one which had the least original label. Again in the example of fig.4.51.12 it would have been v_4 rather than v_9 . For completeness sake the result of this method on the r-tree of fig. 4.51/2 is shown in fig. 4.51.17 with the accompanying table (4.51.16) showing the variation in its sumlayers.

i	1	2	3	4	5	6
sumlayer[i]	1	3	3	5	2	1
	1	3	4	4	2	1
	1	4	3	4	2	1

(table 4.51.16)

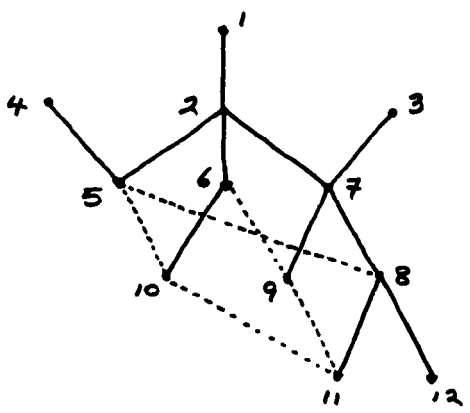


(fig. 4.51.17)

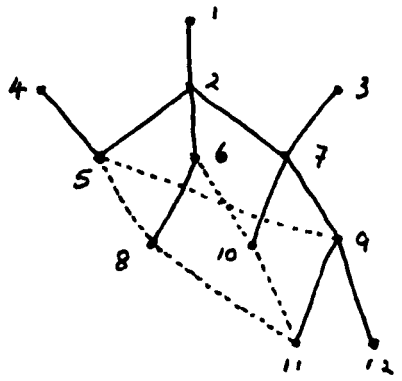
We note that as we start with all the points hanging down (i.e. the values of layer[i] are the maximum possible for all i), they have initially only one degree of movement . That is they can only move upwards , i.e. take lower values of layer[i] . Once some points have had their layer values altered , there will be for some points the choice of either having their layer values further decreased or increased again.This problem is side-stepped in this program by allowing the points to have only one degree of freedom all the time, i.e. they can only move upwards or stay where they are but never move down. Thus there comes a time when any movement of the points (by the alteration of their layer values) either increases the maximum sumlayer value or increases the maximum sum of any two adjacent sumlayers. At this instant the program (or method) stops. More sophisticated methods could be devised which allowed movement in either direction. This introduces added difficulties such as which point to choose and in which direction is it to move or when is the method to terminate.

4.51.2 Labelling of the Graph.

The author found this the most difficult task of all, and feels that the method finally adopted in this subchapter (and program) is only just adequate. The problem was to find a method of labelling the modified T_m, the mushrooming r-tree of the previous subchapter. As hinted earlier , the method is to label the points in successive and increasing valued layers.Thus all the points in layer [1] are first labelled, starting from one. Then all the points in layer [2] are labelled and so on until all layers contain labelled points. In the example of the previous sub-chapter (i.e. fig. 4.51.15.) it might lead to an initial solution as shown in fig. 4.51.21. with $ld(G') = 5$.

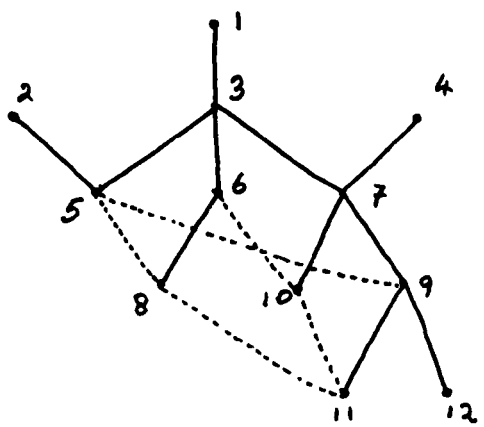


(fig. 4.51.21)



(fig. 4.51.22)

An improvement, perhaps an obvious one, would be to label first, within a layer, those that are adjacent to an earlier labelled point in the layer above them. Thus the figure under consideration will be labelled as indicated in fig. 4.51.22. This does not decrease ld(G) in this particular case. But it can be seen that instead of lines that contributed to ld(G) equaling 5 we have now only one: (2-7). This can be eliminated if we introduce the idea of 'slack', whereby, if there is any slack between layers (e.g. between the first layer, v_1 , and the next to be labelled in layer [2]) while labelling, other points obtain precedence. This is illustrated in fig. 4.51.23.



(fig. 4.51.23)

A further few crude rules are found in the description of the program in Appendix 3.2.

4.52 Analysis of Stage 2.

This was found to be quite a fast method of obtaining a permutation or labelling of the graph whose ld was considerably reduced . In the random data examples reductions in the order of a third or higher were obtained. There is lots of scope for improvement either in the manipulation or the labelling of the graph . There is no need to alter the basic idea but merely refine the rules (of manipulation and/or of the labelling).

4.6 Stage 3.

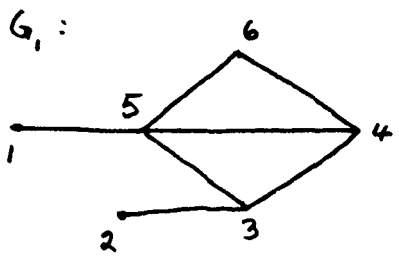
4.61 Introduction.

This the final Stage, improves upon the solution obtained by Stage 2 and if neccessary finds a permutation with minimum bandwidth . The algorithm was the indirect result of a talk given by G.Alway some two and a half years ago . It is similar in principle to the one published by Alway and Martin [34] but the method of attack and explanation is quite different . Alway and Martin discuss their algorithm in terms of binary words and patterns. Here the emphasis is on the corresponding graph and the changes in its ld due to various labellings on it. Suppose as before, that the list of permutations be referred to as L(z) and that within this list the permutations are ordered lexicographically in increasing order of magnitude, e.g. L(3)=(1,2,3);(1,3,2);(2,1,3);(2,3,1);(3,1,2);(3,2,1).

Consider one of the permutations of L(z) and let its members be held in K[i], an Algol array. If we allow the identity permutation to correspond to the original labelling of the graph, then the labelling of the graph corresponding to the permutation K[i] is as follows : $v_{K[i]} = v'_i$, i.e. the point originally labelled K[i] is to be relabelled i . If we write out the adjacency matrix cor-

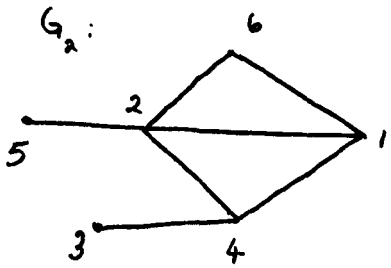
responding to the identity permutation , then the matrix corresponding to the permutation K[i] is obtained by replacing the ith row and columns by the $K[i]^{th}$ ones.

For example consider the graph G_1 in fig. 4.611. Let this correspond to the identity permutation and have binary matrix representation as in A_{G_1}. The graph G_2 and its matrix A_{G_2} corresponding to the permutation (4, 5, 2, 3, 1, 6) can be pictured as in fig. 4.612 .



(fig. 4.611)

$$A_{G_1} = \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$



(fig. 4.612)

$$A_{G_2} = \begin{array}{c} \begin{array}{cccccc} 4 & 5 & 2 & 3 & 1 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

We see that $K[i] = 4, 5, 2, 3, 1, 6$ for $i = 1, 2, .., 6$. The correspondence between the two labelled graphs is that within G,

v_4 has been relabelled 1, i.e. $v_4 = v_1'$,

v_5 2, i.e. $v_5 = v_2'$,

. . . . . ,

v_6 6, i.e. $v_6 = v_6'$.

Thus if we go through $L(z)$ and calculate the ld for each permutation, we shall be able to find a permutation with ld_m. Another way of doing this is to note the ld of the identity permutation and then search through $L(z)$ noting only those permutations which yield successively smaller values of ld. The last one noted will be the one corresponding to $ld_m(G)$. This is in fact what the algorithm does. It surveys the $z!$ possible permutations using stringent conditions to reject unsuitable permutations such that a comprehensive scan can be made in a much shorter time.

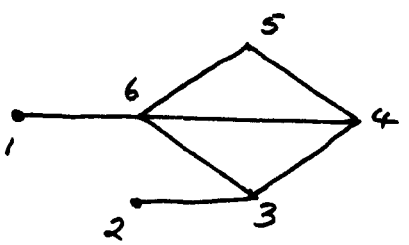
As it is impossible to carry within the computer all $L(z)$ permutations, we have to generate within $K[i]$ each permutation that we wish to examine. It is during this generating process that we incorporate the rules of rejection in order to build up our complete permutation.

4.62 Generating the K[i] (or Rules of Choice).

The method of choosing permutations from L(z) is to fill in, without repetition, the values of K[i] from the set, L , of numbers (or labels) $\{1,2,3, \ldots ,z\}$. The filling in of the array K[i] corresponds to the relabelling of the graph . Thus if we allocate a number to K[1] and then another to K[2] and so on, this will correspond to the label 1 being attached to the point originally labelled K[1] , label 2 being attached to the point originally labelled K[2] and so on. The labels to be applied, rather than the points to be labelled, are chosen in natural order.

We must evolve some rules for choosing the numbers K[1],K[2], ... ,K[z] such that these rules correspond to the sequential examination of L(z). Let us associate with each label i the set of unlabelled points U(i). The rule will be that we allocate to K[i] , for increasing i, <u>the least member of U(i)</u> . Consider the graph of fig. 4.611 . We have $U(1) = \{1, 2, 3, 4, 5, 6\}$ and hence K[1]:= 1 . We have a dynamic pointer that now moves to K[2] indicating that this is the next location to be filled. We have $U(2) = \{2, 3, 4, 5, 6\}$ and so K[2]:= 2. When the pointer finally moves to K[6] we obtain within K[i] the identity permutation, the first in L(z).

Let us consider the next permutation which we know to be (1, 2, 3, 4, 6, 5). The pointer is at K[6]. We reject the content of K[6] and see if there is another member within U(6) to choose. There is not and so we move the pointer down (or leftwards) to K[5] . U(5) = $\{5, 6\}$. We now alter the content of K[5] by <u>choosing the least member within U(5) greater than (the old) K[5]</u> . This is the previous mentioned rule but with greater restriction. Thus K[5]:= 6 . The pointer moves up to K[6] and we use the first mentioned rule to obtain K[6]:= 5 . The graph and matrix corresponding to this permutation are shown in fig. 4.621 .

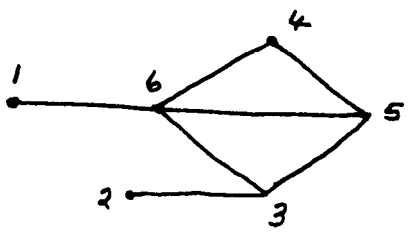


(fig. 4.621)

$$\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 6 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

It is worth emphasising the situation when either rule is to be used. Let us refer to the first rule mentioned, that of choosing the least member of U(i) , as Rule (of Choice) 1 and the second, that of choosing the least member of U(i) greater than the previous K[i], as Rule (of Choice) 2 . If it is required to fill in K[j] then Rule 1 is used if the pointer has just moved <u>up</u> from K[j-1] to K[j] , and Rule 2 if the pointer has just moved <u>down</u> from K[j+1] to K[j] .

Consider the next permutation in the above mentioned example. We know that this will be (1,2,3,5,4,6) but let us see how it is obtained . The pointer is at K[6] and moves down to K[5]. We see that Rule 2 indicates that no member of U(5) can be allocated to K[5]. So the pointer moves down to K[4] . We note that U(4) = $\{4, 5, 6\}$. Rule 2 gives us the point 5 to be allocated to K[4]. (This means that the point 5 has been allocated the label 4.) The pointer moves now up to K[5] and we use Rule 1 to fill in K[5]. We have U(5) = $\{4, 6\}$ and thus K[5] := 4 . Finally the pointer moves up to K[6] and K[6] := 6 . The corresponding graph and matrix are shown in fig. 4.622 .



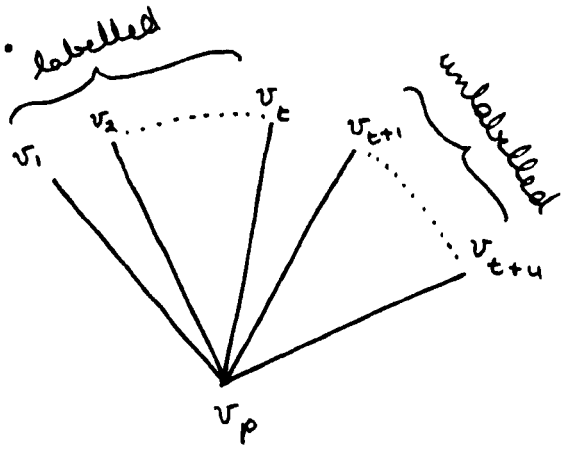
(fig. 4.622)

$$\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 5 & 4 & 6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Thus given a permutation , whether partial or complete, we can by the use of the above mentioned rules find the next complete permutation in L(z) . We could use this to search the list L(z) examining each permutation π_i in turn in order to evaluate $b(\pi_i)$ and hence obtain $b_m(\pi_i)$. However this is impracticable for z greater than seven or eight. Thus we introduce some further tests in order to reject intermediate permutations in which we are not interested because their ld is necessarily greater than the value we are looking for. These tests will be called Tests for Rejection.

4.63 Tests for Rejection.

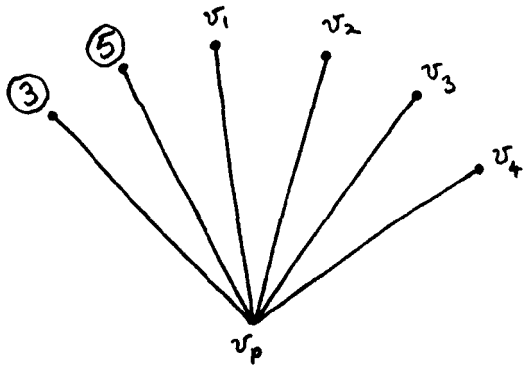
The Tests (for Rejection) can be best explained in terms of the labelling of the graph . Suppose we have a typical unlabelled point v_p which is attached to t labelled points $v_1, v_2, \ldots, v_t$ and u unlabelled points $v_{t+1}, \ldots, v_{t+u}$ as in fig. 4.631.



(fig. 4.631)

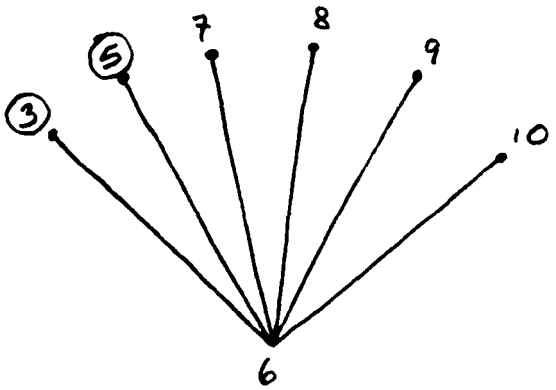
Further suppose that we wish to consider assigning the next unused label 1 to v_p and that $dif(v_p', v_i) = r$ (for $i = 1, \ldots, t$) : that is, if we assign the label 1 to v_p , the maximum difference between it and any of the previously labelled points is r . We can state that for any subsequent labelling of the points adjacent to v_p , $dif(v_p', v_i) \geq \max\{u,r\}$, for $i = 1, \ldots, t, t+1, \ldots, t+u$. In other words, if we assign the label 1 to v_p and whatever the subsequent labelling to the unlabelled points of the graph, we cannot hope to obtain an ld for the graph of less than $\max\{u,r\}$.

(fig. 4.632)



For example consider the configuration of fig. 4.632 with two points already labelled 3 and 5 . Suppose that the next label to be assigned is 6 . This corresponds to filling in K[6]. The test is: can 6 be assigned to v_p such that the final labelling has an ld less than or equal to, 3 say. The answer is no! $dif(v_p', v_i) = dif(4, v_i) \geq 4$ (for i = 1, 2, 3, 4), thus the least ld we can expect if we allocate 6 to v_p is 4 as shown in fig. 4.633 .

(fig. 4.633)



The correspondence with the array K[i] is this. U(6) contains among its members $\{v_1, v_2, v_3, v_4, v_p\}$. A member of U(6) has to be chosen for allocation to K[6]. We have just shown that if we were only interested in permutations whose ld is equal to or less than 3, we would not consider the point v_p since it has four adjacent points as yet unlabelled. This is our Test for Rejection 1 .

The mechanism for carrying this out is as follows. For each label i , we have already associated with it the set of unlabelled points U(i). We now define a subset N(i) (the set of Needed or Neighbours) of U(i). N(i) contains all the members of U(i) which are adjacent to K[1], K[2], ... ,K[i-1] and do not appear in K[1],K[2],...,K[i-1]. N(i) can thus be defined in terms of N(i-1),K[i-1] and U(i-1). We can define N(i) symbolically as N(1) = 0 and

$$N(i) = \{N(i-1) - K[i-1]\} \bigcup \left\{ \begin{array}{c} \text{all points ad-} \\ \text{jacent to } v_{K[i-1]} \end{array} \right\} \bigcap U(i-1) ,$$

for i > 1 .

N(i) can be thought of as a priority subset within U(i) from which the choice for K[i] is sometimes preferentially made.

Suppose we are searching for a labelling whose ld is less than LD and that the situation arises where the number of members in N(j) = LD + 1 . This means that in the graph we have (LD + 1) unlabelled points ad-

jacent to already labelled points. Even if we allocate the next (LD + 1) labels to these points in N(j) we would have a labelling whose ld is greater than LD . Thus Test (for Rejection) 1 corresponds to finding that the number of members in N(j) is greater than LD. This means that the choice of the point in K[j-1] is inadequate and that any permutation which has its first (j-1) elements the same as K[i] (for i = 1 to j-1) , can be safely neglected. The result is that we skip down L(z) to the first permutation whose (j-1) element is different from that in K[j-1] .

It is possible while filling in K[i] to obtain a partial permutation such that within the graph two adjacent labelled points v_i, v_l say, have $\mathrm{diff}(v_i', v_l') > LD$. The correspondence in the array K[i] is that there are more than LD other points between the positions occupied by v_i and v_l . That is if K[s] = i and K[t] = l, then $|t-s| > LD$. Thus every time K[j] is filled, a backcheck is made to see if any labelled points adjacent to $v_{K[j]}$ appear in positions greater than LD places away. This is Test (for Rejection) 2. We reject the current choice of K[j] if the backcheck is positive.

4.64 Summary of Rules and Tests.

As a result of the introduction of the set N(i), we can make one more point about Rule (of Choice) 2. We mentioned that the choice is to be the least member of U(j) greater than the present member of K[j]. Suppose N(j) contains LD members then the choice of members for K[j] is in fact going to be much more restricted. We cannot afford to choose any member other than from N(j). If we did not, N(j+1) would contain all the members of N(j) plus the points adjacent to the new K[j]. This would result in Test 2 rejecting the choice in the next iteration. Hence we anticipate this by stipulating that if N(i) is full ,i.e. contains LD members, the choice will be from N(i) and not U(i).

Rules (of Choice) for K[j].

1/. The pointer has just moved up from K[j-1] to K[j]. The choice for K[j] will be the least member of U(j) (or N(j)).

2/. The pointer has just moved down from K[j+1] to K[j]. The choice will be the least member of U(j) (or N(j)) greater than the old value of K[j].

Tests (for Rejection) of K[j].

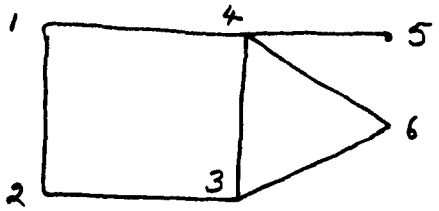
1/. The number of members in N(j) is greater than LD .

2/. There are two adjacent points of the graph whose positions in the permutation are more than LD places apart.

4.65 The Algorithm.

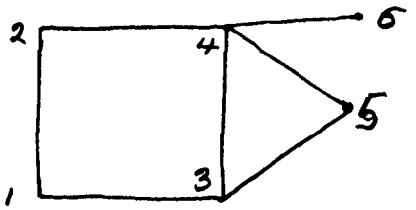
The algorithm works in two parts. It alternates between choosing an element from U(i) to be associated with label i , and that of testing whether it falls foul of one of the Tests. Once we have filled in all the values of K[i] we have a permutation, ld , which is equal to or less than LD. We put LD:= ld - 1 and continue the process from this last permutation. Eventually we either allow the algorithm to test L(z) completely (and thus be sure of obtaining ld_m) or prematurely terminate the process. The actual program has been adapted so that, if it is required, LD can be input by the User. Thus if we have other evidence that ld_m is going to be somewhere near the value LD , we insert this value into LD and the skips down the list will be consequently greater. As an example of the method consider the graph of fig. 4.651 . Suppose we wish to find a permutation yielding an ld of 2. We put LD = 2 and the various stages of the state of K[i] are shown in table 4.653.

The table is almost self-explanatory. As it is read from left to right and down the page so does the algorithm work. The U(j) column is only filled in when reference is made to it. Within the computer U(j) is continually updated. Test 2 is made after K[j] is filled in and hence appears to the right of the K[i]. The labelling corresponding to the final labelling is shown in fig. 4.652.



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

(fig. 4.651)



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

(fig. 4.652)

j	N(j)	U(j)	Rule or Test	K[i] 1	2	3	4	5	6	Test 2.
1	0	1,2,3,4,5,6	RC 1	1						
2	2,4		RC 1	1	2					
3	3,4		RC 1	1	2	3				
4	4,6		RC 1	1	2	3	4			TR 2
3	3,4		RC 2	1	2	4				
4	3,5,6		TR 1							
3	3,4		RC 2	1	2	*				
2	2,4		RC 2	1	4					
3	2,3,5,6		TR 1							
2	2,4		RC 2	1	*					
1	0	1,2,3,4,5,6	RC 2	2						
2	1,3		RC 1	2	1					
3	3,4		RC 1	2	1	3				
4	4,6		RC 1	2	1	3	4			
5	5,6		RC 1	2	1	3	4	5		
6	6		RC 1	2	1	3	4	5	6	TR 2
5	5,6		RC 2	2	1	3	4	6		
6	5		RC 1	2	1	3	4	6	5	

* None

(table 4.653)

4.66 Analysis of Stage 3.

The program, Stage 3, is based on the algorithm just described and was reasonably fast on small sets of data, i.e. $z < 30$, but not as fast as was hoped for larger values of z. This conclusion is reached on the basis that for $z \leq 90$, Stage 3 was used in conjunction with Stage 2 and rarely improved its result. When, due to the time it was taking, Stage 3 was prematurely terminated, the partial permutation in K[i] was printed out and this indicated that the algorithm was not skipping down the list L(z) very quickly (see Chpt. 4.67) . However no comparison , bad or otherwise, can be made with that as published by Alway and Martin as they do not include any experimental results. The method can be improved if we were to make the jumps or skips down the list greater. This can be achieved by refining either the Rules of Choice and/or Tests for Rejection. That is define further rules and tests which would either choose more appropriate points to insert in K[j] or reject the choice for K[j] on the grounds that it would not lead to a desirable permutation.

The Tests as described in 4.63 were in terms of the immediate neighbours (or adjacent points) to a point. That is, if we gave a point v_i the label 1 we were only con-

cerned with the effect on N(1), the subset of points immediately adjacent to previously labelled points . We could extend the tests so as to cover unlabelled points two, three or four lines away . But this means that the time spent in testing for the suitability of a point is going to increase. There will be a point whereby the increase in the size of the skips down L(z) is going to be nullified by the time spent choosing and rejecting unsuitable permutations.

4.7 Conclusion.

The three stages, as programmed for the KDF9 in Algol, were tried on two sets of data. One set was randomly produced and the other was gathered from various sources. When possible, Stage 3 alone was attempted on the same data and the accompanying table (table 4.71) gives an idea of the result of Stages 1, 2, and 3, and Stage 3 alone and their respective times. On the random data, $z>40$, it was not possible to run all the three stages to their conclusion, and thus the times in their column indicate how long they ran before they were terminated.

We see that, for the practical examples illustrated in figs. 4.72 and 4.73 , as opposed to the random ones, all three stages give close results. That is Stage 1 either gave ld or just one less, and Stage 2 gave the minimum ld or one above. However the random examples did not fare so well. Stage 3, except for the first one, did not improve upon the permutation as supplied by Stage 2. This can be accounted for slightly by the fact that rarely was Stage 3 ever allowed to run to its natural end. Stage 2 was usually allowed to finish, and it reduced the initial ld value (as set up by the random matrix generator) by at least a third. How much it would improve upon

an intelligent guess or attempt at an initial labelling is debatable. The resulting graphs produced by the random matrix generator were so complex and naturally non-planar (i.e. they could not be drawn on a two-dimensional plane without lines intersecting each other) that it would be impossible to envisage the graphs, let alone attempt to label them. In the cases where the graphs were gathered from various sources (usually civil and electrical engineering problems and chemical structures) the improvement may not be so significant, but this is because with practice the author acquired certain intuitive and non-rigorous rules for the labelling of graphs. The practical problem as illustrated in figs. 4.72 and 4.73 have the least ld labelling on the graphs.

Random Data Table.

Title	% density	Z	b+	Initial value of Id(G)	Value of E(G) + Id(G) in all three stages / Times for each stage in minutes/seconds. 1	2	3	3 only
Random	25	20	46	18	6 / 50s.	11 / 36s.	10 / 30/42	
"	20	40	151	37	13 / 10/59	25 / 30/20	NONE / 10/45	33 / 95/20
"	10	60	157	56	14 / 18/8	32 / 9/36	NONE / 21/33	55 / 20/27
"	5	60	110	51	11 / 3/45	26 / 4/7	NONE / 4/46	51 / 10/15
"	5	70	122	64	12 / 5/34	26 / 6/44	NONE / 5/23	64 / 15/8
"	5	80	153	78	13 / 5/17	35 / 8/10	NONE / 5/22	78 / 20/3
"	5	90	195	79	15 / 5/9	42 / 7/34	*	*
"	10	90	369	87	23 / 11/19	61 / 25/33	*	*
"	5	120	324	110	*	72 / 25/27	*	*
"	3	150	304	142	*	61 / 31/3	*	*
"	3	200	540	192	*	126 / 42/49	*	*

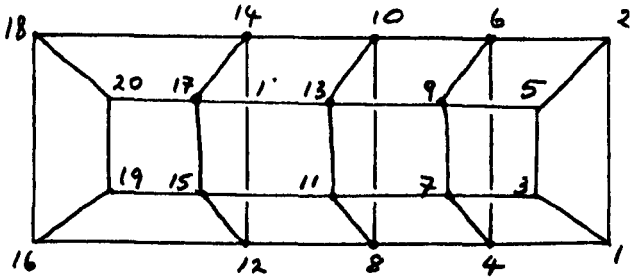
* Machine not large enough to compute these stages for the given data.

(table 4.71)

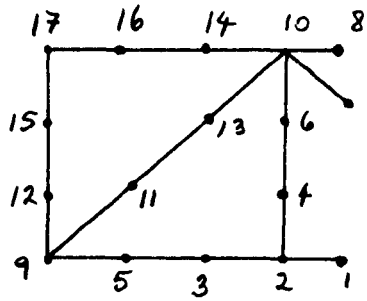
Practical Data Table.

Title	z	br	Initial value of ld(G)	Value of E(G) & ld(G) in all three stages. / Times for each stage in minutes/secs..			
				1	2	3	3 only
Rectangle-3	20	36	15	4 / 37s.	4 / 33s.	4 / 8s	7 / 15/14
Flanged/1	17	18	7	3 / 23s.	5 / 15s.	4 / 7/13	5 / 14/24
Flanged/2	27	30	6	4 / 1/22	5 / 1/46	4 / 14s	4 / 20/10
Flanged/3	34	39	6	4 / 2/25	4 / 1/10	4 / 16s	4 / 2/15
Silo/1	18	23	5	3	4	3	
Elec. Pylon	16	40	11	6 / 33s.	7 / 42s.	7 / 2/9	10 / 10/11
Tri-sected Semi-squares	24	59	10	7 / 1/35	8 / 1/11	7 / 6/42	10 / 10/10
Icosohedron	12	30	11	5 / 14s	6 / 27s.	6 / 8s.	6 / 7/6
Harp	25	36	19	3 / 1/11	4 / 18s	4 / 9/37	19 / 10/10
Cage	22	38	10	5 / 49s.	6 / 2m.	5 / 8/1	10 / 10/11
B_{10}/H_{10}	10	24	8	5 / 9s.	6 / 5s.	5 / 4s.	5 / 1/7
Antiprism	8	16	7	4 / 6s.	4 / 15s.	4 / 2s.	4 / 6s.
B_{18}/H_{22}	18	41	12	4 / 35s	5 / 31s.	4 / 12/16	12 / 12/16
Door	28	42	23	4 / 1/27	6 / 53s.	6 / 8/10	23 / 18/1
Globe	17	32	5	5 / 3s.	6 / 1/40	5 / 10s.	5 / 6s.

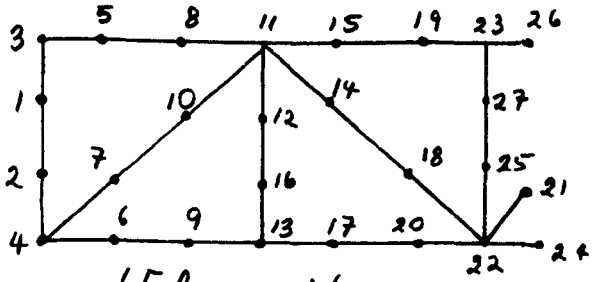
(table 4.71 A)



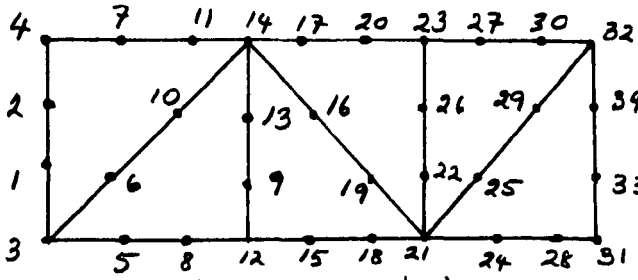
(Rectangle/3)



(Flanged/1)

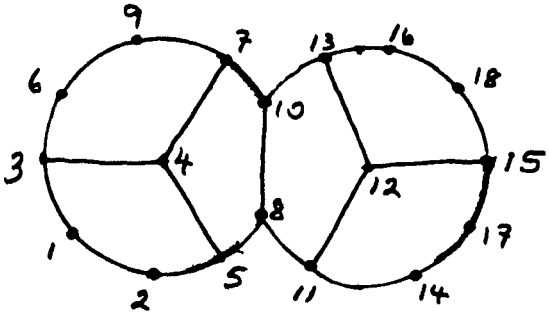


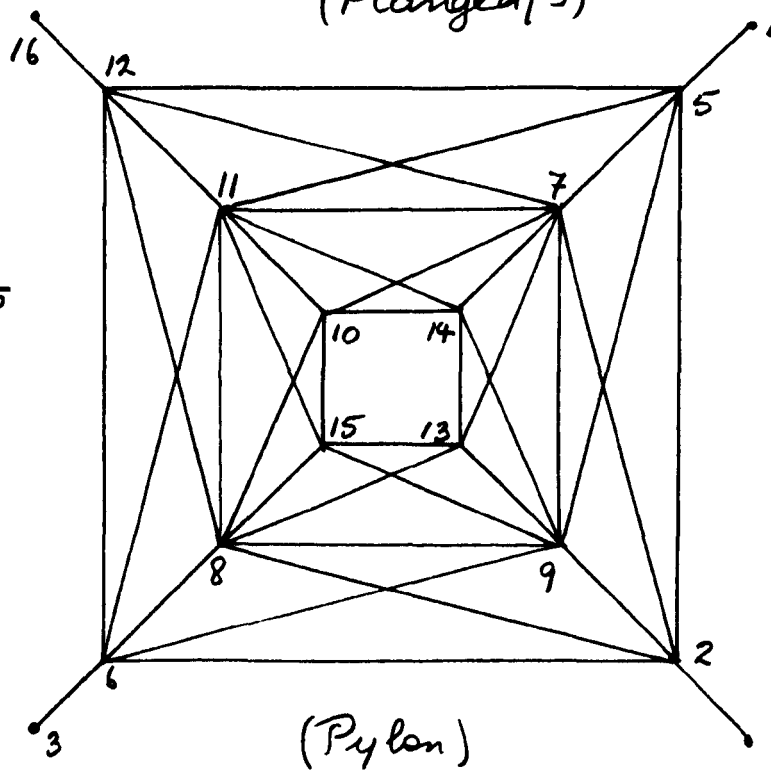
(Flanged/2)



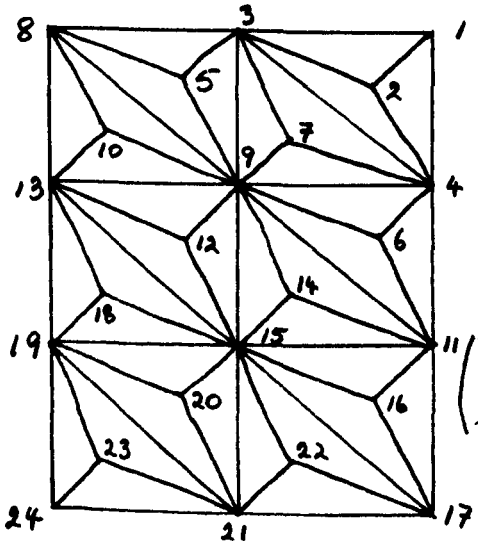
(Flanged/3)

(Silo/1)



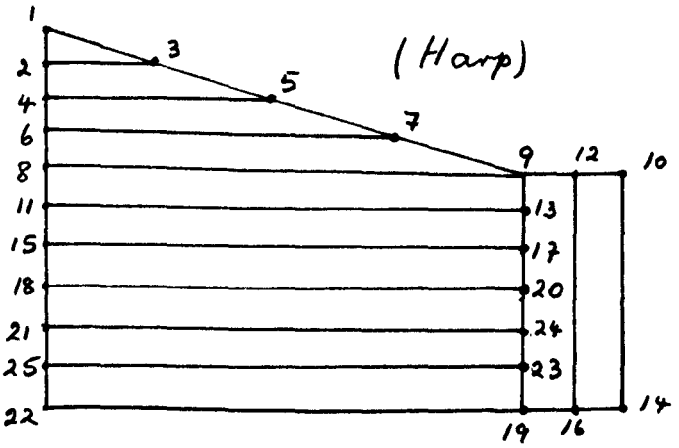


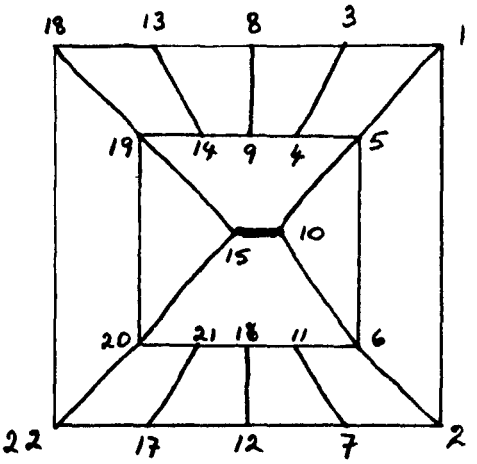
(Pylon)

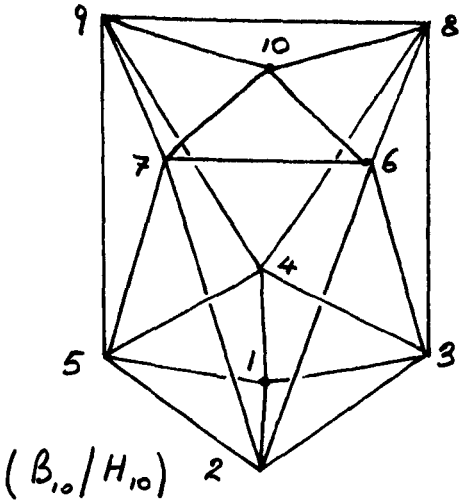


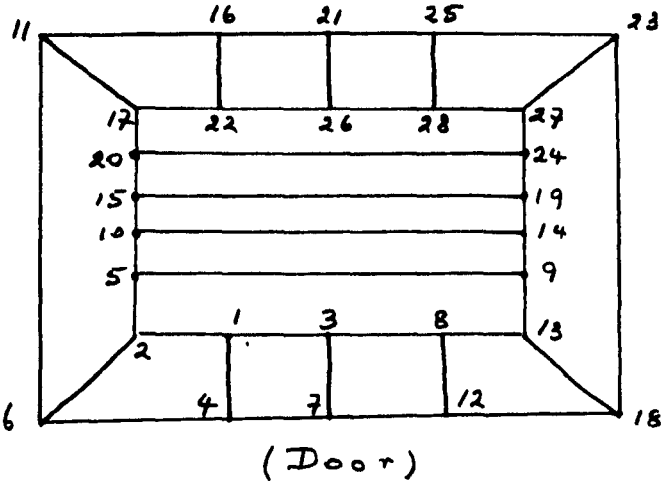
(Tri-sected semi-squared)

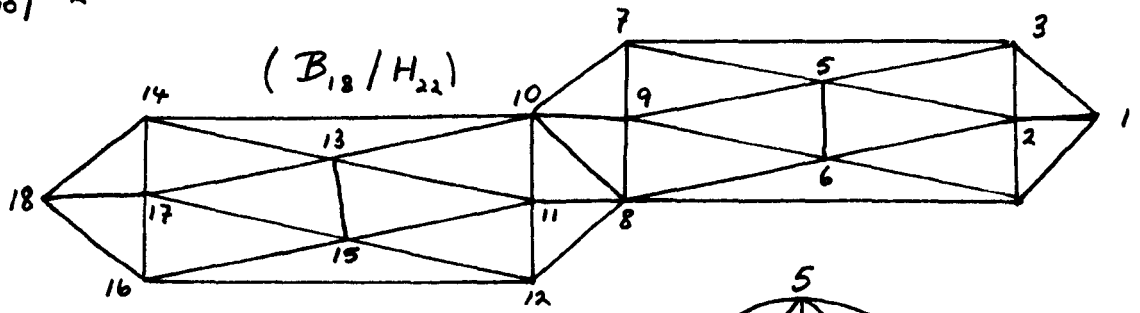
(figs. 4.72)

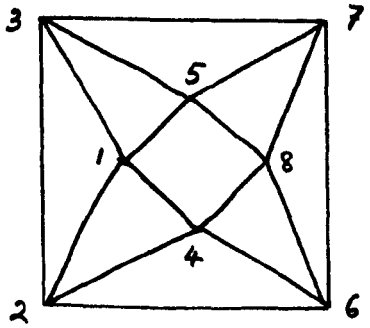




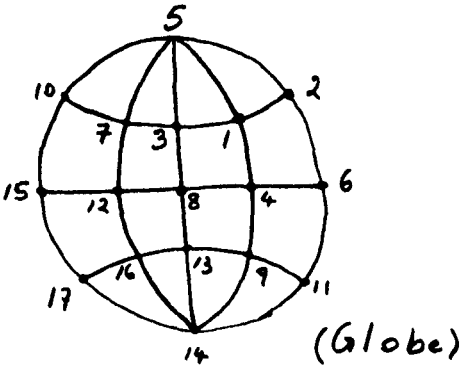








(fig. 4.73)



V SHORTEST DISTANCES ON A DIGRAPH .

5.1 General Discussion.

Within a graph we defined (in Ch. 1.23.1) $\mu_m[v_i, v_j]$ to be the shortest path between v_i and v_j, and $l(\mu_m[v_i, v_j])$ to be the length of, or the number of lines in it. Let us extend this definition to cost associated digraphs so that for any two points v_i, v_j within it, we define $\mu_m[v_i, v_j]$ to be the path with <u>least cost</u> (or <u>shortest distance</u>) from v_i to v_j, i.e. if $(u_1, u_2, \ldots\ldots, u_k)$ were to be any path from v_i to v_j then

$$\text{cost}(\mu_m[v_i, v_j]) = \text{Min}\left(\sum_{f=1}^{k} \text{cost}(u_f)\right).$$

If there is no path from v_i to v_j then the shortest distance between them in that direction is put at infinity.

We can now define three closely related problems. The problems are to find the shortest distance (and/or routes) between

1) two specific points of the digraph,
2) one specific point and the remainder of the points of the digraph, and
3) all pairs of points of the digraph.

Considerable work has been done on these problems, (a general discussion on all the methods will be found in [55]) , and all the algorithms to solve them fall into two categories. The first, called by some authors the matrix method , finds the shortest distances between all pairs of points simultaneously (and hence also solving problems 1) and 2) at the same time). The matrix methods give us the shortest distances between all pairs of points, but extra computation is necessary if the routes or paths yielding these distances are also required. The second method,sometimes referred to as the tree method, is used specifically to solve problems 1) and 2). However, it can be applied repeatedly and hence to solve problem 3). The tree methods, due to the nature of their construction, yield both the shortest distance and a shortest path between pairs of points.

The author investigated the matrix methods and wrote a program for the most efficient one within this category, the Cascade method of Farbey et al [49]. The tree methods were then examined and two programs were written using the techniques developed within this thesis.

The matrix methods are very easy to code (for a computer), and the Cascade method was found to be especially easy and elegant. However the matrix algorithms are very space consuming (requiring storage of the order z^2, at least), and do not compensate for this by a reduction in speed. The tree technique, as in the programs written by the author, required slightly more laborious programming and as a result are more complicated to follow. But they use far less store (of the order $z+br$) and do provide a shortest route while computing the shortest distance between pairs of points. A brief discussion of each method will follow.

5.2 The Matrix Methods.

We have a cost associated digraph with no loops or multiple connections, and whose costs associated with each arc are non-negative . The information can be stored, as described in Ch. 2.11.1 , in an adjacency type cost matrix D, where d_{ij} represents the cost to be associated with the arc (i - j). If there is no arc joining v_i to v_j , we put d_{ij} equal to infinity. For computational purposes this was set at 10^4 . We also make , for all i, $d_{ii} = 0$. We define a new operation min-add, denoted by the operator * , between two square matrices A, B, such that A * B = C where C is (z × z) and

$$c_{ij} = \min_{\forall k} \{ a_{ik} + b_{kj} \} \quad \text{--------(5.2 A)}$$

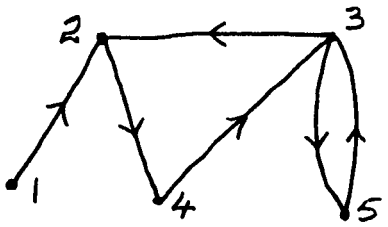
Suppose that we now perform this operation upon D, the cost matrix of above. We have $D^2 = D * D$, and D^2 will give the length of the shortest paths, of not more than two arcs between all pairs of points.This is very easy to see when we examine the operation as defined by (5.2A) in more detail. We vary k over all points of the digraph so that d^2_{ij} ($\in D^2$) will automatically have the least value of the distance from v_i to a third point, and from there to v_j . (This third point may be v_j itself).

Similarly, we can obtain D^3, the matrix of shortest paths of not more than three arcs between all pairs of points by the operation $D^2 * D$ or $D * D^2$. D^4 can be obtained from $D^3 * D$, $D * D^3$ or $D^2 * D^2$ thus giving us an indication of how to accelerate the process. This process terminates when either $D^{m+1} = D^m$ or $D^{2n} = D^n$ (depending upon which method is adopted). It can be seen that this will be a lengthy process.

Let us asses the amount of storage required if we choose the accelerated method, i.e. we repeatedly perform $D^{2\delta} := D^{\delta} * D^{\delta}$, replacing D^{δ} by $D^{2\delta}$ each time. Because we have to compare $D^{2\delta}$ with D^{δ}, we shall have to store both of them thus requiring a storage of the order of $2xz^2$ elements.

5.21 The Cascade Algorithm.

This algorithm is a great improvement upon the general matrix methods just described. It uses far less store, of the order of z^2 and requires only two matrix min-add operations only. It forms the matrices D^A and D^B in a similar way as the general matrix method forms D^2 and D^3. The general matrix method computed the elements of D^{d+1} (or D^{2d}) from those of the matrix obtained in the previous iteration, i.e. D^d. The Cascade Algorithm however, forms D^A within D itself. That is, the elements of D are replaced one by one, by those of D^A and the min-add operations are carried out over the elements of this 'hybrid' matrix. The ordering of the computation of the elements of D^A is all important and it is downwards by rows and from left to right, i.e. we compute $d^A_{11}, d^A_{12}, \ldots, d^A_{1z}, d^A_{21}, \ldots, d^A_{2z}, \ldots, d^A_{z1}, \ldots, d^A_{zz}$. D^B is formed from D^A (as D^A was formed from D) but in the reverse order, i.e. by rows upwards and from right to left. d^B_{ij} ($\in D^B$) now holds the value of the shortest distance from v_i to v_j. Farbey et al prove that these two passes are sufficient and have provided a very efficient algorithm. As an example of the method the matrices D , D^A and D^B are shown next to the corresponding cost associated digraph of fig. 5.211.



(fig. 5.211)

$$\mathbf{D} \equiv \begin{bmatrix} 0 & 2 & X & X & X \\ X & 0 & X & 1 & X \\ X & 3 & 0 & X & 1 \\ X & X & 2 & 0 & X \\ X & X & 1 & X & 0 \end{bmatrix}$$

$$\mathbf{D}^A \equiv \begin{bmatrix} 0 & 2 & X & 3 & X \\ X & 0 & 3 & 1 & X \\ X & 3 & 0 & 4 & 1 \\ X & 5 & 2 & 0 & 3 \\ X & 4 & 1 & 5 & 0 \end{bmatrix} \qquad \mathbf{D}^B \begin{bmatrix} 0 & 2 & 5 & 3 & 6 \\ X & 0 & 3 & 1 & 4 \\ X & 3 & 0 & 4 & 1 \\ X & 5 & 2 & 0 & 3 \\ X & 4 & 1 & 5 & 0 \end{bmatrix}$$

(where X stands for infinity)

However in common with the other matrix methods, the Cascade method does not give us the minimal routes . The authors do indicate how this could be done but it entails double the storage and of course will take much longer. Land and Stairs [51] have extended the method to deal with large digraphs by means of partitioning but the essence of simplicity as characterized by their original algorithm has been lost.

5.3 The Tree Methods.

Most tree methods written to solve problem 2) have one point in common. The minimal solution is in the form of a maximal spanning directed rooted tree where a shortest route between the root and any other point is a path of the directed rooted tree (d-r tree). G.Dantzig [47] prefers to start with any maximal spanning d-r tree and by means of additions and deletions to it, obtain the minimal solution . To each point of the d-r tree is attached its current distance from the root (via a path of the d-r tree). An arc l_1 is chosen from the cotree and added to the d-r tree. If this results in a point having lesser distance from the root, one of the arcs l_2 in the resulting circuit is deleted and returned to the cotree. (l_2 will be that arc of the d-r tree which has the same terminal point as l_1). We also alter the current distances appropriately. If when we have inserted the arc l_1 into the d-r tree there is no improvement, we return l_1 back to the cotree. Another arc l_3 is chosen from the cotree and the process repeated. The algorithm terminates when no improvement occurs for the addition of any arc of the cotree. This method is slow and inefficient because in the worst case we may have to examine every possible path out of the root. At each stage the choice is made from all the arcs of the cotree

to see if any of them might improve the solution, for the rejection of an arc does not necessarily mean that it cannot appear in the final solution.

Moore [63] suggests a much improved algorithm. He builds a least cost maximal spanning d-r tree from the root outwards such that at any stage of this mushrooming process, only a small subset of the arcs of the co-tree are examined. It is very similar to the authors second program called Shortest Route 2 (which was written before the author realized the existence of Moores algorithm).

The author wrote two programs, Shortest Route 1 and 2 to solve problem 2) . Shortest Route 1 is a specialised program which assumes that the cost of each arc is the same. Shortest Route 2 is the more generalized program which gives the least cost maximal spanning d-r tree for each point of the digraph in turn (and hence solves problem 3)). In both cases , as a result of the representation of the data, the size of the problem to be solved is not limited by z, the number of points but by br, the number of arcs in the digraph. Using the branches list representation (Chpt. 2.11.1) the more generalized method, Shortest Route 2, requires storage of the order 2 br + 5 z. (However in the Algol program, because of the inability of being able to do any list processing, the storage had to be

set at 2br + 9z . 5z locations never being , at any instant, more than two fifths full of necessary information.)

The discussion has been primarily the use of tree methods on the solving of problems 2) and 3) . T. Nicholson [60a] has devised an algorithm which solves problem 1) . This is also a type of mushrooming process . He mushrooms out , as we do in both the Shortest Routes methods , but from the two points simultaneously . When the two d-r trees (one with reverse direction to the other) meet he obtains as a result the shortest route and distance between them. He uses a technique first proposed by Minty (in [55]) in order not to cover more points of the digraph than necessary. He carries a minimum distance counter for each d-r tree such that the current distance of any point from (or to) the root is the least distance if it has value less than the corresponding counter. Thus he confines himself to expanding from the points that have current distance value equal to the counter distance. This is necessarily a slow process if at each iteration only one point at a time is added and this will occur if the costs (of each arc) are either all unequal or very nearly so.

5.31 Shortest Route 1.

We assume in this case that the cost to be associated with each arc is unity . Thus the cost of a path between any two points is equal to the number of arcs within it. We can now use the mushrooming r-tree technique to solve problem 2) .

A root point is chosen and we insert into the d-r tree (which we are going to build up) all points adjacent from the root. These points will be a distance of one unit away from the root. In the next or second iteration we examine these points and include in the d-r tree all points adjacent to them and not already in the d-r tree. These adjacent points will be a distance of two units away from the root. We continue these iterations so that at the kth iteration we include in the d-r tree all points v_j , adjacent to those just added in the previous iteration and not already in the d-r tree, i.e. $l(\mu[v_{root}, v_j]) = k$. The process terminates when no new points can be added to the d-r tree.

Thus we mushroom out from the root examining all minimal routes leading away from it . As soon as a point is introduced into the d-r tree we examine all points adjacent from it . This means that all arcs eminating from that point have been examined. Thus in the next and subsequent iterations the point and its arcs will never be con-

sidered again. As a result the building of a maximal spanning d-r tree requires only br examinations of the adjacency matrix (or in the actual program , of the branches list representation).

In the program the mushrooming d-r tree is stored in the below array and when it has been constructed, we find the shortest distance from the root to any point, and a path yielding this distance, by running down the d-r tree from the point to the root. In this and the next program this results in the route being given back to front. Instead of the first label in the sequence being the root and the last the terminal point of the path , we have the terminal point appearing first and the root, last. This is in order to aid the programming . Otherwise, while running down the d-r tree, we would have to store the points and then output them in the reverse order.

Shortest Route 1 is similar in idea to, but more powerful than, Mintys solution to problem 2) [52].The difference is that he confines himself to graphs where he obtains a string model of the graph [Chpt. 4.51], holds it up by the root and hence obtains a mushrooming r-tree. This simple idea, for obvious reasons, cannot be extended to digraphs.

5.32 Shortest Route 2.

Within a digraph it is more usual to find the associated costs to be different (e.g. the distances of a road) and so the algorithm just described is inadequate. The one to be described in not altogether different from the previous one and is similar in theory to dynammic programming. Let us define $f_k(i,j)$ to be the least distance from v_i to v_j, of k or less arcs. Dynamic programming formulation of the problem would be $f_1(i,j) = \text{cost}(i,j)$ and

$$f_{k+1}(i,j) = \underset{\forall l}{\text{Min}}\{f_k(i,l) + \text{cost}(l,j)\}.$$

If v_i remains constant, in our case it will be the root of the mushrooming r-tree, we obtain $f_1(j) = \text{cost}(\text{root},j)$,

$$f_{k+1}(j) = \text{Min}\{f_k(l) + \text{cost}(l,j)\},$$

over all v_l adjacent to v_j.

Let the incomplete mushrooming d-r tree of the kth iteration be denoted by T_k, where the paths in T_k are the shortest distance paths from the root to any point in T_k of k or less arcs. We thus have $f_k(j)$ for all v_j in T_k. Denote by M_k the set of points whose f_k values have been either computed or recomputed in the kth iteration. M_k will not contain the points of T_k whose f_k value equals their f_{k-1} value.

Consider the expansion of the d-r tree in the $(k+1)^{th}$ iteration. A point v_t adjacent to v_s in M_k (from whose points the expansion is being made) will fall into one of three categories :

1/. It is not a member of T_k nor of T_{k+1},

2/. it is not a member of T_k but is of T_{k+1},

3/. it is a member of T_k .

Suppose v_t belongs to the first category. We incorporate it into T_{k+1} by means of the arc $(s - t)$ and put $f'_{k+1}(t) := f_k(s) + cost(s,t)$. The dash over the f denotes that the value contained within f_{k+1} need not necessarily be the final one . It becomes final at the start of the next iteration.

For practical purposes the second and third categories can be regarded together. Suppose that v_t belongs to either T_k or T_{k+1} ,i.e. $f_k(t)$ or $f'_{k+1}(t)$ has already been computed. We test to see whether the path to v_t from the root via v_s has lesser cost than the existing one. We test if $f_k(t) \leq f_k(s) + cost(s,t)$, for v_t in T_k (or if $f'_{k+1}(t) \leq f_k(s) + cost(s,t)$, for v_t in T_{k+1}) . If the condition is satisfied (i.e. the existing path has equal or less cost than the proposed new one) we make $f'_{k+1}(t) := f_k(t)$. If the condition is not satisfied , we alter the path from

the root to v_t by adding the arc $(s - t)$ to T_k and deleting the other arc incident to v_t in the resulting circuit, and making $f'_{k+1}(t) := f_k(s) + cost(s,t)$.

It now remains to state that for all points v_j in T_k for which $f_{k+1}(j)$ has not been computed (because they were not adjacent to a member in M_k) will have $f_{k+1}(j) := f_k(j)$. If M_k is empty, this means that $T_k = T_{k+1}$ and that for any point within T_k , we have not found a shorter distance from the root of $k + 1$ arcs. Hence this indicates the termination of the process.

Computationally we only have to carry one array $f[j]$. $f[j]$ will store the latest value of $f_k(j)$. When we form $f_{k+1}(j)$ we insert the new value into $f[j]$,overwriting the previous value (whether it had been $f_k(j)$ or $f'_{k+1}(j)$). The array $f[j]$ corresponds to the 'labels' of Moores algorithm as described in [55]. Finally, we can regard all the three categories as one. This is achieved by initially making $f[j] := \infty$ (or some other large number) for all j in the digraph. When we meet a point v_t we test whether $f[t] >$ or $\leq f[s] + cost[s,t]$. If $>$ holds true, we alter $f[t]$ and the path from the root to v_t , otherwise we do nothing. With a little thought it will be seen that this does in fact cover all three categories.

If we let $n(M_k)$ be the number of points in M_k , it was found that for non-planar digraphs $\sum n(M_i)$ over all i of the computation, varied between two and three times the number of points in the digraph . This seems to agree with Moores proposition that for planar digraphs, a point will not on average occur more than twice in $\sum M_i$ (for all M_i in the computation).

5.3 Conclusion.

The Cascade and Shortest Route 2 programs were run on sets of random data with the number of points varying from 10 to 85 . Table 5.31 shows the time taken by each program for each set of data. We see from the time ratio column (Shortest Route 2 divided by Cascade) that Cascade is anything up to twice as fast for small digraphs and just faster for medium sized digraphs. However at z=70 and density of 5 % , Shortest Route 2 takes no longer and for z=85 and density of both 5 % and 9 % , it is faster.

Let us see how the two algorithms, as examples of each of the two main methods , compare with each other. The Cascade algorithm will be obviously inefficient on problems 1) and 2) . To use it on the two mentioned types of problems would be similar to using a sledgehammer to crack a nut. Let us assess the two algorithms on problem 3) where we do not require the shortest routes. We see that the Cascade algorithm will be faster for small and medium sized digraphs , regardless of their density . However for large sparse digraphs, Shortest Route 2 is as good if not faster than Cascade. If we wish to find the shortest routes as well as the shortest distances (to problem 3)) Shortest Route 2 will be as fast or faster than the appropriately modified Cascade algorithm.

Title	z	br	density	Time taken by Cascade	Time taken by S.R.2	S.R.2 / Cascade
14/1	10	51	50 %	4 s.	10 s.	2.5
14/2	10	24	25 %	3 s.	8 s.	2.7
14/3	10	5	5 %	2 s.	5 s.	2.5
14/4	30	428	48 %	47 s.	1m.45s.	2.2
14/5	30	249	28 %	47 s.	1m.35s.	2.0
14/6	30	64	7 %	45 s.	1m.31s.	2.0
14/7	30	32	4 %	43 s.	55 s.	1.3
14/8	50	958	38 %	3m.5s.	5m.28s.	1.8
14/9	50	492	20 %	3m.7s.	4m.41s.	1.5
14/10	50	206	8 %	3m.4s.	4m.22s.	1.4
14/11	50	95	4 %	2m.56s.	3m.52s.	1.3
14/12	70	1283	26 %	8m.2s.	10m.40s.	1.3
14/13	70	402	8 %	8m.1s.	8m.42s.	1.1
14/14	70	192	4 %	8m.	7m.59s.	1.0
14/15	85	665	9 %	13m.44s.	13m.16s.	1.0
14/16	85	350	5 %	13m.42s.	13m.12s.	1.0

(table 5.31)

VI THE TRANSPORTATION PROBLEM .

6.1 General Discussion.

The author reveals no new method or theory to solve the Transportation Problem. This is a short note on how the problem can be tackled by means of tree manipulation , the original method being due to H.I.Scoins, and how it could be improved by using some of the ideas expounded in previous chapters. Let us briefly state the problem.

We have m supply depots (or transmitters) each capable of supplying a_i ($1 \leq i \leq m$) units of goods and n demand depots (or receivers) each requiring b_j ($1 \leq j \leq n$) units. We are also given a cost matrix $c_{ij} \in C$, which indicates the cost of transporting one unit from supply depot i to demand depot j . We also specify that $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$. The problem is, given the above conditions which is the cheapest way to meet the demand at the receivers from the transmitters. Mathematically, if x_{ij} were to represent the number of units sent from transmitter i to receiver j , we have $\sum_{i=1}^{m} x_{ij} = b_j$ and $\sum_{j=1}^{n} x_{ij} = a_i$, for all i and j , and we wish to minimise the function

$$\mathcal{b}(x_{ij}) = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} .$$

Any basic feasible solution

will contain only $(n + m - 1)$ x not equal to zero. In fact the basic feasible solution can always be represented as a spanning tree of the graph of $n + m$ points. For a further discussion on the subject reference should be made to [61,62].

The general method of solution is to obtain an initial basic feasible solution and then iterate upon it always finding better solutions, till the best solution is converged upon . Thus there is a natural division into two parts: that of finding the initial tree and that of manipulating this tree into others with better cost solutions.

6.2 Obtaining the Initial Tree.

There is more than one method of obtaining an initial basic solution: the minimum row method, the minimum matrix method , the N-W corner method to name a few. However according to [61] there is no advantage in using any particular method in order to obtain a smaller initial cost or reduce the number of iterations in the second stage. It seems that little work has been done in order to establish either mathematically or statistically the superiority of any one method.

The method chosen to be used in the program was the modified North-West corner method. This was incorporated in the procedure called Treeform . The procedure described the tree in the below array, the ordering of the tree not being made till the second stage. The ordering is necessary in order to be able to manipulate the (basic solution) tree into another with lesser cost.

6.3 Obtaining the Final Solution.

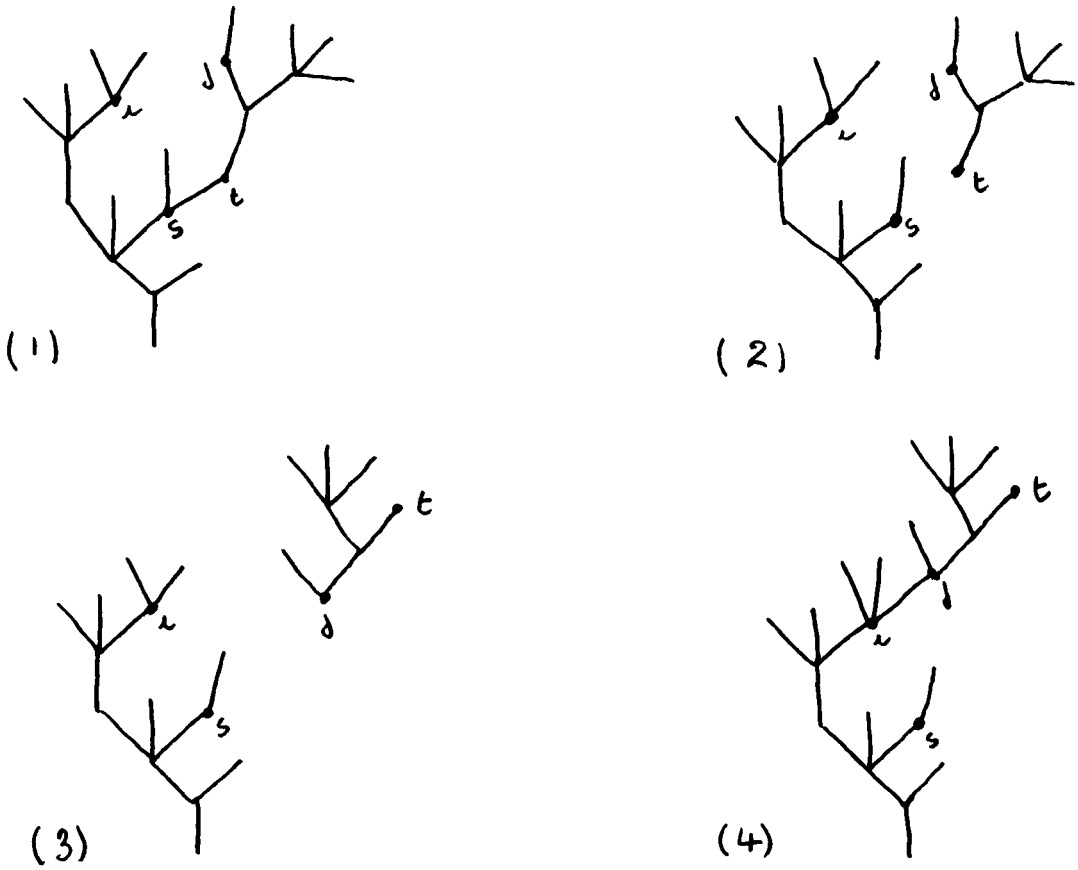
The method used in the program is sometimes referred to as the Stepping-stone Algorithm. A slight explanation will follow but it will be illustrated in terms of the basic tree solution . Suppose we have a basic feasable solution T_k as a result of the kth iteration. We now compute the shadow costs for each of the $n + m$ points. The shadow cost of the root is made equal to zero. From the root we go up the ρ-tree (using the process procedure) calculating the other $n + m - 1$ shadow costs from the formula shadow[j]:= cost[i,j] - shadow[i] , where shadow[i] has already been computed. In the program the formula becomes shadow[j]:= cost[i,below[j]] - shadow[below[j]] .

Having computed the shadow costs, shadow[i], for all $n + m$ points the matrix is searched for the smallest value of cost[i,j] - shadow[i] - shadow[j] ≤ 0 --(5.3A). Scoins in his method adopts a threshhold technique which combines the above process and the other of finding the first value of (5.3A) which is smaller than some preset level.

Having noted the desirable link (i-j) to be inserted, a simple circuit is formed in the ρ-tree and

the link to be deleted (within the circuit) quickly determined, (s-t) say. (s-t) will be the link in the circuit with the least associated load . The ρ-tree is now severed into two at (s-t) . The subtree is inverted so as to make v_i (or v_j) as its root and added to the main ρ-tree at v_j (or v_i), thus defining a new tree T_{k+1} . This can be illustrated pictorially in fig. 5.31.

It will be noted from the figures in fig. 5.31 that in forming T_{k+1}, it will be unnecessary to recalculate all the shadow costs again. The only ones which will differ from those in T_k will be those in the subtree that has been inverted and attached elsewhere. Thus we can use the procedure process upon the subtree with v_j (in the figure) as root and recompute the shadow costs of its points. If the subtree is small, this will result in a large saving of work. Otherwise for each iteration the shadow costs of each of the n + m points will have to be recomputed. The procedures belonging to the program are printed in Appendix 5 and for any one who is interested , they do appear with quite self explanatory comments.



(fig. 5.31)

In (1) we have T_k and from the shadow costs we have worked out that (i-j) is to be inserted and (s-t) to be deleted. The deletion is done in (2) and the subtree at v_t is inverted so that v_j is its new root as in (3). It is now reattached to the main tree at v_i to form T_{k+1}.

6.4 Conclusion.

The program as written by the author was not any faster than the one written by H.I.Scoins. That is the ρ-tree representation and manipulation program was not found to be faster than the r-tree one. The reason is that the procedures used within the authors program were written originally for the work of Chapters 3 and 4, where efficiency was not all important. In those chapters it was required to test ideas and this resulted in a small number of slow and inefficient procedures whose purpose was mainly to test some aspects of manipulation rather than find the best way of doing the actual manipulation. Thus there is scope for further work in finding how the most efficient program of each method compares with other.

R E F E R E N C E S .

Chapter 1.

1/. Berge,C. : Theory of Graphs and its Applications [Transalated by A. Doig, J.Wiley, London,1962]

2/. Cartwright,D.,Harary,F. and Norman ,R. : Structural Models. [John Wiley,New York, 1965].

3/. Cayley,A. : Collected Mathematical Papers. Cambridge 1889 - 1897. Vol. 3 p.242, Vol.9 p.202,p427, Vol.11 p.365, Vol.13, p.26,p.265.

4/. Konig,D. : Theorie der Endlichen und Unendlichen Graphen. [Leipzig, 1936.]

5/. Ore,O. : Theory of Graphs. [Amer. Math. Soc., Colloqium Publications, Vol. XXXVIII (1962)].

6/. ------ : Structures on Directed Graphs. [Annals of Mathematics, v.63 (1956) p.383].

7/. Riordan,J. : An Introduction to Combinatorial Analysis . [John Wiley , New York , (1958)]

8/. Trent,H.M. : A Note on the Enumeration and Listing of all possible Trees in a Connected Linear Graph. [Proc. Nat. Acad. Sciences, U.S.A. v.40 (1954)].

Chapter 2.

9/. Bott,R. and Mayberry,J. : Matrices and Trees. [Economic Activity Analysis,Wiley,N.Y.1954,p.391]

10/. Erdos,P.,Goodman,A.,Pasa,L. : Representation of Graphs. [Canad. Jr. of Math. v.20, (1966) p.106]

11/. Neville,E.H. : The codifying of tree structures . [Proc.Camb.Phil.Soc. v.49 (1953) p.381]

12/. Obruca,A.K. : An Investigation into Flat Rooted Trees. [M.Sc. Dissertation, Univ. of Durham (1963)]

13/. Okado,S. : Algebraic and Topological Foundations of Network Synthesis. [Proc.Symp.on Modern Network Synthesis , N.Y. ,1955, p.283]

14/. Perko,A. : Some Computational Notes on the Shortest Route Problem. [B.C.J. ,v.8 ,(1965) p.19]

15/. Solomon,E.W. : A Comprehensive Program for Network Problems. [B.C.J. v.3 (1961) p.89]

Chapter 3. 15/a. Barachet,L. : Graphic Soln. of the Travelling Salesman Problem. [Oper. Res. v. 5, (1957) p.841]

16/. Bellman,R. : Dynamic Programming Treatment of the Travelling Salesman Problem. [J.A.C.M. v.9 (1962)]

17/. Croes,G.A. : A Method for Solving Travelling Salesman Problems. [Oper.Res. v.6 (1958) p.791]

18/. Dantzig,G. et al : On a L.P. Combinatorial Approach to the Travelling Salesman Problem. [Rand Research Memorandum, RM - 2321 , (1959)]

19/. ------- : Solution of a Large Scale Travelling Salesman Problem. [Oper.Res. v.2 (1954) p.393]

20/. Flood,M. : The Travelling Salesman Problem. [Oper. Res. ,v.4 , (1956) p.61]

21/. Gilbert,E.N. : Random Minimal Trees. [S.I.A.M. v.13 , (1965) p.376]

22/. Hammersley,J. and Handscomb,D. : Monte Carlo Methods. [Methuen (Monographs) , London , 1964 , p.48]

23/. Held,M. and Karp,R. : A Dynamic Programming Approach to Sequencing Problems. [S.I.A.M. v.10 (1962) p.196]

24/. Heller,I. : The Travelling Salesman Problem. Part 1, Basic Facts. [An Introduction to L.P. by Charnes, Cooper and Mellon. , J.Wiley , 1953]

25/. ------- : On the Travelling Salesman Problem. [Proc. 2nd Symp. on L.P. (1955) NBS and HQ USAF]

26/. Karel,C et al : An Algorithm for the Travelling Salesman Problem. [Oper.Res. v.11 (1963) p.972]

27/. Kruskal,J.B. : On the Shortest Spanning Subtree of a Graph and a Travelling Salesman Problem. [Proc. Amer. Math. Soc. v.7 (1956) p.48]

28/. Kuhn,H.W. : The Travelling Salesman Problem.[Proc. of the VIth Symp. in Appl. Math. 1956 (edit.) Curtiss]

29/. Land,A.H. and Morton,G. : A Contribution to the Travelling Salesman Problem.[J.Roy.Stat.Soc.(B)v.17(1955)]

30/. Loberman,H. and Weinberger,A. : Formal Procedures for Connecting Terminals with a Minimal Total Wire Length. [J.A.C.M. v.4 , (1957) p.428]

31/. Miller,C., Tucker,A. and Zemlin,R. : Integer Programming Formulation of the Travelling Salesman Problem. [J.A.C.M. v.7 , no.4 (1960) p.326]

32/. Obruca,A.K. : Algorithm Mintree. [B.C.Bul. v.8(1964)]

33/. Robacker,J. : Some Experiments on the Travelling Salesman Problem. [Rand Report, 1955]

Chapter 4.

34/. Alway,G. and Martin,D. : An algorithm for reducing the Bandwidth of a Matrix of Symmetrical Configuration. [B.C.J. v.8 (1965) p.264]

35/. Braun,F.H. : Machine Analysis of Networks and its Applications. [I.B.M. Techn. Report, TR 00.855]

36/. Carre,B.A. : The Partitioning of Network Equations for Block Iteration. [B.C.J. v.9 (1966) p.84]

37/. Harper,L.H. : Optimal Assignements of Numbers to Vertices. [S.I.A.M. v.12 (1964) p.131]

38/. Livesley,R. : The Analysis of Large Structural Systems. [B.C.J. v.3 (1961) p.34]

39/. Porter,S. : The Use of Linear Graphs in Gauss Elimination. [S.I.A.M. Review, v.3 (1961) p.119]

40/. Sato,N. and Tinney,W. : Technique for the Exploiting the Sparsity of the Network Admittance Matrix. [I.E.E.E. Trans.on Power App. and Syst. N.69,p.944]

40/a. Livesley, R.K. : Matrix Methods of Structural Analysis. [Pergamon Press, 1964]

41/. Varga,R.S. : Matrix Iterative Analysis. [London, Prentice-Hall Int. (1962).]

42/. Wilson,L.B. : Solution of Certain Large Sets of Equations on Pegasus using Matrix Methods. [B.C.J. v.2 (1959) p.130]

42/a. Garwick, J. : Solution of a linear system with a band coefficient matrix [B.I.T. vol.3 (1963) p207].

Chapter 5.

43/. Beardwood,J.,Halton,J. and Hammersley,J. : The Shortest Path through many Points. [Proc. Camb. Phil. Soc. v.55 (1959) p.299]

44/. Bellman,R. : On a Routing Problem. [Quart.Appl. Math. XVI , no.1 (1958)]

45/. Busacker,R. and Saaty,T. : Finite Graphs and Networks: An Introduction with Applications. [McGraw Hill, 1965, p.58.]

46/. Clarke,S.,Krikorian,A. and Rausen,J.: Computing the N Best Loopless Paths in a Network.[SIAM v.11(1963)]

47/. Dantzig,G. : Discrete Variable Extremum Problems. [Oper. Res. v.5 (1957) p.266]

48/. Erdos,P. and Gallai,T. : On Maximal Paths and Circuits of a Graph. [Acta Math.Acad.Hung. v.10 (1959)]

49/. Farbey,B.,Land,A. and Murchland,J. : The Cascade Algorithm for finding the Minimum Distances on a Graph. [Private Communication, 1966]

50/. Hoffman, and Pavley : A Method for the Solution of the N Best Path Problem.[JACM v.6(1959)p.566]

51/. Land,A. and Stairs,S. : The Extension of the Cascade Algorithm to Large Graphs.[Private Comm.,1966]

52/. Minty,G.J. : A Comment on the Shortest Route Problem. [Oper. Res. v.5 (1957) p.724]

53/. Moore,E. : The Shortest Path through a Maze. [Proc. of an Int.Symp.on the Theory of Switching,1957] [The Annals of the Computation Laboratory of Harvard University, (1959) Harvard Univ. Press]

54/. Narahon,Pandit : The Shortest Route Problem, an Addendum. [Oper.Res. v.9 (1961) p.129]

55/. Pollack,M. and Wiebenson,W.: Solutions of the Shortest Route Problems: A Review. [Oper.Res. v.8 (1960) p.225]

56/. Prim,R.C. : Shortest Connection Matrix Network and Some Generalizations. [Bell Systems Tech.Jr, v.36 (1957) p.1389]

57/. Rapaport,H. and Abramson,P. : An Analog Computer for finding an Optimum Route through a Communucation Network. (1959) I.R.E. Trans.Comm.Syst.CS-7 p.37]

58/. Robbins,H.E. : A Theorem on Graphs, with an Application to a Problem of Traffic Control. [Amer. Math. Monthly, v.46 (1939) p.281]

59/. Shimbel,A. : Structures in Communication Nets.[Proc. of Symp.on Inform.Netwks.,Polyt.Inst.of Brooklyn(1954)]

60/. Verblunsky,S. : On the Shortest Path through a Number of Points. [Proc.Amer.Math.Soc. v.6 (1957) p.904]

60/a. Nicholson, T.A. : Finding the Shortest Route between two points in a Network. [British Joint Comp. Conf. Eastbourne, 1966].

Chapter 6.

61/. Gass,S. : Linear Programming Methods ans Applications. McGraw Hill, 1958 , p.137.

62/. Vajda,S. : Mathematical Programming. Addison Wesley. 1961, p.117.

APPENDIX 1.

App. 1.1 Trees.

When applying trees to solve or help solve a given problem , we find all three types : trees, r-trees and ρ-trees. Trees and r-trees are represented in the below array. This means that trees are made rooted by designating one of their points as the root . However if it is required to manipulate them (e.g. Toptree) then the r-trees have to be planarized or ordered and made into ρ-trees. This, as explained in Chapter 2 , is in order to be able to process them. The representation to be used will be the rd,lu one in all cases.

The rd,lu representation takes slightly longer to find the below of a point than does the below, posnbr representation. But the procedure which accomplishes this, is very simple and can be defined recursively as

```
integer procedure belnext(rd,a);
value a;  integer a;   integer array rd;
belnext:= if rd[a]<0 then -rd[a]
          else if rd[a]=a then a
               else belnext(rd,a);
```

Similarly positive neighbour can be defined recursively as (overleaf)

```
integer procedure  nbr next(rd,a);
value a;  integer a;  integer array  rd;
nbr next:= if rd[a]=a then  0
           else  if rd[a]>0 then  rd[a]
                 else  nbr next(rd,-rd[a]);
```

The rd,lu representation with the addition of the above two procedures can accomplish all that the below, pos nbr representation could, plus the ability of being able to go up the ρ-tree.

The most useful procedure written within this thesis is the next one to be defined and it gives us the means of analysing each point of the ρ-tree in turn. It is an integer procedure called process, which takes the value of a different point upon each call. The first time a boolean identifier is set true (and within the procedure it is reset false) and the call yields the root of the subtree specified by another identifier boot. That is, the procedure will take on all the values of the points of the subtree at v_{boot} , v_{boot} being the first to be presented. When all the points have been exhausted, the procedure becomes zero. The ordering, as can be seen from the procedure, is up left and across .

```
integer procedure  process(rd,lu,first,boot);  value boot;
integer boot;   boolean first;   integer array rd,lu;
begin   own integer b;
        if first then   begin   first:=false;
                                process:=b:=boot;
                        end
        else  process:=b:=if lu[b]>0 and lu[b]≠b then lu[b]
                          else  nbr next(rd,b);
end process;
```

As explained before, process will not yield a point, before the point below it has already been presented. Thus in Chapter 6 , this has been put to good use, when computing the shadow costs of each point. We let the shadow cost at the root equal zero and the ordering of the computation of the other shadow costs is by means of process . The procedure is reasonably efficient, however it will take slightly longer to obtain the points of a high ϱ-tree than those of a shorter ϱ-tree due to the more frequent use of recursive calls.

The last three procedures will be of general use to those who wish to manipulate ϱ-trees whilst using the rd, lu representation. The next few procedures are more specialized and within the thesis have been used in Chapters 3 (for which they were originally written) and 6.

Rather than store the generation number of each point within a ρ-tree and recomputing them if and when we alter the ρ-tree, a recursive procedure was written which gave the generation number of a point when called. This admittedly is slower, but if not used too often, justifies itself by the saving in space. The method is to go down the ρ-tree until the root is reached and in the process count the number of steps taken.

```
integer procedure gnrd(rd,a);
value a; integer a; integer array rd;
gnrd:= if rd[a]=a then 1 else gnrd(rd,belnext(rd,a))+1;
```

Given a ρ-tree, T_c, it may be required to cut it into two parts, T^1 and T^2, by deleting (c-d) where v_d will remain connected to the root of T_c, i.e. belong to the lower subtree T^1. The procedure written to accomplish this was called Treecut. The procedure Newtree, when given a ρ-tree rooted at v_c, alters its configuration so that it becomes rooted at v_b. Finally the procedure Fixtop joins or connects the two subtrees T^1 and T^2 at the points v_a of T^1 to v_b of T^2. All these three procedures are used in Chapter 3, and 6. ~~whereas Newtree is omitted in Chapter 5, Short-cut Routes 2, as we are not interested in changing the root of T, but merely in joining the root of T to a different point in T~~.

All three procedures use a procedure called topleft, which when given a point, v_α of T, will yield the top most left point of the subtree within which v_α is the root.

```
integer procedure  topleft(a,lu);
value a; integer a;  integer array lu;
begin   integer j,k;
    j:=a;
    for k:=lu[j] while k > 0 and j≠k do  j:=k;
    topleft:=j;
end topleft;


procedure  Treecut(c,d,rd,lu);    value c,d;
integer c,d;    integer array  rd,lu;
begin   integer h,j,k,l,t;
    k:=lu[d];
    l:=rd[c];
    t:=topleft(c,lu);
    if k≠c then     begin   for j:=k,rd[h] while j≠c do  h:=j;
                            rd[h]:=l;
                    end
```

```
    else    if d+1=0 then   lu[d]:=if lu[t]=t then d else lu[t]
        else    begin   j:=topleft(l,lu);
                        lu[j]:=if lu[t]=t then j else lu[t];
                        lu[d]:=l;
                end;
    rd[c]:=c;   lu[t]:=t;
end  Treecut ;


procedure  Fixtop(b,a,rd,lu);   value b,a;
integer b,a;    integer array  rd,lu;
begin   integer  u,v,w;
    v:=lu[a];
    u:=topleft(b,lu);
    if v<0 or v=a then
            begin     rd[b]:=-a;
                      lu[u]:= if v=a then  u else  v;
            end
    else    begin   rd[b]:=v;
                    w:=topleft(a,lu);
                    lu[u]:= if lu[w]=w then u else lu[w];
                    lu[w]:= -a;
            end;
    lu[a]:=b;
end  Fixtop;
```

```
procedure   Newtree(b,c,rd,lu);   value b,c;
integer  b,c;   integer array rd,lu;
begin   integer j,k,l;
        integer array  red[0:abs(genrd(rd,b)-genrd(rd,c))+2];

procedure  newrd(a);  value a;  integer  a;
begin   integer j,m;
    if  a=1 then
        begin   if lu[red[a+1]]≠red[a]  then
            begin   for j:=lu[red[a+1]],pd[j] while j≠red[a]
                        do m:=j;      rd[m]:=rd[rd[m]];
            end;
            rd[red[a+1]]:= if  lu[red[a]]>0  and

            lu[red[a]]≠red[a]  then  lu[red[a]] else  -red[a]
        end
    else   begin    if lu[red[a+1]]≠red[a] then
                begin   for j:=lu[red[a+1]],rd[j]
                        while  j≠red[a]  do   m:=j;
                        rd[m]:=rd[pd[m]];
                end;
                rd[red[a+1]]:=  if  lu[red[a]]≠red[a-1] then
                     lu[red[a]] else   if  rd[red[a-1]] > 0
                     then  rd[red[a-1]]  else   -red[a];
        end;
end newrd ;
```

```
procedure   newlu(a,k,l,b);  value  a,k,l,b;
integer  a,k,l,b;
if red[a-1]≠lu[red[a]] then
    begin   lu[topleft(red[a],lu)]:=if a≠k then -red[a]
                 else  topleft(red[a],lu);
            if a≠k then  lu[red[a]]:=red[a+1];
    end
else   if a=k and l>0  then
    begin   lu[topleft(l,lu)]:= topleft(l,lu);
            lu[red[a]]:=l;
    end
    else   if a=k then   lu[red[a]]:= red[a]
            else   lu[red[a]]:=red[a+];

    if  b≠c then
        begin   red[1]:=b;  for j:=2,j+1 while red[j-1]≠c do
            begin   red[j]:=belnext(rd,red[j-1]); k:=j; end;
                l:=rd[red[k-1]];    red[0]:=red[k+1]:=0;
                for j:=k-1 step -1 until 1 do   newrd(j);
                rd[red[1]]:=red[1];
                for j:=k step -1 until 1 do   newlu(j,k,l,b);
        end;
end Newtree;
```

The following 7 procedures are used to order a r-tree so that it may be manipulated by some of the previously described procedures. An ordering is defined on the points so that each point has a value, quite arbitrary of course, for the array pos nbr. This is done by means of the procedure Planarize. The representation is now altered into the rd, lu representation by means of the procedures Convrd and Convlu . An example of the use of these procedures is in Toptree . The spanning tree for a graph is obtained in a rooted form and before it can be manipulated by the procedures Treecut, Newtree, etc., the r-tree has to be ordered. We specify an ordering on the points of the spanning tree which does not affect the problem at all.

```
procedure   Planarize(bel,posnbr,z);  integer z;
    integer array  bel,posnbr;
comment  This procedure obtains the r-tree in the bel array
and transforms it into a ρ-tree given by the arrays  bel
and  pos nbr . ;
begin   integer j;     integer array carry[0:z];
    for j:=0 step 1 until z do  carry[j]:=0;
    for j:=1 step 1 until z do
    begin   posnbr[j]:=carry[bel[j]];
        carry[bel[j]]:=j;   end;
```

```
    for j:=1 step 1 until z do
          if bel[j]=0 then  bel[j]:=j;
end Planarize;

procedure  Convrd(bel,posnbr,z,rd);    value z;
    integer z;  integer array bel,posnbr,rd;
comment  this converts the (bel,posnbr) representation of
 the ρ-tree into the rd  representation. ;
begin  integer  j,root;
    for j:=1 step 1 until z do
        if bel[j]=j then root:=j;
    for j:=1 step 1 until z do
    rd[j]:= if posnbr[j]≠0 then  posnbr[j] else
        if j=root  then  j  else -bel[j];
end Convrd ;

procedure  Convlu(bel,posnbr,z,lu);   value z,bel,posnbr;
    integer z;   integer array bel,posnbr,lu;
comment  this converts the (bel, posnbr) representation of
 theρ-tree into the lu representation ( again losing the
ordering or planar property of the original ρ-tree ). ;
begin  integer j,root;  integer array  nab,nsuc[1:z];
    Na(bel,posnbr,z,nab);        Refl(bel,posnbr,z);
    Succes(bel,posnbr,z,nsuc);
```

```
    for j:=1 step 1 until z do   if bel[j]=j then root:=j;
    for j:=1 step 1 until z do
    lu[j]:= if nab[j]≠0 then nab[j]
    else  if  nsuc[j]=root  then j  else  -bel[nsuc[j]];
end  Convlu;

procedure  Na(bel,posnbr,z,nab);      integer z;
           integer array  bel,posnbr,nab;
comment  this procedure finds the most negative or left
above for every point  j , placing the value in nab[j].
(In fact nab[j] is equivalent to  above(j) for the  v_j
of the ρ-tree.) If the point has no negative above, i.e.
 is an end point then nab is put equal to zero. ;
begin   integer j,k,l;
    for l:=1 step 1 until z do
    begin   nab[l]:=0;
        for k:=1 step 1 until z do
        if (bel[k]=l) and (k≠l)  then
        begin   for j:=2,j+1 while z≥j do
            if posnbr[j]=k then
            begin   k:=j;  j:=1;  end;
            nab[l]:=k;  goto Na1;
        end;
   Na1: end;
end Na ;
```

```
procedure   Refl(bel,posnbr,z);    integer z;
                    integer array  bel,posnbr;
comment  this computes the reflected tree of  (bel,posnbr)
placing the new ρ-tree in the same arrays bel,posnbr. The
reflected ρ-tree of a given ρ-tree is the one which has the
same belows but has opposite sign convention. That is the
positive neighbours of the original ρ-tree become negative
neighbours in the new one. ;
begin  integer j,k;   integer array rn[1:z];
   for j:=1 step 1 until z do rn[j]:=0;
   for j:=1 step 1 until z do
      for k:=1 step 1 until z do
         if k=posnbr[j] then rn[k]:=j;
   for j:=1 step 1 until z do  posnbr[j]:=rn[j];
end  Refl;

procedure  Succes(bel,posnbr,z,suc);  integer z;
                 integer array bel,posnbr,suc;
comment  this computes the positive successor of each
point placing the values in suc[i] for i=1...z. ;
begin  integer j,k;
for j:=1 step 1 until z do suc[j]:=0;
for j:=1 step 1 until z do
    if bel[j]=j then
```

```
    begin  suc[j]:=j;
        for k:=1 step 1 until z do
            suc[k]:=su(bel,posnbr,suc,k);
    end
end Succes;


integer procedure   su(bel,posnbr,suc,a);    value a;
        integer a;   integer array  bel,posnbr,suc;
comment  this evaluates the positive successor of a point
a recursively. On coming out of this procedure any inter-
mediate successors that  have been computed are retained
 and assigned to the corresponding identifiers suc[i] ;
if suc[a]≠0 then su:=suc[a] else
if nbr[a]≠0 then su:=nbr[a] else
su:=suc[a]:=su(bel,posnbr,suc,bel[a]);
```

APPENDIX 2.

App.2.1 Toptree Program.

We define three further procedures to be used in the Toptree program for the Travelling Salesman Problem. One is called Free ends which when applied to a ρ-tree , inserts in an array ends[1:z] the labels of all the end points of that ρ-tree . This procedure is used when T_ℓ has been divided into two parts or subtrees by the deletion of a line . We find the best line with which to join the two parts by means of the procedure Join pts . The procedure branches count is used to find the degree of a point.

```
procedure  Free ends(rd,lu,ends,root,z,k);
integer root,z,k;  integer array  rd,lu,ends;
begin   integer j,c; boolean first;
    first:=true;
    for j:=1,j+1 while ends[j-1]≠0 do
    begin   k:=j;
        for c:=0, if c≠0 then c else topleft(root,lu)
            while lu[c]>0 and lu[c]≠c   do
            c:=ends[j]:=process(rd,lu,first,root);
    end;
```

```
    if  rd[lu[root]]=-root  then  ends[k]:= root
        else  k:=k-1;
end Free ends ;

procedure  Join pts(cost,ends1,n1,below cut,ends2,n2,
above cut,a,b,dist);   integer n1,n2,a,b,dist,above cut,
below cut;   integer array  cost,ends1,ends2;
begin   integer  j,k;
    dist:=10;
    for j:=1 step 1 until n1 do
    for k:=1 step 1 until n2 do
    if  (cost[ends1[j],ends2[k]]<dist) and
        (below cut≠ends1[j] or above cut≠ends2[k])
    then    begin   a:=ends1[j];
                b:=ends2[k];
                dist:=cost[a,b];
            end;
end; Join pts;

integer procedure  br count(j,rd,lu);  value j;
integer j;   integer array  rd,lu;
comment  This evaluated  d(v_j), the degree of the point j.
In other words branches count takes on the value of the num-
ber of lines incident to the point j. ;
```

```
begin   integer l;
    l:=0;
    if  lu[j]>0 and lu[j]≠j then
        for j:=lu[j],rd[j] while j>0 do  l:=l+1;
    br count:=l+1;
end br count;
```

There is one more point to note. The program will always test if the root of T_ℓ is an end point. If not, it assigns an end point of T_ℓ as the root point. This is to ensure that all multi-membered packages of T_ℓ are also multi-membered stars. The test is first made when mintree is input (i.e. $T_\ell = T_m$) and thereafter, every time the ρ-tree T_ℓ is divided into two parts and rejoined together. The program, apart from the previously mentioned procedures, follows next. The section of program up to the label LO is concerned with the input of data. The label PROCESS deals with the division and rejoining of T_ℓ . PRINT is concerned with output of better solutions as they are found. The procedures Time and time are given in App. 3.1 as they were written for the programs in that section.The rest of the program is concerned with data manipulation.

```
begin   library A0,A6;   integer z,count,data nr;
pen(20);  open(30);
write text(30,[[pccc]DHCL21/TOPTREE***1[ccc]]);
data nr:=read(20);
START:  z:=read(20);
begin   integer j,k,a,b,dist,nr1,nr2,root,above cut,
        below cut,pt,n,pa,neg,rem,total,min total;
    integer array cost[1:z,1:z],rdd,luu,ends1,ends2,
        s,rt,cut,distance[1:z],up,p[0:z],lu,rd[1:z,1:z];

        ( All the necessary procedures are declared here,
        i.e. all the ones in App. 1.1 up to Newtree and
        Join pts, Free ends, branches, Time and time. )

for j:=2 step 1 until z do
for k:=1 step 1 until j-1 do
    cost[j,k]:=cost[k,j]:=read(20);
total:=read(20);
writetext(30,[[8c]MIN***TREE***TOTAL**--]);
write(30,format([nddd]),total);
for j:=1 step 1 until z do
begin   rdd[j]:=read(20);
    luu[j]:=read(20);  end;
root:=read(20);
```

```
if  rdd[luu[root]]≠ -root then
begin   rem:=topleft(root,luu);
        Newtree(rem,root,rdd,luu);
        root:=rem;   end;
emit:=read(20);   Time(emit,L0,L0);     timer:=0;
write text(30,[[cccc]COUNT*****TOTAL*************
    *****ROUTE[ccc]]);  count:=n:=0;  min total:=10;

L0:   pt:=root;

L1:   pt:=luu[pt];

L2:   if rdd[luu[pt]]≠-pt  then   else
        if luu[luu[pt]]=luu[pt] then   goto  PRINT
            else  goto L1;
n:=n+1;   pa:=0;
for j:=luu[pt],rdd[j]  while j>0  do   pa:=pa+1;
up[n]:=1;   p[n]:=pa;
for j:=1 step 1 until z do
begin   rd[n,j]:=rdd[j];
    lu[n,j]:=luu[j];  end;
s[n]:=pt;   rt[n]:=root;   cut[n]:=luu[pt];
distance[n]:=total;  below cut:=pt;  above cut:=luu[pt];
neg:=cost[below cut,above cut];
```

```
PROCESS:
Time(0,FIN,FINISH);
Tree cut(above cut,below cut,rdd,luu);
Free ends(rdd,luu,ends1,root,z,nr1);
Free ends(rdd,luu,ends2,above cut,z,nr2);
Joinpts(cost,ends1,nr1,belowcut,ends2,nr2,abovecut,a,b,dist);
Newtree(b,above cut,rdd,luu);
Fixtop(b,a,rdd,luu);
total:=total+dist-neg;
if rdd[luu[root]]≠-root  then
begin   rem:=topleft(root,luu);
        Newtree(rem,root,rdd,luu);
        pt:=root:=rem;
end;
goto L2;

PRINT:
count:=count+1;
total:=total+cost[root,topleft(root,luu)];
if total<min total then
begin   write text(30,[[c]]);
        min total:= total;
        write(30,format([ndddd]),count);
        write(30,format([4sndddd]),total);
         write text(30,[[6s]]);
```

```
        for j:= root,luu[k] while j≠k  do
        begin   k:=j;
            write(30,format([nddd]),j);
        end;
end;

L3:   if n=0 then goto  FINISH;
if up[n]>p[n]  then
begin   n:=n-1;
    if n≤0 then goto FINISH  else  goto L3;
end
else
begin   for j:=1 step 1 until z do
    begin   rdd[j]:=rd[n,j];
        luu[j]:=lu[n,j];  end;
    root:=rt[n];  pt:=s[n];
    if  rdd[cut[n]]<0 then
    begin   above cut:=s[n];
        pt:=below cut:=belnext(rdd,s[n]);
    end
    else   begin    below cut:=s[n];
                cut[n]:=above cut:=rdd[cut[n]];
            end;
    up[n]:=up[n]+1;
    neg:=cost[below cut,above cut];
```

```
    total:=distance[n];
    goto  PROCESS;
end;

FIN:    emit:=read(20);   Time(emit,L0,L0);
timer:=timer+time;
writetext(30,[[ccc]TIME**TAKEN**SO**FAR*---]);
write(30,format([ndd.d]),timer/60);
writetext(30,[**MINUTES.]);
writetext(30,[[cc]COUNT**SO**FAR**--]);
write(30,format([ndddd]),count);
end;

FINISH:  write  text(30,[[cccc]END***OF***PROGRAM[10s]]);
writetext(30,[TOTAL**TIME**TAKEN**IN**MINS**--]);
write(30,format([nddd.d]),timer/60);
write text(30,[TOTAL**COUNT***--]);
write(30,format([ndddd]),count);
data nr:=data nr-1;
if data nr = 0 then   else goto  START;
close(30);  close(20);
end→
```

The maximum storage used is of the order $3 \times z^2 + 10 \times z$. If we know the number of lines which lie in multi-membered packages of mintree , we can reduce the storage declaration . Suppose there are n lines which lie within multi-membered packages (i.e. $n = (z - 1) -$ (number of one-membered packages)). Then the number of arrays rdd and luu can be reduced from z to n. The storage required is of the order $(z^2 + 10 \times z + 2 \times z \times n)$ where $n < z$. In a large number of cases n is half z giving us a storage limit of the order $2 \times z^2 + 10 \times z$.

App. 2.2 Data for Input.

The data for input has to be in the following format :-

data nr ;	This is the number of blocks of data to follow.
z ;	A block of data starts from here and z is the number of points in it.
c_{21} ;	
c_{31} ; c_{32} ;	Now follows the lower triangle of
. . . . .	the cost matrix.
c_{z1} ; c_{z2} ; ... ; c_{zz} ;	
total ;	Cost of mintree .
rd[1] ; lu[1] ;	
rd[2] ; lu[2];	This is the rd, lu representation
. . . . .	of mintree.
rd[z] ; lu[z] ;	
root ;	(of mintree) if it is unknown just insert 1.
interupt duration;	This is the time in seconds for the procedure Time and also indicates the end of a block of data.

→→→

App. 2.3 Random Data Preparation and Mintree.

A short program, called Steering Program, was written which prepared random data and processed it into a form suitable for input into Toptree. As mentioned in Chpt. 3.7 , the z points are scattered at random onto a rectangle with sides of length 99 units. The distances between all pairs of points is calculated and inserted into the cost matrix . The second part of the program applied the procedure Min tree to the resulting cost matrix and described it in the below representation. This was manipulated by the third part of the program, so as to obtain the tree in the rd,lu representation. All relevant information is output on paper tape in a suitable format for re-input into Toptree. The procedure Mintree as published in BCJ by the author is named here Min tree 1 .

```
begin   integer z;
library 1;   open(30);  open(20);   open(10);
L:  write text(30,[[p]DHCL21******STEERING***PROGRAM[cccc]]);
z:=read(20);      write(10,format([nddd;c]),z);
begin   integer j,k,total;   integer array   cost[1:z,1:z],
        below[0:z],link to,link from,rd,lu,x,y[1:z];
    ( The following procedures are now declared: Planarize,
        Convlu,Convrd, Refl, Na, su, Success and random )
```

```
procedure   Min tree1(cost,z,below,total);
integer z,total;  integer array  cost,below;
begin   integer m,n,a,b,s,count;
integer procedure  root(a);  value a;  integer a;
    root:= if below[a]=0 then a else  root(below[a]);
procedure   change(a,b);   value a,b;  integer a,b;
begin   integer  m;
    for m:= if a=0 then b else below[a] while a≠0 do
    begin   below[a]:=b;  b:=a;  a:=m;  end;
end change;

    for m:=1 step 1 until z do  below[m]:=0;
    count:=total:=0;
A:  s:=₁₀10;
    for m:=2 step 1 until z do
    for n:=1 step 1 until m-1 do
        if  cost[m,n]<s then
        begin   s:=cost[m,n];  a:=m;  b:=n;  end;
    if  root(a)≠root(b) then
    begin   count:=count+1;  change(a,b);
                total:=total+s;
        if count=z-1  then goto  FINISH;
    end;
    cost[a,b]:=₁₀10;   goto  A;
FINISH:  end Min tree1;
```

```
j:=random(1,100,read(20));
for j:=1 step 1 until z do
begin   x[j]:=random(1,100,0);
    write(30,format([8snddd]),x[j]);
    y[j]:=random(1,100,0);
    write(30,format([ndddd]),y[j]);
    if j=5 or j=10 or j=15 then write text(30,[[c]]);
end;
write text(30,[[6c]]);
for j:=2 step 1 until z do
begin   write text(30,[[c]]);  write text(10,[[c]]);
    for k:=1 step 1 until j-1 do
    begin   cost[j,k]:=cost[k,j]:=
            sqrt((x[j]-x[k])↑2+(y[j]-y[k])↑2);
        write(30,format([nddddd]),cost[j,k]);
        write(10,format([ndddd;]),cost[j,k]);
    end;
end;
Min tree1(cost,z,below,total);
write text(30,[[6c]LINKS***OF**THE**TREE[ccc]]);
for j:=1 step 1 until z do
begin   write(30,format([ndddddd]),j);
    write(30,format([4snddc]),below[j]);
end;
```

```
    write text(30,[[ccc]TOTAL**--]);
    write text(10,[[cccc]]);
    write(30,format([ndddddc]),total);
    write(10,format([ndddd;cc]),total);
    begin   integer array   nbr[1:z];
        Planarize(below,nbr,z);
        Convlu(below,nbr,z,lu);
        Convrd(below,nbr,z,rd);
        for j:=1 step 1 until z do
        begin   write(10,format([-ndddddd;]),rd[j]);
            write(10,format([5s-nddd;c]),lu[j]);
        end;
        charout(10,61);     gap(10,50);
        for j:=1 step 1 until z do
        if rd[j]=j then  write(10,format([ndd;c]),j);
        charout(10,61);
    end;
end;   close(30);   close(20);   close(10);
end→
```

The data for input is as follows: z (the number of points); random (an eleven digit random number to start the random number geerator); →

App. 2.4 Dynamic Programming Program.

This is the dynamic programming program mentioned in Chpt. 3.6 . The data format ready for input is as follows :

data nr ;	Number of blocks of data.
z ;	Number of points in graph.
c_{21} ;	Lower triangle of the cost matrix.
c_{31} ; c_{32} ;	
. . . . .	
c_{z1} ; c_{z2} ; ... ; $c_{z\overline{z-1}}$;	
→→	

The complete program is as follows.

```
begin   integer n,count;
integer procedure  fact(n);  value n;  integer  n;
begin  integer j,k;  k:=1;
    for j:=1 step 1 until n do k:=k×j;
    fact:=k;
end  fact;

integer  procedure  C(n,m);  value n,m;  integer  n,m;
    C:=fact(n)÷(fact(n-m)×fact(m));
```

```
library 1;  open(20);;  open(30);;
write text(30,[[pcc]DHCL021[16s]DYNAMIC***PROGRAMMING[c]
[14s]FOR***THE***TRAVELLING***SALESMAN***PROBLEM[ccccc]]);
count:=read(20);

START : n:=read(20);  write text(30,[[cccc]]);
begin   integer  j,k,l,m,v,w,order,tally,note,rem,charge,top;
    integer array   cost[1:n,1:n],route,vector[1:n],
            total[1:2,1:(n-1)×C(n-2,(n-2)÷2)],
            perm[1:4,1:(n-1)×C(n-2,(n-2)÷2)];

procedure   Sequence(produce,n,m,vector,order);   value  n,m;
integer  n,m,order;   integer  array  vector;  boolean produce;
begin  integer  s,j,t,ordersub,z;
    integer procedure  sum(s,p,k,bool);   value s,p,k,bool;
        integer s,p,k;   boolean bool;
    begin   integer total,l;
        total:=0;  if p=m-1  then  total:= k
        else    if  k≠0  then
            begin   for l:= if bool then s else k
                step -1  until  if bool then s-k+1 else 1 do
                    total:=total+sum(s,p+1,l,false);
            end;
        sum:= total;
    end sum;
```

```
if  produce  then
begin   ordersub:=order;    vector[1]:=1;    s:=n-m+2;
    for  j:=2 step 1 until m-1 do
    begin   vector[j]:=0;
        for t:= s, t-1  while  vector[j]=0  do
        begin   z:= sum(s,j,t,true)×(n-m+1);
            if ordersub > z  then
            begin   vector[j]:= vector[j-1] + t + 1;
                ordersub:=ordersub - z;
                s:=s-t;
            end;
        end;
    end;
    vector[m]:=50;
    for j:=2,j+1 while  vector[m]=50 do
    if  vector[j]-vector[j-1]>1 then
    begin   z:=vector[j] - vector[j-1];
        if z > ordersub  then
        vector[m]:=ordersub + vector[j-1]
        else ordersub:=ordersub - z+1;
    end;
end   else
begin   order:=0;  t:=1;  s:=n-m+2;
    forj:=2 step 1 until m-1 do
```

```
    begin   z:=vector[j] - vector[j-1] -1;
        order:= order + sum(s,j,z,true)×(n-m+1);
        if  vector[m]>vector[j]  then  t:=t+1;
        s:=s-z;
    end;
    order:=order + vector[m] - t;
end;
end  Sequence;

procedure   Permutate;
begin   integer  rem;
    if m ≤ 6  then  perm[v,order]:=perm[w,tally] ×
        100 + vector[m]   else
    begin   rem:=perm[w,tally] + 100000 00000;
            perm[v,order]:=(perm[w,tally] -100000 00000× rem)
                 × 100 + vector[m];  perm[v+2,order]:=
            perm[w+2,tally] × 100 + rem;
    end;
end   Permutate;

    for j:=2 step 1 until n do
        for k:=1 step 1 until j-1 do
    cost[j,k]:=cost[k,j]:=read(20);
    for j:=1 step 1 until  (n-1)×C(n-2,(n-2)+2)  do
        for k:=1 step 1 until 4 do  perm[k,j]:=0;
```

```
v:=1;   w:=2;
for j:=2 step 1 until n do
begin   total[v,j-1]:=cost[1,j];
    perm[v,j-1]:=100+j;  end;
for m:=3 step 1 until n do
begin   v:=if v=1 then 2 else 1;
    w:=3-v;   top:=(n-1)×C(n-2,m-2);
    for  order:=1, order+1  while  order ≤ top  do
    begin   Sequence(true,n,m,vector,order);
        total[v,order]:= 1 00000 00000;
        for l:=2 step 1 until m-1 do
        begin   route[m-1]:=vector[l];  k:=1;
            for j:=k while k< l, k+1 while k< m-1 do
            begin   route[k]:=vector[j];
                k:=k+1;   end;
            Sequence(false,n,m-1,route,tally);
            charge:=total[w,tally]+cost[vector[l],vector[m]];
            if  charge < total[v,order]   then
            begin   total[v,order]:=charge;
                    Permutate;   end;
        end;
    end;
end;
```

```
    charge:= 1 00000 00000;
    for j:=1 step 1 until n-1 do
    if  charge > total[v,j] + cost[n-j+1,1]   then
    begin   charge:=total[v,j] + cost[n+1-j,1];
        note:=j;
    end;
    write text(30,[[cc]TOTAL**---]);
    write(30,format([ndddddddc]),charge);
    write text(30,[[cc]ROUTE**---]);
    for j:=1 step 1 until n do
    begin   if j > 6   then  v:=v+2;
        rem:=(perm[v,note]+ 100) × 100;
        route[j]:=perm[v,note] - rem;
        write(30,format([ndddd]),route[j]);
        perm[v,note]:=perm[v,note] + 100;
        if j> 6 then  v:=v-2;
    end;
    count:=count-1;
    if count≠0 then goto START;
end;   close(20);   close(30);
end→
```

App. 2.5 Specimen Output.

This was the Toptree output for the (20 x 20) Croes Problem. The lower cost triangle is given below followed by the output on the next page.

29;
41; 72;
9; 72; 70;
18; 50; 54; 60;
6; 39; 35; 20; 17;
42; 60; 59; 24; 74; 26;
48; 34; 88; 73; 93; 60; 97;
74; 25; 19; 79; 0; 32; 65; 63;
43; 46; 72; 51; 76; 63; 64; 27; 71;
51; 25; 87; 43; 30; 84; 13; 42; 91; 66;
7; 35; 38; 58; 55; 21; 23; 62; 5; 30; 9;
36; 14; 24; 4; 84; 26; 3; 32; 85; 57; 26; 86;
93; 20; 68; 47; 42; 96; 78; 20; 51; 8; 6; 27; 12;
58; 35; 63; 29; 47; 75; 15; 26; 72; 71; 99; 34; 28; 19;
11; 83; 80; 22; 91; 14; 30; 5; 53; 19; 33; 72; 24; 77;
51; 27; 58; 48; 21; 13; 56; 80; 8; 25; 8; 45; 60; 14;
61; 86; 40; 27; 59; 51; 22; 52; 49; 10; 99; 59; 19; 22;
30; 95; 89; 88; 24; 16; 13; 47; 90; 83; 92; 32; 12; 54;
44; 30; 24; 91; 80; 83; 58; 36; 39; 40; 31; 77; 20; 77;

22;
75; 62;
28; 79; 91;
72; 97; 59; 87;
64; 47; 75; 4; 31;

DHCL021/TOPTREE 1

MIN TREE TOTAL -- 154

COUNT	TOTAL	ROUTE									
1	347	19	5	9	3	20	18	10	14	11	17
		6	4	1	12	2	13	7	15	8	16
2	286	12	19	5	9	3	20	18	10	14	11
		17	6	4	1	16	8	15	7	13	2
5	278	2	8	16	15	7	13	19	5	9	3
		20	18	10	14	11	17	6	4	1	12
8	277	8	16	15	7	12	1	4	6	17	11
		14	10	18	20	3	9	5	19	13	2
135	269	2	13	4	15	7	19	5	9	3	20
		18	10	14	11	17	6	12	1	16	8
141	260	8	16	15	7	19	5	9	3	20	18
		10	14	11	17	6	12	1	4	13	2

END OF PROGRAM COUNT -- 1728

Count is the iteration number within which the improved solution was found. The count number after the words END OF PROGRAM indicates the number of solutions within S_m which the program analysed.

APPENDIX 3.

The program for the bandwidth minimisation is basically in Algol but a little Usercode (KDF9 Machine language) is used to facilitate the segmentation and the relabelling of magnetic tapes. The three segments of the program correspond to the three Stages (described in Chpt. 4). At the finish of each segment , all data which is relevant to the next segment is written onto a magnetic tape (initially zero but labelled AKPOXMBM the first time the program is called). Due to the inadequecy of the KDF9 Algol compiler, the magnetic tape transfer procedures are slightly more complicated than they have to be , because all the integer arrays have to be transferred to real arrays and then copied to the magnetic tape. There are a few input and output procedures which are common to all three segments and they, with a few others, will be described now.

```
procedure  Mag read 1(dv);   value dv;  integer dv;
comment  This is one of the first instructions to be obeyed
during Segments  2 and 3 . This establishes the size of the
various arrays to be declared and read from the magnetic
tape (AKPOXMBM) . ;
begin   readbinary(dv,mag,[A]);
    datanr:=mag[1];      rm:=mag[2];
```

```
    emit:=mag[3];          tt:=mag[4];
    bd:=mag[5];            bounds:=mag[6];
    ld:=mag[7];            z:=mag[8];
    br:=mag[9];            b:=mag[10];
    bstop:=mag[11];        ma:=mag[12];
    mb:=mag[13];           mc:=mag[14];
end  Mag read 1;

procedure next segment(v);    value v;   integer v;
comment  This procedure is used to jump to the next
segment. Thus the last instruction obeyed in Segment 1
is this procedure with v set equal to 2. At the end of
the next segment  v  is set equal to 3 and in the last
one it is set back to  1 . ;
KDF9 3/0/0/0;
    V0=B 20 36 36;
1;  J2EN; ERASE; J1;
2;  J3EJ; LINK; ERASE; J2;
3;  E3; SHC+12; SHL+6; [v]; OR; SHL+12;
     V0; OR; SHC+18; E2; SET1; OUT;
ALGOL; comment end of next segment;


procedure  Mag read 2(dv);  value dv;  integer dv;
comment  Having established the size of the various arrays
```

by means of Mag read 1 we now read down the rest of the data by means of this procedure.;

```
begin   integer j,k,b,c;
    procedure arnext(array);  integer array  array;
    begin   if b=50 then  readbinary(dv,mag,[A]);
            b:= if b=50 then 1 else b+1;
            array[k]:=mag[b];  end;

    for j:=1 step 1 until 3 do
    begin   b:=50;  c:= if j=1 then br else z;
        for k:=1 step 1 until c do
            if j=1 then  arnext(branches)
            else  if j=2 then arnext(rowstart)
                else  arnext(nrinrow);
    end;
end  Mag write2;
```

procedure Mag write(dv); value dv; integer dv;
comment One of the last few instructions to be obeyed before the end of a segment is this procedure which writes up onto the magnetic tape all the information necessary to its functioning. This includes the branches, rowstart and nrinrow array. ;

```
begin   integer j,k,b,c;
    procedure  idnext(array);  integer array array;
```

```
    begin   b:= if b=50 then 1 else b+1;
        mag[b]:=array[k];
        if b=50 or k=c then  writebinary(dv,mag,[A]);
    end;
    b:=0;        for j:=datanr,rm,emit,tt,bd,bounds,ld,z,
    br,b,bstop,ma,mb,mc  do
    begin   b:=b+1;  mag[b]:=j;   end;
    for j:=b+1 step 1 until 50 do  mag[j]:=0;
    writebinary(dv,mag,[A]);
    for j:=1,2,3 do
    begin   b:=0;   c:= if j=1 then br else z;
        for k:=1 step 1 until c do
            if j=1 then idnext(branches)
            else  if j=2 then idnext(rowstart)
                else idnext(nrinrow);
    end;
end Mag write;

procedure  Matrix Output 1(branches,rowstart,nrinrow,z,br,
density,b,bstop,dv);   integer  z,br,density,b,bstop,dv;
integer array  branches,rowstart,nrinrow;
comment  This procedure is used whenever it is required to
output any details of the graph. This is done usually at
the end of every segment. ;
```

```
begin   integer j,k;
    if dv=30 then  writetext(30,[[c]**Z******BR*******
        DENSITY*****B******B*STOP[cc]]);
    write(dv,format([ndd;]),z);
    write(dv,format([ndddddd;]),br+2);
    write(dv,format([9snd;]),density);
    write(dv,format([ndddddd;]),b);
    write(dv,format([ndddddd;cc]),bstop);
    if dv=30 then  writetext(30,[[cc]ROW***NR*IN*ROW*
                ********MATRIX[c]]);
    for j:=1 step 1 until z do
    begin   newline(dv,if j-1=10×((j-1)÷10) then 2 else 1);
        if dv=30 then write(30,format([nd;]),j);
        write(dv,format([ndddddd;]),nrinrow[j]);
        writetext(dv,[[ssss]]);
        for k:=0 step 1 until nrinrow[j]-1 do
        write(dv,format([nddd;]),branches[rowstart[j]+k]);
    end;
end  Matrix Output 1;

procedure  Matrix Input(branches,rowstart,nr in row,z,br);
integer z,br;   integer array  branches,rowstart,nr in row;
comment  This is a standard input procedure used not only
in this program but in others. It accepts the matrix in
```

the branches list form and the layaout is as described in Chpt. 2.21 .;

```
begin   integer s,j,k;
    s:=1;
    for j:=1 step 1 until z do
    begin   nr in row[j]:=read(20);
        rowstart[j]:=s;
        for k:=1 step 1 until nr in row[j] do
        begin   branches[s]:=read(20);
                s:=s+1;   end;
    end;
end  Matrix input;

procedure  Perm branches(branches,rowstart,nrinrow,S,br,z);
integer  br,z;
integer array  branches,rowstart,nrinrow,S;
comment  Given a matrix in the branches list representation
and a permutation of  L(z)  in the array  S , this proce-
dure will permutate the rows and columns of the matrix cor-
respondingly (i.e. pre and post-multiply the equivalent ad-
jacency matrix by the vector corresponding to the array S).;
begin   integer j,s,k;   integer array copy branches[1:br],
    inv S,inv rowstart,inv nrinrow[1:z];
```

```
    for j:=1 step 1 until z do
    begin   inv S[S[j]]:=j;
        inv rowstart[j]:=rowstart[S[j]];
        inv nrinrow[j]:=nrinrow[S[j]];
    end;
    for j:=1 step 1 until br do
        copybranches[j]:=inv S[branches[j]];
    s:=1;
    for j:=1 step 1 until z do
    begin   nrinrow[j]:=inv nrinrow[j];
        rowstart[j]:=s;
        for k:=inv rowstart[j] step 1 until
        inv rowstart[j]+nrinrow[j]-1 do
        begin   branches[s]:=copy branches[k];
                s:=s+1;     end;
    end;
end  Perm branches;

procedure  Time(a,L1,L2);  value a;  integer a;  label L1,L2;
comment  This time control procedure is of great help in
circumstances where it is impossible to estimate how long
a program is going to take for some set of data. Into  a
is inserted, when this procedure is called the first time,
the time interval (in seconds) between monitor typewritter
```

queries. To continue the program the reply is 0.→ . If it is required to jump to the label L1 then a reply of 1.→ is called for and similarly a reply of 2.→ makes the program jump to label L2 .;

```
KDF9  2/2/0/9;
    V3=B 02 07 64 51 55 45 00 00;
    V4=B 61 65 45 62 71 00 00 34;
    V6=Q 0/AV3/AV5;
    V7=B 37 75 00 00 00 00 00 20;
    V8=B 02 00 00 41 47 41 51 56;
    V9=Q 0/AV8/AV8;
[a]; SHA24; DUP; J1=Z; DUP; =V2; =V1; SET9; OUT; =V0; EXIT;
1;  ERASE; SET9; OUT; V0; -; V2; -; J2≥Z; EXIT;
2;  V2; V1; +; =V2; V6; =Q15; V9; =Q14;
3;  TWEQ15; V5; SHC6; V7; -; ZERO; J4≠;ERASE; EXIT;
4;  SET1; J5≠; ERASE; J[L1];
5;  SET2; -; J6=Z; TWQ14; VR; J3;
6;  J[L2];
ALGOL;
```

```
integer procedure time;
comment The first time this procedure is called it notes
the time on the 24 hour clock and it will take on the value
of the total number of seconds which have elapsed since its
```

last call. ;

<u>KDF9</u> 3/0/0/0;

V0=0;

SET9; OUT; DUP; V0; CAB; =V0; -; SHA-24; <u>EXIT</u>;

<u>ALGOL</u>;

App. 3.1 Segment 1.

The first segment includes Stage 1, input procedures and a facility to generate random data. The main procedure is called Ld comp . This is a control procedure which is called once for each new set of data. By means of Gr inspect, it finds the next point to be removed from the graph (Chpt. 4.43) and accomplishes this by means of Cut point . The resultant graph is then rearranged ready for the next iteration by means of Gr rearrange, Gr add and Gr subtract. At the start of each iteration E(resultant graph) is computed by means of Ld calc . The procedures used will now follow each having a brief comment to describe their workings.

```
procedure Ld calc(rem); integer rem;
comment Given the graph in (bbranches, etc.) the procedure Min g roots finds the maximum path length and all pairs of points yielding this value . Next for each pair of these points, the procedure Loop count is entered to find the number of distinct paths between them. We thus have the values of all the variables in formula ( 4.4 A ) . The value of E(G') is inserted in the identifier rem. ;
begin integer loop,j,k,sum,b,d,g,n,m;
    integer array ends1,mod[1:zz+1],ends2[1:bounds];
```

```
    Min g roots(bbranches,rrowstart,nnrinrow,zz,bbr,ends1,
    ends2,mod,b,d,g);   loop:=-1;
    for j:=1 step 1 until b do
        for k:=mod[j] step 1 until mod[j+1]-1 do
    begin   Loop count(bbranches,rrowstart,nnrinrow,zz,bbr,
    sum,ends1[j],ends2[k],n,m);   if sum>loop then loop:=sum;
    end;
    rem:=entier((zz+loop+m-2)/m);
end  Ld calc;

procedure  Min g roots(branches,rowstart,nrinrow,z,br,ends1,
ends2,mod,b,d,g); integer z,br,g,b,d;  integer array branches,
rowstart,nrinrow,ends1,ends2,mod;
comment  This uses the mushrooming r-tree technique as des-
cribed in Chpt. 4.41. in order to find the root and highest
point of all the mushrooming r-trees with maximum height.
These two sets of points are stored in the integer arrays
 ends1 and ends2 . ;
begin   integer gee,a,s,olds,nr,c,j,k,root;
    integer array gen,list[1:z];
    for j:=1 step 1 until z do  ends1[j]:=ends2[j]:=0;
    g:=0;     d:=nr:=b:=0;
    for root:=1 step 1 until z do
    begin   for j:=1 step 1 until z do gen[j]:=0;
```

```
        a:=s:=1;  olds:=gee:=0;  list[a]:=root;  gen[root]:=1;
A:      for j:=olds+1,j+1 while j≤s do
         for k:=rowstart[list[j]]  step  1  until
           rowstart[list[j]]+nrinrow[list[j]]-1 do
           if  gen[branches[k]]=0  then
        begin   c:=branches[k];   gen[c]:=gen[list[j]]+1;
                a:=a+1;     list[a]:=c;
                if gee<gen[c] then gee:=gen[c];   end;
        olds:=s;   s:=a;  if a≠z then  goto  A  else
        if gee<g then  goto  FINISH;
        if g<gee  then  d:=b:=0;  b:=b+1;   g:=gee;
        ends1[b]:=root;  mod[b]:=d+1;
        for j:=olds+1 step 1 until s do
        begin   d:=d+1;
            if d>bounds then   begin   writetext(30,[[4c]NON-
CATASTROPHIC**FAILURE**DUE**TO**VALUE**OF**BOUNDS**BEING**TOO
**SMALL**---**RESULTING**PROGRAM**THUS**INCOMPLETE]);
d:=mod[b]-1;  b:=b-1;   goto FINISH;  end;
            ends2[d]:=list[j];
        end;
FINISH:
    end;
    mod[b+1]:=d+1;  g:=g-1;
end  Min g roots;
```

```
procedure  Loop count(branches,rowstart,nrinrow,z,br,loop,
endpt1,endpt2,n,m);  integer  z,br,loop,end pt1,end pt2,n,m;
comment  This procedure is given two points of the graph
and builds a spanning or mushrooming r-tree from each point
one generation at a time. The process stops as soon as the
two rooted trees come into contact. A note is made of the
number of common points and from this is deduced the num-
ber of distinct paths between the two respective roots. ;
integer array  branches,nrinrow,rowstart;
begin   integer j,k,a,b,s,t,old s,old t;
    integer array delete[1:z],path[-z:z],list[1:2,1:z];
    n:=2;        s:=t:=1;        old s:=old t:=m:=path[0]:=0;
    loop:=-1; a:=b:=1; list[1,1]:=endpt1; list[2,1]:=endpt2;
    for j:=1 step 1 until z do  begin   path[j]:=j;
                                        path[-j]:=-j;
                                        delete[j]:=0;  end;
    delete[endpt1]:=z;   delete[endpt2]:=-z;
  A:   for j:=old s+1,j+1  while  j≤s  do
       for k:=rowstart[list[1,j]] step 1 until
            rowstart[list[1,j]]+nrinrow[list[1,j]]-1 do
    if delete[branches[k]]<0 and path[delete[branches[k]]]≠0
                and path[delete[list[1,j]]]≠0 then
       begin   loop:=loop+1;   path[delete[branches[k]]]:=
               path[delete[list[1,j]]]:=0;  end
```

```
    else  if  delete[branches[k]]=0 or delete[branches[k]]=-z
    then  begin   delete[branches[k]]:=if j=1 then a else
                          delete[list[1,j]];  a:=a+1;
                     list[1,a]:=branches[k];  end;
    old s:=s;     s:=a;  m:=m+1;   n:=n+s-old s;
    if loop≥0 then  goto  FINISH;
    for j:=old t+1,j+1 while j≤t do
        for k:=rowstart[list[2,j]]  step  1  until
            rowstart[list[2,j]]+nrinrow[list[2,j]]-1   do
if delete[branches[k]]>0 and path[delete[branches[k]]]≠0
            and (path[delete[list[2,j]]]≠0 or j=1) then
        begin   loop:=loop+1; path[delete[branches[k]]]:=
                path[delete[list[2,j]]]:=0;   end
    else  if  delete[branches[k]]=0 then
        begin   delete[branches[k]]:=if j=1 then -b else
                delete[list[2,j]];     b:=b+1;
                list[2,b]:=branches[k];     end;
    old t:=t;     t:=b;  m:=m+1;   n:=n+t-old t;
    if loop<0 then goto A;
FINISH: n:=n-(s-olds)-(t-oldt)+loop+1;
end Loop count;
```

```
procedure  Gr inspect(rem);  integer  rem;
comment  The theory of  Chpt. 4.43  is used and rem will
contain the label of that point v  which has p(m,j)≤p(m,i)
for all i. Before the matrix  P  is computed, the procedure
Min g2 roots is entered in order to find the maxmin path
length  m . ;
begin   integer  s,t,j,sum,a;
    Min g2 roots(a);
    begin   integer array matrix[1:a,1:zz];
        for s:=1 step 1 until a do
            for t:=1 step 1 until zz do
        begin   sum:=0;  for j:=rrowstart[t] step 1 until
                rrowstart[t]+nnrinrow[t]-1 do   sum:=
            sum+(if s≠1 then matrix[s-1,bbranches[j]] else 1);
            matrix[s,t]:=sum;
        end;
        sum:=₁₀10;
        for j:=zz step -1 until 1 do
            if matrix[a,j]<sum then
        begin   sum:=matrix[a,j];
                rem:=j;  end;
    end;
end Gr inspect;
```

```
procedure  Min g2 roots(g);  integer g;
comment  this uses the mushrooming r-tree technique in
order to compute  m ,the heigth of the tallest or highest
mushrooming rtree and hence the maxmin path length. ;
begin   integer root,a,j,k,c,s,olds,gee;
    integer array  gen,list[1:zz];
    g:=0;
    for root:=1 step 1 until zz do
    begin   for j:=1 step 1 until zz do  gen[j]:=0;
        a:=s:=gen[root]:=1;
        olds:=gee:=0;
        list[a]:=root;
        A:  for j:=olds+1,j+1 while j≤s do
            for k:=rrowstart[list[j]] step 1 until
                rrowstart[list[j]]+nnrinrow[list[j]]-1 do
            if gen[bbranches[k]]=0 then
        begin   c:=bbranches[k];  gen[c]:=gen[list[j]]+1;
                a:=a+1;  list[a]:=c;
                if gee<gen[c] then gee:=gen[c];  end;
        olds:=s;      s:=a;   if a≠zz then  goto A;
        if gee>g then  g:=gee;
    end;
    g:=g-1;
end  Min g2 roots ;
```

```
procedure  sort(a,n);  integer n;  integer array a;
comment  A simple sorting procedure used by Cut point. ;
begin   integer b,i,j;
    i:=0;
L:   i:=i+1;  j:=n;  if i=n then  goto SF;
M:   if j=i then  goto L;   j:=j-1;
    if a[j]<a[j+1] then   begin   b:=a[j+1]; a[j+1]:=a[j];
                                  a[j]:=b;  end;
    goto M;
SF: end  sort;

procedure  Cut point(a,can,bool,branches,rowstart,nrinrow,
z,br);  value a;  integer a,z,br;  integer array  can,branches,
rowstart,nrinrow;  boolean bool;
comment  Having decided which point to eliminate from the graph
by means of the procedure  Gr inspect , this is now affected.
The fact that the  resulting graph may become disconnected is
not taken into account. The graph also has to be relabelled
in order that the remaining graph will be labelled consecu-
tively from one upwards. ;
begin   integer j,k,b,c,d,r,n,l;  integer array S,nig[1:z];
    d:=a;  r:=1;  nig[r]:=n:=if bool then a else abs(can[d]);
    if n>z then  goto  CPF;
    for j:=0 step 1 until nrinrow[n]-1 do
```

```
        if nrinrow[branches[rowstart[n]+j]]=1 then
    begin   r:=r+1;  nig[r]:=branches[rowstart[n]+j];
        sort(nig,r);  end;
    c:=r;  r:=-1;
CP2: r:=r+1;  if r=c then goto  CP1;  a:=nig[r+1];
    if a≠n then  b:=z-r-1 else  begin  b:=z;  l:=0;  end;
    if a=b then goto CP2;
    for j:=1 step 1 until a-1,a+1 step 1 until b-1,b+1 step
    1 until z do  S[j]:=j;   S[a]:=b;  S[b]:=a;
    if not bool then  for j:=1 step 1 until z-1 do
        if sign(can[j])=sign(can[d]) and abs(can[j])=a  then
            can[j]:=sign(can[j])×b;
    Perm branches(branches,rowstart,nrinrow,S,br,z);
    goto  CP2;
CP1: n:=nrinrow[z];
    for j:=rowstart[z] step 1 until br do
    begin   for k:=rowstart[branches[j]] step 1 until
        rowstart[branches[j]+1]-1 do  if branches[k]=z then
            for l:=k step 1 until rowstart[z]-2 do
                branches[l]:=branches[l+1];
       C:   nrinrow[branches[j]]:=nrinrow[branches[j]]-1;
        for l:=branches[j]+1 step 1 until z-1 do
            rowstart[l]:=rowstart[l]-1;
    end;
```

```
    z:=z-c;    br:=br-2×n;
CPF: end  Cut point;

procedure  Gr rearrange(a);  integer a;
comment  This procedure tests by means of  Gr cont  if the
graph is still connected. If it is not it takes certain
steps,by means of  Gr seperate, so that the program does
not later fail catastrophically. ;
begin   integer j,cz1,cbr1,cz2,cbr2,na,nb,zzz,bbbr;
    integer array can[1:z];

procedure Gr seperate;
comment  If the graph is disconnected, this procedure
will relabel the graph so that the points in each com-
ponent are consecutively labelled. ;
begin   integer j;  integer array branches2[1:bbbr],
        nrinrow2,rowstart2[1:zzz];
    for j:=1 step 1 until zzz do
    begin   rowstart2[j]:=rrowstart[j];
        nrinrow2[j]:=nnrinrow[j];    end;
    for j:=1 step 1 until bbbr do branches2[j]:=bbranches[j];
    cz2:=cz1:=zzz;    cbr1:=cbr2:=bbbr;
    for j:=zzz step -1 until 1 do
```

```
    if can[j]>0 then   Cutpoint(j,can,false,bbranches,
    rrowstart,nnrinrow,cz1,cbr1)  else   Cutpoint(j,can,
    false,branches2,rowstart2,nrinrow2,cz2,cbr2);
    for j:=1 step 1 until bbbr-cbr1 do
    bbranches[cbr1+j]:=branches2[j]+cz1;
    for j:=1 step 1 until zzz-cz1 do
    begin   rrowstart[cz1+j]:=rowstart2[j]+cbr1;
            nnrinrow[cz1+j]:=nrinrow2[j];  end;
end Gr seperate;

boolean procedure  gr cont;
comment  By means of mushrooming  a quick check is made to
see if the graph is connected. The root of the mushrooming
r-tree is v1 . If the mushrooming r-tree does not contain
all the points of the graph then bool is set false to in-
dicate disconnectedness otherwise bool is set true. ;
begin   integer j,k,a,s,olds,c;    boolean bool;
    integer array gen,list[1:zzz];
    bool:=true;   for j:=1 step 1 until zzz do
    begin  can[j]:=-j;  gen[j]:=0;  end;
    gen[1]:=a:=s:=1;   olds:=0;    list[a]:=1;
    A:  for j:=olds+1,j+1 while j≤s do
        for k:=rrowstart[list[j]] step 1 until
            rrowstart[list[j]]+nnrinrow[list[j]]-1 do
```

```
        if gen[bbranches[k]]=0 then
    begin   c:=bbranches[k];  gen[c]:=gen[list[j]]+1;
        a:=a+1;   list[a]:=c;  end;
    olds:=s;  s:=a;  if olds≠s and a≠zzz then goto A;
    if olds=s then
    begin   bool:=false;  for j:=1 step 1 until s do
            can[list[j]]:=list[j];  end;
    gr cont:=bool;
end  gr cont;

    for j:=1 step 1 until zz do  copycut[j]:=copybr[j]:=0;
    zzz:=zz;   bbbr:=bbr;  na:=nb:=0;   a:=1;
  GR1:  if gr cont then  goto  GRF;   Gr seperate;
    a:=a+1;   na:=na+cz2;   nb:=nb+cbr2;
    zzz:=copycut[a]:=cz2;
    bbbr:=copybr[a]:=cbr2;  goto  GR1;
GRF: copycut[1]:=zz-na;  copybr[1]:=bbr-nb;   a:=a-1;
end Gr rearrange;
```

The following two procedures are simply graph manipulative routines in order to prepare the graph for re-entry into the Ld calc iterative loop.

```
procedure  Gr add(a);  value a;  integer a;
begin   integer j;
```

```
    for j:=1 step 1 until br-cutbr[cp-1] do
        branches[cutbr[cp-1]+j]:=bbranches[j]+cutpt[cp-1];
    for j:=1 step 1 until z-cutpt[cp-1] do
    begin rowstart[cutpt[cp-1]+j]:=rrowstart[j]+cutbr[cp-1];
        nrinrow[cutpt[cp-1]+j]:=nnrinrow[j];    end;
    for j:=1 step 1 until a  do
    begin   cutpt[cp-1+j]:=cutpt[cp-2+j]+copycut[j];
            cutbr[cp-1+j]:=cutbr[cp-2+j]+copybr[j];
    end;
    cp:=cp+a;  cutpt[cp]:=z;    cutbr[cp]:=br;
end  Gr add;

procedure Gr subtract;
begin   integer j;
    for j:=1 step 1 until z do
    begin   nnrinrow[j]:=nrinrow[j];
        rrowstart[j]:=rowstart[j];  end;
    for j:=1 step 1 until br do  bbranches[j]:=branches[j];
    for j:=1 step 1 until z-zz do
    begin   rowstart[j]:=rrowstart[j+zz]-bbr;
        nrinrow[j]:=nnrinrow[j+zz];      end;
    for j:=1 step 1 until br-bbr do
        branches[j]:=bbranches[j+bbr]-zz;
```

```
    for j:=1 step 1 until cp-1 do
    begin   cutpt[j]:=cutpt[j+1]-zz;
        cutbr[j]:=cutbr[j+1]-bbr;   end;
    cutpt[cp]:=z;   cutbr[cp]:=br;  Gr add(0);
end  Gr subtract;


procedure  Ld comp(branches,rowstart,nrinrow,z,br,ld,bounds);
value branches,rowstart,nrinrow,z,br;  integer z,br,ld,bounds;
integer array  branches,rowstart,nrinrow;
comment  All the previous procedures just described within
this sub-appendix are declared within this the main control
procedure. ;
begin  integer rem,zz,bbr,j,a,cp,point;  integer array copycut,
copybr,nnrinrow,rrowstart[1:z],cutbr,cutpt[0:z],bbranches[1:br];
            ( procedures dwclared here )

    for j:=1 step 1 until z do  cutpt[j]:=cutbr[j]:=0;
    zz:=cutpt[1]:=z;      bbr:=cutbr[1]:=br;
    ld:=cutpt[0]:=cutbr[0]:=0;       cp:=1;
    for j:=1 step 1 until br do bbranches[j]:=branches[j];
    for j:=1 step 1 until z do
    begin   nnrinrow[j]:=nrinrow[j];
        rrowstart[j]:=rowstart[j];   end;
```

```
LD1: Ld calc(rem);   writetext(30,[[cc]LD1[c]]);
     if rem>ld then ld:=rem;  Gr inspect(point);
     Cut point(point,nrinrow,true,bbranches,rrowstart,
     nnrinrow,zz,bbr);   z:=zz+cutpt[cp-1];
     br:=bbr+cutbr[cp-1];  if zz=0 then cp:=cp-1;
     Gr rearrange(a);   Gr add(a);    Time(0,LDF,LDF);
     rem:=0;
LD2: zz:=cutpt[1];   bbr:=cutbr[1];
     Gr subtract;    rem:=rem+1;
     if zz≤2 and rem<cp then  goto  LD2;
     if rem≤cp and zz>2  then  goto LD1;
LDF:
end Ld comp;
```

procedure Mat Rand(branches,rowstart,nrinrow,z,br, density,const); integer z,br,density,const; integer array branches,rowstart,nrinrow;

comment This procedure uses a random number generator in order to generate random matrices. It is entered with two identifiers already set. Density is a number lying in the range (1,100) and will indicate the required density of the final matrix. Const is a positive number which indicates the amount of leeway or exactness required from the given density. The smaller the value of const the more

exact has the final density to be . The usual range was from 1 to 5 . In the practical examples const was set at 0 or 1 for z = 60 and upwards. For smaller z , const was increased up to 4 or 5 : especially if the required density was low. If the procedure fails to find a matrix fitting the initial conditions after ten attempts, it jumps out of the procedure, Stage and program. To remedy this either increase const or increase density for the same value of z. ;

```
begin   integer j,k,l,c,m,n,p,q;
    c:=0;
MRS:  n:=0;
    for j:=1 step 1 until z do
    begin   m:=0;  rowstart[j]:=n+1;
        if j>1 then  for p:=1 step 1 until j-1 do
                     for q:=rowstart[p] step 1 until
                         rowstart[p]+nrinrow[p]-1 do
                if branches[q]=j then
                begin   if n=br then  goto  MRFAIL;
                    n:=n+1;     branches[n]:=p;
                end;
        if j=z then  goto  MR;
        for k:=j+1 step 1 until z do
        begin   l:=random(1,100,0);
```

```
            if l≤density-const then
            begin   if n=br then  goto  MRFAIL;
                n:=n+1;  branches[n]:=m:=k;   end;
        end;
        if m=0 then
        begin   if n=br then  goto  MRFAIL;
            n:=n+1;  branches[n]:=random(j+1,z,0);  end;
      MR:  nrinrow[j]:=n-rowstart[j]+1;
    end;

    l:=100×n/(z×z-z);   k:= if density<35 then 50 else
    if density<60 then 70 else 100;
    if 10×l+k<11×density or 10×l-k>9×density  then
    begin   if density>10 then   else  if l-10<density  then
        goto  MRFIN;    writetext(30,[[ccc]FAILURE**IN**MAT*
        *RAND******FINAL**DENSITY**NOT**ACCURATE**ENOUGH**--
        --**DENSITY**WAS*]);  write(30,format([ndd]),l);
        c:=c+1;  if c≠10 then goto MRS  else  goto FAIL;
    end
    else  goto MRFIN;
MRFAIL: writetext(30,[[ccc]FAILURE**IN**MAT**RAND******OVER
        FLOW**OF**BRANCHES]);   c:=c+1;
    if c≠10 then goto MRS else goto FAIL;
MRFIN:  br:=n;
end  Mat Rand;
```

```
integer procedure random(a,b,c); value a,b,c; integer a,b,c;
comment This is a standard random number generator. Originally
it was copied from CACM but it has been mutilated
slightly in order to take any starter number. The first time
into this procedure and eleven digit number is inserted for
the a value, thereafter a takes the value zero. The bounds
for the random number are given by b and c . b being the
lower bound and c the upper. ;
begin own integer x,j,k,l;
    if c≠0 then
    begin x:=c;
        j:=34 359 738 368;
        k:=68 719 476 736;
        l:=137 438 953 472;
    end;
    x:=5×x;

   R:  if x≥l then x:=x-l;
    if x≥k then x:=x-k;
    if x≥j then x:=x-j;
    if x>10↑9 then goto R;
    random:=x/j×(b-a)+a;
end random;
```

The complete program is as follows.

```
begin   array  mag[1:50];
    integer  datanr,z,br,b,bd,ld,bstop,density,rm,randnr,
            n,m,con,tc,tt,ma,mb,mc,bounds,emit;

        ( The procedures next segment and  relabel
                are declared here. )

    open(20);     open(30);
    writetext(30,[[p]************MATRIX***BANDWIDTH***
    MINIMISATION[c]************----------------------
    ----------***************DATA**TITLE***]);
    copytext(20,30,[:;]);
    datanr:=read(20);      z:=read(20);
    rm:=read(20);          randnr:=bd:=0;
    if rm≥8  then
    begin   density:=read(20);     con:=read(20);
        randnr:=read(20);      rm:=rm-8;
        br:=z↑2xdensity/100+2xzx(1-density/100);
    end  else   br:=2xread(20);
    if rm=2 or rm=6  then  ld:=read(20)  else  ld:=0;
    if rm>3 then    begin   bd:=read(20);
                bstop:=read(20);
            end   else  bd:=bstop:=0;
```

```
    emit:=read(20);
    bounds:= if rm≠2×(rm÷2) then  read(20)  else  0;
    b:=ma:=mb:=mc:=0;
begin   integer array branches[1:br],rowstart,nrinrow[1:z];

        ( All the procedures mentioned in App. 3.0 and
            this one are declared now. )

if datanr=1 then
begin   find(100,[********]);   relabel(100,[AKPOXMBM]);
    interchange(100);  end   else   find(100,[AKPOXMBM]);
interchange(100);
Time(emit,FAIL,PREND);   tc:=time;   tt:=0;
if z↑2<bounds and bounds≠0 then bounds:=z↑2;
if randnr≠0   then
begin   writetext(30,[[8c]STAGE**0***--------***RANDOM**MATRIX
        **GENERATOR[c]--------[4c]INITIAL**RANDOM**NR*=*]);
    write(30,format([nddsdddsdddsddd]),randnr);
    randnr:=random(1,100,randnr);
    writetext(30,[[4c]INITIAL**DENSITY*=*]);
    write(30,format([nd]),density);
    writetext(30,[[10s]CON*=*DENSITY*VARIATION*=]);
    write(30,format([ndd]),con);
    Mat Rand(branches,rowstart,nrinrow,z,br,density,con);
end  else  Matrix Input(branches,rowstart,nrinrow,z,br);
```

```
writetext(30,[[8c]********INPUT***MATRIX[c]********
                    --------------[ccc]]);
Matrix Output 1(branches,rowstart,nrinrow,z,br,100×br/(z×z-z),
ld,bstop,30);
if rm≠2×(rm÷2) then
begin   Ld comp(branches,rowstart,nrinrow,z,br,ld,bounds);
    writetext(30,[[8c]STAGE**1***--------***LD**COMP
    UTATION*****LD*=]);   write(30,format([ndd]),ld);
    writetext(30,[[c]--------]);
    tc:=time;  tt:=tc+tt;
    writetext(30,[[4c]TIME**TAKEN*=*]);
    write(30,format([ndd]),tc÷60);
    writetext(30,[.]);
    write(30,format([nd]),(tc/60-tc÷60)×60);
    writetext(30,[**MINUTES]);
end;
PREND:FAIL:
Magwrite(100);    interchange(100);
writetext(0,[END**OF**STAGE*-*1*---]);
end;   close(20);  close(30);   close(100);   next segment(2);
end→
```

App. 3.2 Segment 2.

As in the last segment Min g roots is used to find the roots of all mushrooming r-trees with maximum height. We then take each root in turn and iterate the following sequence.

Graph label will be the control procedure. It builds a mushrooming r-tree for a given root and then orders it by means of the procedures Planarize, Convlu and Convrd. We have now the rd, lu representation of the r-tree as described in Chpt. 4.51.1 . We enter the procedure Rearrange tree which manipulates the layers of the r-tree into a sausage formation as described in the latter part of Chpt. 4.51.1 .

```
procedure Rearrange tree(rd,lu,gen,nrabove,nrsides,wght,
below,z,root);  integer z,root;  integer array rd,lu,
gen,nrabove,nrsides,wght,below;
comment  After filling in the initial values of all the
relevant arrays, Calc box is entered to compute sumfloat
for the ρ-tree. This is followed by Calc float which as-
sociates with each point of the ρ-tree one of the integers
from -3 to +3 . This will indicate whether a point can be
moved down or not and whether it has to be moved down if
some point below it moves down. This is followed by Float
```

search which searches the ρ-tree for a suitable point to move down in order to decrease sumlayer. Having found a suitable point Readjust is entered in order to alter the layer values associated with the points that have just moved down . Procedure backtest is used to find if we can repeat the above iteration (by testing whether the movement of any points will decrease $ld_{max}(G)$). If not we finish with Readjust tree. ;

```
begin   integer  d,j,k,l,pem,qem,rem,sem,tem,av,bv,endpt,gmax;
    boolean first;   integer array  box,gox,float,mark[1:z];
procedure Calcbox;
begin   gmax:=0;
    for j:=1 step 1 until z do  box[j]:=0;
    for j:=1 step 1 until z do
    begin   box[gen[j]]:=box[gen[j]]+1;
        if   gen[j]>gmax   then   begin   gmax:=gen[j];
                                          endpt:=j;   end;
    end;
end Calc box;

procedure  Calc float;
begin   for j:=1 step 1 until z do  float[j]:=0;
    for j:=endpt,belnext(pd,d) while j≠d do
```

```
begin   d:=j;  float[j]:=3;  end;
for j:=1 step 1 until z do
    if float[j]=0 then
begin   for k:=rowstart[j] step 1 until
     rowstart[j]+nrinrow[j]-1 do
    if gen[branches[k]]>gen[j] and belnext(pd,branches[k])≠j
     and float[j]≤1  then    float[j]:=float[j]+2
    else  if gen[branches[k]]<gen[j] and
          belnext(pd,j)≠branches[k] and (float[j]=0 or
         float[j]=2) then  float[j]:=float[j]+1 ;
    if float[j]=0 then float[j]:=4;
end;
for j:=1 step 1 until z do
    for k:=rowstart[j] step 1 until
         rowstart[j]+nrinrow[j]-1 do
if belnext(pd,j)≠branches[k] and belnext(pd,branches[k])≠j
    and j>branches[k]  then marking(j,branches[k]);
for j:=1 step 1 until z do
begin   if (abs(float[j])=1 or abs(float[j])=3)
              and j≠root then
    begin   first:=true;
         for k:=process(pd,urth,first,j) while k≠0 do
         if gen[k]≤gen[belnext(pd,k)] then
```

```
            begin   d:=nbrnext(pd,k,j);
                for l:=process(pd,urth,first,j) while l≠d do ;
                if d=0 then  goto C1;  end
            else if abs(float[k])=4 then  float[k]:=
                sign(float[k])  else  if abs(float[k])=2 then
                float[k]:=sign(float[k])×3;
        C1: end;
    if (abs(float[j])=2 or abs(float[j])=3) and j≠root then
        for k:=belnext(pd,j),belnext(pd,k) while float[k]<0
        and k≠root do  float[k]:= if float[k]=-4 then 2 else
                                  if float[k]=-1 then 3 else
                                  if float[k]=-3 then 3 else
                                  if float[k]=-2 then 2 else
                                     float[k];
    end;
    float[root]:=3;
end  Calc float;

procedure  Float search;
begin   sem:=rem:=tem:=qem:=pem:=0;
    for j:=1 step 1 until gmax-1 do
    if (box[j]+box[j+1]>sem or(box[j]+box[j+1]=sem and
     (box[j]>tem orbox[j+1]>tem))) and mark[j]=0  then
    begin   sem:=box[j]+box[j+1];
```

```
        tem:=if box[j]≥box[j+1] then box[j] else box[j+1];
        pem:=if box[j]≥box[j+1] then j else j+1;
        rem:=if pem=j then j+1 else j;
        qem:=j;      end;
end Float search;

procedure  marking(a,b); value a,b;  integer a,b;
begin   integer aa,bb,j,k,l,s;
    integer array can[1:z+1];
    if gen[a]=gen[b] then goto M;
    aa:=if gen[a]<gen[b] then a else b;
    bb:=if aa=a then b else a;
    for j:=1 step 1 until z+1 do  can[j]:=0;
    l:=0;
    for j:=aa,belnext(pd,k) while j≠k and gen[j]≠gen[k]
     do    begin   l:=l+1; can[l]:=j; k:=j;  end;
    s:=0;
    for j:=bb,belnext(pd,j) while s=0 and j≠root do
    for k:=1 step 1 until l do  if can[k]=j then s:=k;
    if s=0 then s:=l+1;      k:=gen[aa];
    if s>2 then  for j:=2 step 1 until s-1 do
        if gen[can[j]]≥k  then  goto  M
        else    begin   float[can[j]]:=-abs(float[can[j]]);
                    k:=gen[can[j]];   end;
M: end marking;
```

```
boolean procedure  backtest;
begin   first:=false;
    for j:=1,j+1 while not first and j<gmax do
        if mark[j]=0 then first:=true;
    backtest:=first;
end backtest;

procedure Readjust;
begin   integer a,b,c,d,e;
    integer array  maj,min[1:z];
    a:=b:=0;     c:=d:=-1;
    for j:=1 step 1 until z do
    if   gen[j]=pem and float[j]≠3   then
    begin   a:=a+1;  maj[a]:=j;  end
    else   if  gen[j]=rem and float[j]≠3   then
    begin   b:=b+1;  min[b]:=j;  end;
    if a=0 and b=0 then  goto R3;
    for j:=0,j+1 while c≠a-1 do
        for k:=1 step 1 until a do
        if nrabove[maj[k]]+nrinrow[maj[k]]=j and
         (float[maj[k]]=1 or float[maj[k]]=4)  then
            begin  c:=k;  goto R1;  end
        else  if nrabove[maj[k]]+nrinrow[maj[k]]=j
                then  c:=c+1;
```

```
    for j:=0,j+1 while d≠b-1 do
        for k:=1 step 1 until b do
        if nrabove[min[k]]+nrinrow[min[k]]=j and
        (float[min[k]]=1 or float[min[k]]=4)
        then  begin  d:=k;  goto R1;  end
        else  if nrabove[min[k]]+nrinrow[min[k]]=j
            then  d:=d+1;
  R3: mark[qem]:=1;   goto BT;
  R1: first:=true;
for j:=1 step 1 until gmax do  mark[j]:=0;
    e:= if d=-1 then maj[c] else min[d];
    for j:=process(pd,urth,first,e) while j≠0 do
    R4: if gen[j]≤gen[belnext(pd,j)]+1 and j≠e then
        begin   d:=nbrnext(pd,j,e);  j:=d;
            for c:=process(pd,urth,first,e) while c≠d do ;
            if d=0 then goto R5 else goto R4;
        end
        else   begin  gen[j]:=gen[j]-1;  gox[j]:=1;  end;
  R5: Calcbox;  Float search;
    if tem≤av and sem≤bv  then  goto  R2;
    for j:=1 step 1 until z do   gen[j]:=gen[j]+gox[j];
    goto RTF;
R2: for j:=1 step 1 until z do
        if wght[j]=0 then nrabove[j]:=gox[j]:=0 else
```

```
        begin   nrabve[j]:=gox[j]:=0;
            for k:=urth[j],pd[k] while k>0 do
            if gen[k]>gen[j] then nrabove[j]:=nrabove[j]+1;
        end;
    av:=tem;  bv:=sem;
end Readjust;

    for j:=1 step 1 until z do
    begin   wght[j]:=nrabove[j]:=brcount(j,pd,urth)-1;
        float[j]:=box[j]:=nrsides[j]:=gox[j]:=mark[j]:=0;
    end;
    first:=true;  gen[root]:=0;  av:=bv:=₁₀10;
    for j:=process(pd,urth,first,root) while j≠0 do
        gen[j]:=gen[belnext(pd,j)]+1;
    for j:=1 step 1 until z do
        nrsides[j]:=nrinrow[j]-wght[j]-1;
    nrsides[root]:=0;
  CB: Calcbox;
  CF: Calcfloat;
  FS: Floatsearch;    if pem=1 then  goto RTF;
      Time(0,RTF,Q);
   R: Readjust;   goto CB;
  BT: if backtest then goto FS;
 RTF:end Rearrange tree;
```

Having manipulated the spanning -tree into a suitable configuration procedure Label tree is called. This again consits of smaller procedures. As explained in Chpt. 4.51.2, each succeding layer is tackled in turn and all the points within it are labelled . Procedure Label abas, when given a point labelled list[j], will label all the points adjacent to it and the layer below it . It will also label all points within the layer above which are two lines away from itself. Continuity then labels all points adjacent to list[j] and not included by Label abas. These points will be those formed by considering the lines of the cotree. Finally any subtree of which list[j] is the root , is investigated to see if any of its points lie in the layer below. This is done by the procedure Up n down. Now all the remainder of the points in the layer above not already labelled will be labelled by means of Proc .

```
procedure Label tree(pd,urth,gen,wght,nr above,nr sides,label,
below,list,z,root,ld,hm); integer z,root,ld,hm; integer
array pd,urth,gen,wght,nr above,nr sides,label,below,list;
begin  integer g,a,s,old s,u,j,c,l,b;
procedure Label abas(r);  integer r;
begin  integer array la,ma[1:z];
    integer j,k,l,b,c,d,e,f;
```

```
procedure  Label asides;
begin   for j:=1 step 1 until b-1 do
        if wght[ma[j]]>0 then
        for k:=urth[ma[j]],pd[k] while k>0 do
            if gen[k]=g and label[k]=0 then
            begin   a:=a+1;   list[a]:=k;
                label[k]:=a;  btest(k);
            end;
end  Label asides;

    b:=1;
    for j:=1 step 1 until z do    la[j]:=ma[j]:=0;
    for j:=belnext(pd,r),if wght[r]>0 then urth[r] else
        r,pd[j] while j>0 do  ifgen[j]=g then
        for k:=1,k+1 while k≤b do
            if nrinrow[j]≤la[k] or k=b then
            begin   L: b:=b+1;
                f:=j;  d:=nrinrow[j];
                for l:=k step 1 until b do
                begin   M: c:=la[l];  la[l]:=d;  d:=c;
                    e:=ma[l];    ma[l]:=f;    f:=e;
                end;
                k:=b;
            end;
    if b=1 then  goto  LA;
```

```
    for j:=1 step 1 until b-1 do
        if label[ma[j]]=0 then
    begin   a:=a+1;       list[a]:=ma[j];
        label[ma[j]]:=a;  btest(ma[j]);  end;
    Label asides;
LA: end Label abas;

procedure  btest(a);   value a;   integer a;
for u:=rowstart[a] step 1 until  rowstart[a]+nrinrow[a]-1 do
    if label[branches[u]]=0 then
    else    if abs(label[branches[u]]-label[a])>ld  then
        begin   ld:=abs(label[branches[u]]-label[a]);
            hm:=1;   end
        else    if  abs(label[branches[u]]-label[a])=ld
            then  hm:=hm+1;

procedure Continuity(r);   integer r;
begin  integer k,l,p,b,c,d,e,f;  integer array  la,ma[1:z];
    for k:=1 step 1 until z do  la[k]:=ma[k]:=0;
    b:=1;   label[0]:=₁₀4;
    for k:=rowstart[r] step 1 until
        rowstart[r]+nrinrow[r]-1 do
        if label[branches[k]]=0 and gen[branches[k]]=g then
            for l:=1 step 1 until b do
```

```
            if label[below[branches[k]]]≤la[l] or l=b then
        begin   N: b:=b+1;  f:=branches[k];
            d:=label[below[branches[k]]];
            for p:=l step 1 until b do
            begin   P: c:=la[p];  la[p]:=d;  d:=c;
                e:=ma[p];   ma[p]:=f;   f:=e;  end;
            l:=b;
        end;
    if b>1 then   for k:=1 step 1 until b-1 do
            begin   a:=a+1;     list[a]:=ma[k];
                label[ma[k]]:=a;  btest(ma[k]);  end;
end  Continuity;

procedure  Up n down(r);  value r;  integer r;
begin   integer e,f;
    for e:=rowstart[r] step 1 until
        rowstart[r]+nrinrow[r]-1 do
        for f:=rowstart[branches[e]] step 1 until
         rowstart[branches[e]]+nrinrow[branches[e]]-1 do
    if gen[branches[f]]=g and label[branches[f]]=0 then
    begin   a:=a+1;       list[a]:=branches[f];
        label[branches[f]]:=a;  btest(branches[f]);
    end;
end  Up n down;
```

```
procedure  Proc(r);  value r;  integer r;
begin  integer j;  boolean second;
    second:=true;
    for j:=process(pd,urth,second,r) while j≠0 do
        if label[j]=0 and gen[j]=g then
    begin   a:=a+1;       list[a]:=j;
        label[j]:=a;  btest(j);
    end;
end  Proc;

    for j:=1 step 1 until z do  label[j]:=list[j]:=0;
    g:=a:=1;  old s:=ld:=hm:=0;  list[1]:=root;
    label[root]:=1;  Proc(root);   g:=2;   s:=a;
    A: for j:=olds+1,j+1 while j≤s do
    begin   c:=a;   Label abas(list[j]);
        Continuity(list[j]);
        b:=a;
        if c≠b then
    begin   for l:=c+1 step 1 until b do   Up n down(list[l]);
            for l:=c+1 step 1 until b do   Proc(list[l]);
    end;
    end;
    olds:=s;   s:=a;   g:=g+1;   if s≠z then  goto A;
end  Label tree;
```

```
procedure  Graph Label(branches,rowstart,nr in row,z,br,
endpt,ld,hm,S);    integer  z,br,endpt,ld,hm;   integer
array branches,rowstart,nr in row,S;
begin   integer  j,old s,s,a,root,k;  integer array  rd,
    lu,gen,nrabove,nr sides,below,nbr,label,list,wght[0:z];
    root:=endpt;
    for j:=1 step 1 until z do   gen[j]:=below[j]:=
            nrabove[j]:=nrsides[j]:=0;
    list[1]:=root;  a:=s:=1;  old s:=0;
   TL: for j:=olds+1,j+1 while j≤s do
    for k:=rowstart[list[j]] step 1 until
        rowstart[list[j]]+nrinrow[list[j]]-1 do
        if below[branches[k]]=0 and branches[k]≠root then
        begin   a:=a+1;  list[a]:=branches[k];
            below[branches[k]]:=list[j];  end;
    olds:=s;    s:=a;   if s≠z then goto  TL;
    Planarize(below,nbr,z);
    Convlu(below,nbr,z,lu);
    Convrd(below,nbr,z,rd);
    Rearrange tree(rd,lu,gen,nrabove,nrsides,wght,below,
    z,root);   Label tree(rd,lu,gen,wght,nrabove,nrsides,
    label,below,S,z,root,ld,hm);
end Graph label;
```

```
procedure  S Output(S,z);  integer z;  integer array S;
begin   integer j;
    for j:=1 step 1 until z do
    begin   write(30,format([nddddd]),S[j]);
        if j=10×(j÷10) then  newline(30,1);
    end;
end  S Output;
```

The complete program is as follows.

```
begin  integer  datanr,z,br,ld,endpt,rm,emit,b,bstop,bd,tt,
    f,tc,ec,bsmal,hm,h,ma,mb,mc,bounds;
    array mag[1:50];
```

(The procedures Mag read1 and next segment
are declared now.)

```
find(100,[AKPOXMBM]);   Magread1(100);
open(30);
begin   integer array  branches[1:br],rowstart,nrinrow,
                    S,Seq,ends[1:z];
```

(All the rest of the procedures declared in
this sub-Appendix are now declared with
those of App. 3.0.)

```
Mag read2(100);
rewind(100);
if rm=2 or rm=3 or rm=6 or rm=7 then
begin   Time(emit,FAIL,FAIL);
    tc:=time;  h:=₁₀10;
    for f:=1 step 1 until z do Seq[f]:=f;
    bsmal:=z-3;   writetext(30,[[p]STAGE**2***--------***
    APPROXIMATE**LABELLING**OF**GRAPH[c]--------]);
    Min g roots(branches,rowstart,nrinrow,z,br,ends,ec,f);
   P: endpt:=ends[ec];
    Graph Label(branches,rowstart,nrinrow,z,br,endpt,b,hm,S);
    if bsmal>b or bsmal=b and hm<h then
    begin   writetext(30,[[8c]A**PERMUTATION**WHICH**YIELDS
        **B*=*]);  write(30,format([ndcc]),b);
        S Output(S,z);
        for f:=1 step 1 until z do  Seq[f]:=S[f];
        bsmal:=b;  h:=hm;
    end;
    Time(0,R,Q);
   R: ec:=ec-1;  if ec>0 then  goto P;
   Q: Perm branches(branches,rowstart,nrinrow,Seq,br,z);
    writetext(30,[[4c]******REARRANGED**MATRIX[c]
        ******----------------[cc]]);
    b:= if rm=2 or rm=6 then bsmal else ld+(bsmal-ld)×3/4;
```

```
    Matrix Output1(branches,rowstart,nrinrow,z,br,
    b,bstop,30);     tc:=time;      tt:=tt+tc;
    writetext(30,[[4c]TIME**TAKEN**=*]);
    write(30,format([ndd]),tc÷60);
    writetext(30,[.]);
    write(30,format([nd]),(tc/60-tc÷60)×60);
    writetext(30,[**MINUTES]);
    end  else   b:= if bd≠0 and bd≥ld then bd
            else  if ld≠0 and ld>bd then ld+1 else z-3;
interchange(100);
Magwrite(100);
interchange(100);
writetext(0,[END**OF**STAGE*-*2*------]);
FAIL:
end;  close(30);  close(100);  next segment(3);
end
→
```

App. 3.3 Segment 3.

There is one main procedure only called Min Band Width. Within it there are three small procedures called test, next unused and next nun (standing for next necessary and unused).

```
integer procedure  test;
comment;      At any moment during the program the integer
array  S[i] will be partially or totally filled. When this
procedure is applied, it takes on the value of the ld of
the partial permutation as described  in  S[i] .
begin   t:=0;      lim:= if r+b>z then z else r+b;
    for j:=r+1,j+1 while j≤lim and t=0  do
       if j-b>1 then
    for k:= if r≤b then 1 else r-b  step 1 until j-b-1 do
        for l:=0 step 1 until  nr in row[S[k]]-1 do
        if  branches[rowstart[S[k]]+l]=S[j]  then  t:=k;
    test:=t;
end  test;
```

```
integer procedure next unused(a);   value a;  integer a;
comment this will give the lowest valued unlabelled point
The unlabelled points are kept in the array  unused nr[j].
unusednr[k] = 0  if it has been labelled already and  = k
otherwise ;
begin   t:=0;
    if a>0 and a≤z then
    for a:=a,a+1 while  t=0 and a≤z  do
       if unused nr[a]≠0 then  t:=a;
    next unused:=t;
end  nextunused;
```

```
integer procedure  next nun(a);  value a;  integer a;
comment  Within the integer array  needed nr[j,k] is kept
the set  N(j)  of Chpt. 4.63 .  Thus if a point v_k  lies
in  N(j) then  needed nr[j,k] = k  else it  = 0 . This
procedure at step j will give the lowest unlabelled point
which lies in  N(j). ;
begin   t:=0;
    if a>0 and a≤z then
    for a:=a,a+1 while  t=0 and a≤z  do
    if unused nr[a]≠0 and needed nr[r,a]≠0  then   t:=a;
    next nun:=t;
end  nextnun;
```

The program has two main entries EN1 and EN2 . EN1 corresponds to Rule of Choice 1 and in particular that section of program after label T . EN2 corresponds to Rule of Choice 2. Label M corresponds to Test for Rejection 2. Test for Rejection 1 occurs in the two lines just before label T. When a complete permutation is reached a jump is made to label L . Here the true value of b (or ld) of the permutation S[i] is made and outputed with the array S[i].

```
procedure  Min Band Width(branches,rowstart,nrinrow,
    Seq,z,b,bstop,tc);   integer  z,b,bstop,tc;
integer array  branches,nrinrow,rowstart,Seq;
begin   integer  a,r,n,j,k,l,s,t,f1,lax,cut,lim,f count,
    e count,e sum;   integer  array  needednr[1:z,1:z],
    unusednr,tcont,S[1:z];   boolean  up,down;

        ( The procedures  next unused, next nun and
                test are declared here . )

    if bstop<1 or bstop≥z-1 then
    begin   writetex    (30,[[cc]B*STOP**OUT**OF**BOUNDS
        ***NEW**VALUE**SET**EQUAL**TO*0[cc]])
        bstop:=1;
    end;
```

```
    f1:=format([ndddd]);
    up:=down:=false;  esum:=tc:=0;
    if b<b stop then b:=b stop  else if b≥z then b:=z-1;

START:   for j:=1 step 1 until z do
         S[j]:=unused nr[j]:=t count[j]:=0;
    for j:=1 step 1 until z do
         for k:=1 step 1 until b do   needed nr[j,k]:=0;
    r:=1;        cut:= f count:= e count:=0;
    for j:=1 step 1 until z do   Seq[j]:=j;
 EN 1: r:=r-1;    e count:=e count+1;
    for j:=1 step 1 until z do   unused nr[j]:=j;
    if r>0 then
         for j:=1 step 1 until r do  unused nr[S[j]]:=0;
    if  r=cut  then
    begin   r:=r+2;   S[r]:=0;
         S[r-1]:= next unused(S[r-1]+1);
         if S[r-1]=0 then  goto  FAIL;
         goto  EN 1;   end;
    lax:=b-tcount[r];
    if S[r+1]≠0 then  goto T;
    for j:=1 step 1 until z do  needed nr[r,j]:=0;
    if r>cut+1 then
       for j:=1 step 1 until z do
```

```
needed nr[r,j]:= if needed nr[r-1,j]≠0  then  j else  0;
for j:=rowstart[S[r]] step 1 until
    rowstart[S[r]]+nr in row[S[r]]-1  do
    needed nr[r,branches[j]]:=branches[j];
for j:= if r-b≤cut then cut+1 else r-b step 1 until r do
    needed nr[r,S[j]]:=0;
t:=0;
for j:=1 step 1 until z do
    if needed nr[r,j]≠0 then     t:=t+1;
t count[r]:=t;
if t=0  then    begin   cut:=r;    r:=r+1;
                     S[r]:=0;    goto  EN 1;
                end;
lax:=b-t;   if lax<0   then   goto EN 1;
T: a:=S[r+1]+1;
lim:= if z-r≥b then b else z-r;
if  lax>0 and nextunused(a)=0  or  lax≤0 and
    nextnun(a)=0  then  goto EN 1;
for k:=1,k+1 while  k≤lim  do
begin   t:= if k=2 then 1 else a;
    if  lax>0  then
    begin   a:=S[r+k]:=next unused(t);
        if needed nr[r,a]=0 then  lax:=lax-1;
         unused nr[a]:=0;   end
```

```
        else   begin   a:=S[r+k]:=next nun(t);
                       unused nr[a]:=0;  end;
    end;
    goto F;

EN 2: e count:=e count+1;
    k:=n:=if r+b>z then z else r+b;
    for j:=1 step 1 until z do  unused nr[j]:=j;
    for j:=1 step 1 until n do unused nr[S[j]]:=0;
    lax:=0;
    if needed nr[r,S[k]]=0  then
    begin   if nextunused(S[k])≠0 then
        begin   S[n]:=next unused(S[k]);
                goto  F;   end
            else   lax:=lax+1;
    end;
 M: unused nr[S[k]]:=S[k];
    k:=k-1;   if k=r then  goto EN 1;
    if needed nr[r,S[k]]=0 then  lax:=lax+1;
    if  S[k]>S[k+1]then  goto M;
    a:=if lax>0 then  next unused(S[k]) else  next nun(S[k]);
    unused nr[a]:=0;   unused nr[S[k]]:=S[k];
    S[k]:=a;   lax:=lax-1;
    if needed nr[r,S[k]]≠0 then  lax:=lax+1;
    a:=1;
```

```
 for k:=k+1 while k≤n do
 begin   lax:=lax-1;   a:=S[k]:=
         if lax≥0 then  next unused(a) else  nextnun(a);
     unused nr[S[k]]:=0;
     if needed nr[r,S[k]]≠0  then  lax:=lax+1;
 end;
F: f count:=f count+1;
 if tc<100 then tc:=tc+1
 else  begin  tc:=0;  Time(0,TFAIL,PREND);  end;
 if test≠0 then  goto EN 2;
 if r<z-b then   begin   r:=r+2;  S[r]:=0;
                         goto EN 1;   end;
L: b:=b-1;   s:=0;
 for r:=b,r+1 while  r≤z-b and s=0 do  s:=test;
 if s=0 then  goto  L;
 writetext(30,[[8c]A**PERMUTATION**WHICH**YIELDS**B*=]);
 write(30,format([ndcc]),b+1);
 for j:=1 step 1 until z do  begin   Seq[j]:=S[j];
                     write(30,format([ndddd]),S[j]);
                     if j=10×(j÷10) then  newline(30,1);
                              end;
 writetext(30,[[3c]F*COUNT**WAS*]);
 write(30,f1,fcount);
 writetext(30,[[3s]-------***E*COUNT**WAS*]);
```

```
    write(30,f1+1,ecount);
    down:=true;  e sum:=e sum+e count;
    if up or b<bstop  then  goto  FINISH  else
    begin  r:=s;       cut:=0;
            ecount:=fcount:=0;
            if tcount[r]≤b then  goto EN2;
            goto EN 1;  end;

 FAIL: writetext(30,[[7c]UNABLE**TO**FIND**PERM**FOR**B*=]);
    write(30,format([ndcc]),b);
    writetext(30,[[c]F*COUNT**WAS*]);
    write(30,f1,fcount);
    writetext(30,[[3s]-------***E*COUNT**WAS*]);
    write(30,f1+1,ecount);
    up:=true;
    esum:=esum+ecount;
    if down then  goto  FINISH;
    b:=b+1;
    goto  START;
TFAIL: esum:=esum+ecount;
FINISH: b:=b+1;
    writetext(30,[[5c]END**OF**MIN**BAND**WIDTH**----
            **TOTAL**E*COUNT**WAS]);
    write(30,f1+2,e sum);
end  Min Band Width;
```

```
procedure  Dot Output(branches,rowstart,nrinrow,z,br);
integer z,br;  integer array branches,rowstart,nrinrow;
comment  This procedure is given the matrix in the bran-
ches list notation and outputs the corresponding adjacen-
cy matrix in terms of dots and crosses. The dots stand
for zeroes and the crosses for ones. This enables one to
visualise the bandwidth of the matrix. ;
begin  integer  j,k,l,s;
    writetext(30,[[6c]DOT/CROSS***REPRESENTATION**OF**
        THE**FINAL**MATRIX[ccc]]);
    if z>60 then
    for j:=1 step 1 until z do
    begin  writetext(30,[[c]]);
        for k:=1 step 1 until z do
        begin  s:=0;
            for l:=1 step 1 until nrinrow[j] do
                if branches[rowstart[j]-1+l]=k then  s:=1;
        if s=1 then  writetext(30,[X]) else writetext(30,[.]);
        end;
        end;
    if z<60 then
    for j:=1 step 1 until z do
    begin  writetext(30,[[c]]);
```

```
        for k:=1 step 1 until z do
        begin   s:=0;
            for l:=1 step 1 until nrinrow[j] do
                if branches[rowstart[j]-1+l]=k then  s:=1;
      if s=1 then  writetext(30,[X*]) else writetext(30,[.*]);
        end;
    end;
    writetext(30,[[cccc]]);
end;
```

The complete program is as follows.

```
begin   integer  b,z,br,s,j,k,datanr,bstop,bd,ld,emit,tt,tc,
        bounds,rm,ma,mb,mc;   array  mag[1:50];

        ( The procedures  Mag read1, next segment and
                relabel are now declared. )

open(30);  find(100,[AKPOXMBM]);  Magread1(100);
begin  integer array branches[1:br],rowstart,nr in row,S[1:z];

        ( All the procedures of this sub-appendix and
                those in  App. 3.0  are now declared. )
```

```
Mag read2(100);
rewind(100);
if rm>3 and b>bstop and b>ld  then
begin   Time(emit,PREND,PREND);
    tc:=time;
    writetext(30,[[p]STAGE**3**--------***MIN**BAND**
                    WIDTH[c]--------]);
    if bstop=0 or bstop<ld then bstop:=ld;
    b:=b-1;
    if bd<b and bd≠0 then b:=bd;
    Min Band Width(branches,rowstart,nrinrow,S,z,b,
                    bstop,tc);
    Perm branches(branches,rowstart,nrinrow,S,br,z);
    writetext(30,[[8c]*******FINAL***MATRIX[c]*****
            ***-------------[cc]]);
    Matrix Output1(branches,rowstart,nrinrow,z,br,b,
                    bstop,30);
    tc:=time;       tt:=tt+tc;
    writetext(30,[[4c]TIME**TAKEN**=*]);
    write(30,format([ndd]),tc÷60);
    writetext(30,[.]);
    write(30,format([nd]),(tc/60-tc÷60)×60);
    writetext(30,[**MINUTES]);
end;
```

```
Dot Output(branches,rowstart,nrinrow,z,br);
writetext(30,[[10c]END**OF**ANALYSIS**BY**MATRIX**BANDWIDTH
**MINIMISATION[cccc]TOTAL**TIME**TAKEN**=*]);
write(30,format([ndd]),tt÷60);
writetext(30,[.]);
write(30,format([nd]),(tt/60-tt÷60)×60);
writetext(30,[**MINUTES.SECONDS]);
PREND:
if datanr<0 then  relabel(100,[********]);
writetext(0,[END**OF**STAGE*-*3*------------[cc]]);
end;   close(30);   close(100);
    if datanr≥0 then  nextsegment(1);
end
→
```

App. 3.4 Input Specification.

The input layout is as follows :-

```
: < TITLE > ;
data nr ;
z ;
rm ;
        density ;  }
        con ;      }     if rm > 8 .
        rand nr ;  }
br ;    -------------    if  rm < 8 .
        ld ;     -----   if  rm = 2 or 6 .
bd ;  }
      }   ---------      if  rm > 3 .
b stop ;}
emit ;
bounds;   -----------    if rm = 1, 3, 5 or 7 .
< Branches list >
→→→
```

If this is the first set of data (or using previous terminology , it is the first block of data), then data nr is set equal to 1 . If it is the last, it is set to -1 , otherwise it is put equal to 0 .

z is the number of points in the graph.

rm is a control variable on the three segments. Depending upon its value the program will complete the corresponding segments or stages . If rm > 8 then the matrix will be randomly produced within the first segment by Mat Rand . If it is less than 8 , the matrix is being input in the branches list representation. The table for rm is :

rm	1	2	3	4	5	6	7
	9	10	11	12	13	14	15
Stages	1	2	1,2	3	1,3	2,3	all

The density is a number between 0 and 100.

con is usually between 0 and 5.

rand nr is an eleven digit random number.

br is the number of lines in the graph.

ld is the approximate ld_m for the graph, if it is unknown insert 0 .

bd is an upper bound on the ld_m , again if this is unknown insert the maximum possible number, z .

b stop is the value of ld at which Stage 3 is to stop. If this is unknown or immaterial insert 0 .

emit is the time interuption frequency in seconds.

bounds is an upper bound for the arrays end1 and end2. It is convenient to set it a bit high at 2000.

App. 3.5 Specimen Output.

```
MATRIX  BANDWIDTH  MINIMISATION          DATA TITLE FLANGED/2
-------------------------------

    INPUT MATRIX
    ------------

 Z       BR       DENSITY      B        BSTOP

27;      30;        9;         0;        0;


ROW      NR IN ROW        MATRIX

 1;        2;        2;    3;
 2;        3;        1;    4;    5;
 3;        2;        1;    6;
 4;        2;        2;    7;
 5;        2;        2;    8;
 6;        2;        3;    9;
 7;        2;        4;   10;
 8;        2;        5;   11;
 9;        2;        6;   12;
10;        3;        7;   13;   14;

11;        5;        8;   12;   15;   16;   17;
12;        2;        9;   11;
13;        2;       10;   18;
14;        2;       10;   15;
15;        2;       14;   11;
16;        2;       11;   19;
17;        2;       11;   20;
18;        2;       13;   21;
19;        2;       16;   21;
20;        2;       17;   22;

21;        5;       18;   19;   23;   24;   25;
22;        3;       20;   26;   27;
23;        1;       21;
24;        1;       21;
25;        2;       21;   26;
26;        2;       22;   25;
27;        1;       22;


STAGE 1 -------- LD COMPUTATION    LD = 4
--------

TIME TAKEN  =    1.22 MINUTES
```

STAGE 2 -------- APPROXIMATE LABELLING OF GRAPH

A PERMUTATION WHICH YIELDS B = 6

27	22	20	26	17	25	23	24	11	16
19	21	8	12	15	18	5	9	14	13
2	6	10	1	4	3	7			

A PERMUTATION WHICH YIELDS B = 5

26	25	23	22	27	21	24	20	18	19
17	15	16	13	11	12	14	10	8	9
7	5	4	6	2	3	1			

A PERMUTATION WHICH YIELDS B = 5

3	1	6	2	9	4	5	12	7	8
11	10	15	16	17	13	14	19	20	18
24	21	22	23	25	27	26			

REARRANGED MATRIX

Z	BR	B	B STOP
27;	30;	5;	0;

ROW	NR IN ROW	MATRIX		
1;	2;	2;	3;	
2;	2;	4;	1;	
3;	2;	1;	5;	
4;	3;	2;	6;	7;
5;	2;	3;	8;	
6;	2;	4;	9;	
7;	2;	4;	10;	
8;	2;	5;	11;	
9;	2;	6;	12;	
10;	2;	7;	11;	

11;	5;	10;	8;	13;	14;	15;
12;	3;	9;	16;	17;		
13;	2;	17;	11;			
14;	2;	11;	18;			
15;	2;	1;	19;			
16;	2;	12;	20;			
17;	2;	12;	13;			
18;	2;	14;	22;			
19;	2;	15;	23;			
20;	2;	16;	22;			
21;	1;	22;				
22;	5;	20;	18;	24;	21;	25;
23;	3;	19;	27;	26;		
24;	1;	22;				
25;	2;	22;	27;			
26;	1;	23;				
27;	2;	23;	25;			

TIME TAKEN = 1.46 MINUTES

STAGE 3 --------- MIN BAND WIDTH

A PERMUTATION WHICH YIELDS B = 5

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27			

F COUNT WAS 19 -------- E COUNT WAS 20

A PERMUTATION WHICH YIELDS B = 4

1	2	3	4	5	6	7	8	9	10
11	13	12	14	15	17	16	18	19	20
21	22	23	24	25	26	27			

F COUNT WAS 22 -------- E COUNT WAS 22

END OF MIN BAND WIDTH ---- TOTAL E COUNT WAS 42

FINAL MATRIX

Z	BR	B	B STOP
27;	30;	4;	4;

ROW	NR IN ROW	MATRIX				
1;	2;	2;	3;			
2;	2;	4;	1;			
3;	2;	1;	5;			
4;	3;	2;	6;	7;		
5;	2;	3;	8;			
6;	2;	4;	9;			
7;	2;	4;	10;			
8;	2;	5;	11;			
9;	2;	6;	13;			
10;	2;	7;	1;			
11;	5;	10;	8;	12;	14;	15;
12;	2;	16;	11;			
13;	3;	9;	17;	16;		
14;	2;	11;	18;			
15;	2;	11;	19;			
16;	2;	13;	12;			
17;	2;	13;	20;			
18;	2;	14;	22;			
19;	2;	15;	23;			
20;	2;	17;	22;			
21;	1;	22;				
22;	5;	20;	18;	24;	21;	25;
23;	3;	19;	27;	26;		
24;	1;	22;				
25;	2;	22;	27;			
26;	1;	23;				
27;	2;	23;	25;			

TIME TAKEN = 0.14 MINUTES

```
DOT/CROSS  REPRESENTATION  OF  THE  FINAL  MATRIX

. X X . . . . . . . . . . . . . . . . . . . . . . . .
X . . X . . . . . . . . . . . . . . . . . . . . . . .
X . . . X . . . . . . . . . . . . . . . . . . . . . .
. X . . . X X . . . . . . . . . . . . . . . . . . . .
. . X . . . . X . . . . . . . . . . . . . . . . . . .
. . . X . . . . X . . . . . . . . . . . . . . . . . .
. . . X . . . . . X . . . . . . . . . . . . . . . . .
. . . . X . . . . . X . . . . . . . . . . . . . . . .
. . . . . X . . . . . . X . . . . . . . . . . . . . .
. . . . . . X . . . X . . . . . . . . . . . . . . . .
. . . . . . . X . X . X . X X . . . . . . . . . . . .
. . . . . . . . . . X . . . . X . . . . . . . . . . .
. . . . . . . . X . . . . . . X X . . . . . . . . . .
. . . . . . . . . . X . . . . . . X . . . . . . . . .
. . . . . . . . . . X . . . . . . . X . . . . . . . .
. . . . . . . . . . . X X . . . . . . . . . . . . . .
. . . . . . . . . . . . X . . . . . . X . . . . . . .
. . . . . . . . . . . . . X . . . . . . . X . . . . .
. . . . . . . . . . . . . . X . . . . . . . X . . . .
. . . . . . . . . . . . . . . . X . . . . X . . . . .
. . . . . . . . . . . . . . . . . . . . . X . . . . .
. . . . . . . . . . . . . . . . . X . X X . . X X . .
. . . . . . . . . . . . . . . . . . X . . . . . . X X
. . . . . . . . . . . . . . . . . . . . . X . . . . .
. . . . . . . . . . . . . . . . . . . . . X . . . . X
. . . . . . . . . . . . . . . . . . . . . . X . . . .
. . . . . . . . . . . . . . . . . . . . . . X . X . .

END OF ANALYSIS BY  MATRIX  BANDWIDTH  MINIMISATION

TOTAL TIME TAKEN  =  3.22 MINUTES.SECONDS
```

APPENDIX 4.

App. 4.1 Cascade.

The main procedure was called, appropriately Cascade. It is as follows :

```
procedure  Cascade(mat,z);
integer z;  integer array mat;
comment        mat[i,j]  contains the cost associated matrix
and mat[i,i] = 0 .   If v_i  is not  adjacent  to v_j  , then
mat[i,j] =  10.  At the end of the second pass   mat[i,j]
will contain the length of the shortest path from v_i to v_j  ;
begin   integer j,k,l,min;
    for j:=1 step 1 until z do
        for k:=1 step 1 until z do
        if j≠k then
    begin   min:=mat[j,k];
        for l:=1 step 1 until z do
            if min>mat[j,l]+mat[l,k] then
            min:=mat[j,k]:=mat[j,l]+mat[l,k];
    end;
    for j:=z step -1 until 1 do
        for k:=z step -1 until 1 do
```

```
        if j≠k then
    begin   min:=mat[j,k];
        for l:=z step -1 until 1 do
            if min>mat[j,l]+mat[l,k] then
                min:=mat[j,k]:=mat[j,l]+mat[l,k];
    end;
end  Cascade;
```

The data for this and the other two programs was in the branches list representation. Thus another procedure Transform was written in order to transform the data back into the adjacency type form.

```
procedure Transform(branches,rowstart,nrinrow,z,br,mat,di,dist);
integer z,br,di; integer array branches,rowstart,nrinrow,mat,dist
begin   integer j,k,l;
    for j:=1 step 1 until z do
        for k:=1 step 1 until z do
        mat[j,k]:=9⏨3;
    for j:=1 step 1 until z do   mat[j,j]:=0;
    for j:=1 step 1 until z do
        for k:=rowstart[j] step 1 until
            rowstart[j]+nrinrow[j]-1 do
        mat[j,branches[k]]:= if di>0 then dist[k] else 1;
end  Transform;
```

The resultant shortest path distance matrix was output by means of the procedure Mat out :-

```
procedure  Mat out(mat,z);  integer z;  integer array  mat;
begin   integer j,k;
    writetext(30,[[p]CL21*****SHORTEST**DISTANCE**MATRIX
        **---**CASCADE**METHOD[8c]]);
    for j:=1 step 1 until z do
    begin   if j≠(j÷11)×11  then  newline(30,1)
                else  newline(30,2);
        for k:=1 step 1 until z do
        begin   if k=(k÷11)×11 then space(30,2);
            if mat[j,k]<10^3 then write(30,format([nddd]),
                mat[j,k]) else  writetext(30,[***X]);
        end;
    end;
end  Mat out;
procedure  Distance input(br,dist);  integer br;
        integer array  dist;
comment     The costs to be associated with each line  was
read in by means of this procedure, the ordering being the
same as the lines concerned. ;
begin   integer j;
    for j:=1 step 1 until br do  dist[j]:=read(20);
end Distance input;
```

App. 4.2 Shortest Route 1 .

The control procedure was titled Shortest Route. It assigned the root to each point in turn and built a mushrooming r-tree. As soon as the r-tree had been built, it was output. This was accomplished by running down the r-tree from each point of the digraph in turn and outputting the successive belows. If a point does not belong to the mushrooming r-tree (i.e. there is no path from the root to that point, this is indicated by its distance vector, gen , being still equal to $_{10}4$.

```
procedure  Tree span(branches,rowstart,nrinrow,z,br,gen,
below,root);   integer  root,z,br;     integer array
branches,rowstart,nrinrow,below,gen;
comment  This is the procedure which builds a mushrooming
r-tree in the array below for the root point  root . ;
begin   integer  j,a,b,c,k,old a;
    integer array list[1:z];
    for j:=1 step 1 until z do
    begin   below[j]:=-1;
        gen[j]:= ₁₀6;  end;
    below[root]:=old a:=gen[root]:=0;
    list[1]:=root;   a:=b:=1;
```

```
TS:  for j:=olda+1,j+1 while j≤a do
     for k:=rowstart[list[j]] step 1 until
          rowstart[list[j]]+nrinrow[list[j]]-1 do
     if below[branches[k]]=-1 then
  begin   c:=branches[k];
     below[c]:=list[j];
     gen[c]:=gen[list[j]]+1;
     b:=b+1;  list[b]:=c;
  end;
  olda:=a;   a:=b;
  if b≠z and olda≠a then goto TS;
end  Tree span;

procedure  Shortest routes(branches,rowstart,nrinrow,z,br);
integer z,br;   integer array  branches,rowstart,nrinrow;
begin   integer j,k,root,l;   integer array  below,gen[1:z];
  writetext(30,[[p]CL21******SHORTEST***PATHS***THROUGH*
     **A***GRAPH[cccc]ORIGIN****DESTINATION***DISTANCE*
     ********ROUTE[cc]]);
  for root:=1 step 1 until z do
  begin   Tree span(branches,rowstart,nrinrow,z,br,gen,
          below,root);  write(30,format([nddd]),root);
     for k:=1 step 1 until z do
          if k≠root then
```

```
begin   write(30,format([8snddd]),k);
    l:=k;   if below[k]=-1 then  writetext(30,
    [[11s]NONE])  else  write(30,format([6sndd
            dddd]),gen[k]);
    if below[k]=-1  then  goto  SR1;
    writetext(30,[[8s]]);
    for l:=l,below[l] while l≠0 do
        write(30,format([ndddd]),l);
    SR1:   writetext(30,[[c]]);
    if k≠z then writetext(30,[[4s]]);
end;
end;
end  Shortest routes;
```

The final program comprised of the above two procedures and the procedure Matrix Input.

App. 4.3 Shortest Route 2 .

The program is very similar to Shortest Route 1 . The difference is that the d-r tree is built up within the control procedure, Shrt route 2 , and not in another (i.e. Treeform). Two other procedures are used : Matrix input and Dist input.

```
procedure Shrt route 2(branches,rowstart,nrinrow,z,br,dist);
integer z,br;  integer array branches,nrinrow,rowstart,dist;
begin   integer  root,j,k,l,m,s,olds,a,c,d;
    boolean  first;
    integer array  below,gen[1:z],list[1:5×z];
integer procedure distance(a,b);  value a,b;  integer a,b;
    for m:=rowstart[a] step 1 until
        rowstart[a]+nrinrow[a]-1 do
        if branches[m]=b then  distance:=dist[m];
    writetext(30,[[p]CL21******SHORTEST***PATHS***THROUGH
    ***A***COST***ASSOCIATED***GRAPH[cccc]ORIGIN****DESTI
    NATION***DISTANCE[9s]ROUTE[cc]]);
    for root:=1 step 1 until z do
    begin   for j:=1 step 1 until z do  gen[j]:=⌀5;
        list[1]:=below[root]:=root;
        gen[root]:=olds:=0;        s:=a:=1;
```

```
ST: for j:=olds+1,j+1 while j≤s do
        for k:=rowstart[list[j]] step 1 until
         rowstart[list[j]]+nrinrow[list[j]]-1 do
            if gen[branches[k]] > gen[list[j]] +
                dist[k] then
    begin   c:=branches[k];   d:=list[j];
        a:=a+1;     below[c]:=d;
        if a > 5×z then  goto  NOGO;
        list[a]:=c;  gen[c]:=gen[d]+dist[k];
    end;
    if a=s then  goto  SRTF;
    olds:=s;        s:=a;        goto ST;
NOGO:  writetext(30,[[cc]NOGO***FAILURE***ARRAY**LIST**TOO
        **SMALL*---*ALTER**UPPER**BOUND[cc]]);  goto  SFIN;

SRTF: write(30,format([nddd]),root);
    for l:=1 step 1 until z do
        if l≠root then
    begin   write(30,format([8snddd]),l);
        k:=l;
        if gen[k]=5 then  writetext(30,[[11s]NONE])
            else  write(30,format([10sndd]),gen[k]);
        if gen[k]=5 then  goto SRTF1;
        writetext(30,[[8s]]);
```

```
        for k:=k,below[k] while k≠root do
            write(30,format([ndddd]),k);
        write(30,format([ndddd]),root);
        SRTF1:   writetext(30,[[c]]);
        if l≠z then  writetext(30,[[4s]])
        else writetext(30,[[c]]);
    end;
    end;
SFIN: end  Shrt route 2;
```

If the procedure should fail (at NOGO) because the array list is too small, the upper bound declaration for list should be raised from 5xz to some higher figure e.g. 6 or 7 times z. If this method was to be programmed in a list processing language, we would have to declare space for only 2 x z words because the useful part of the array list is from list[olds] to list[a] and this will never exceed twice the number of points in the digraph (Chpt. 5.22).

App. 4.4 Specimen Input and Output.

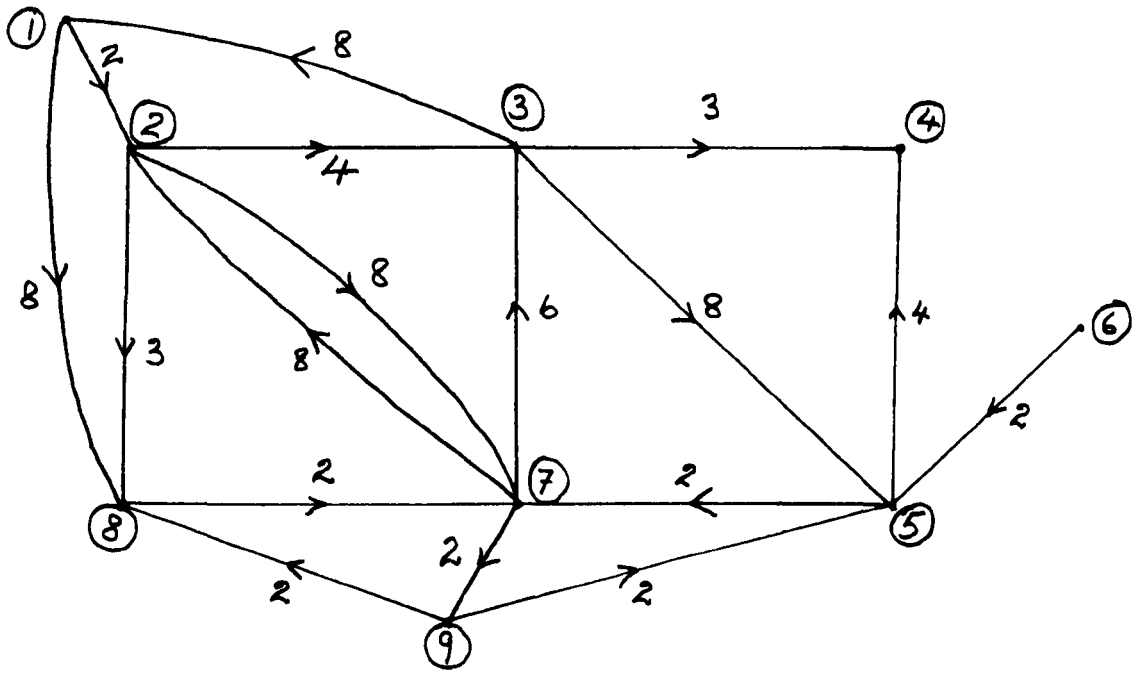
The input layout or format for Cascade and Shortest Route 2 is of the form : -

```
data nr ;
z ;  br ;

< Branches list representation >
→
di ;   ------------  This has a value only for Cascade.
                      If all the distances are unity
                      then di < 0 and the following
                      cost matrix is not read, other-
                      wise  di > 0 .
< Cost Matrix > -----  The cost element of each line is
                      read in, the ordering  being the
                      same as the branches list.
→→→
```

For Shortest Route 1 , the input is as above up to the first end of message (→). For test purposes the same data tape was used for all three programs except that di was punched in by hand for Cascade.

Below is given the digraph of a simple problem with the output from the Cascade method program and the program Shortest Route 2.



CL21 SHORTEST DISTANCE MATRIX --- CASCADE METHOD

0	2	6	9	11	X	7	5	9
12	0	4	7	9	X	5	3	7
8	10	0	3	8	X	10	13	12
X	X	X	0	X	X	X	X	X
16	10	8	4	0	X	2	6	4
18	12	10	6	2	0	4	8	6
14	8	6	8	4	X	0	4	2
16	10	8	10	6	X	2	0	4
18	12	10	6	2	X	4	2	0

CL21 SHORTEST PATHS THROUGH A DIGRAPH

ORIGIN	DESTINATION		DISTANCE	ROUTE					
1	2	2	2	2	1				
	3		6	3	2	1			
	4		9	4	3	2	1		
	5		11	5	9	7	8	2	1
	6		NONE						
	7		7	7	8	2	1		
	8		5	8	2	1			
	9		9	9	7	8	2	1	
2	1		12	1	3	2			
	3		4	3	2				
	4		7	4	3	2			
	5		9	5	9	7	8	2	
	6		NONE						
	7		5	7	8	2			
	8		3	8	2				
	9		7	9	7	8	2		
3	1		8	1	3				
	2		10	2	1	3			
	4		3	4	3				
	5		8	5	3				
	6		NONE						
	7		10	7	5	3			
	8		13	8	2	1	3		
	9		12	9	7	5	3		
4	1		NONE						
	2		NONE						
	3		NONE						
	5		NONE						
	6		NONE						
	7		NONE						
	8		NONE						
	9		NONE						
5	1		16	1	3	7	5		
	2		10	2	7	5			
	3		8	3	7	5			
	4		4	4	5				
	6		NONE						
	7			2		7	5		
	8		6	8	9	7	5		
	9		4	9	7	5			

6	1	18	1	3	7	5	6
	2	12	2	7	5	6	
	3	10	3	7	5	6	
	4	6	4	5	6		
	5	2	5	6			
	7	4	7	5	6		
	8	8	8	9	7	5	6
	9	6	9	7	5	6	
7	1	14	1	3	7		
	2	8	2	7			
	3	6	3	7			
	4	8	4	5	9	7	
	5	4	5	9	7		
	6	NONE					
	8	4	8	9	7		
	9	2	9	7			
8	1	16	1	3	7	8	
	2	10	2	7	8		
	3	8	3	7	8		
	4	10	4	5	9	7	8
	5	6	5	9	7	8	
	6	NONE					
	7	2	7	8			
	9	4	9	7	8		
9	1	18	1	3	7	5	9
	2	12	2	7	5	9	
	3	10	3	7	5	9	
	4	6	4	5	9		
	5	2	5	9			
	6	NONE					
	7	4	7	5	9		
	8	2	8	9			

APPENDIX 5.

There is only one sumcheck made while inputting the data fot this program , Transportree . This is to ensure that $\sum_{1}^{m} a_i = \sum_{1}^{n} b_j$ (see Chpt. 6). After reading in all the data, Tree find is used to find an initial basic feasable solution. The solution is left in the below form . The tree is then ordered by means of Planarize, Convlu and Convrd into the rd, lu representation.

```
procedure  Treeform(supply,demand);  value supply,demand;
    integer array  supply,demand;
comment    The modified  N-W corner method is used to find the
initial basic feasable solution ( or initial tree ). This is
rooted at an arbitrary point and described in the below array;
begin
    procedure rowsearch;
    begin   max:=10;
        for j:=m+1 step 1 until m+n do
            if cost[a,j]<max and below[j]=-1 then
            begin   max:=cost[a,j];
                    b:=j;  end;
    end rowsearch;
```

```
procedure  colsearch;
begin   max:=10;
    for j:=1 step 1 until m do
        if cost[j,a]<max and below[j]=-1 then
        begin   max:=cost[j,a];
                b:=j;   end;
end  colsearch;

sum:=below[1]:=0;     count:=a:=1;
for j:=2 step 1 until n+m do   below[j]:=-1;

RS:   rowsearch;
  below[b]:=a;   count:=count+1;
  load[b]:=if supply[a]<demand[b] then
      supply[a] else  demand[b];
  supply[a]:=supply[a]-load[b];
  demand[b]:=demand[b]-load[b];
  sum:=load[b]×cost[a,b]+sum;
  if count=n+m then  goto  FIN;
  if supply[a]≥demand[b] then  goto  RS;
  a:=b;

CS:   colsearch;
  below[b]:=a;   count:=count+1;
  load[b]:=if supply[b]<demand[a] then
      supply[b] else demand[a];
```

```
    supply[b]:=supply[b]-load[b];
    demand[a]:=demand[a]-load[b];
    sum:=sum+load[b]×cost[b,a];
    if count=n+m then goto  FIN;
    if demand[a]≥supply[b] then  goto  CS;
    a:=b;   goto  RS;
FIN:
end  Tree form;
```

The main iterative loop starts at TOP and finishes at FINISH . First the shadow costs are computed by means of Shad cost. Initially this computation is done for all the points of the tree and thereafter , only for the points of the subtree which has been cut of .

```
procedure  Shad cost;
begin   first:=true;   shadow[1]:=0;
    for j:=process(rd,lu,first,root) while j>0 do
        if j≠1 then
    begin   a:=belnext(rd,j);
        shadow[j]:=postcost(a,j)-shadow[a];
    end;
end  shad cost;
```

This is now followed by Extra link which finds the next link to be inserted. If there is no line which will improve the solution, the identifiers endpt1 and endpt2 are made equal to zero. As mentioned in Chpt. 5.3 there are many ways of doing this. Two procedures were written: one searched through for the smallest value of cost[i,j]-shadow[i]-shadow[j] and the other for the first negative value of that same expression.

```
procedure  Extra 1 link;
begin   cut1:=cut2:=0;
    for j:=1 step 1 until m do
        for k:=m+1 step 1 until m+n do
        if cost[j,k]<shadow[j]+shadow[k] then
        begin   cut1:=j;   cut2:=k;
                goto  E1;   end;
  E1:
end  Extra link;
```

The circuit is completed by means of Loop form. Within the circuit, the smallest load link is found by means of Smallest link and then the loads along the loop are altered by means of Recalc loop.

```
procedure  Smallest link;
begin   inc:=10;
    for j:=1 step 2 until apex-1,lm step -2 until apex+1 do
        if load[loop[j]]<inc then
        begin   inc:=load[loop[j]];
            sl1:=loop[j];
            sl2:=belnext(rd,sl1);
            lo:=j;  end;
end Smallest link;

procedure  Loop form;
begin   ga:=gnrd(rd,cut1);
    gb:=gnrd(rd,cut2);
    c:=if ga≥gb then cut1 else cut2;
    d:=if c=cut1 then cut2 else cut1;
    if c≠cut1 then  begin   j:=ga;  ga:=gb;
                            gb:=j;  end;
    e:=0;  f:=n+m;
    for j:=c,belnext(rd,k) while ga≥gb do
    begin   e:=e+1;   loop[e]:=k:=j;
            ga:=ga-1;  end;
    apex:=e;
    loop[n+m]:=d;
```

```
    if loop[e]≠d then
        for j:=belnext(rd,loop[e]) while j≠loop[e] do
        begin   e:=e+1;    loop[e]:=j;
            if  loop[e]=belnext(rd,loop[f]) then
            begin   apex:=e;
                for k:=f step 1 until n+m do
                begin   e:=e+1;
                    loop[e]:=loop[k];
                end;
                goto LC1;
            end;
            f:=f-1;
            loop[f]:=belnext(rd,loop[f+1]);
        end;
LC1:    lm:=e;
end  Loop form;

procedure  Recalc loop;
begin   for j:=1 step 2 until apex-1,
            lm step -2 until apex+1 do
        load[loop[j]]:=load[loop[j]]-inc;
    for j:=2 step 2 until apex-1,
        lm-1 step -2 until apex+1 do
        load[loop[j]]:=load[loop[j]]+inc;
```

```
if lo>apex then
    begin   for j:=lo step 1 until lm-1 do
                load[loop[j]]:=load[loop[j+1]];
            load[loop[lm]]:=inc;
    end
else    begin   for j:=lo step -1 until 2 do
                load[loop[j]]:=load[loop[j-1]];
                load[loop[1]]:=inc;
            end;
for j:=1 step 2 until lm-1 do
    sum:=sum-inc×postcost(loop[j],loop[j+1]);
for j:=2 step 2 until lm-2 do
    sum:=sum+inc×postcost(loop[j],loop[j+1]);
sum:=sum+postcost(loop[1],loop[lm])×inc;
end Recalc loop;
```

We have thus so far found a line to delete within a feasable solution, found another to insert (so as to keep the solution basic and feasable) and altered the loads along the affected route accordingly . However the information is not as yet in the required form . The tree has to be manipulated. This is accomplished by Treecut and Fixtop. Treecut cuts the tree into two parts ar smallest link and the tree is rejoined into a single tree by means of Fixtop at the points given by Extra link.

A final procedure Print was written so as to monitor what was happening in the program . Every out number of iterations (out being input with the rest of the data) , the tree and its cost was output . At the end the final solution , its cost and the number of iterations is output with the time taken to complete the computation.

```
procedure  Print;
begin   f:=format([ndddd]);
    writetext(30,[[4c]TOTAL***COST**IS]);
    write(30,f,sum);
    writetext(30,[[6c]*****ROUTE*********LOAD*****COST
        [c]*****-----[10s]----[5s]----[2c]]);
    first:=true;
    for j:=process(rd,lu,first,1)  while j>0 do
        if j≠1 then
    begin   write(30,f,j);   writetext(30,[[s]]);
        write(30,f,belnext(rd,j));  writetext(30,[[7s]]);
        write(30,f,load[j]);   writetext(30,[[4s]]);
        write(30,f+1,load[j]×postcost(belnext(rd,j),j));
    end;
end  Print;
```

The complete program now follows on the next page.

```
begin   integer  n,m,z,datanr,out,pr;
    open(20);     open(30);
    data nr:=read(20);
START:  out:=read(20);
    m:=read(20);      n:=read(20);    z:=n+m;
    begin   integer  sum,j,k,lo,sl1,sl2,p,b,count,ga,gb,cut1,
                 cut2,c,e,iterns,root,d,f,nr,lu,max,a,apex,lm,inc;
    integer array  cost[1:m,m+1:m+n],supply[1:m],demand[m+1:m+n],
                 below,nbr,shadow,loop,load,rd,lu[1:m+n];
    boolean  first;

        ( All the procedures described in this appendix
            with  Planarize, Convlu, Convrd, process,
            belnext, Success, Na, su, topleft, Treecut,
            Fixtop, nbr next and Refl are now declared. )

    iterns:=a:=b:=0;   root:=1;
    for j:=1 step 1 until m do
    begin   supply[j]:=read(20);
        b:=b+supply[j];   end;
    for j:=m+1 step 1 until m+n do
    begin   demand[j]:=read(20);
        a:=a+demand[j];   end;
```

```
for j:=1 step 1 until m do
    for k:=m+1 step 1 until m+n do
    cost[j,k]:=read(20);
if a≠b then  begin   writetext(30,[[6c]FAILURE**IN**INPUT
                     *---*DEMAND**NOT**EQUAL**TO**SUPPLY]);
                     goto  FAIL;   end;

Treeform(supply,demand);
writetext(30,[[p]CL21******AKPO/TREE/TRANSPORT[8c]
        INITIAL***SOLUTION[2c]]);
Planarize(below,nbr,z);
Convlu(below,nbr,z,lu);
Convrd(below,nbr,z,rd);
Print;

TOP:  iterns:=iterns+1;
Shad cost;
Extra 2 link;
if cut1=0then  goto FINISH;
Loop form;
Smallest link;
Recalc load;
Treecut(sl1,sl2,rd,lu);
nr:=if lo<apex then loop[1] else loop[lm];
lu:=if nr=loop[1] then loop[lm] else loop[1];
```

```
    root:=loop[if lo>apex then lm else 1];
    Newtree(nr,sl1,rd,lu);
    Fixtop(nr,lu,rd,lu);
    if(iterns+out)≠(iterns/out) then  goto  TOP;
    writetext(30,[[4c]SOLUTION***FOR***ITERATION***NUMBER]);
    write(30,format([ndddcc]),iterns);
    Print;
    goto  TOP;
FINISH: writetext(30,[[8c]FINAL****SOLUTION[c]-----****----
        ----[4c]NUMBER**OF**ITERATIONS**=]);
    write(30,format([ndddc]),iterns-1);
    Print;
FAIL:  datanr:=datanr-1;
    if datanr≠0 then goto  START;
    writetext(30,[[8c]END***OF***TREETRANSPORT[6c]]);
end;
close(20);  close(30);
end→
```

App. 5.1 Specimen Input and Output.

The minimum storage required for this method is of the order $m \times n + 6(m + n)$, where m is the number of transmitters and n the number of receivers. The program described in the last sub-appendix requires a further $2 \times (m + n)$ locations to carry the initial tree in both the below, posnbr and rd,lu representations. In order to reduce this, it would have been necessary to have split the program into two parts : one would compute the initial tree and output it in the rd, lu representation on paper tape , and the other part would read in the initial tree , analyse it and find the minimal solution. The split would occur at the label TOP: in the program. Data for input should be in the following form :-

data nr ; number of blocks of data.

out ; iteration output counter.

m ; n ;

b_1 ; b_2 ; .... ; b_n ;

a_1 ; a_2 ; .... ; a_m ;

c_{11} ; c_{12} ; . . . ; c_{1m} ;

c_{21} ; c_{22} ; . . . ; c_{2m} ;

. . . . . .

c_{n1} ; c_{n2} ; . . . ; c_{nm} ;

→→

Here is a specimen problem with the data ready for input and the output from the program.

```
1;
6;
6;  12;

10;  14;   9;  15;  11;  13;
 6;  6;  5;  7;  4;  8;  6;  5;  3;  7;  6;  9;

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

→→→

(Program output)

```
CL21     AKPO/TREE/TRANSPORT

INITIAL  SOLUTION

TOTAL  COST  IS  149
```

ROUTE		LOAD	COST
13	1	4	4
2	13	2	4
15	2	1	2
4	15	2	2
18	4	3	6
6	18	6	6
12	6	1	4
3	12	7	7
17	3	2	8
5	17	4	12
16	5	7	28
8	6	6	18
14	4	5	10
9	4	5	10
11	2	4	8
10	2	7	14
7	1	6	6

SOLUTION FOR ITERATION NUMBER 6

TOTAL COST IS 129

ROUTE		LOAD	COST
13	1	6	6
7	1	4	4
4	7	2	2
15	4	3	3
14	4	5	10
9	4	5	10
6	9	0	0
8	6	4	12
5	8	2	6
16	5	3	12

2	16	4	8
11	2	4	8
10	2	6	12
3	10	1	1
12	3	8	8
17	5	6	18
18	6	9	9

FINAL SOLUTION

NUMBER OF ITERATIONS = 11

TOTAL COST IS 109

ROUTE		LOAD	COST
17	1	1	2
5	17	5	15
12	5	6	6
3	12	2	2
9	3	0	0
4	9	1	2
7	4	6	6
15	4	3	3
14	4	5	10
6	9	4	4
18	6	9	9
10	3	7	7
8	1	3	6
2	8	3	9
16	2	7	14
11	2	4	8
13	1	6	6

END OF TREETRANSPORT

END 1M 9S