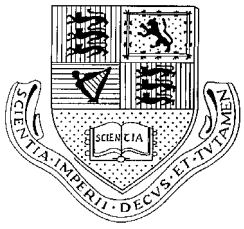
An Algorithm for Calculating the Reduced Costs on a Graph

by

Constantino Carlos Reyes Aldasoro



This report is submitted in partial fulfilment of the requirements for the Degree of Master of Science (M.Sc.) and the Diploma of Imperial College (D.I.C.)

Department of Electrical and Electronic Engineering
Imperial College of Science Technology and Medicine
University of London

September 1994

ABSTRACT

The problem of calculating the Reduced Costs of all arcs on a graph is considered. For each arc on the graph, the problem is to determine the arc with maximum cost on the fundamental path on the corresponding spanning tree. A new algorithm for this problem is proposed. It is based on the construction of a Binary Tree by sequential deletion of arcs in a descending order of costs. The tree is composed of leaf nodes representing the actual vertices in the graph and intermediate nodes representing the branches of the Minimum Spanning Tree. Using the Binary Tree, the Reduced Costs of any chord is determined by the Nearest Common Ancestor of the leaf nodes corresponding to the chord vertices. Computational results are presented for graphs of various densities. The algorithm's performance is compared to the path labelling algorithm of Carpaneto [1].

Table of Contents

CHAPTER 1: INTRODUCTION

Definitions

Let $G=(V,A)$ be a connected undirected graph composed of a finite set of vertices $V=\{1,2,3,...,n\}$, and a set of arcs $A=\{1,2,3,...,m\}$. For every arc in A there is a pair of vertices (i,j), $i \neq j$, (i,j) is an ordered pair of V and it has a related cost $c(i,j)>0$. For undirected graphs, clearly $c(i,j)=c(j,i)$, but (i,j), (j,i) are stored as two separate arcs. The *maximum cost* of any arc in G is defined as $CMAX$. Any subset of G in which all the vertices are connected and there are no loops (i.e. having n vertices and n-1 arcs) is called a *Spanning Tree*. A subset of G, $T(V,A')$, $A' \subseteq A$ such that $\sum_{\{i,j\} \in A'} c(i,j)$ is minimum, is called the *Minimum Spanning Tree* of the graph. The Minimum Spanning Tree is stored using one pointer per vertex. For each vertex v, $father(v)$ points to the predecessor of v in the spanning tree T. The *root* of the tree has no father.

The Minimum Spanning Tree will be composed of n vertices and n-1 branches and will have its root at vertex 1 by default. The *diameter* of a Minimum Spanning Tree is defined as the longest path inside the tree. After obtaining the Minimum Spanning Tree, the arcs in the graph G will be classified in two groups: *branches* for the arcs that belong to T and *chords* otherwise (i.e. in $\overline{T}$). The *degree* of a vertex represents the number of branches incident to the vertex. The *fundamental path* (P_{pq}) of a chord (p,q) is the set of branches that connect p and q. A branch (k,l) will be defined as the *critical branch* if $c(p,q)-c(k,l)$

$£c(p,q)-c(k',l')$ for every branch $(k',l') \in P_{pq}$. A critical branch is the maximum cost branch on the fundamental path.

A *Binary Tree* is a tree in which each node has degree≤3, one being its father and two being its sons. A node having no sons is called a *leaf node*. The Binary Tree structure will be the constructed from T by progressively removing the set A' and adding a new set $\{n+1, n+2, ..., 2n-1\}$ of $n-1$ *intermediate nodes* and $2n-2$ *new branches*, regarded as T'. The resulting Binary Tree is therefore the union $(T \cup T')$. The *depth* of a node will be the number of ancestors it has in $(T \cup T')$. To express the distance of a node from the root, the terms *shallow* as close to the root, *deep* away from the root will be used. The *Nearest Common Ancestor* of two nodes will be the deepest node that will be an ancestor of both nodes. The *maximum depth* of a leaf node will be the level of the deepest leaf node. The first level is reserved for an intermediate node (node $n+1$) and will be taken as level 0. The *average level* of the leaf nodes will be the average of the levels of all the leaf nodes.

The *Reduced Cost* of a chord will be $\bar{c}(p,q) = c(p,q)-c(k,l)$ where $c(p,q)$ is the cost of the chord and $c(k,l)$ is the cost of the critical branch on the fundamental path P_{pq}.

Some terms from graph theory used along the text are standard (e.g. [2], [9], [5]) The convention for the trees follows [5].

The report will be divided into 5 chapters. The second chapter will provide a description of the algorithm proposed to solve the problem, including the different subroutines employed. The third chapter will present a complete example on a general graph to illustrate the algorithm. The fourth chapter includes an analysis of the performance of the algorithm concerning computational complexity and memory locations for storage as comparison with Carpaneto's Algorithm [1]. A final chapter will include conclusions, and further work.

CHAPTER 2 ALGORITHM FOR THE REDUCED COST PROBLEM ON A GRAPH

The previous approach [1] [9] to the problem of calculating the Reduced Costs, has been through path labelling procedures. These procedures obtain the fundamental path of a chord and obtain the critical branch by successively comparing the cost of each of the branches.

In this work, a new algorithm to obtain the Reduced Costs of a graph is proposed. The main idea will be to obtain the Reduced Costs of the graph through the Nearest Common Ancestor of the delimiting nodes of a chord in a Binary Tree constructed for this purpose. The implementation of the algorithm can be divided, and will be presented, in three major sections. The first part is to obtain the Minimum Spanning Tree of the graph and define some useful variables. The second will consider the construction of a Binary Tree by the deletion of the branches of the MST. The costs of the branches will be embedded in the levels of the Binary Tree. In the third and last section, the Nearest Common Ancestor of every pair of leaf nodes (delimiting a chord) will be located, and from this, the Reduced Cost will be calculated. The first section will be briefly described, and the subroutine used is noted in the references. The other two sections will be described with its particular subroutines. Finally, the computer implementation is described.

2.1 Minimum Spanning Tree

Several implementations have been proposed to find the MST of a graph ([2], [10], [4]). These algorithms are based on the algorithms proposed by Kruskal [6] and later Dijkstra [3] and Prim [8].

The basic Prim Algorithm, the one chosen for the application, is defined as follows[1]:

Step 0. [Initialise] Label all the vertices as "unchosen"; set $T \neg$ a graph with n vertices and no arcs; choose and arbitrary vertex and label it "chosen".

Step 1. [Iterate] While there is an unchosen vertex, do step 2 and stop.

Step 2. [Pick smallest cost arc] Let *(u,v)* be a smallest cost arc between any chosen vertex *u* and any unchosen vertex *v*; label *v* as "chosen" and set $T \leftarrow T + (u,v)$

This algorithm is regarded as "greedy" for it always look for the best option in that moment without considering the effect of it in the future. The over all complexity of the algorithm is $O(n^2)$. The implementation of [4] modifies the basic Prim algorithm and reduces the computational complexity to the order of $O(n+m)$. The modification of the algorithm in [4] takes the advantage of a sparse (e.g. fewer arcs present than the possible ones) graph using a Forward Star configuration (Appendix A) instead of a matrix, this way the $O(n+m)$ complexity can be obtained. The number of arcs will vary from: $M_{\min} = n-1$, to keep a connected graph, up to: $M_{\max} = n(n-1)/2$, all the possible connections between vertices. So, in the worst case, $n+m = \frac{n(n+1)}{2}$, but can be as low as $n+m = 2n-1$. The latter, of course, would be a trivial case since the graph would be the MST by itself.

[1] Names have been changed to keep Terminology coherent.

2.2 Binary Tree Construction

Once the Minimum Spanning Tree is obtained, the next step is to transform it into a Binary Tree. The construction of the Binary Tree will be done by a sequential removal of the branches from the MST. For every branch removed a new intermediate node and two new branches will be added and the resulting tree will be rearranged. After n-1 removals, one for every branch in the MST, the Binary Tree will have been constructed. The process of removing branches (Figure 2.1) will imply for each branch to be removed:

a) deleting the branch linking vertices v (father) and w (son), this leaves two unconnected subtrees.
b) Rearranging each subtree to leave v and w as respective roots.
c) Creating a new intermediate node $(n+i)$ that belongs to T'.
d) Linking subtrees by making v and w children of the new intermediate node.

The most involved step, and the only one that will have a dynamic number of operations, is rerooting the subtrees. The rearrangement of the these will concern only one of subtrees; the one adjacent to the father of the branch; v. Only the vertices that lie on the path from v to the shallowest vertex in T (not in T') will be rearranged.

In the first case, this vertex will be to the root itself but after the first step, the path will be from the father of the branch up to the last vertex, i.e. not an intermediate node. The reorganisation of the vertices will be a modification of the parental status. Namely, the father and son of every branch on a path will swap positions for all cases except for v, who will become son of the new node. Figure 2.2 shows this rerooting when branch c-d is removed. The thicker line emphasises the path up from v to the shallowest vertex.

The levels of the tree are obtained from the root level (0 by default). Each time an intermediate node is added, its level will be obtained adding one to the level of the father. The levels of the leaf nodes will be determined once the Binary Tree is completed, and their position along the tree will not vary any more.

The process of branch removal can be done up to the moment when the degree of all nodes is either 1 or 2 i.e. every vertex is a leaf node, or a father of a single leaf node. The branch that remains will be of a smaller value than the critical branch on the fundamental path, determined by the Nearest Common Ancestor, and is allowed to remain since a single path is obtained. Nevertheless, because the computational complexity of using a stopping criterion to detect that the state of degree ≤ 2 is reached, it is better to continue the removal of branches until all the original vertices become leaf nodes. These extra steps will not imply too much work since for all cases the father, will be son of a new node and then the rerooting is not necessary and the removal process is short. The stopping criterion will imply a *O(n)* search through the array of nodes for every iteration.

The idea behind removing the branches of the Minimum Spanning Tree is the following. For a given MST like the one shown in Figure 2.3, the removal of branch (*k*,*l*), where *(k,l)* has the highest arc cost, will divide the tree into two subtrees. For any chord connecting vertices *p* and *q*, where *p* and *q* belong to the two subtrees separated by the cutset *(k,l)*, the critical branch in the fundamental path will always be (*k*,*l*). This can lead to the calculation of the Reduced Cost of any chord. For a second branch *(k',l')* to be removed, the Reduced Cost of any chord which connect the two subtrees created by the new cut will be resolved by using branch *(k',l')* the subtree will be again divided. The process is repeated for every pair of vertices (*p'*,*q'*).

2.3 Nearest Common Ancestor and the Reduced Cost

In the third section, the Nearest Common Ancestor of every pair of leaf nodes (that delimit a chord) will be found, and through a simple calculation, the critical branch in the fundamental path and the Reduced Cost of the corresponding chord will be determined.

First, the Nearest Common Ancestor is obtained following this steps:

Step 1: The levels of the leaf node pair are compared to determine which one is deeper.
Step 2 If the levels are different, climb the path of the deepest until the levels are equal.
Step 3 When they have the same level, check if they are the same. If it is, that node is the NCA, stop.
Step 4 If not, climb up once both, and continue until NCA is reached.

The process of climbing the deepest node will be defined as an *individual climb.* It will be a single node decreasing the level by one in his path to the root. In other words, the node will take the position of its father each time. This individual climb is done only by the deepest node until it reaches the level of the shallowest node. In contrast, when both leaf nodes reside in the same level, a *pair climb* will be performed. This can be done from the beginning of the process, or after a number of individual climbs of the deepest node. The pair climb is done as the individual climb; each node decreases the level by one in the path to the root until the NCA is reached.

In the best case, the father of one of the leaf nodes will be the NCA, the worst case will be that the root of the tree is the NCA and one of the leaf node is one of the deepest level. It

should be noted, that it is not possible that one of the vertices is the NCA of a pair since all the vertices will be leaf nodes in the Binary Tree.

The critical branch of the fundamental path of a chord is determined by a simple subtraction:

$$\text{critical branch } \left(\text{on the fundamental path } P_{pq}\right) = \text{NCA}(p,q) - n$$

that will lead to the definition of Reduced Cost, and the end of the algorithm.

2.4 Computer Implementation

The data of the graph can be an input to the program in both Matrix and Forward Star configurations. If the data is in a Matrix form, it will be converted to a star as it is being read and will be stored in the star variables. This will be done in order to manipulate the data in the more convenient way for the Minimum Spanning Tree construction.

The subroutine MSTREE uses the implementation of [4] to obtain the Minimum Spanning Tree. The subroutine will receive as input the values of the graph in a Forward Star configuration: (APT(·)), (ALIST(·)) and (ACOST(·)), the number of vertices (n) and arcs (m), and a dummy variable (NP1 = n+1) to indicate the termination of the data. The output produced will be a father list for each node (F(·)), and the cost for that father-son branch (BRANCHCOST(·)) and the total cost of the tree (TCOST). Other internal variables are also used, for further detail the reader should refer to [4]. In order to solve the Reduced Costs problem, it was necessary to store the costs of the arcs selected as branches in a different array.

The order in which the branches should be removed is given by their order of costs in the MST (from higher to lower). The SORTING subroutine will create the index in decreasing order of the (BRANCHCOST(·)) array. It is important not to modify the order of the original array for it will be used again afterwards. The SORTING is a Quicksort and has order $O(n\log_2 n)$ computational time. The index will be stored in (POINTER(·)) which will be used later to generate two new arrays. One will correspond to the father, and one to the son delimiting the branch; (BF(·)) and (BS(·)). The arrays will determine the proper order in which the branches should be removed. At the same time that these arrays are generated, the array (ACOST(·)) will be modified. The cost of the positions corresponding to the branches will be changed to infinity: *(CMAX*+1*)*. In the general graph, if the value of a pair *(i,j)* is greater than *CMAX*, that will mean that there is no connection between *i* and *j*. The idea is to disable these arcs from the array. By doing so, in the analysis of the Nearest Common Ancestor of a pair of leaf nodes, chords will be differentiated from branches.

The REMOVE subroutine will delete the branches from the MST and will add intermediate nodes and new branches in order to create the Binary Tree. It will use as input the father array (F(·)), and the ordered branch array (BF(·)) and (BS(·)), and will rearrange (F(·)) adding the intermediate nodes and give (LEVEL(·)) as output.

The subroutine NEAREST will obtain the Nearest Common Ancestor, the critical branch in the fundamental path and the Reduced Cost for every pair of vertices delimiting a chord. The input variables will be (F(·)), (n), (m), and (LEVEL(·)) as the Binary Tree in which the ancestors are to be looked for, (CMAX), (APT(·)), (ALIST(·)), and (ACOST(·)) to differentiate the chords from the branches, and (BRANCHCOST(·)) and (POINTER(·)) to determine the Reduced Cost. The output will be a single value, the Reduced Cost for each chord.

The two algorithms, to construct the Binary Tree (procedure REMOVE), and the Nearest Common Ancestor location (procedure NEAREST), are presented below in the form of the subroutine developed.

Procedure REMOVE

```
F(1) = 0,  F(n+1) = 0,  LEVEL(n+1) = 0          Initialising variables and
                                                 setting  first level
DO  I = 1, n-1
    IF BF(I) = F(BS(I)) THEN
       v = BF(I),  w = BS(I)                     Find father and son of the
    ELSE                                         chord and assign them to
       v = BS(I),  w = BF(I)                     v and w respectively
    ENDIF
    fv = F(v),  F(v) = n+I,  F(w) = n+I          Make v and w sons of new
                                                 node and prepare the
880    CONTINUE                                  rearrangement by swapping
       pv = v,  cond = 0                         parental status
       IF fv < n THEN
         v = fv,  fv = F(fv),  F(v) = pv
         IF fv = 0 THEN cond = 1                 The root is reached
       ELSE
         F(n+I) = fv,   cond = 1                 An intermediate node
                                                 is reached
         IF LEVEL(n+I) < 0 THEN
               LEVEL(n+I) = LEVEL(F(n+I))+1      The level is set
       ENDIF
    IF cond≠1 GOTO 880                           The tree has been
                                                 rearranged

CONTINUE                                         At the end of the cycle, the
                                                 Binary Tree has been
                                                 constructed
```

Procedure NEAREST

```
DO  I = 1, n
    DO  J = APT(I), APT(I+1)-1

        IF ACOST(J) ≤CMAX THEN          Discriminate branches from chords

          IF LEVEL(I) ≤LEVEL(ALIST(J)) THEN
             X = I                      Assign shallowest leaf node
             Y = ALIST(J)               to X and the deepest to Y
          ELSE
             X = ALIST(J)
             Y = I
          ENDIF

          LX = LEVEL(X)
          LY = LEVEL(Y)

          IF LX ≠ LY THEN               If the levels are not equal
502             CONTINUE                climb the deepest until they
             Y = F(Y)                   remain in the same level
             LY = LY-1                  individual climb
             IF LX ≠ LY GOTO 502
          ENDIF

        IF X = Y GOTO 512               When the levels are equal find out if
510        CONTINUE                     the nodes are just one, if not, climb
          X = F(X)                      up both until NCA is found
          Y = F(Y)                      pair climb
          IF X ≠ Y GOTO 510

512       NCA = X                       The NCA is used to obtain
        CRITICAL = BRANCHCOST(POINTER(NCA-n))  the critical branch and the
        REDUCED = ACOST(J)-CRITICAL     Reduced Cost
        ENDIF

    CONTINUE                            The process is repeated
CONTINUE                                for all the chords
```

2.5 Figures

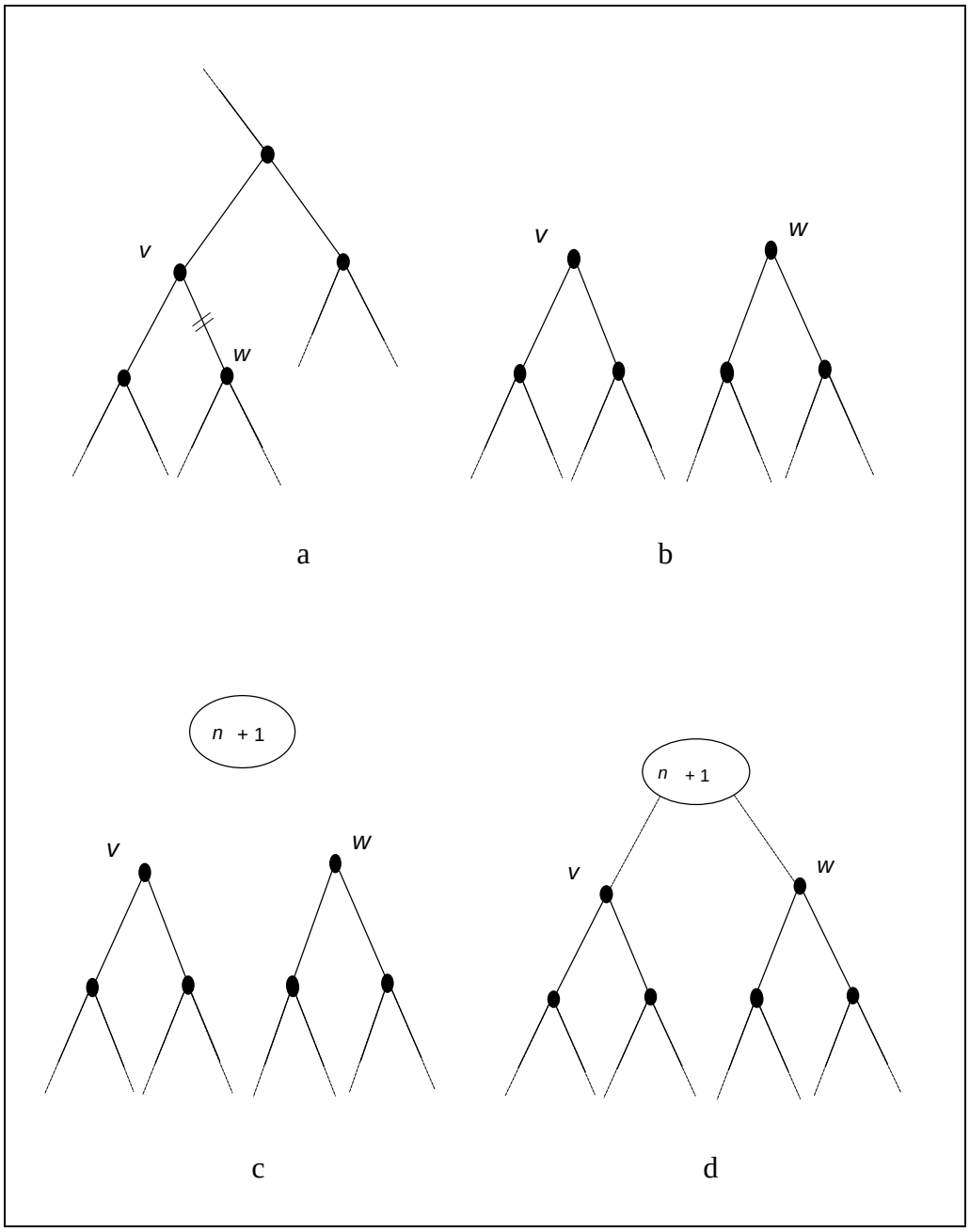
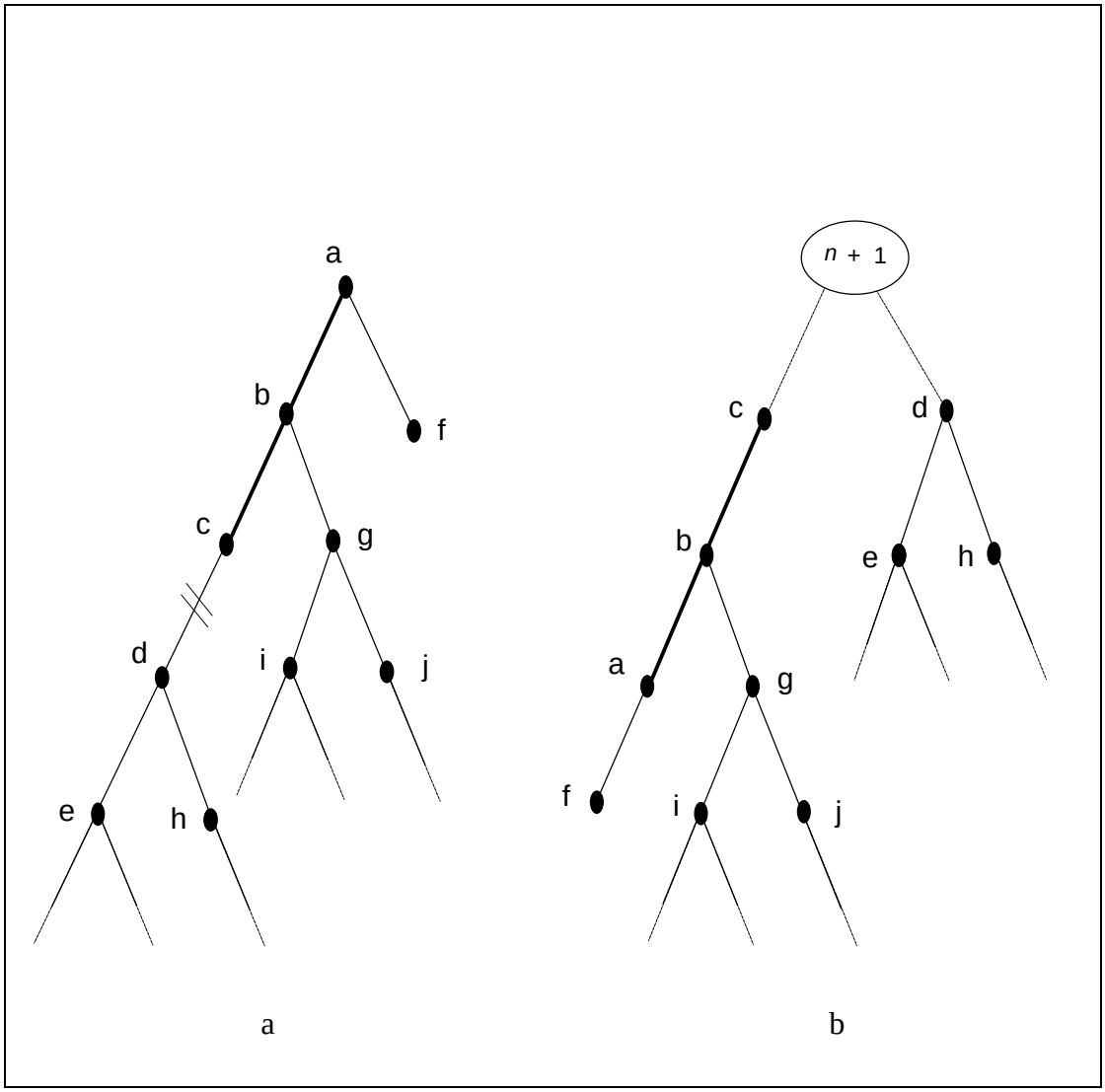


Figure 2.1



x

Figure 2.2

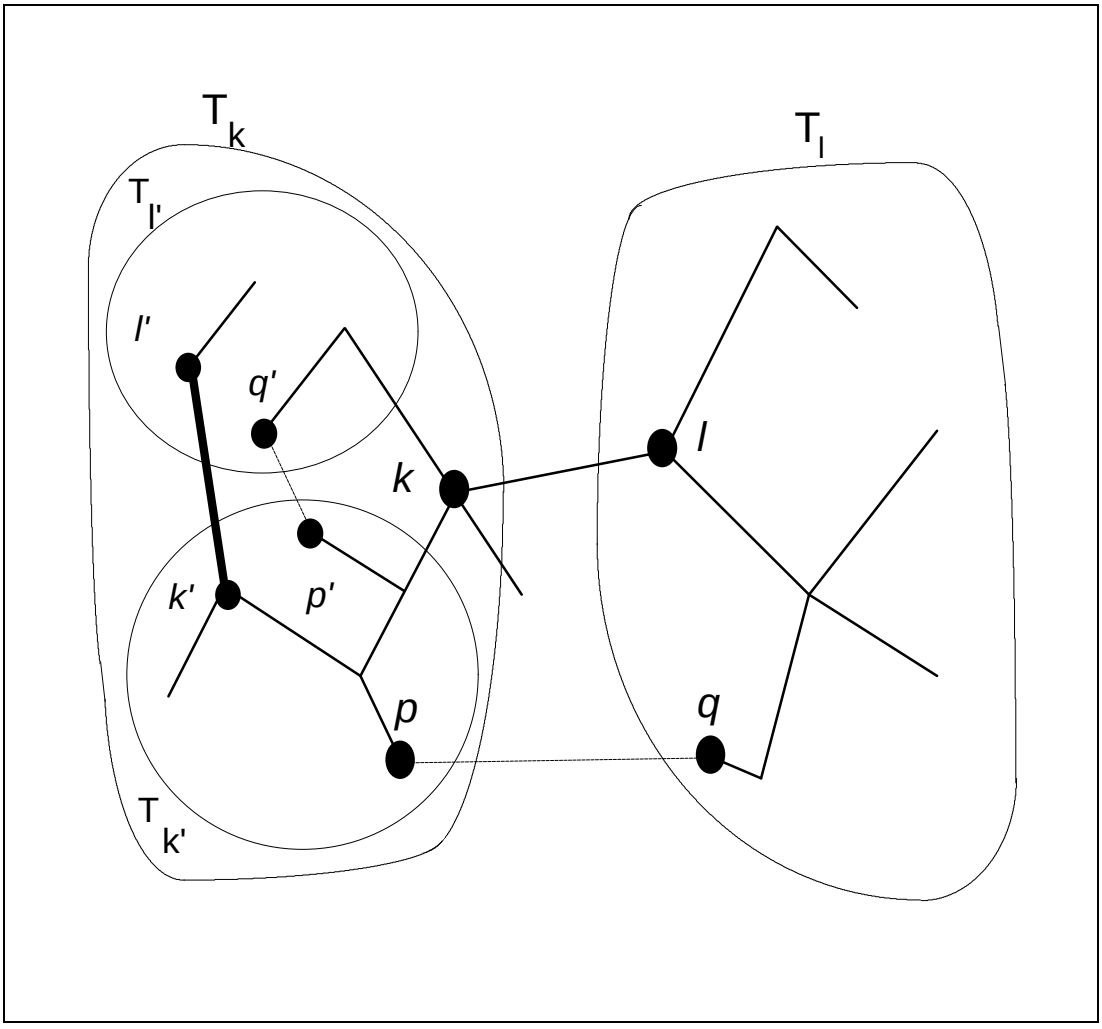


Figure 2.3

CHAPTER 3 AN EXAMPLE

3.1 Description of the example

The algorithm previously described was implemented in FORTRAN 77 running on a UNIX system. It was tested with different graphs, both general and complete, in different densities and sizes. The algorithm was successful in solving the Reduced Costs problem with graphs of size 240 vertices with 28,658 arcs.

To illustrate the algorithm, a solved example is presented. The graph shown below in Figure 3.1 represents a general graph with 16 vertices (**bold** numbers), 72 directed arcs, with a maximum arc cost of 62. The fifteen arcs in italics constitute the MST.

The Minimum Spanning Tree obtained from this graph is represented in Figure 3.2. The solid lines are the branches of the MST, and the dotted lines are the chords . The branches are sorted from highest to lowest value and labelled to be removed in that particular order to form the Binary Tree. (Figure 3.3)

The MST previously obtained is considered to be rooted at vertex 1, so it can be redrawn as in the Figure 3.4.a. The connections between vertices are the same, only that here a status of father or son is given to the vertices for a corresponding chord.

Here, a double line is depicted over a specific chord; from 7 to 6, this represents the first branch to be removed, i.e. the one with highest value. The path from the father of this

branch to the root is emphasised with a thicker line as in 3.4.b. This is done to show the reversal in order of the path from the father to the root. The first intermediate node to be added will become the root of the Binary Tree, the intermediate nodes are shown inside an ellipse and the new branches are dot and dashed lines. All these additions will be static and will not be modified along the process.

Figure 3.4.c represents the Tree after branches 7 to 6 and 8 to 11 have been removed, and intermediate nodes 17 and 18 have been added. Similar graphs will be obtained for each of the fifteen steps, one for every branch in the MST.

Figure 3.5.a represents the tree after nine steps. In this moment the critical branch in fundamental path through the MST can be obtained without further removal of branches. As explained earlier, the process will continue for all the n-1 steps, represented, with their corresponding levels in Figure 3.5.b. The characteristics of the levels are the following:

$$maximum\ depth(\ example\ ) = 7$$

$$average\ level(\ example\ ) = 5$$

Once the Binary Tree, with the respective levels for the leaf nodes, is obtained, the Nearest Common Ancestor is obtained for each chord. The NCA is related to the critical branch in the fundamental path of a chord in the following way. Suppose the chord analysed is from 13 to 14. The NCA is 18 as shown in Figure 3.6.

The path from p to q (13-14) goes 13-10-7-8-11-15-14, like in Figure 3.7. The branches have their values and the number inside the box corresponds to the order in which the branches are to be removed. The critical branch is obtained through its order in the branch set from the following subtraction:

$$\text{critical branch } (k,l) = 18 - 16 = 2$$

The value corresponding to the branch labelled as 2 is 23 and in Figure 3.7 is underlined as well as the cost of the chord; 41. The Reduced Cost of the chord is immediate from the definition, or alternatively:

$$\bar{c}(p,q) = c(p,q) - \max c(k,l) \qquad (k,l) \in P_{p,q}$$

$$\bar{c}(13,14) = 41 - 23 = 18$$

The process is repeated for every chord of the graph shown in Figure 3.8

The analysis of all the chords is listed below in the table 3.1, with the NCA for every pair of leaf nodes that correspond to a chord, the critical branch to be removed, and the Reduced Cost for the chord.

3.2 Tables and Figures

chord		NCA *(p,q)*	critical branch		Reduced Cost		
p	*q*		*k*	*l*	*c(p,q)*	*c(k,l)*	$\bar{c}$ *(p,q)*
1	3	20	3	2	47	21	**26**
1	6	17	6	7	50	25	**25**
1	7	20	3	2	34	21	**13**
2	8	20	3	2	29	21	**8**
3	4	20	3	2	38	21	**17**
3	6	17	6	7	29	25	**4**
4	9	19	9	8	34	22	**12**
5	7	17	6	7	36	25	**11**
5	10	17	6	7	62	25	**37**
5	13	17	6	7	46	25	**21**
7	11	18	11	8	39	23	**16**
8	12	18	11	8	29	23	**6**
9	11	18	11	8	35	23	**12**
9	12	18	11	8	41	23	**18**
10	11	18	11	8	34	23	**11**
11	13	18	11	8	46	23	**23**
11	16	21	16	15	44	20	**24**
12	15	23	15	11	35	15	**20**
12	16	21	16	15	39	20	**19**
13	14	18	11	8	41	23	**18**
14	16	21	16	15	36	20	**16**

Table 3.1

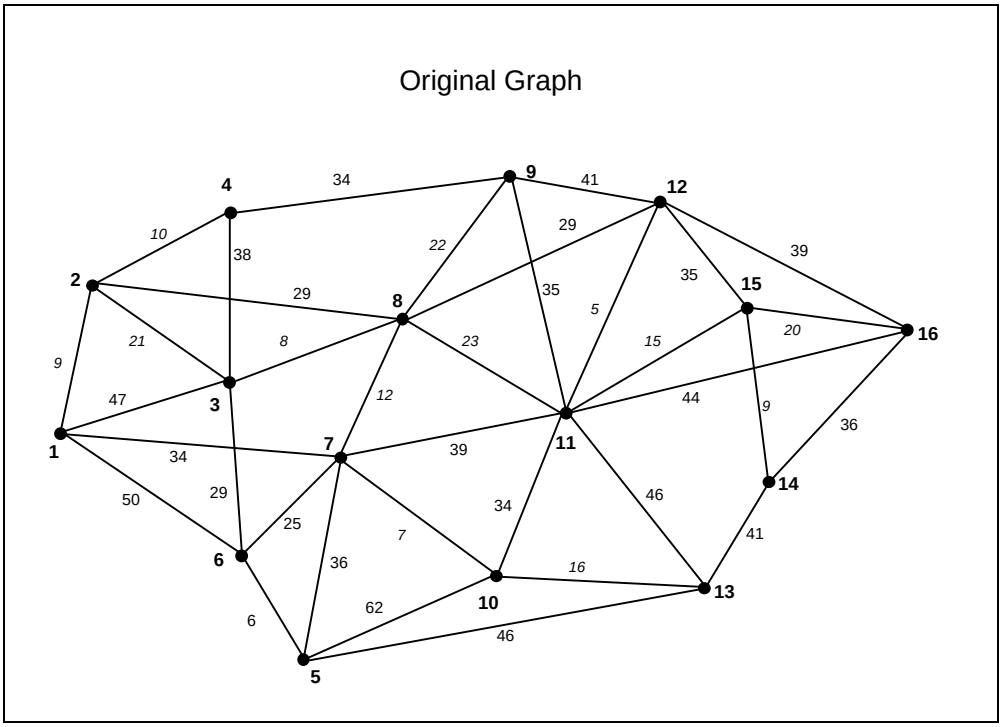


Figure 3.1

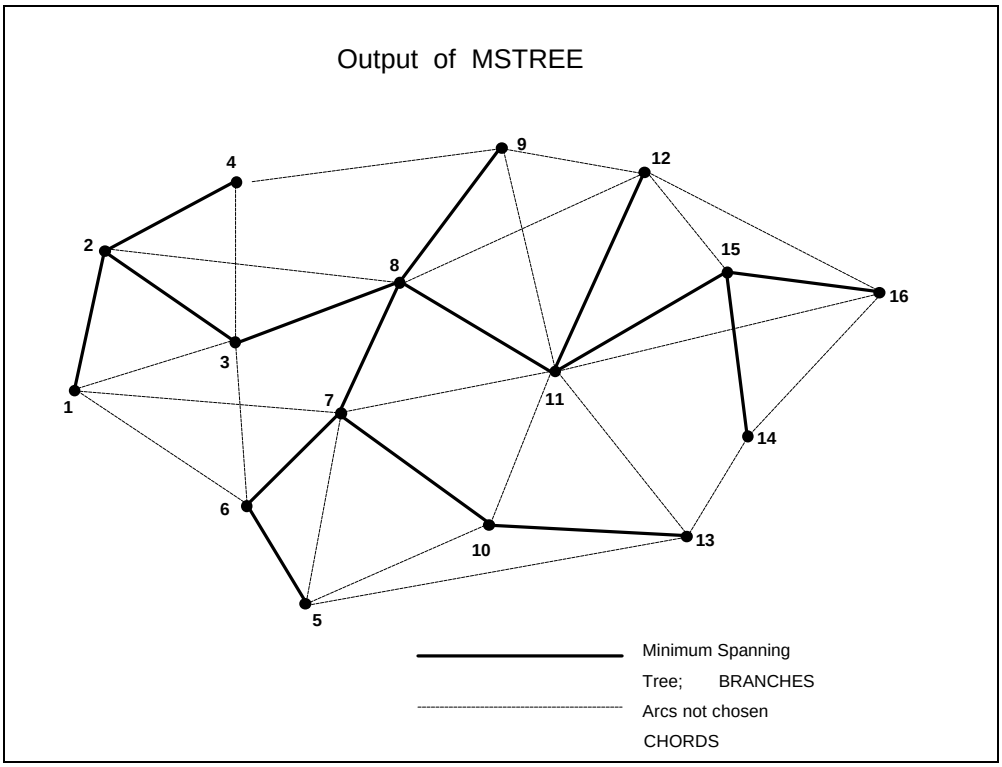


Figure 3.2

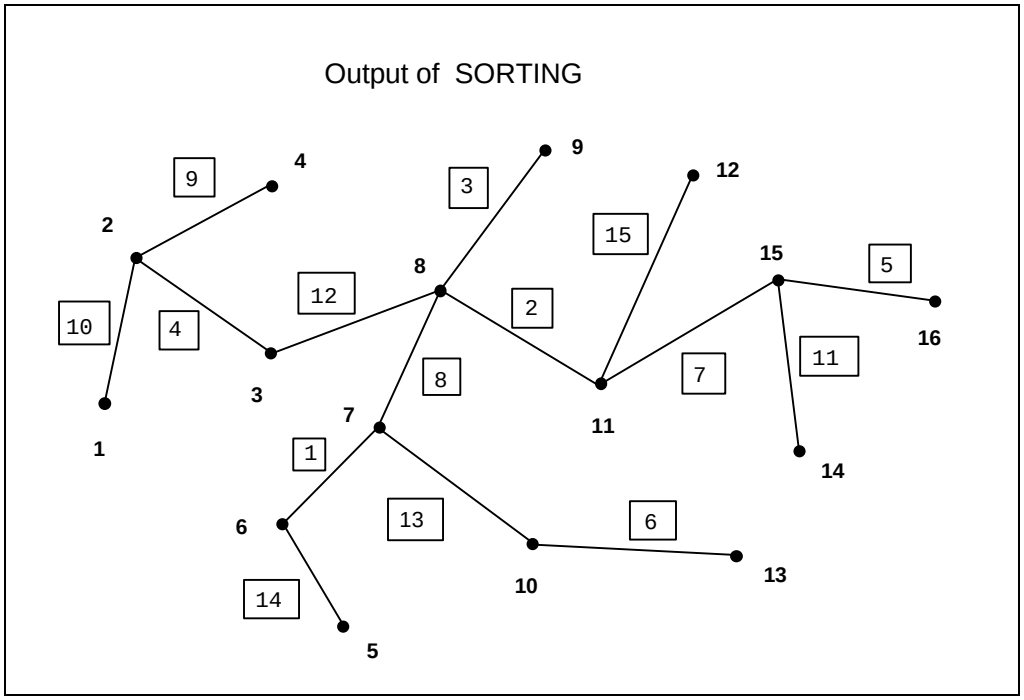


Figure 3.3

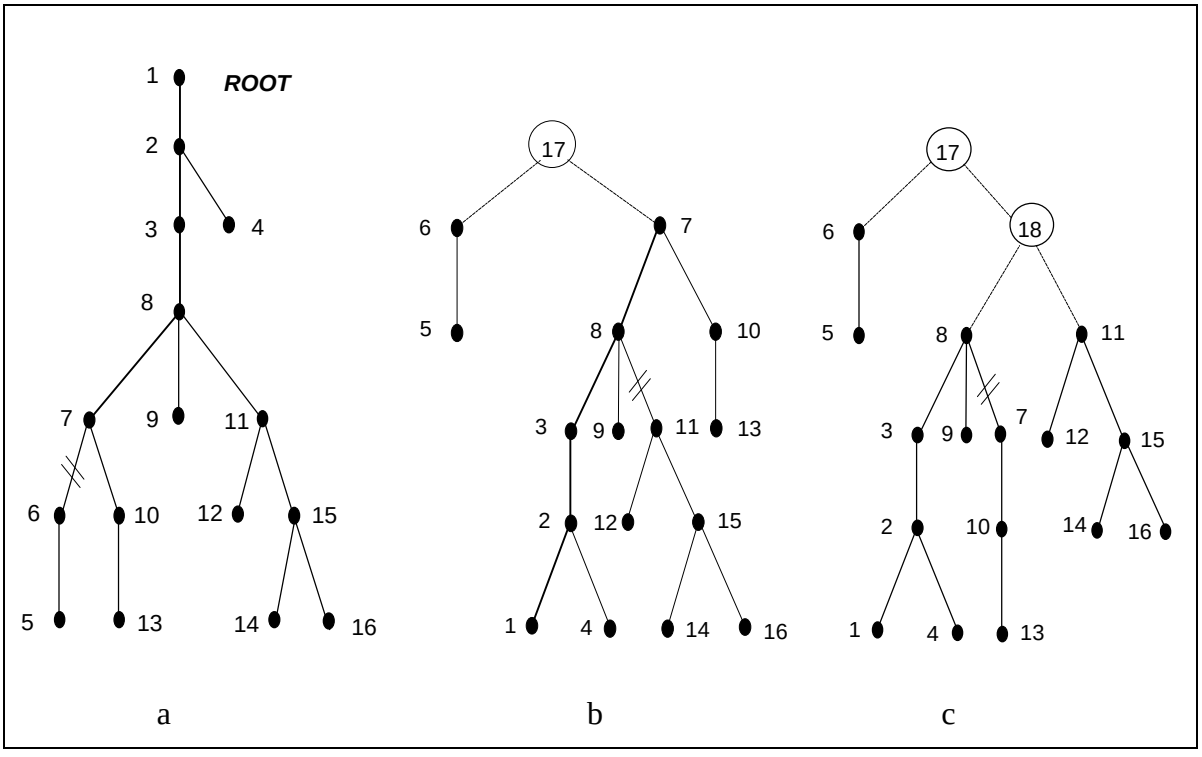


Figure 3.4

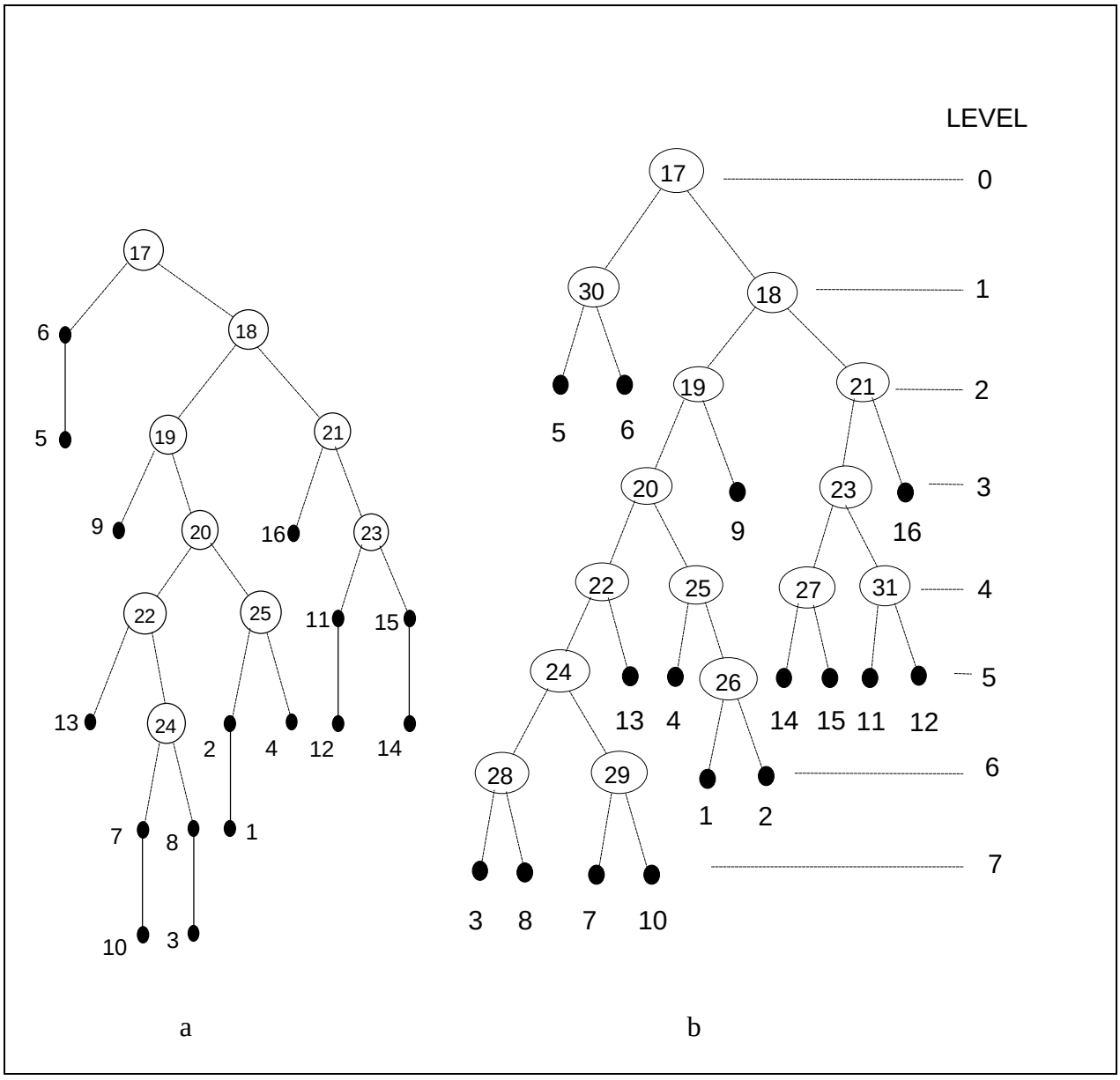


Figure 3.5

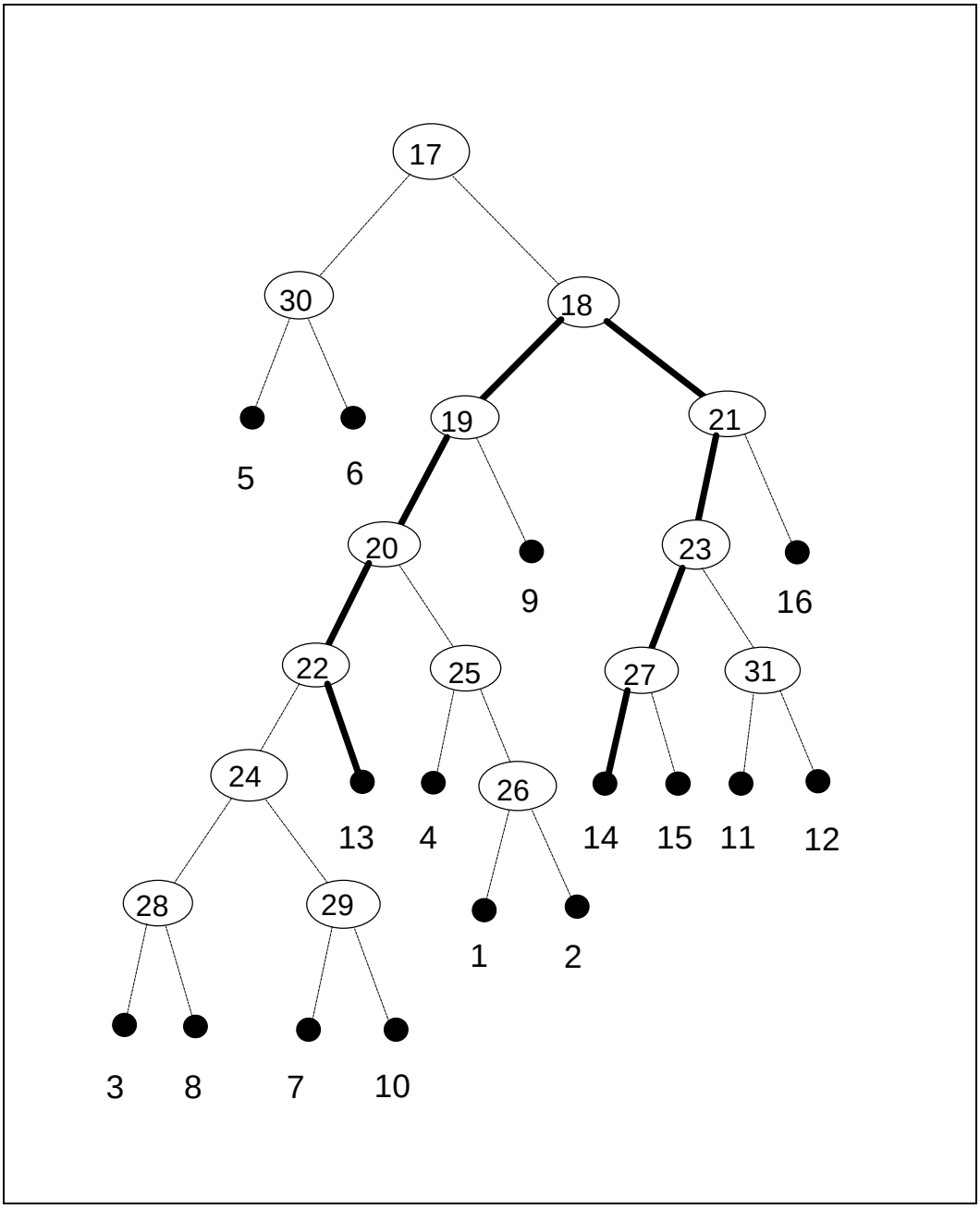

17
30
18
5
6
19
21
20
9
23
16
22
25
27
31
24
13
4
26
14
15
11
12
28
29
1
2
3
8
7
10

Figure 3.6

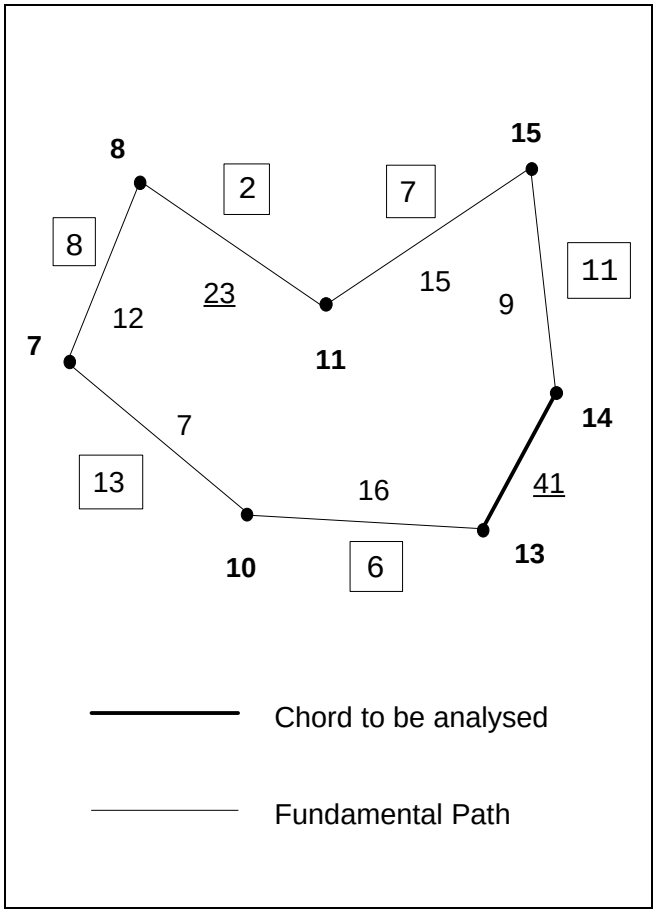

8
15
2
7
8
11
23
15
9
12
7
11
14
7
13
16
41
10
6
13
Chord to be analysed
Fundamental Path

Figure 3.7

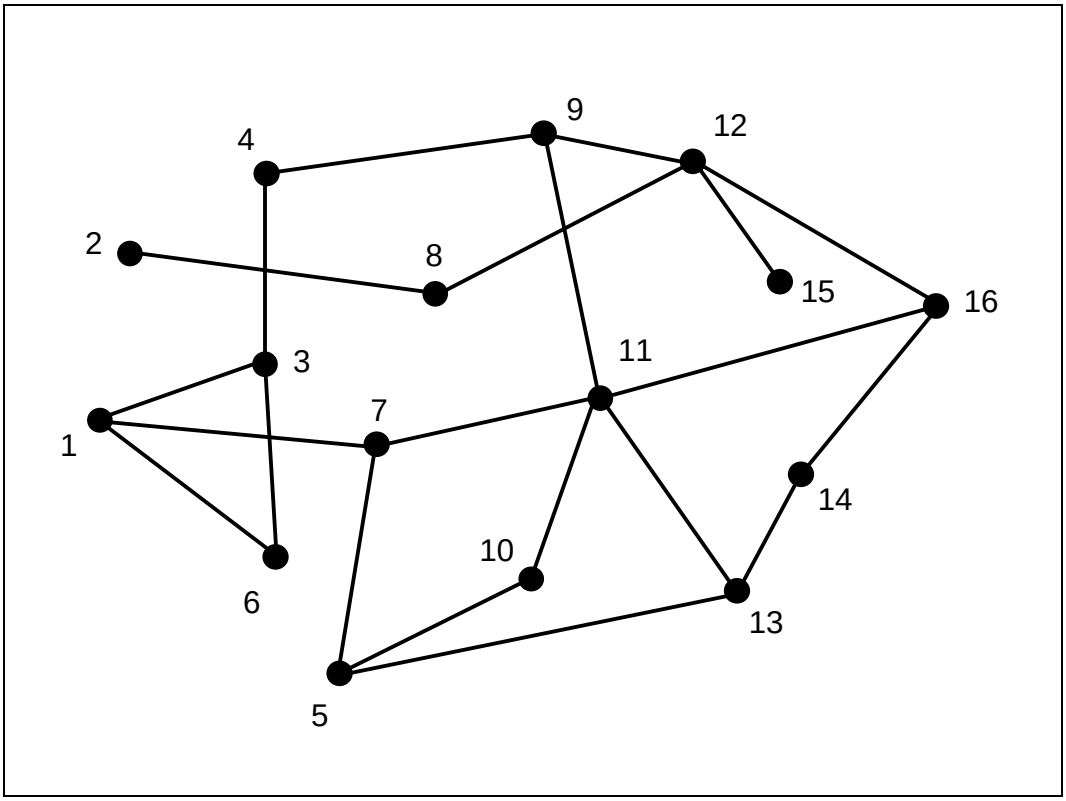


Figure 3.8

CHAPTER 4 COMPUTATIONAL PERFORMANCE

4.1 Computational Complexity and Storage Requirements

To analyse the performance of the algorithm, three important cases were studied. The three cases represent special types of the Minimum Spanning Tree obtained from a complete graph. Figure 4.1 shows a complete graph of 8 vertices, K_8. By varying the cost ordering on the arcs, the three following types of MST can be obtained.

4.1.1 Path Tree

In Figure 4.1, the arc order depending on the cost is shown in squares. The MST obtained from the complete graph in Figure 4.1 will be called a Path Tree since it forms a path from the lowest to the highest labelled vertex. This is due to the order in which the arcs will be chosen to form the MST. The diameter of this tree is given by:

$$diameter(Path\,Tree) = n - 1$$

This is the maximum diameter possible. The second important characteristic of this tree is that the order of the branches to be removed alternates from one side to another starting on the higher side as depicted in Figure 4.2. This special MST configuration provides a worst-

case scenario for the Binary Tree construction moreover a worst-case for the Nearest Common Ancestor location.

Binary Tree Construction:

This particular arrangement will imply that the vertices which will modify their parental status while removing the branches will always be the maximum possible. The series shown in Figures 4.3.a to 4.3.h represent the process of removing the branches. It should be noticed that the branch to be removed is, in all cases, the deepest. When the branch to be removed is the deepest, the number of vertices that should change their parental status will be *n* for the first remove, *n*-1 for the next one, and so on until the last remove where two vertices become the children of the last intermediate node.

The total number of changes will be of order $O(n^2 - n)$, being the worst case for a Binary Tree construction. The final state in the Figure 4.3 represents the Binary Tree. The characteristics of the levels are:

$$maximum\ depth(Path\ Tree) = n - 1$$
$$average\ level(Path\ Tree) = \frac{n-1}{n} + \frac{n-1}{2}$$

Nearest Common Ancestor location:

The process to obtain the NCA results in a case where the leaf nodes will never be on the same level, except for one pair. So, at least one individual climb will be done in every case. Besides all the individual climbs, a pair climb will be done as well.

The worst case of the NCA will be between a chord joining the deepest and shallowest leaf nodes. In the example of the 8 vertices complete graph, K_8, the two chords

are (4, 8) and (5, 8). They will have to traverse through the complete structure of intermediate nodes from the deepest level up to level one. As shown above, the maximum depth is *n*-1 and therefore the number of levels (including level zero) is maximum and will be:

$$levels = n$$

The minimum bound on the number of levels is for a balanced Binary Tree, section 4.1.3. Then, for the worst case chord, *n* climbs need to be performed. Continuing for all the chords incident to a particular vertex will diminish up to one pair climb, which is the best case. Therefore, the total number of climbs necessary for each chord incident to a vertex is $(n^2 - n)/2$. If this is repeated for the *n* vertices the total number of operations will be of order $O(n^3)$. This is the case for a complete graph that is obviously the worst case. This number of chords for a complete graph is:

$$chords = \frac{(n-1)(n-2)}{2}.$$

4.1.2 Star Tree

The Star configuration can be obtained from the complete graph shown in Figure 4.4. The only difference with Figure 4.1 is the ordering of arc values that will generate a MST shown in Figure 4.5.a. This Star Tree will have special characteristics for the generation of the Binary Tree and the NCA determination. In particular, the structure has the best case performance for the Binary Tree Construction and shares with the Path Tree the worst case performance for the NCA location.

Binary Tree construction:

For the Binary Tree construction, the ordering of the arc costs in this star is not relevant. Any order will give the same number of operations for the construction.

The remove process is performed on any branch connecting a vertex to the central vertex. At any stage in the construction, the central vertex is always the father of all remaining vertices (i.e. the ones not affected by previous removals). As shown in the example, the branch vertices simply take the new intermediate node as their new father. There are no parental swaps since there is no path of original vertices between the father and the nearest intermediate node. For each branch, the removal process involves a fixed number of operations, and a $O(n)$ complexity as a whole. This can be achieved in linear time. The sequence from b to h in Figure 4.5 shows how the path to be rearranged is not composed of several vertices, unlike the Path Tree. This is the case although the branch to be removed is also in the deepest level.

After completing the Binary Tree construction, the resulting structure is the same as that for the Path Tree. The only difference is the vertices' levels.

4.1.3 Water Wheel Tree

Finally, a third special case of the MST, now called the water wheel is presented. The arc cost ordering of a the anti-symmetric MST on K_8 is shown in Figure 4.6.

This tree has the characteristic that during the removal process the parental status change will be minimum. The number of branches to be removed for each level of the tree goes increasing in geometrical way; 1, 2, 4, 8,... as shown in the sequence of Figure 4.7.a to

4.7.d. The Binary Tree obtained (Figure 4.7.d), will be a balanced structure. A Binary Tree with a balanced structure is the one that has the minimum depth. If the number of vertices is a factor of 2^x where x is an integer, the tree will have a symmetric structure and is called a Complete Binary Tree. In this case, the number of levels will be determined by the number of vertices according to the following formula:

Let

$$N = 2n \qquad \text{(number of nodes in the Binary Tree plus one)}$$

and then

$$levels = \lceil \log_2 N \rceil$$

This will be the minimum bound for the number of levels in the Binary Tree. The maximum number as expressed for the Star and Path Trees will be *n*. For a Complete Binary Tree case, the maximum depth will equal the average level, since all the leaf nodes are at the same depth.

$$maximum\ \ depth(Balanced\ Tree) = average\ level\ (Balanced\ Tree) = levels$$

This gives a best case performance for the Binary Tree construction and for the Nearest Ancestor location, therefore leading to a best overall performance case.

The remove process for the construction of the Binary Tree will be like in the Star case, order $O(n)$ since the branch to be removed will always have as father a node that will be son of either the root or an intermediate node.

The Nearest Common Ancestor location will be quite different since all the leaf nodes will be on the same level. This means that no individual climbs will have to be done. The number of climbs for all the arcs in the graph will be then:

$$climbs = \sum_{i=1,}^{levels_{\min}} i(2^{i+2})$$

Even for the best case, the computational complexity of the algorithm is still of order $O(n^2)$.

4.2 Storage:

The dynamic memory requirements for the program are the following:

7 arrays of size *n*	APT, BRANCHCOST, LABEL, CHAIN POINTER, BS and BF
2 arrays of size *m*	ACOST, ALIST
2 arrays of size 2*n*	F, LEVEL
1 array of size *CMAX*	ADRS

The variables (APT(·)), (ACOST(·)) and (ALIST(·)) will store the information of the graph in a forward star configuration. This can result in a more efficient storage if the graph is sparse. For a complete graph though, the matrix configuration is better because it requires only one location to store the information of links between vertices and their cost, while the star will need one for each. Nevertheless, the usage of the Forward Star in the MST construction results in a faster algorithm, capable of analysing graphs fairly big [4]. Appendix A shows a comparison of memory spaces using Matrix or Forward Star configurations in relation with the sparsity of the graph.

(ADRS(·)), (CHAIN(·)) and (LABEL(·)) are used only in the Minimum Spanning Tree generation, they are local variables and will not be used further on. (F(·)) is of size 2*n* since it will be the predecessor list for the vertices of the graph and the intermediate nodes

added for the Binary Tree. (F(·)) and (BRANCHCOST(·)) are generated in the MST subroutine and will be used along the program.

The SORTING subroutine will generate an index array to determine the order in which the branch should be removed, but will not modify the array itself since it will be used again for calculating the Reduced Costs. The array sorted will be (BRANCHCOST(·)) and the index will be (POINTER(·)). This index will be used to form the arrays BF(·) and BS(·) which will keep the vertices defining the branches to be removed. Finally, (LEVEL(·)) will store the position in the Binary Tree for vertices and intermediate nodes.

The subroutine used for the MST by itself will need $4*n + 2*m +$ (CMAX) memory locations, the other arrays $(3*n+2*2n)$ do not represent a considerable increase for the memory.

4.3 Experimental Performance

The algorithm was implemented in FORTRAN Language in a UNIX system. To evaluate the performance, a previous approach to the problem [1] was also implemented and run with the same experimental data. Interesting results were obtained by observing iterations and the CPU time for the program subroutines. After various graphs of different sizes and densities were analysed, the following results were obtained: for complete graphs and general, nearly complete graphs, the algorithm shown in [1] requires a much smaller number of iterations to solve the Reduced Costs problem, but as the general graphs become more sparse, the number of iterations is considerably reduced. For general graphs with a densities less than 23% of the total possible arcs[2], the algorithm proposed was faster.

[2]This result is the average obtained of the general random graphs tested.

Tables 4.1 to 4.3 show the results obtained for some of the graphs tested. Figures 4.8 to 4.10 plot the average made of the CPU times for each group of graphs, and 4.11 an overall average of the process.

To determine the density of the graph, a series of complete graphs with random values for the arcs were generated. The value *CMAX* was used as a threshold, this value was very close to the actual density of the graphs.

The CPU times for the existing algorithm tend to be constant, with some variations, for the series of graphs analysed. On the other hand, the algorithm proposed in this project, is decreasing monotonically as the density grows smaller. This becomes particularly important when the 23% of density is reached; below this value the CPU times (on average) of the proposed algorithm become smaller that the existing algorithm. For densities around 5% the reduction in CPU time can be three to four times smaller.

As the size of the graph is increased, the advantage of this reduction becomes more important. The densities of general graphs used for real applications (road maps, train railings) are not close to the complete graphs that are only interesting for theoretical analysis. The graphs tested were not very similar to the theoretical cases of section 4.1. This can be seen in Table 4.4 that shows the average level and maximum depth obtained for the graphs previously mentioned and the general graphs.

The memory locations required for the storage of the array, not concerning the graph definition are about the same. The important difference relies in the graph characteristics storage. The algorithm in [1] uses a matrix while the algorithm proposed uses the Forward Star. The matrix does not take advantage of the sparsity of the graph, contrary to the Forward Star that reduced the amount of memory as the graph grows more sparse (Appendix A).

4.4 Tables and Figures

n	*CMAX*	number of arcs (directed)	Density (%)	CPU time (seconds 10^{-2})								Average (seconds 10^{-2})	
				Proposed Algorithm				Carpaneto Algorithm				Proposed	Carpaneto
100	100	9900	100.00	9	10	11	11	2	4	3	3	10.25	3
	50	4948	49.98	5	5	6	5	3	3	3	3	5.25	3
	40	3914	39.54	4	4	5	4	2	3	2	2	4.25	2.25
	35	3412	34.46	4	4	4	4	2	2	3	3	4	2.5
	30	2968	29.98	4	3	5	4	3	2	4	2	4	2.75
	27	2670	26.97	1	3	2	2	2	3	3	2	2	2.5
	25	2476	25.01	2	2	3	2	3	3	2	4	2.25	3
	23	2270	22.93	2	2	2	2	2	4	3	3	2	3
	20	1968	19.88	2	1	2	2	3	3	3	2	1.75	2.75
	15	1454	14.69	1	2	1	1	4	3	4	3	1.25	3.5
	10	940	9.49	1	1	1	1	4	1	3	3	1	2.75

Table 4.1 Graphs with 100 nodes

n	*CMAX*	number of arcs (directed)	Density (%)	CPU time (seconds 10^{-2})								Average (seconds 10^{-2})	
				Proposed Algorithm				Carpaneto Algorithm				Proposed	Carpaneto
150	100	22350	100.00	24	24	24	24	5	2	5	5	24	4.25
	50	11116	49.74	12	13	14	13	7	8	5	6	13	6.5
	40	8912	39.87	10	11	10	11	6	8	5	7	10.5	6.5
	35	7836	35.06	8	10	8	9	5	3	7	5	8.75	5
	30	6764	30.26	6	7	8	7	6	7	6	3	7	5.5
	27	6118	27.37	6	6	6	8	4	7	6	6	6.5	5.75
	25	5680	25.41	6	6	6	6	8	6	7	8	6	7.25
	23	5224	23.37	5	6	6	6	5	5	6	7	5.75	5.75
	20	4580	20.49	4	4	6	5	5	8	4	5	4.75	5.5
	15	3378	15.11	5	3	3	3	6	5	4	5	3.5	5
	10	2242	10.03	4	2	2	2	6	7	4	6	2.5	5.75
	5	1094	4.89	2	1	1	1	5	3	6	4	1.25	4.5

Table 4.2 Graphs with 150 vertices

n	*CMAX*	number of arcs (directed)	Density (%)	CPU time (seconds 10^{-2})								Average (seconds 10^{-2})	
				Proposed Algorithm				Carpaneto Algorithm				Proposed	Carpaneto
220	100	48180	100.00	56	56	55	56	16	15	16	16	55.75	15.75
	80	38596	80.11	45	45	46	45	13	15	12	14	45.25	13.5
	50	24060	49.94	28	27	27	29	11	10	11	11	27.75	10.75
	40	19304	40.07	23	22	24	24	12	14	14	13	23.25	13.25
	35	16910	35.10	17	23	22	23	13	10	12	12	21.25	11.75
	30	14532	30.16	17	16	17	19	15	17	16	14	17.25	15.5
	27	13122	27.24	17	15	15	16	14	12	14	11	15.75	12.75
	25	12182	25.28	14	15	16	15	14	8	9	10	15	10.25
	23	11246	23.34	12	13	11	11	13	10	12	12	11.75	11.75
	20	9730	20.20	11	9	9	8	11	11	12	12	9.25	11.5
	15	7210	14.96	8	9	8	8	9	9	11	10	8.25	9.75
	10	4786	9.93	5	5	4	6	15	10	13	14	5	13
	5	2344	4.87	2	2	3	2	9	10	14	8	2.25	10.25

Table 4.3 Graphs with 220 vertices

n	*average level*			*maximum depth*		
	random	Path Tree	Balanced Tree	random	Path Tree	Balanced Tree
100	30.39	50.49	6.71	46.17	99	7
150	39.85	75.493	7.853	57.2	149	8
220	52.14	110.495	8	79.5	219	8

Table 4.4 Comparison of average level and maximum depth

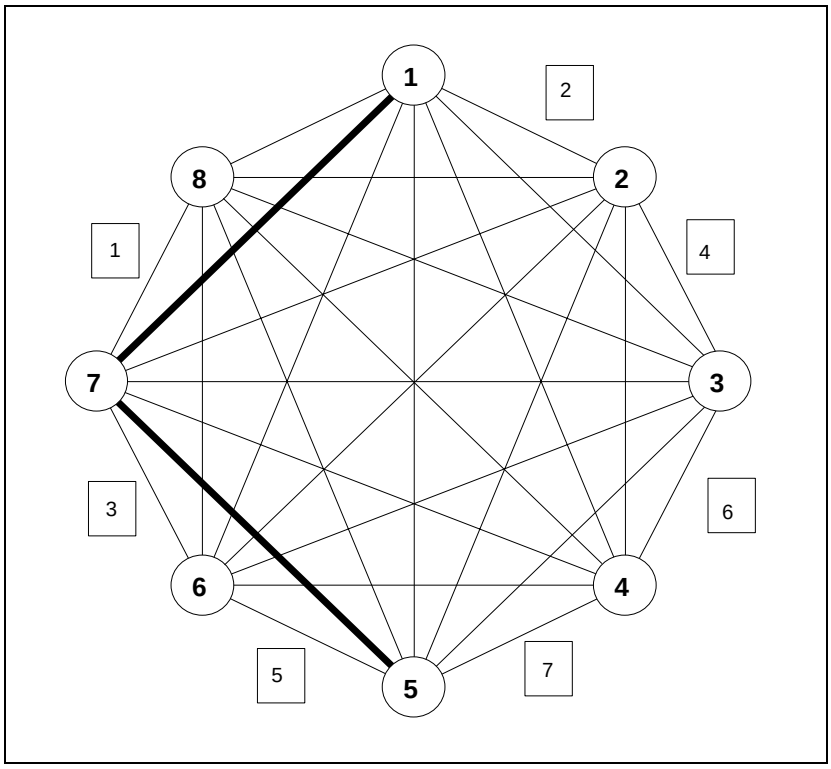


Figure 4.1

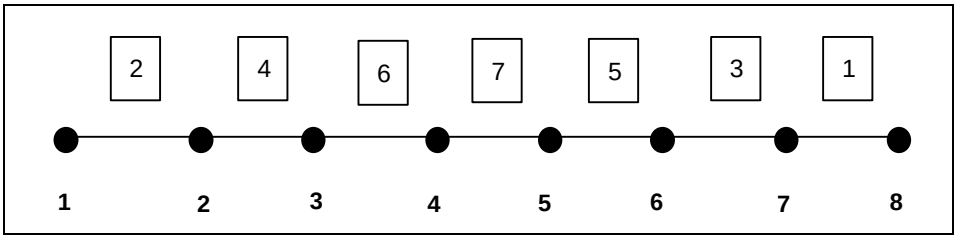


Figure 4.2

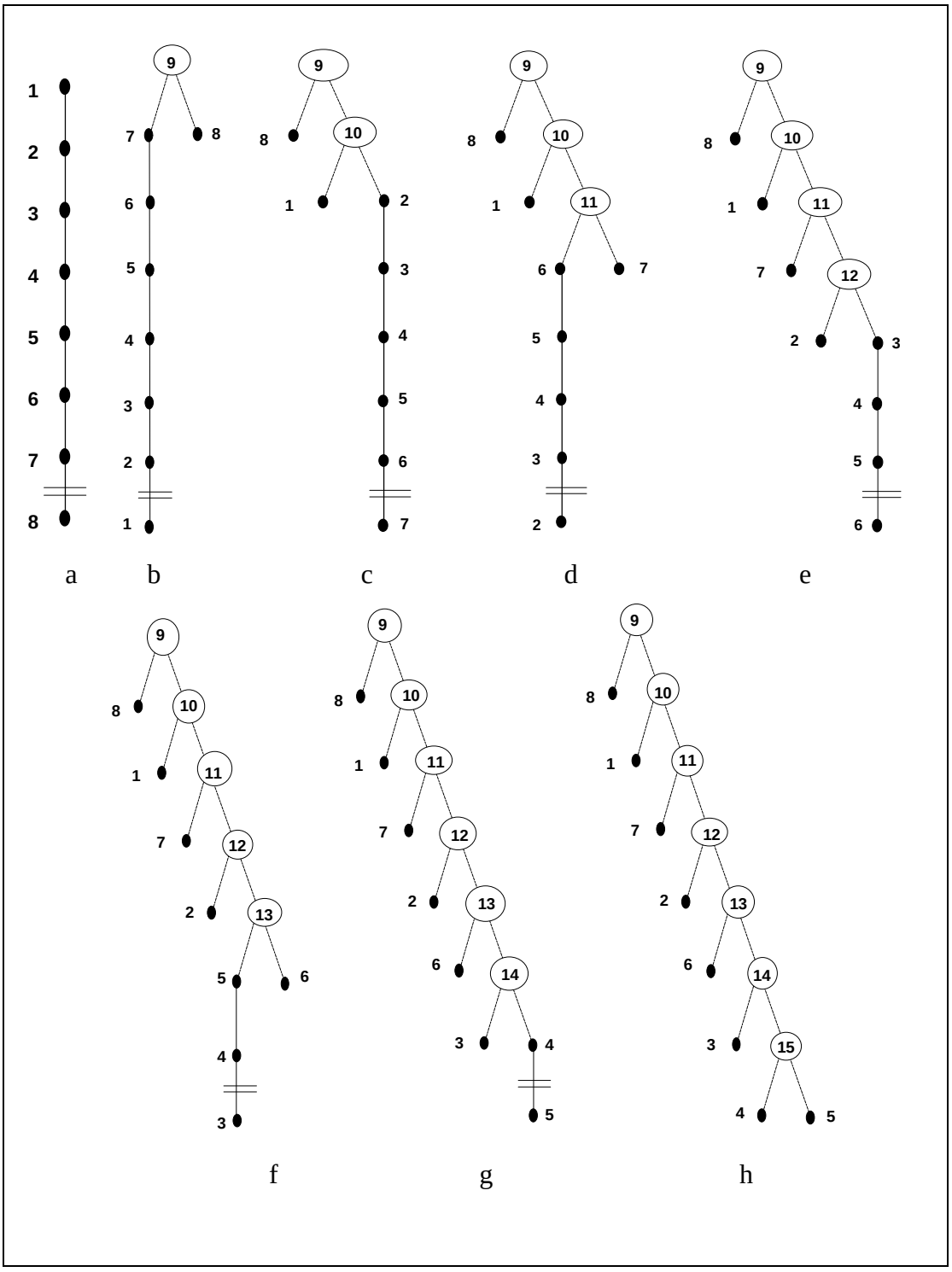


Figure 4.3

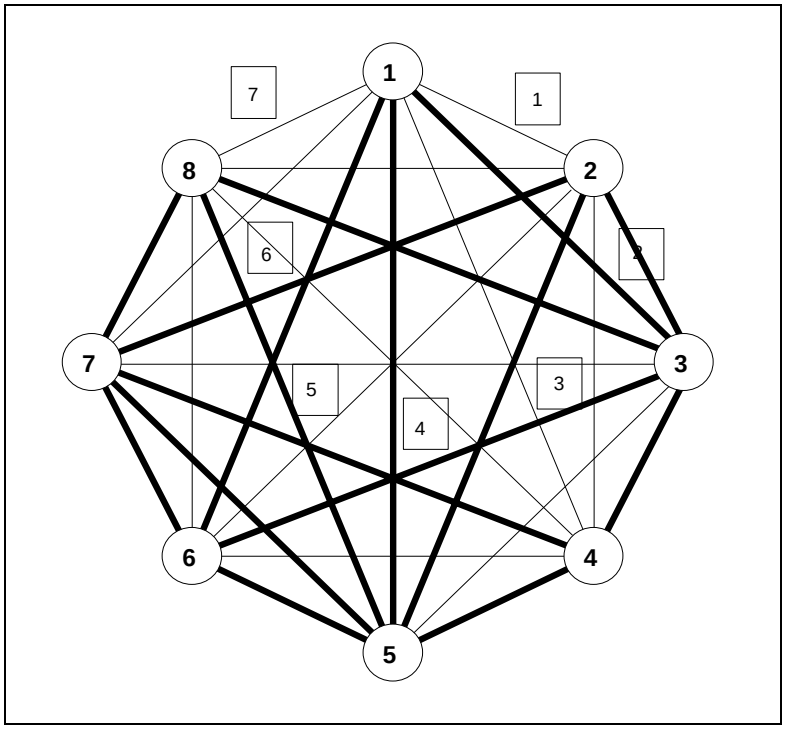


Figure 4.4

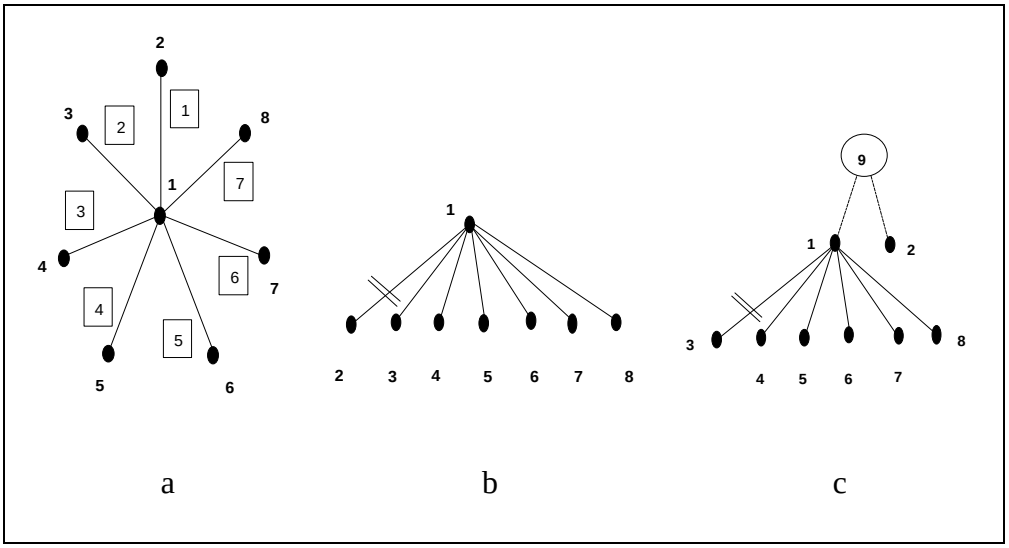


Figure 4.5

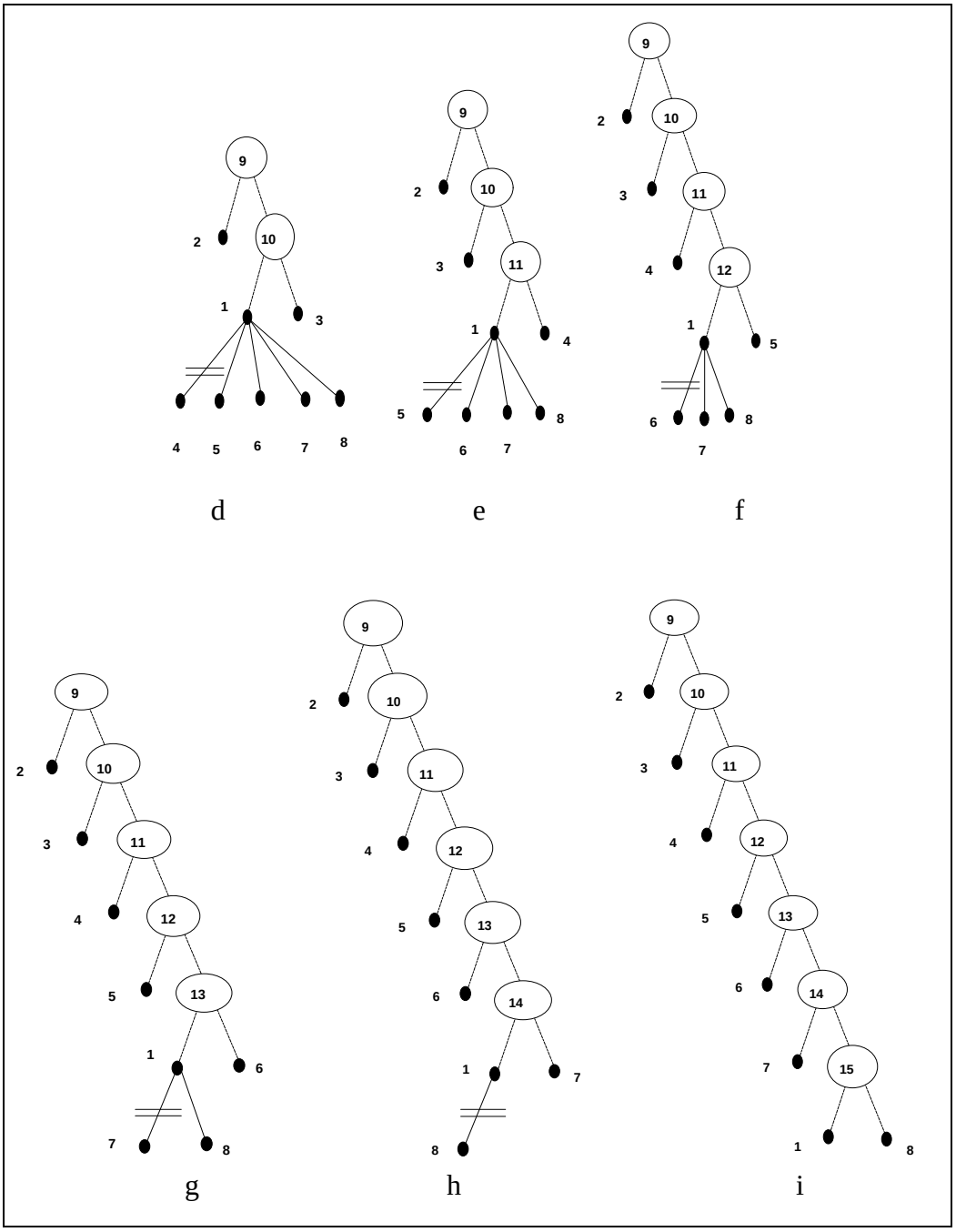


Figure 4.5 (continued)

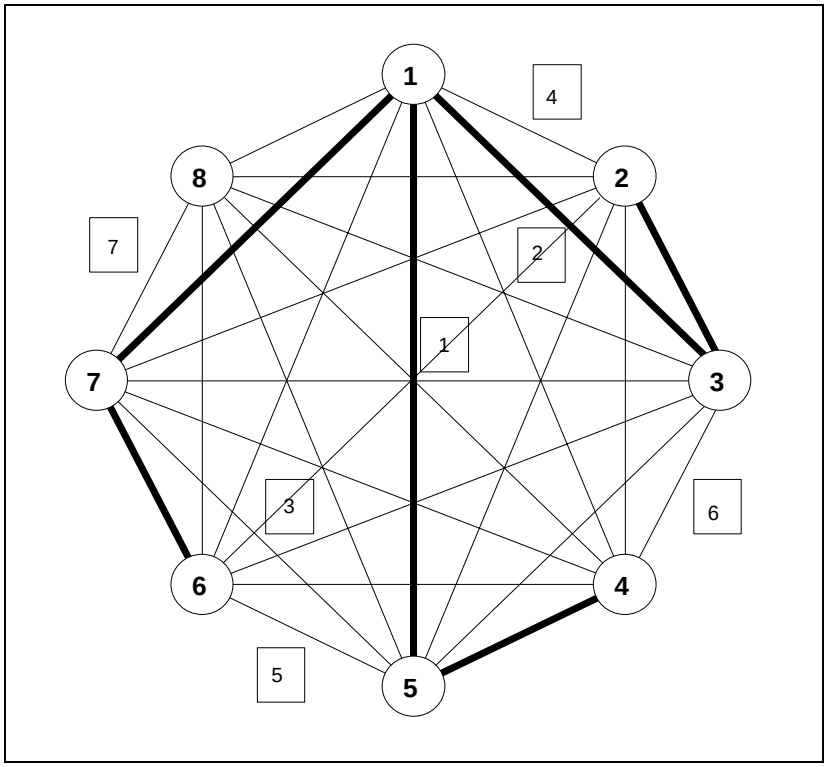


Figure 4.6

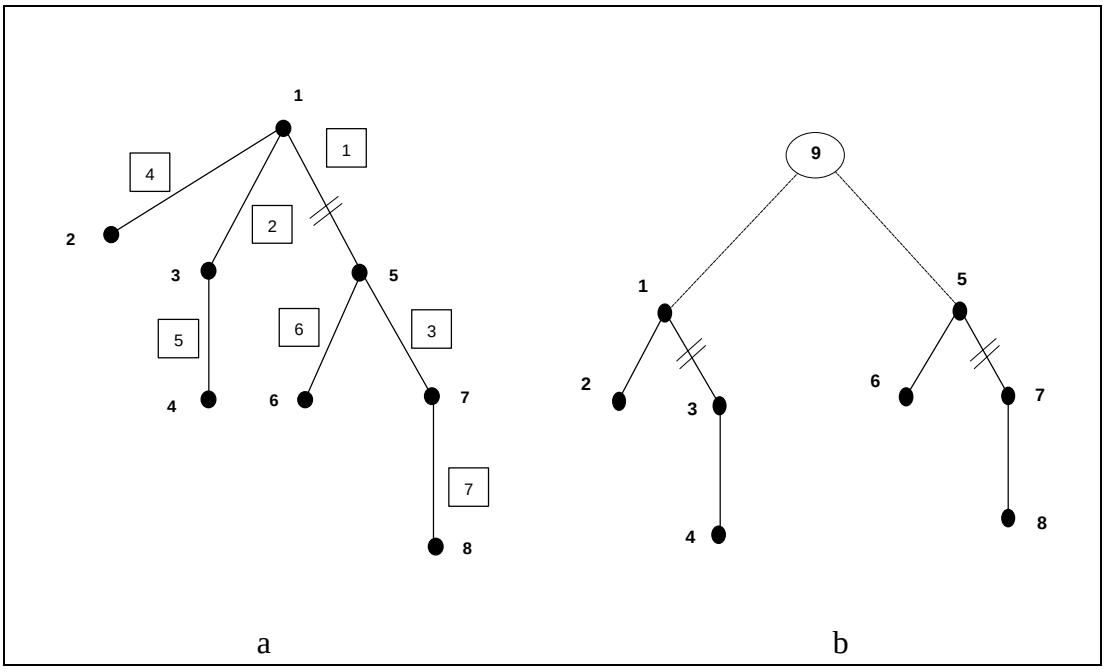


Figure 4.7

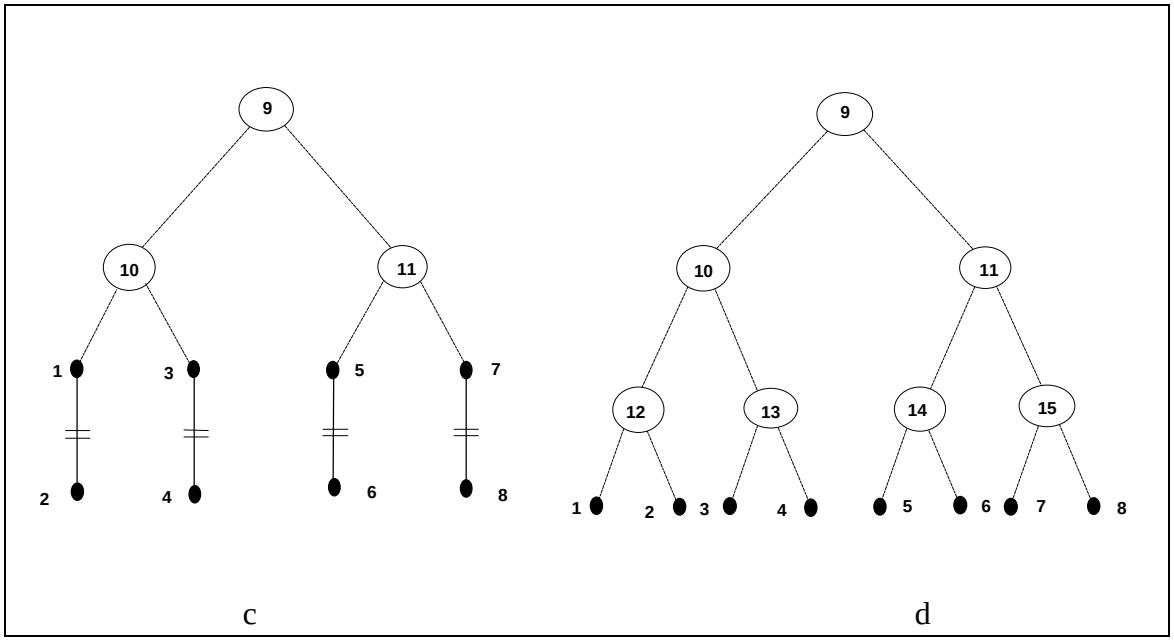


Figure 4.7 (continued)

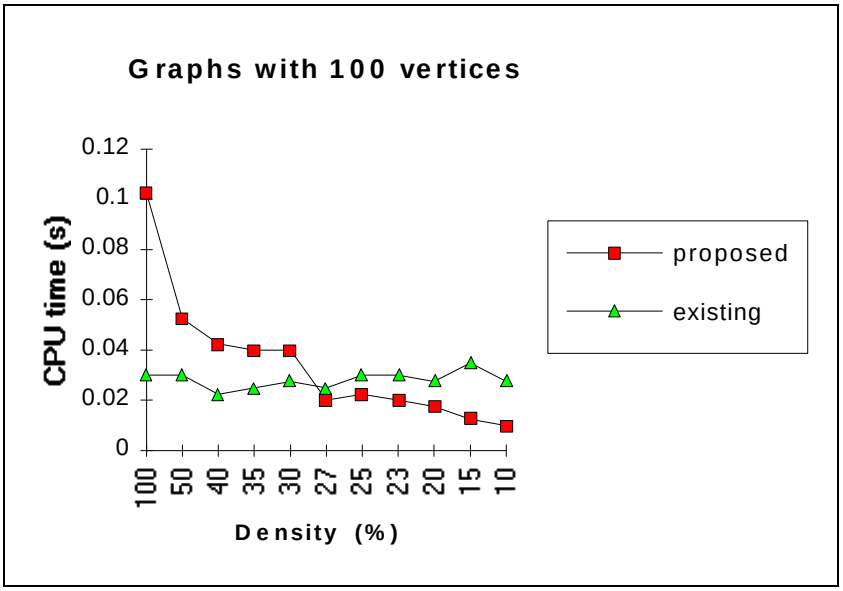


Figure 4.8

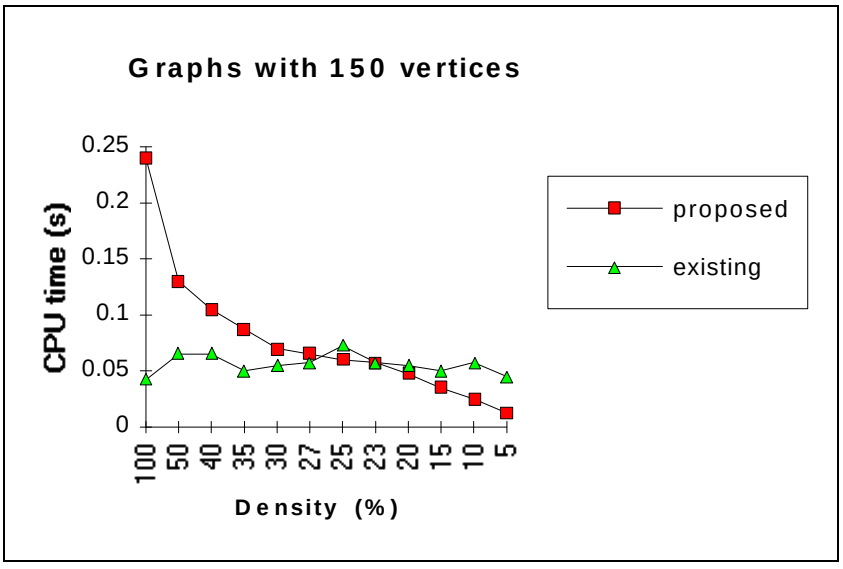


Figure 4.9

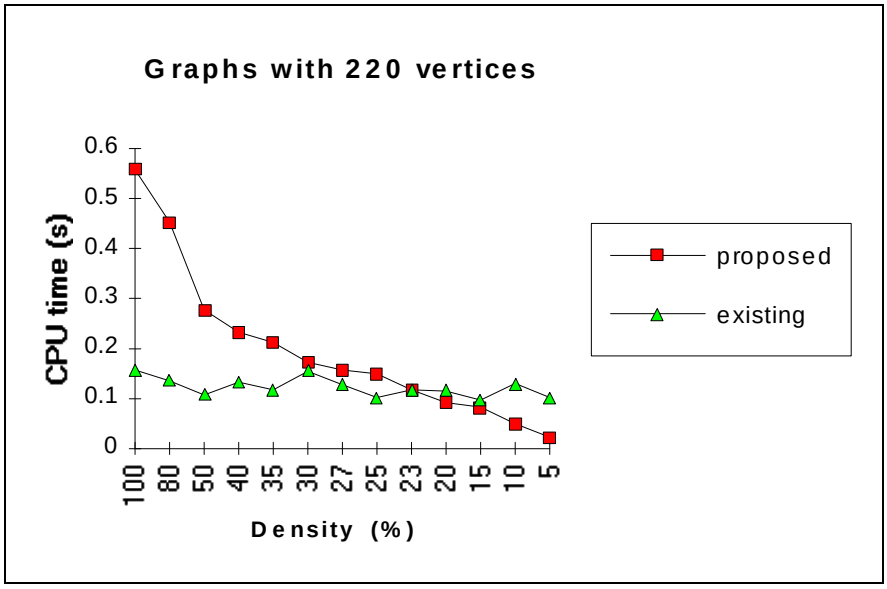


Figure 4.10

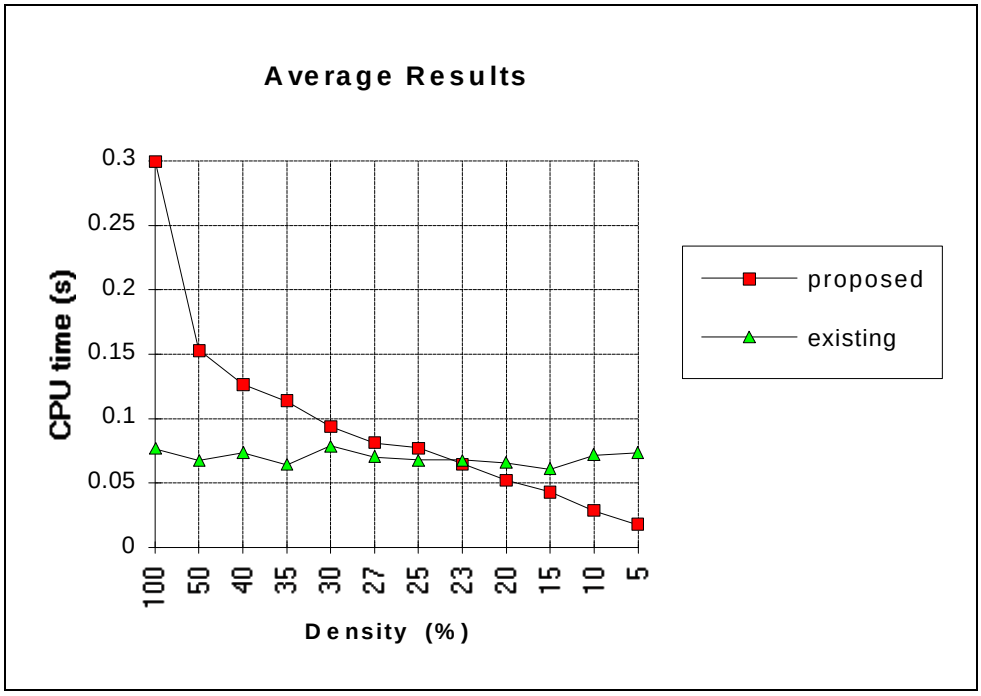


Figure 4.11

CHAPTER 5 CONCLUSIONS

A new algorithm for solving the Reduced Costs problem was proposed and successfully implemented. The algorithm was based in the construction of a Binary Tree from the Minimum Spanning Tree of a graph. This Binary Tree provided enough information, through the Nearest Common Ancestor location, to calculate the Reduced Cost of all chords of the original graph.

Extensive number of experiments was carried out, with different structures for the Minimum Spanning Tree, as well as graphs with random connections and costs. After the results were compared with an existing algorithm, it was noticed that for general sparse graphs, the number of iterations required to solve the Reduced Costs problem was smaller for the proposed algorithm. For complete and very dense graphs, the previously existing required a smaller number of iterations. Graphs having around 23% or less of the possible arcs incident to each vertex were solved faster with the proposed algorithm. This covers a large range of graphs with practical applications in Operational Research problems.

Further work over this algorithm should be done to reduce the computational complexity, specially for the worst case analysed. This should be reduced to compete with the previously existing algorithm when the Reduced Costs of dense graphs are being required. Maier [7] proposed a rather complicated structure to store the information of a tree that can lead into a faster algorithm. It can be studied if the structure can be used for the Reduced Costs solution based on the NCA of the Binary Tree.

REFERENCES:

[1] Carpaneto, G. Fichetti, and Toth, "New Lower Bounds for the Symmetric Travelling Salesman Problem", *Mathematical Programming*, V5 1989

[2] Cheriton, D., Tarjan, R.E., "Finding Minimum Spanning Trees", *SIAM Journal of Computing*, V5 (1976).

[3] Dijkstra, E.W. "A note on two problems in connection with graphs", *Numerical Math.* 1, 5 (Oct. 1959).

[4] Haymond, R.E., Jarvis, J.P., Shier, D.F., "Algorithm 613 Minimum Spanning Trees for Moderate Integer Weights", *ACM Transactions on Mathematical Software*, V 10 N 1 March 1984.

[5] Knuth, Donald E., *The Art of Computer Programming, Volume 1, Fundamentals*, Addison-Wesley Publishing Company.

[6] Kruskal, J. B. Jr. "On the shortest spanning subtree of a graph and the travelling Salesman problem", *Proc. American Mathematical Society*, 7, 1956.

[7] Maier D., "An Efficient Method for Storing Information in Trees", *SIAM Journal of Computing*, V 8 N 4, November 1979.

[8] Prim, R.C., "Shortest Connection Networks and some Generalisations", *Bell Syst. Tech. J.* 36 (1957).

[9] Volgenant, Ton, Jonker, Roy, "The Symmetric Travelling Salesman Problem and Edge Exchanges in Minimal 1-trees", *European Journal of Operations Research* 12 (1983)

[10] Whitney, K.A., "Algorithm 422 Minimal Spanning Tree [H]", *Communications of the ACM*, V15 N4 April 1972.

APPENDIX:

Forward Star Configuration

The Forward Star configuration used in the program consists of three different arrays used to store the characteristics of the graph. The first one, (APT), will act as a pointer to the other two arrays: (ALIST) and (ACOST), where the relation of directed arcs and their cost will be stored respectively. (ALIST) will have in the first locations the values of the vertices adjacent to vertex 1, in ascending order. Their costs will be stored in the same position in (ACOST). The list of the vertices adjacent to vertex 2 will follow those for vertex 1. If an arc (1,2) exists, it will be stored twice: as (1,2) and (2,1). (APT) will store the position where the list of that particular vertex begins

The first disadvantage of the star over the matrix is that it records a same arc two times, while less than half of the positions of a matrix can be used:

$$m(i, j) \quad 1 \le i \le N, \quad i < j \le N$$

The second disadvantage of the star is that the matrix need only one location for the link and the cost of and arc, for if the value of the location is higher than the maximum value of the arcs, it means that there is no connection for the vertices.

Figure A.1.1 shows the Forward Star for the example in chapter 3. The MST subroutine will use an extra dummy position, but it is not important for the information stored.

The disadvantage of the matrix over the star is that it will use a fixed amount of memory locations determined by the number of vertices, this regardless of the number of arcs adjacent to each of them. For a general graph with a relatively low number of arcs (compared to the possible ones) the matrix will be wasting many memory locations. The Forward Star has a dynamic structure that will be reduced if the graph is sparse. A graph comparing the number of memory locations used with both configurations is shown in Figure A.1.2

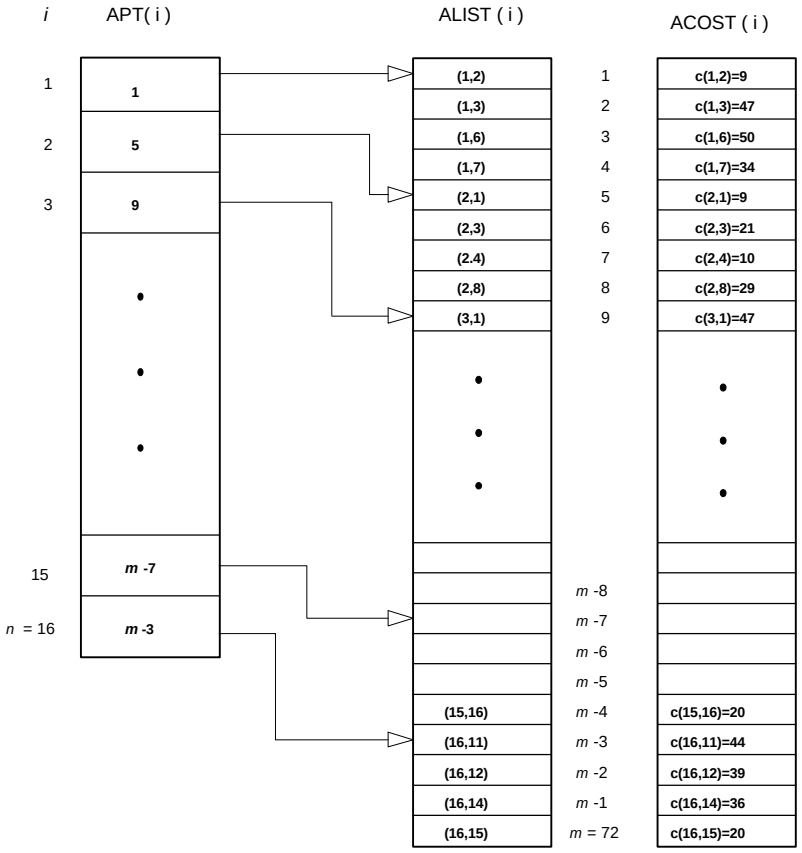


Figure A.1.1

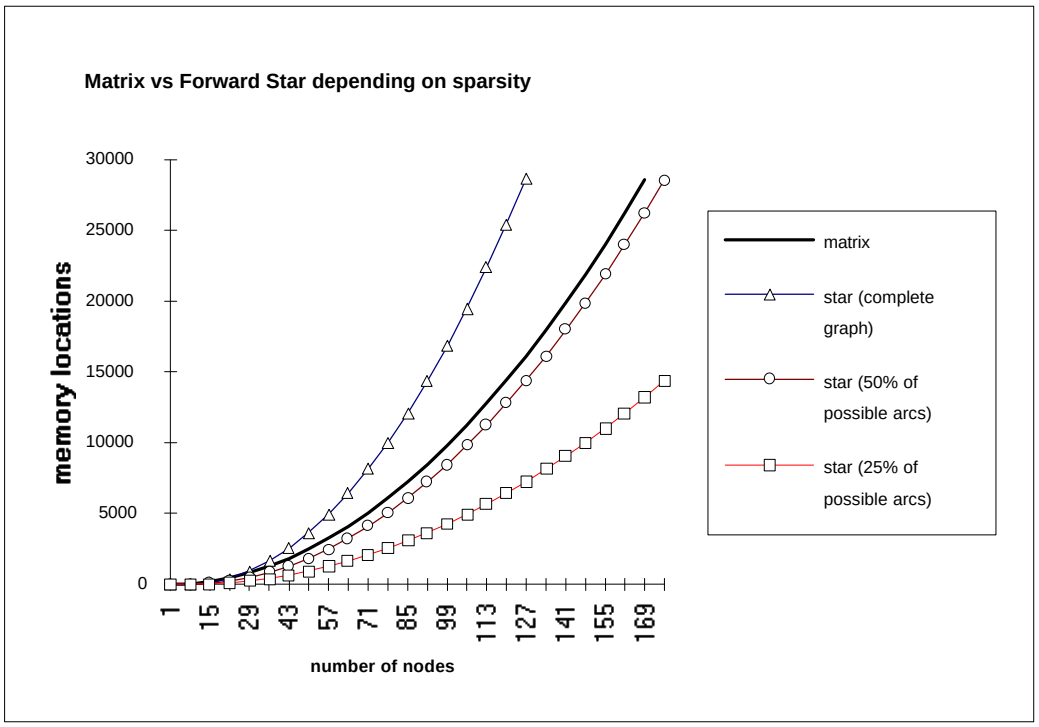


Figure A.1.2