CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

Canada

Design and Analysis of Graph Algorithms:

Spanning Tree Enumeration, Planar Embedding and Maximal Planarization

Rajagopalan Jayakumar

A Thesis

in

The Department

of

Electrical Engineering

Presented in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

August, 1984

ISBN 0-315-30667-X

ABSTRACT

Design and Analysis of Graph Algorithms:

Spanning Tree Enumeration, Planar Embedding and Maximal Planarization

Rajagopalan Jayakumar, Ph. D.
Concordia University, 1984.

This thesis is concerned with the design and analysis of some graph algorithms and is organized into two parts.

In Part I a detailed computational complexity analysis of a spanning tree enumeration algorithm due to Char is given.

First the analysis is presented for general graphs. An expression for the number of sequences generated by the algorithm is then derived and a few properties of the algorithm are established. The complexity of this algorithm is shown to be $O(n^3t)$ where n is the number of vertices of the graph and t is the number of spanning trees. Two heuristics aimed at reducing the number of sequences generated are proposed for selecting the initial spanning tree and an implementation using path compression is also described.

Analysis of Char's algorithm for special graphs is then carried out. A class of graphs for which the algorithm is of complexity O(nt) is identified. Certain interesting results relating to the complete graph, the ladder, and the wheel, which belong to this class, are obtained.

Next an efficient implementation of Char's algorithm, called algorithm MOD-CHAR, is developed. Classes of graphs for which algorithm MOD-CHAR is of complexity O(nt) are identified. It is shown that when applied on large complete graphs ($n \geq 8$), algorithm MOD-CHAR requires, on the average, at most 10 computational steps to generate a spanning tree.

Finally, a computational evaluation of Char's algorithm in comparison with an algorithm due to Gabow and Myers is presented.

In Part II of the thesis, efficient algorithms to obtain a planar embedding of a planar graph and to determine a maximal planar subgraph of a nonplanar graph are developed.

First the planar embedding problem is considered. An embedding procedure which involves placing the vertices at different horizontal and vertical levels in the plane is developed. The vertical levels of the vertices are decided by their st-numbers and an O(n) algorithm is presented to

determine the horizontal levels of the vertices. Another O(n) algorithm to determine the order in which edges entering a vertex from lower numbered vertices should be drawn is also developed. A procedure to draw by hand the edges without crossovers is then described.

Next the maximal planarization problem is considered. Certain results relating to a planarization algorithm due to Ozawa and Takahashi are first established. It is shown that this algorithm does not, in general, determine a maximal planar subgraph. A new maximal planarization algorithm of complexity $O(n^2)$ is then developed.

ACKNOWLEDGEMENTS

I would like to record my deep sense of gratitude to my thesis supervisors Dean M.N.S. Swamy and Professor K. Thulasiraman for their excellent guidance during the course of this research.

I am delighted to make special mention of all the help and encouragement I have received from Dean Swamy and of the deep interest which Professor Thulasiraman has shown in my work during both my Ph. D. research at Concordia University, Montreal, Canada and M. S.. research earlier at the Indian Institute of Technology, Madras, India. I am grateful to them for all these things.

I would also like to thank Concordia University for the University Graduate Fellowship awarded to me from September 1981 to August 1983.

Thanks are due to all my friends for keeping my spirits alive which made this thesis possible.

TO

THE MEMORY OF

MY MOTHER

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The impact of technological innovations on developments in mathematics can hardly be underestimated. These innovations make possible design of large and complex systems. Such systems require sophisticated mathematical tools for their analysis and design; and this leads to the introduction of new mathematical concepts as well as to a deeper study of already known concepts. For example, the availability of VLSI technology and computers has provided great impetus to increased research in a variety of mathematical disciplines. Graph theory is one of the areas of applied mathematics in which recent developments have largely been influenced by the complexities of modern systems.

The role of graph theory in unifying the study of several engineering and scientific disciplines is now well recognized. This unification has become possible because of the fact that for most systems, their behaviour is characterized by properties which arise mainly as a result of the constraints imposed by their structure, namely the way the different elements/subsystems of the systems are interconnected and graph representations of these systems clearly capture their behaviour. Thus graph theory has proven useful in many ways. One is to study the behaviour

of a system as revealed through its structure, the other is to analyse a system for its structural properties and the third is to design a structure having specified properties. However, graphs which arise in real-life problems are extremely large and complicated. An inevitable result of this has been the search to develop computationally efficient algorithms to solve graph problems. Thus began, about two decades ago, a period of intense research on what is now called **Algorithmic Graph Theory.**

In this thesis we make several contributions to this branch of graph theory. We discuss the design and analysis of algorithms for two graph problems, namely spanning tree enumeration, and planar embedding and maximal planarization. Thus the thesis is organized into two parts.

Part I consisting of Chapters 2 to 6 is concerned with the complexity analysis and the design of efficient implementations of a spanning tree enumeration algorithm due to Char. We also give an evaluation of this algorithm in comparison to other known efficient spanning tree enumeration algorithms.

Part II consisting of Chapters 7 to 9 develop efficient algorithms for the planar embedding and maximal planarization problems based upon Lempel, Even and Cederbaum's planarity testing algorithm. To make this part

self-contained, we briefly discuss in Chapter 7 this planarity testing algorithm and its implementation using PQ-trees.

In Chapter 10, we summarize the results of the thesis and point out a few problems for further investigation.

PART I

SPANNING TREE ENUMERATION

CHAPTER 2

SPANNING TREE ENUMERATION ALGORITHMS

A connected acyclic subgraph of a connected graph G having all the vertices of G is called a spanning tree of G. The spanning tree is perhaps one of the most important subgraphs in graph theory, insofar as engineering applications are concerned. For example, a number of results in electrical network theory are based on the concept of a spanning tree. The number of independent Kirchhoff's equations, methods for formulating sets of independent network equations and the topological formulas for network functions are all stated in terms of the single concept of a spanning tree. In addition to these, spanning trees have been used in chemical identification, scheduling and distribution problems and a variety of other applications [1]-[3].

In the topological analysis of a linear system, the problem ultimately reduces to that of finding the set of all the spanning trees in an associated graph [4]. Bedrosian [5] used the set of all the spanning trees of a graph in what is called multilevel maser analysis. All the spanning trees of a graph are also required in the computation of Tutte's polynomial [6] which generalizes the chromatic polynomial of a graph, and in determining symbolic reliability expressions for communication networks [7].

Because of its wide range of applications, the problem of enumerating all the spanning trees of a graph has received considerable attention in the literature. A number of different algorithms based on various concepts have been developed to enumerate all the spanning trees of a graph [4], [8]. Chase [8] classifies those algorithms developed before 1970 according to their underlying principles. Most of these algorithms suffer from one or more of the following disadvantages.

(i) Complicated tests to determine whether a set of edges of the given graph constitutes a spanning tree or not.

(ii) Involved procedures to avoid duplication of spanning trees.

(iii) Extensive manipulations of the graph during the generation process.

Since the number of spanning trees of a graph grows exponentially with the size of the graph, efficiency of these algorithms is of paramount importance. However, complexity analysis is not available for many of the spanning tree enumeration algorithms reported in the literature. Based on his complexity analysis for complete graphs, Chase [8] has concluded that factoring algorithms, which find the spanning trees as a set of Cartesian products, are the most efficient. A recent complexity analysis by Kajitani [9] for a factoring algorithm substantiates this observation.

In 1965, Minty [10] presented a simple algorithm to enumerate all the spanning trees of a graph. This algorithm is based on the following principle. If e is an edge of a graph G, then the set of all the spanning trees of G can be classified into two groups - those which contain e and those which do not contain e. Note that the spanning trees of G which contain the edge e can be obtained from the spanning trees of the graph constructed by contracting e in G, and the spanning trees of G which do not contain the edge e are the same as those of the graph constructed by removing e from G. Thus the spanning trees of G can be obtained from the spanning trees of certain graphs constructed from G using the edge contraction and removal operations. Algorithms of this type are known to be the most efficient [2]. Read and Tarjan [11] presented an implementation of Minty's algorithm which requires $O(m+n+mt)$ time and $O(m+n)$ space for a graph having m edges, n vertices and t spanning trees. While enumerating all the spanning trees of G, Minty's algorithm does not generate any subgraph which is not a spanning tree. Moreover, in this algorithm the spanning trees are generated without duplication. But the graph G is manipulated extensively during the enumeration process, since the spanning trees of G are obtained from those of certain graphs derived from G.

Read and Tarjan's implementation of Minty's algorithm generates the spanning trees of G by starting with an

arbitrary edge of G and then adding certain appropriately selected edges of G. During this process the partial subgraph generated at an intermediate step may not be connected. In 1978, Gabow and Myers [12] presented an implementation of Minty's algorithm in which the partial subgraph formed at each step is guaranteed to be connected. Growing the trees this way, Gabow and Myers achieved $O(m+n+nt)$ time and $O(m+n)$ space complexities for their algorithm. Even though Gabow and Myers' algorithm is as efficient as theoretically possible, it still has the disadvantage of requiring extensive graph manipulation.

Earlier, in 1968, Char [13] had presented a conceptually simple and elegant algorithm to enumerate all the spanning trees of a graph. This algorithm starts with a reference spanning tree, called the initial spanning tree, and determines all the other spanning trees of G. A very simple procedure is used to determine whether a set of edges of G is a spanning tree or not, and the spanning trees are enumerated without duplication. Moreover, no manipulation of the graph is required during the enumeration process. However, in addition to spanning trees, Char's algorithm generates certain subgraphs which are not spanning trees of G. Thus the complexity of Char's algorithm depends on the number of non-tree subgraphs generated. Char had not performed any complexity analysis of his algorithm.

In a recent computational complexity analysis of Char's algorithm [14]-[15], it has been shown that the complexity of the algorithm depends on the choice of the initial spanning tree, and that for a number of special graphs this algorithm is of $O(m+n+nt)$ time complexity. In the general case, the complexity is $O(m+n+n(t+t_0))$, where t_0 is the number of non-tree subgraphs generated. Using the crude bound $t_0 \leq n^2 t$, the worst-case complexity of Char's algorithm becomes $O(m+n+n^3 t)$. However, experimental results presented in [14] suggest that Char's algorithm might be faster than Minty's and Gabow and Myers' algorithms. Perhaps, this is because Char's algorithm does not require extensive graph manipulation in contrast to the other two algorithms. Moreover, Char's algorithm has a number of very interesting properties from the point of view of computational complexity theory.

In this part of the thesis we perform a detailed complexity analysis of Char's algorithm and present several interesting results relating to this algorithm.

In Chapter 3 we first present Char's algorithm and then establish an elegant expression for $(t+t_0)$ in terms of the numbers of spanning trees of certain graphs constructed from G. We then present a systematic method, using certain concepts from electrical network theory, to compute this number. Based on this expression, we carry out a

computational complexity analysis of Char's algorithm for general graphs. The expression for $(t+t_0)$ indicates that the number t_0 depends on the initial spanning tree used in the enumeration. Thus an interesting problem is to find an initial spanning tree which leads to the minimum value for t_0. With the view to reducing the value of t_0, we also develop in Chapter 3 two heuristic procedures for constructing appropriate initial spanning trees. We then present experimental results illustrating the reduction in the value of t_0 achieved when the initial spanning trees constructed by the two heuristics are used in Char's algorithm. Finally, we present in this chapter an implementation of Char's algorithm using the principle of path compression which achieves considerable reduction in the actual number of comparisons made by the algorithm while testing whether a set of edges constitutes a spanning tree or not [16].

Chapter 4 presents analysis of Char's algorithm for special graphs. Specifically, we consider those graphs for which the algorithm requires $O(m+n+nt)$ time and hence is optimal. We first identify a class of graphs for which Char's algorithm is optimal. We derive the number $(t+t_0)$ of subgraphs generated by the algorithm when applied on complete graphs, ladders and wheels using a star tree as the initial spanning tree, and show that the algorithm has linear time complexity in these cases. More interestingly,

in the cases of ladders and wheels, we develop expressions for the total number of computational steps required by Char's algorithm and show that in these cases, the algorithm requires, on the average, at most 4 computational steps per spanning tree generated. We conclude Chapter 4 with the definition of the concept of min-tree-number, which is essentially equal to the minimum value of t_0 for a graph. We outline some results on the min-tree-number and state two conjectures which are supported by our computational experiences with Char's algorithm.

In Chapter 5 we design a highly efficient implementation of Char's algorithm. We call the new algorithm MOD-CHAR. We prove that MOD-CHAR has a better asymptotic complexity than Char's algorithm. We also show that for large complete graphs MOD-CHAR requires, on the average, at most 10 computational steps per spanning tree and identify a class of graphs for which MOD-CHAR is of O(m+n+nt) time complexity.

In Chapter 6 we present a comparison of Gabow and Myers' algorithm with Char's algorithm and algorithm MOD-CHAR. Even though Gabow and Myers' algorithm has a better asymptotic time complexity than the other two algorithms, it is found to require more execution time. This is because of the extensive graph manipulations performed by Gabow and Myers' algorithm. To make an

evaluation which is independent of implementation details, we base our comparison on the number of basic computational steps performed by each of these three algorithms when applied on a number of randomly generated graphs.

Without any loss of generality, we assume that all the graphs considered in this thesis are connected.

CHAPTER 3

COMPUTATIONAL COMPLEXITY OF CHAR'S ALGORITHM

In this chapter we discuss the computational complexity of Char's algorithm for general graphs. In order to make our presentation self-contained, we describe, in Section 3.1, Char's algorithm and determine its complexity in terms of the number of subgraphs generated by the algorithm. In Section 3.2, we develop a formula for the number of subgraphs generated by Char's algorithm and present a systematic method to compute this number from the given graph. We also discuss the complexity of Char's algorithm in detail and show that the complexity depends on the initial spanning tree used in the enumeration. With a view to reducing the number of non-tree subgraphs generated by Char's algorithm, we develop in Section 3.3 two heuristics to select an initial spanning tree. Finally, in Section 3.4, we present an implementation of Char's algorithm using the principle of path compression which considerably reduces the actual number of comparisons made by the algorithm. We also show that this implementation has the same asymptotic complexity as the original straightforward implementation.

3.1 Char's Algorithm

Consider an undirected graph G = (V,E) having n = |V| vertices and m = |E| edges. Let the vertices of G be denoted as 1, 2, ..., n. Consider any sequence λ = (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) of vertices of G such that DIGIT(i), $1 \leq i \leq n-1$, is a vertex adjacent to vertex i in G. Each such sequence λ corresponds to a subgraph $G_\lambda = (V_\lambda, E_\lambda)$ of G such that

and
$$V_\lambda = \{1, 2, \ldots, n\},$$
$$E_\lambda = \{(1,\text{DIGIT}(1)), (2,\text{DIGIT}(2)), \ldots, (n-1,\text{DIGIT}(n-1))\}.$$

Note that not all the edges in E_λ are necessarily distinct. Char's algorithm is based on the following [14].

Tree Compatibility Property

The sequence (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) represents a spanning tree of graph G if and only if for each $k \leq n-1$, the first vertex not less than k in the sequence k, DIGIT(k), DIGIT(DIGIT(k)), ... is greater than k. □

This can be shown as follows. Let λ be a sequence having the tree compatibility property, and let G_λ be the corresponding subgraph of G. The tree compatibility property ensures that all the n-1 edges in G_λ are distinct.

Furthermore, in G_λ there is a path from each vertex to the vertex n. Thus G_λ is connected. Since G_λ has n vertices, n-1 edges and is connected, it is a spanning tree of G.

Clearly there are $\prod_{i=1}^{n-1} \deg(i)$ such (n-1)-digit sequences possible for a graph G where deg(i) is the degree of vertex i in G. Char's algorithm generates some of these sequences and classifies those sequences which have the tree compatibility property as <u>tree sequences</u> and those sequences which do not have the property as <u>non-tree sequences</u>. It may be noted that if (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) is a tree sequence, then, in the corresponding spanning tree, DIGIT(i), $1 \leq i \leq n-1$, is the vertex next to i in the path from vertex i to vertex n.

To start with, Char's algorithm selects a reference spanning tree, called the <u>initial spanning tree</u>, of G by performing a breadth-first search on G. The vertices of G are numbered as n, n-1, ..., 1 in the order in which they are visited during the search. These numbers are used thereafter to represent the vertices of G. Using the initial spanning tree, an array REF is defined as REF(i) = FATHER(i), $1 \leq i \leq n-1$, where FATHER(i) is that vertex from which vertex i is visited during the search. Since we number FATHER(i) before numbering vertex i, it follows that REF(i) > i, $1 \leq i \leq n-1$. Therefore the sequence λ_0 = (REF(1), REF(2), ..., REF(n-1)) has the tree

compatibility property. In fact, λ_0 represents the initial spanning tree, and so it is called the initial tree sequence.

It should be pointed out that in Char's algorithm, any spanning tree can be used as the initial spanning tree, provided the vertex numbering is done such that in the corresponding tree sequence, DIGIT(i) > i, $1 \leq i \leq n-1$. In fact, as we shall see later, it is this requirement on vertex numbering that makes Char's algorithm very efficient. One easy way to achieve this requirement is to perform a depth-first search or a breadth-first search on the initial spanning tree and number the vertices as n, n-1, ..., 1, in the order in which they are visited during the search. Then the initial tree sequence will be (REF(1), REF(2), ..., REF(n-1)) where, as before, REF(i) = FATHER(i) > i, $1 \leq i \leq n-1$. Note that for a given spanning tree more than one vertex numberings satisfying the above requirement are possible.

In Char's algorithm the graph G is represented by the adjacency lists of all of its vertices except vertex n, such that the first entry in the adjacency list of any vertex v is REF(v) and the other neighbours of v are arranged in any order in the list. The enumeration starts with the initial tree sequence (REF(1), REF(2), ..., REF(n-1)). Given a tree sequence (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)), to generate

the next sequence, we test whether DIGIT(n-1) is the last entry in the adjacency list of vertex n-1. If not, we set DIGIT(n-1) to the entry next to the current value of DIGIT(n-1) in the adjacency list of vertex n-1 and obtain the next sequence. On the other hand, if DIGIT(n-1) is the last entry in the adjacency list of vertex n-1, we set DIGIT(n-1) to REF(n-1) and proceed to test DIGIT(n-2). If DIGIT(n-2) is also the last entry in the adjacency list of vertex n-2, we set DIGIT(n-2) to REF(n-2) and proceed to test DIGIT(n-3) and so on until we find a new sequence.

Suppose we have obtained a new sequence (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)). Let k be the highest integer such that in this sequence DIGIT(k) $\neq$ REF(k). Consider the sequence of vertices k, DIGIT(k), DIGIT(DIGIT(k)), ... and let j be the first vertex in this sequence which is not less than k. Now the following two cases arise.

(i) If $j > k$, then the new sequence is a tree sequence. In this case the sequence is listed and we proceed to generate the next sequence.

(ii) If $j = k$, then the new sequence is a non-tree sequence. In this case, if DIGIT(k) is not the last entry in the adjacency list of vertex k, we set DIGIT(k) to the entry next to the current value of DIGIT(k) in the adjacency list of vertex k and obtain the next sequence. If DIGIT(k) is the last entry in the adjacency list of vertex k, we set DIGIT(k) to REF(k) and proceed to test DIGIT(k-1).

The enumeration stops when replacement is attempted with DIGIT(0). In this case DIGIT(i) = REF(i), $1 \leq i \leq n-1$, and hence the corresponding spanning tree is the initial spanning tree. Formally Char's algorithm may be given in ALGOL-like notation as follows. Here, by SUCC(DIGIT(i)) we mean the entry next to DIGIT(i) in the adjacency list of vertex i.

Char's Algorithm to Enumerate all the Spanning Trees of a Graph.

```
procedure CHAR;
comment procedure CHAR enumerates all the spanning trees of
        a connected n-vertex graph G represented by the
        adjacency lists of its vertices.
begin
  select an initial spanning tree of G;
  perform a depth-first search or a breadth-first search on
  the initial spanning tree and renumber the vertices as n,
  n-1, ..., 1 in the order in which they are visited during
  the search;
  find FATHER(i), 1 ≤ i ≤ n-1;
  for i:= 1 to n-1 do
    begin
      REF(i) := FATHER(i);
      DIGIT(i) := REF(i)
    end;
  output the initial tree sequence DIGIT(i), 1 ≤ i ≤ n-1;
```

```
i := n-1;
while i ≠ 0 do
  begin
    if SUCC(DIGIT(i)) ≠ nil
       then begin
         DIGIT(i) := SUCC(DIGIT(i));
         if (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) is a tree
            sequence
           then begin
             output the tree sequence;
             i := n-1
           end
       end
       else begin
         DIGIT(i) := REF(i);
         i := i-1
       end
  end
end CHAR;
```

Now we illustrate Char's algorithm by enumerating all the spanning trees of the graph G shown Fig. 3.1(a). First we perform a breadth-first search starting at vertex b and obtain the initial spanning tree shown in Fig. 3.1(b). During the search we number the vertices of G and these numbers are shown within parentheses in Figs. 3.1(a) and (b). For each v, $1 \leq v \leq 4$, REF(v) is given below.

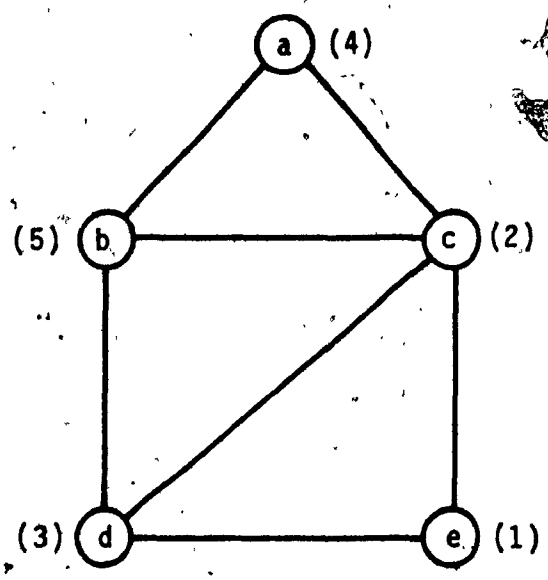


Figure 3.1(a)

Graph G

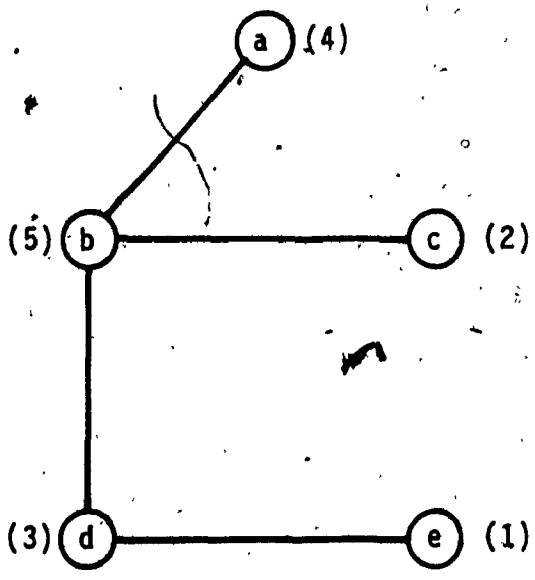


Figure 3.1(b)

Initial Spanning Tree of G

v	REF(v)
1	3
2	5
3	5
4	5

Thus the initial tree sequence is (3 5 5 5). We represent the graph G by the following adjacency lists.

v	Adj(v)
1	3,2
2	5,4,3,1
3	5,2,1
4	5,2

Starting with the tree sequence (3 5 5 5), Char's algorithm generates the following sequences where the tree sequences are denoted by a plus sign (+).

+	3	5	5	5
+	3	5	5	2
+	3	5	2	5
+	3	5	2	2
	3	5	1	5
+	3	4	5	5
	3	4	5	2
+	3	4	2	5

+	3	3	5	5
+	3	3	5	2
	3	3	2	5
	3	3	1	5
+	3	1	5	5
+	3	1	5	2
	3	1	2	5
	3	1	1	5

	3	4	2	2		+	2	5	5	5
	3	4	1	5		+	2	5	5	2
+	2	5	2	5		+	2	4	1	5
+	2	5	2	2			2	4	1	2
+	2	5	1	5		+	2	3	5	5
+	2	5	1	2		+	2	3	5	2
+	2	4	5	5			2	3	2	5
	2	4	5	2			2	3	1	5
+	2	4	2	5			2	1	5	5
	2	4	2	2						

Next we show that not all the possible sequences for a graph G are generated in Char's algorithm and certain confirmed non-tree sequences are skipped. Let (DIGIT(1), DIGIT(2), ..., DIGIT(k), ..., DIGIT(n-1)) be a non-tree sequence generated by the algorithm which does not have the tree compatibility property at position k. In the subgraph corresponding to this non-tree sequence there is a sequence of edges starting with the edge (k,DIGIT(k)) and ending at vertex k using one or more of the edges (1,DIGIT(1)), (2,DIGIT(2)), ..., (k-1,DIGIT(k-1)). Note that this sequence of edges can either be a circuit passing through vertex k or just the repetition of an edge as (k,DIGIT(k)) and (DIGIT(k),k). The next sequence is obtained by changing DIGIT(k) of this non-tree sequence. Hence generation of all subsequent non-tree sequences which have the sequence (DIGIT(1), DIGIT(2), ..., DIGIT(k)) as a subsequence is

avoided. Thus not all the possible sequences are generated by Char's algorithm.

Consider a tree sequence (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)). As pointed out earlier, it follows from the tree compatibility property that DIGIT(i), $1 \leq i \leq n-1$, is the vertex next to i in the path, in the spanning tree, from vertex i to vertex n. Thus the tree sequence specifies the path from each vertex i to vertex n. So it follows that each tree sequence corresponds to a unique spanning tree. Furthermore, since distinct tree sequences specify distinct sets of paths, they correspond to distinct spanning trees. On the other hand, suppose for a given spanning tree, we obtain the sequence (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) such that DIGIT(i), $1 \leq i \leq n-1$, is the vertex next to i in the path, in the spanning tree, from vertex i to vertex n. Then this sequence will have the tree compatibility property. This means that for each spanning tree, there exists a sequence having the tree compatibility property. Thus there exists a one-to-one correspondence between the set of all the tree sequences and the set of all the spanning trees of G. Since Char's algorithm generates all the sequences which have the tree compatibility property, it follows that the algorithm enumerates all the spanning trees of the graph G. Furthermore, the sequences generated by Char's algorithm are all distinct, and so the spanning trees are generated without duplication. Moreover, only the

adjacency lists of the graph are used in the algorithm and no manipulation of the graph is required during the enumeration process.

We now study the complexity of Char's algorithm. When a new sequence is obtained by changing DIGIT(k) of the previous sequence, it is clear that DIGIT(i) = REF(i) > i, $i \geq k+1$, and DIGIT(1), DIGIT(2), ..., DIGIT(k-1) have the same values as in the previous sequence. Hence the new sequence is to be tested for the tree compatibility property only at position k and this test, in the worst case, involves k comparisons. Hence, at most n computational steps are required to generate and test a sequence. So, if t_0 is the number of non-tree sequences generated by Char's algorithm, in the worst case $n(t+t_0)$ computational steps are required to enumerate all the spanning trees of the given graph and hence Char's algorithm is of time complexity $O(m+n+n(t+t_0))$, which includes the complexity of determining the initial spanning tree also. As regards the space complexity, first note that the graph is represented by a set of n-1 adjacency lists. This representation requires O(m) space. Furthermore, the arrays DIGIT and REF each require O(n) space. Thus Char's algorithm requires O(m+n) space altogether.

3.2 Computational Complexity Analysis for General Graphs

Since the computational complexity of Char's algorithm is $O(m+n+n(t+t_0))$, any complexity analysis of this algorithm would require a study of the number $(t+t_0)$. With this objective in view, we first obtain an expression for $(t+t_0)$. From our discussion in Section 3.1, it is clear that Char's algorithm generates certain (n-1)-digit sequences of vertices of the graph G and classifies each one of them as a tree sequence or a non-tree sequence using the tree compatibility property. We partition the sequences generated by Char's algorithm as follows.

Let $T = \bigcup_{i=0}^{n-1} T_i$ be the set of all the tree sequences such that

(i) $T_0 = \{\lambda_0\}$, and

(ii) T_i, $1 \leq i \leq n-1$, is the set of tree sequences of the form (DIGIT(1), DIGIT(2), ..., DIGIT(i), REF(i+1), REF(i+2), ..., REF(n-1)) with DIGIT(i) $\neq$ REF(i).

Also let $T' = \bigcup_{i=1}^{n-1} T'_i$ be the set of all the non-tree sequences such that T'_i is the set of non-tree sequences of the form (DIGIT(1), DIGIT(2), ..., DIGIT(i), REF(i+1), REF(i+2), ..., REF(n-1)) with DIGIT(i) $\neq$ REF(i). Note that $|T| = t$ is the number of tree sequences and $|T'| = t_0$ is the number of non-tree sequences generated by Char's algorithm.

Now we prove the following.

THEOREM 3.1.

Consider a connected n-vertex undirected graph with its vertices numbered as in Char's algorithm. Let $G_k^{(s)}$, $1 \leq k \leq n-1$, be the graph obtained from G by coalescing the vertices k, k+1, ..., n and let t(k) be the number of spanning trees of $G_k^{(s)}$. If t is the number of tree sequences and t_0 is the number of non-tree sequences generated by Char's algorithm, then

$$t+t_0 = 1 + \sum_{k=1}^{n-1} (\deg(k)-1)\, t(k),$$

where deg(k), $1 \leq k \leq n$, is the degree of vertex k in G.

Proof:

Consider a tree sequence λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), REF(k), REF(k+1), ..., REF(n-1)) generated by Char's algorithm. The spanning tree corresponding to λ_k is then the subgraph $G_k = (V_k, E_k)$, where

$$V_k = V$$

and

$$E_k = \{(1,\text{DIGIT}(1)),\ (2,\text{DIGIT}(2)),\ \ldots,\ (k-1,\text{DIGIT}(k-1)),\ (k,\text{REF}(k)),\ (k+1,\text{REF}(k+1)),\ \ldots,\ (n-1,\text{REF}(n-1))\}.$$

Let $G_k' = (V_k', E_k')$ be the spanning 2-tree obtained from G_k by deleting the edge (k,REF(k)) so that

and
$$V'_k = V_k$$
$$E'_k = E_k - (k, REF(k)).$$

Since REF(i) > i, $1 \leq i \leq n-1$, it follows that, in G'_k the edges (k+1,REF(k+1)), (k+2,REF(k+2)), ..., (n-1,REF(n-1)) are in one component and the vertex k is in the other component. Let $G'_{k,1} = (V'_{k,1}, E'_{k,1})$ be the component containing the edges (k+1,REF(k+1)), (k+2,REF(k+2)), ..., (n-1,REF(n-1)) and let $G'_{k,2} = (V'_{k,2}, E'_{k,2})$ be the component containing the vertex k. Note that in $G'_{k,1}$ and in $G'_{k,2}$ there exists a unique path between every pair of vertices.

Consider any vertex $v \neq REF(k)$, adjacent to vertex k. Let $G''_k = (V'_k, E'_k \cup (k,v))$. Now the following two cases arise.

(i) If $v \in V'_{k,1}$, then it is clear that G''_k is a spanning tree of G. Thus the sequence (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), v, REF(k+1), ..., REF(n-1)) with $v \in V'_{k,1}$ is a tree sequence passing the tree compatibility test at position k.

(ii) If $v \in V'_{k,2}$, then in G''_k the edge (k,v), along with the unique path in $G'_{k,2}$ between the vertices k and v, forms a circuit passing through the vertex k, and hence G''_k is a non-tree subgraph of G generated by Char's algorithm. Thus the sequence (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), v, REF(k+1), ..., REF(n-1)) with $v \in V'_{k,2}$ is a non-tree sequence which does not have the tree compatibility property at

position k.

Since vertex k is adjacent to (deg(k)-1) vertices other than REF(k), there are (deg(k)-1) distinct sequences of the form (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), v, REF(k+1), ..., REF(n-1)), with $v \neq REF(k)$, which have the same DIGIT(1), DIGIT(2), ..., DIGIT(k-1) as λ_k. Each one of these sequences is either a tree sequence or a non-tree sequence depending on whether $v \in V'_{k,1}$ or $v \in V'_{k,2}$, and so all these sequences belong to $T_k \cup T'_k$. Thus if t(k) is the number of tree sequences of the form λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), REF(k), REF(k+1), ..., REF(n-1)), then

$$|T_k \cup T'_k| = (\deg(k)-1)t(k). \qquad (3.1)$$

Since in the spanning tree corresponding to λ_k, the edges (k,REF(k)), (k+1,REF(k+1)), ..., (n-1,REF(n-1)) are present, it follows that t(k) is the number of all the spanning trees of G in which the edges (k,REF(k)), (k+1,REF(k+1)), ..., (n-1,REF(n-1)) are present. Thus t(k) is the number of spanning trees of the graph obtained from G by coalescing the vertices k, k+1, ..., n-1, REF(k), REF(k+1), ..., REF(n-1). But {k, k+1, ..., n-1, REF(k), REF(k+1), ..., REF(n-1)} = {k, k+1, ..., n}, because REF(i) > i, $1 \leq i \leq n-1$, and so t(k) is the number of spanning trees of $G_k^{(s)}$, the graph obtained from G by coalescing the vertices k, k+1, ..., n. Also the total number of sequences generated by Char's algorithm is

$$t+t_0 = |T_0| + \sum_{k=1}^{n-1} |T_k \cup T_k'| = 1 + \sum_{k=1}^{n-1} |T_k \cup T_k'|.$$

From these observations and (3.1) the theorem follows. □

From Theorem 3.1 we get the following.

COROLLARY 3.1.1.

For a complete graph, t_0 is independent of the initial spanning tree.

Proof:

The proof follows if we note that in the case of a complete graph G, the number of spanning trees t(k) of the graph $G_k^{(s)}$ for a given k is the same for any choice of the initial spanning tree and that deg(k) = n-1 for all k, $1 \leq k \leq n$. □

Now we develop a systematic procedure to compute t(k). Let G(w) be a weighted undirected graph in which w(i,j) denotes the weight of the edge (i,j). For any vertex i of G(w), let $\Gamma(i)$ be the set of vertices adjacent to vertex i in G(w). Let

$$d_i = \sum_{j \in \Gamma(i)} w(i,j).$$

By pivotal condensation at vertex i in G(w) we mean the following operation: For each pair of vertices $j_1, j_2 \in \Gamma(i)$, if the edge (j_1, j_2) is already present in G(w), then increase its weight by $w(i,j_1)w(i,j_2)/d_i$; otherwise add to G(w) the edge (j_1, j_2) with the weight $w(i,j_1)w(i,j_2)/d_i$. After all pairs of neighbours of the vertex i are considered, delete from G(w) the vertex i and all the edges incident on it.

Let N be an RLC electrical network and let G(N) be the graph of N such that the weight of an edge is given by the admittance of the corresponding element of N. Let A be a subset of the vertex set $V = \{1, 2, \ldots, n\}$ of N. Let the networks N_A and N_A^0 be defined as

N_A: the network that results after coalescing all the vertices of N which do not belong to A,

N_A^0: the network that results after suppressing all the vertices of N which belong to A.

If T(N), $T(N_A)$, and $T(N_A^0)$ denote the sum of tree-admittance products of the networks N, N_A, N_A^0 respectively, then it has been shown in [17] that

$$T(N) = T(N_A)T(N_A^0). \qquad (3.2)$$

Note that the graph $G(N_A^0)$ of the network N_A^0 can be obtained from the graph G(N) by performing pivotal condensations, in G(N), at all the vertices in A.

Let $G_1(N) = G(N)$ and the graph $G_i(N)$ be obtained from $G_{i-1}(N)$ by performing pivotal condensation at vertex i-1 in $G_{i-1}(N)$. If $A = \{1, 2, \ldots, k-1\}$, and d_i, $1 \leq i \leq k-1$, is the sum of admittances of all the edges incident on vertex i in $G_i(N)$, then as shown in [17]

$$T(N) = d_1 d_2 \ldots d_{k-1} T(N_A^0). \tag{3.3}$$

Comparing (3.2) and (3.3) we get

$$T(N_A) = d_1 d_2 \ldots d_{k-1}.$$

Note that the graph of the network N_A is obtained from G by coalescing the vertices k, k+1, ..., n and hence it is $G_k^{(s)}$. If N is a resistive network with each element of admittance one Siemens, then the admittance product of each spanning tree is one and so $T(N_A)$ is the number of spanning trees of the graph $G_k^{(s)}$. Thus we get the following.

THEOREM 3.2.

Consider a connected n-vertex undirected graph G with its vertices numbered as in Char's algorithm. Let $G_k^{(s)}$ be the graph obtained from G by coalescing the vertices k, k+1, ..., n. Let G_1 be the graph obtained from G by assigning unit weight to each edge of G, and G_i be the graph obtained from G_{i-1} by performing pivotal condensation at vertex i-1 in G_{i-1}. Let d_i be the sum of the weights in G_i of all the edges (i,j), $j \in \Gamma(i)$ where $\Gamma(i)$ is the set of vertices adjacent to i in G_i. If t(k) is the number of spanning

trees of $G_k^{(s)}$, then

$$t(k) = d_1 d_2 \ldots d_{k-1}. \qquad \square$$

From the fact that $t(n) = t$, we get the following corollary of the above theorem.

COROLLARY 3.2.1.

The number of spanning trees of G is given by

$$t = d_1 d_2 \ldots d_{n-1}. \qquad \square$$

Using the above corollary and Theorem 3.2 in Theorem 3.1, we get the following.

THEOREM 3.3.

The number of sequences generated by Char's algorithm is

$$t+t_0 = 1 + t \sum_{k=1}^{n-1} \frac{\deg(k)-1}{d_{n-1} d_{n-2} \ldots d_k}. \qquad \square$$

Now we illustrate the above procedure to compute $(t+t_0)$ for the graph G in Fig. 3.1(a). We obtain the graph G_1 in Fig. 3.2(a) by assigning unit weight to each edge of G. Note that $d_1 = 2$. The graph G_2 is obtained by performing pivotal condensation at vertex 1 in G_1 and it is shown in Fig. 3.2(b). From G_2 we get $d_2 = 7/2$. The graph G_3 is

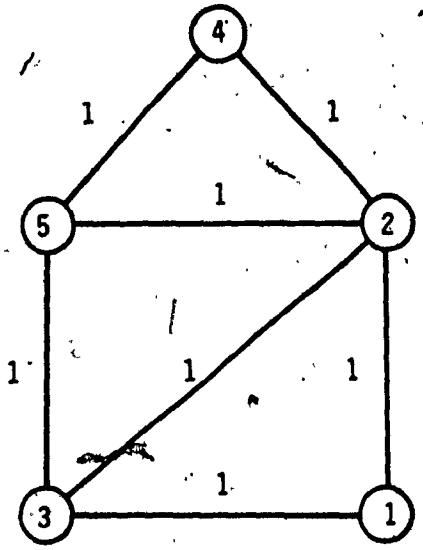


Figure 3.2(a)

Graph G_1

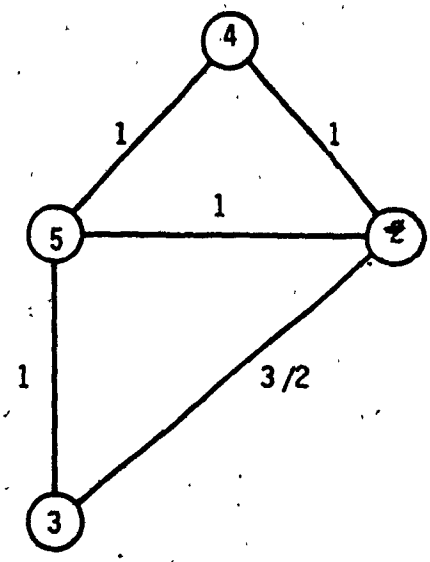


Figure 3.2(b)

Graph G_2

obtained from G_2 by performing pivotal condensation at vertex 2 in G_2 and it is shown in Fig. 3.2(c). From G_3 we get $d_3 = 13/7$. Finally, for the graph G_4 shown in Fig. 3.2(d), $d_4 = 21/13$. Thus for G

$$t = d_1 d_2 d_3 d_4 = 21,$$

and

$$t+t_0 = 1 + t \sum_{k=1}^{4} \frac{deg(k)-1}{d_{n-1} d_{n-2} \cdots d_k} = 35.$$

From the example given in Section 3.1 we can verify that the graph G has 21 spanning trees and that Char's algorithm generates 35 sequences for G.

The value of $(t+t_0)$ given in Theorem 3.1 depends on t(k)'s. Each t(k) is the number of spanning trees of $G_k^{(s)}$, which is obtained from G by coalescing the vertices k, k+1, ..., n. Since the vertices are numbered using the initial spanning tree, the value of $(t+t_0)$ and hence the complexity of the algorithm depends on the initial spanning tree. However, for two different initial spanning trees, the values of t(k) for a given k will be the same if the set of vertices which receive the numbers k, k+1, ..., n is identical in both cases. In other words, the value of t(k) depends on the set of vertices which are assigned the numbers k, k+1, ..., n and not on the edges connecting these vertices. Since this statement is true for all values of k,

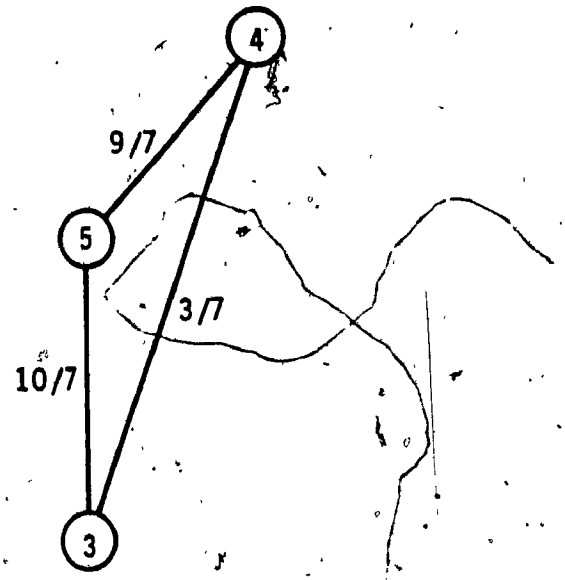


Figure 3.2(c)

Graph G_3

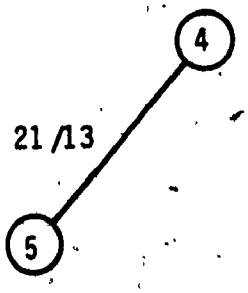


Figure 3.2(d)

Graph G_4

we get the following.

THEOREM 3.4.

Consider any arbitrary numbering of the vertices of a connected undirected graph G. Let S be the set of all the spanning trees of G such that in the sequences corresponding to these spanning trees DIGIT(i) > i, $1 \leq i \leq n-1$. Then the number $(t+t_0)$ of subgraphs generated by Char's algorithm when applied on G (whose vertices are numbered as before) will be the same for all choices of initial spanning trees chosen from the set S. □

For example, consider the graph shown in Fig. 3.3(a) and the two distinct spanning trees shown in Figs. 3.3(b) and 3.3(c). If the vertex numbers are assigned as shown within parentheses, then the sequences representing these trees are

(5 4 4 5 6)

and

(5 3 4 6 6).

Clearly in these sequences DIGIT(i) > i, $1 \leq i \leq n-1$, and Theorem 3.4 is applicable. When these trees are used as the initial spanning trees, the same value of $(t+t_0)$ will be obtained. In fact, for both these initial spanning trees, $t+t_0 = 210$ and $t_0 = 80$.

Having obtained an expression for $(t+t_0)$, we now

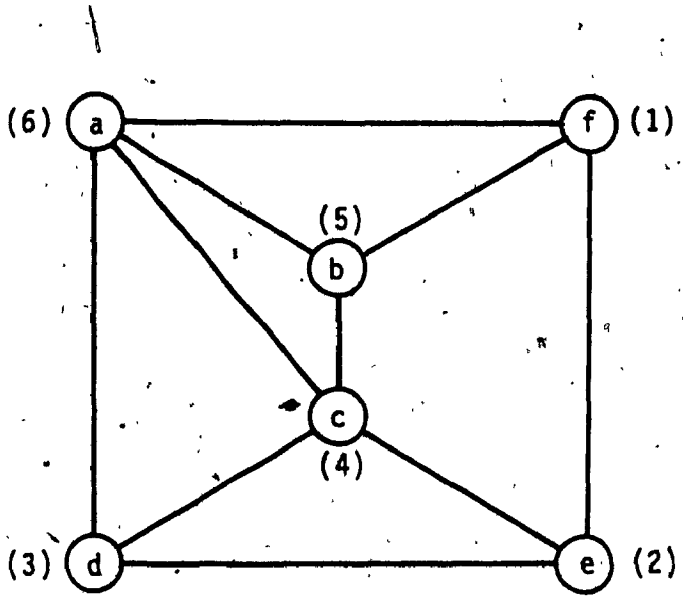


Figure 3.3(a)

Graph G to Illustrate Theorem 3.4

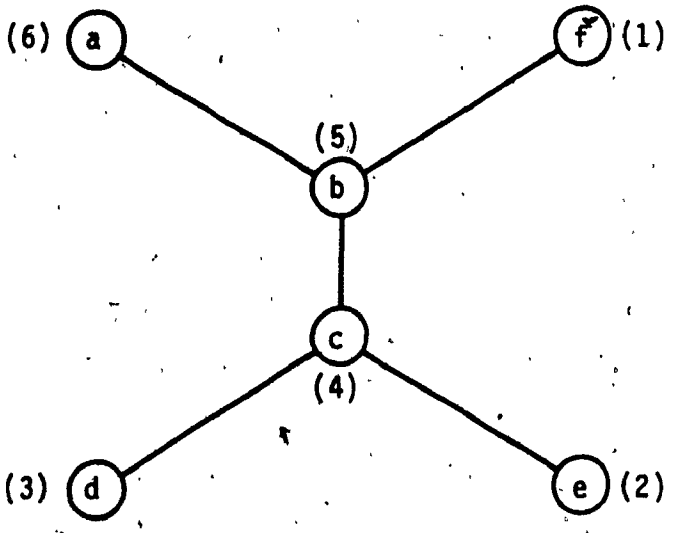


Figure 3.3(b)

A Spanning Tree of G

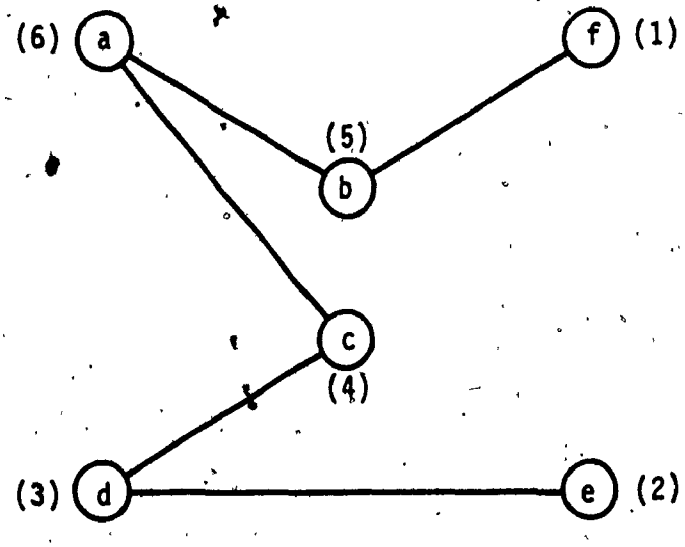


Figure 3.3(c)

Another Spanning Tree of G

consider the computational complexity of Char's algorithm. Consider a sequence λ = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), x, REF(k+1), ..., REF(n-1)), with $x \neq$ REF(k), generated by Char's algorithm. This sequence belongs to $T_k \cup T'_k$. To generate this the algorithm explicitly requires setting DIGIT(i) = REF(i) for each i, $k+1 \leq i \leq n-1$, in addition to setting DIGIT(k) = x. Next the algorithm tests whether λ is a tree sequence or not. This is done by checking the tree compatibility property at position k. This in turn requires checking the existence of a path, from vertex k leading to k or a vertex greater than k, in the subgraph consisting of the edges (1,DIGIT(1)), (2,DIGIT(2)), ..., (k-1,DIGIT(k-1)) and needs at most k comparisons. Thus generating and testing λ involves the following two types of computations.

Type 1: (n-k-1) steps to set DIGIT(i) = REF(i), $k+1 \leq i \leq n-1$.

Type 2: C_k steps to set DIGIT(k) = x and to test λ for the tree compatibility property.

Suppose the sequence λ passes the tree compatibility test. Then it is a tree sequence and the cost of Type 1 computation used in generating λ can be associated with λ. On the other hand, if λ fails the test, then the algorithm generates a new sequence λ' by setting DIGIT(k) to the

vertex next to x in the adjacency list of k. The sequence λ' is then tested for the tree compatibility property. Thus generating λ' does not require Type 1 computation. If λ' also fails the test, the algorithm continues to generate sequences (without using Type 1 computations) until a tree sequence λ'' is generated. The cost of Type 1 computation required in generating λ can therefore be charged to the tree sequence λ''. Thus the cost of each Type 1 computation can be charged to a tree sequence. Clearly the cost of Type 1 computations (in terms of computational steps) for generating all the tree sequences in T_k is given by $|T_k|(n-k-1)$. If we denote by COST1 the total cost of Type 1 computations required in generating all the tree sequences, then

$$\text{COST1} = \sum_{k=1}^{n-1} |T_k|(n-k-1) = \sum_{k=1}^{n-2} |T_k|(n-k-1).$$

But

$$|T_k| = t(k+1)-t(k)$$

for all k, $1 \leq k \leq n-2$. So

$$\text{COST1} = \sum_{k=1}^{n-2} [t(k+1)-t(k)](n-k-1)$$

$$= \sum_{k=2}^{n-1} t(k) - (n-2), \text{ since } t(1) = 1$$

$$= t \sum_{k=2}^{n-1} \frac{1}{d_{n-1}d_{n-2}\cdots d_k} - (n-2). \qquad (3.4)$$

As regards the Type 2 computation, it is required for each sequence in $T_k \cup T_k'$ and for all $1 \leq k \leq n-1$. If C_k^m denotes the maximum number of computational steps required to perform a Type 2 computation for any sequence in $T_k \cup T_k'$, and COST2 denotes the cost of performing all Type 2 computations, then

$$\text{COST2} < \sum_{k=1}^{n-1} C_k^m |T_k \cup T_k'|$$

$$= t \sum_{k=1}^{n-1} \frac{\deg(k)-1}{d_{n-1}d_{n-2}\cdots d_k} C_k^m, \text{ by Theorem 3.3.} \qquad (3.5)$$

Thus the total cost COST of execution of Char's algorithm is

$$\text{COST} = \text{COST1} + \text{COST2}$$

$$\leq t \sum_{k=2}^{n-1} \frac{1}{d_{n-1}d_{n-2}\cdots d_k} - (n-2) + t \sum_{k=1}^{n-1} \frac{\deg(k)-1}{d_{n-1}d_{n-2}\cdots d_k} C_k^m.$$

From (3.4) and (3.5), it is clear that

$$\text{COST1} \leq nt$$

and

$$\text{COST2} \leq n^3 t.$$

So COST is $O(n^3 t)$. Thus the computational complexity of Char's algorithm is $O(m+n+n^3 t)$ where $O(m+n)$ is the complexity of selecting an initial spanning tree and numbering the vertices of the graph. In obtaining this, we have substituted n for the sum

$$\sum_{k=1}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k}.$$

However, this is a very crude bound except in trivial cases. In a number of cases it has been found that COST2 $\leq$ nt. Most of our discussions in the remaining parts of Part I of this thesis will be concerned with a detailed study of COST1 and COST2 and attempts to minimize them.

It is clear from (3.4) and (3.5) that to minimize COST we need to minimize

(i) $$\sum_{k=2}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k},$$

(ii) minimize t_0, or equivalently minimize

$$\sum_{k=1}^{n-1} \frac{\deg(k)-1}{d_{n-1} d_{n-2} \cdots d_k},$$

(iii) minimize C_k^m for each k.

These questions are considered in the following sections.

3.3 Heuristics for Selecting the Initial Spanning Tree

The initial spanning tree used in Char's algorithm can be obtained by performing a breadth-first search (BFS) or a depth-first search (DFS) on the given graph. The implementation of Char's algorithm given in [14] selects the initial spanning tree by performing a BFS starting at a vertex of maximum degree. In this section we consider the question of using DFS for selecting the initial spanning tree. Our objective is to minimize t_0 and C_k^m. For results relating to DFS, [3] may be consulted.

Let T_{DFS} be a DFS tree of the given graph G. Starting at the root of T_{DFS}, let the vertices of G be numbered as n, n-1, ..., 1, in the order in which they are visited during the DFS. As pointed out earlier, with such a numbering, T_{DFS} will have the tree compatibility property and in the corresponding tree sequence DIGIT(i) > i, $1 \leq i \leq n-1$. It should be noted that each ancestor of k in T_{DFS} will have a number greater than k and each descendant will have a number less than k. Furthermore, there are no cross edges in G. In other words, if x and y are two vertices such that neither of them is a descendant of the other in T_{DFS}, then

the edge (x,y) is not in G. Now we prove the following.

THEOREM 3.5.

If vertex k is a leaf in T_{DFS}, then $|T'_k| = 0$.

Proof:

When vertex k is a leaf, the number of any vertex adjacent to k will be greater than k. Therefore, the tree compatibility property is always satisfied at position k. In other words, no non-tree sequence which does not have the tree compatibility property at position k is generated. Hence $|T'_k| = 0$. □

Let δ_k be the number of descendants of vertex k in T_{DFS}. Then

THEOREM 3.6.

$$C_k^m \leq \delta_k$$

Proof:

Recall that C_k^m is the maximum number of computational steps required to perform a Type 2 computation for any sequence in $T_k \cup T'_k$. Also Type 2 computation requires traversing a path from vertex k in the subgraph which does not include the edges (k,REF(k)), (k+1,REF(k+1)), ..., (n-1,REF(n-1)). This, along with the fact that there are no cross edges in G, means that the traversing is done using only the descendants of k. Hence the theorem. □

To minimize t_0, we need to minimize the sum on the right-hand side of the expression for $(t+t_0)$ given in Theorem 3.3. Each term in this sum will be minimized when its numerator is as small as possible and the denominator is as large as possible. Thus it is clear that if the vertices of the given graph G are numbered in such a way that the degrees deg(n-1), deg(n-2), ..., deg(1) of the vertices n-1, n-2, ..., 1 are in the ascending order and the numbers d_{n-1}, d_{n-2}, ..., d_1 are in the descending order, then $(t+t_0)$ will be reduced considerably. Since deg(n) does not appear in the expression for $(t+t_0)$, we can number the vertex having the maximum degree in G as n. In other words, we can start the DFS to find the initial spanning tree at a vertex of maximum degree.

Consider now a DFS spanning tree T_{DFS} of the graph G. Let $\Gamma'(i)$ be the set of ancestors of vertex i in T_{DFS} which are adjacent to i in G and let $d'_i = |\Gamma'(i)|$. We may recall that to find the numbers $d_1, d_2, \ldots, d_{n-1}$, we start with the graph G_1 obtained from G by assigning unit weight to each edge of G. Then d_i is the sum of the weights of the edges incident on i in the graph G_i which is obtained from G_1 by performing pivotal condensations at the vertices 1, 2, ..., i-1. Since pivotal condensation does not reduce the weight of any edge connecting i to any vertex in $\Gamma'(i)$, and since each such edge has a weight of value at least one, it follows that

$$d_i \geq d'_i, \quad 1 \leq i \leq n-1.$$

It is evident from Theorems 3.5 and 3.6 and the above that t_0 could be reduced considerably if we

(i) maximize the number of leaves in T_{DFS},

(ii) maximize the number of ancestors of each vertex during the DFS, and

(iii) minimize the number of descendants δ_k, for each k.

To achieve these objectives, we suggest the following two heuristics for selecting the initial spanning tree using DFS.

Heuristic 1: Start the DFS at a vertex of maximum degree. During the search, when we are at vertex i, choose, from among the neighbours of i, the one having the maximum number of ancestors in the tree developed so far. If more than one vertex has this property, then choose, from among these vertices, the one having minimum degree in G.

Heuristic 2: Start the DFS at a vertex of maximum degree. During the search, when we are at vertex i, choose, from among the neighbours of i, the one having minimum degree in G. If more than one vertex has this property, then choose, from among these vertices, the one having the maximum number of ancestors in the tree developed so far.

We have implemented Char's algorithm using each one of the above two heuristics. In Table 3.2 we give the number of non-tree sequences generated by the algorithm when the initial spanning tree is selected using a BFS (as suggested in [14]), as well as when each one of the above two heuristics is used. The test graphs used in our comparison have been generated randomly using the procedure given in [18], and in Table 3.1 we give the number of vertices and edges of these ten test graphs. From Table 3.2 it is clear that the heuristics considerably reduce the number of non-tree sequences generated by the algorithm. We also note that these two heuristics generate approximately the same number of non-tree sequences. So either one of them can be used in an efficient implementation of Char's algorithm.

3.4 Path Compression

We may recall (Section 3.2) that the cost of execution of Char's algorithm consists of two components - COST1, the cost of Type 1 computations and COST2, the cost of Type 2 computations. Whereas COST1 explicitly depends on the initial spanning tree, COST2 depends on the initial spanning tree as well as the number of comparisons required to test a sequence for the tree compatibility property. The heuristics for the selection of the initial spanning tree discussed in Section 3.3 are aimed at reducing both COST1

Table 3.1

Test Graphs

Graph	Vertices	Edges	Spanning trees
G_1	9	20	24672
G_2	10	20	13931
G_3	10	24	151662
G_4	11	25	151719
G_5	11	30	1360710
G_6	11	35	12897990
G_7	12	30	1592512
G_8	12	30	1820488
G_9	12	35	14689650
G_{10}	13	35	26520950

Table 3.2

Number of Non-tree Sequences Generated

Graph	Number of Spanning trees	Number of non-tree sequences		
		BFS	Heuristic 1	Heuristic 2
G_1	24672	20738	14412	14438
G_2	13931	8778	5308	5310
G_3	151662	120259	66079	67036
G_4	151719	90831	65657	65950
G_5	1360710	1112223	504279	481506
G_6	12897990	7559568	7136979	6971221
G_7	1592512	871944	528193	528128
G_8	1820488	1151321	634183	635357
G_9	14689650	11998877	6179924	6207721
G_{10}	26520950	20921468	9096476	8941338

and COST2. Though the number of comparisons required in a straightforward implementation is also influenced by the initial spanning tree, the actual number of comparisons made during the execution of the algorithm can be reduced considerably by an appropriate choice of a data structure for maintaining the information relating to a tree sequence. In this section we discuss a method which can be used to achieve this.

Consider a sequence λ = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), x, REF(k+1), ..., REF(n-1)) with $x \neq$ REF(k), generated by Char's algorithm. Let G_λ be the corresponding subgraph of the given graph G. Let G'_λ be the subgraph obtained by removing from G_λ the edge (k,x). Clearly G'_λ is a spanning 2-tree of G. To test whether λ is a tree sequence or not, Char's algorithm traverses the sequence of vertices k, x, DIGIT(x), DIGIT(DIGIT(x)), ... until the vertex k or a vertex j > k is encountered. In the latter case, λ is a tree sequence. Suppose λ is a tree sequence, and let P denote the path represented by the sequence of vertices k, x, DIGIT(x), DIGIT(DIGIT(x)), ..., j.

After generating and identifying the tree sequence λ, the algorithm proceeds to generate sequences in which DIGIT(1), DIGIT(2), ..., DIGIT(k-1), and DIGIT(k) are the same as in λ. This is done by changing DIGIT(k+1), DIGIT(k+2), ..., DIGIT(n-1) in an appropriate order. So the

path P will be present in all the ~~subgraphs~~ corresponding to such sequences. Consider now one such sequence λ' which is to be tested for the tree compatibility property at position i. Clearly $i > k$. Let in λ', DIGIT(i) = α. Then to test the tree compatibility property, we need to traverse the sequence P' of vertices i, α, DIGIT(α), DIGIT(DIGIT(α)), ... and so on until vertex i or a vertex greater than i is encountered. If k lies on P', then the sequence of vertices k, x, DIGIT(x), DIGIT(DIGIT(x)), ..., j representing P will be a subsequence of P'. Hence, in such a case, while traversing P', when we encounter k, we can proceed directly to j. In other words, we can effectively compress P' if we keep track of the information relating to the path P. This technique, called path compression, will considerably reduce the actual number of comparisons made during the execution of Char's algorithm. Path compression has also been successfully used in designing several efficient algorithms [19].

To implement Char's algorithm with path compression, we use a new array NEXTVERTEX. Whereas the DIGIT array keeps the adjacency information of each sequence, the NEXTVERTEX array, for a tree, is defined as NEXTVERTEX(i) = j, where j is the first vertex greater than i reachable from vertex i as we traverse the tree from i to vertex n. We create and maintain the NEXTVERTEX array as follows. Since for the initial tree sequence, DIGIT(i) = REF(i) > i, $1 \leq i \leq n-1$,

we initialize NEXTVERTEX(i) = REF(i), $1 \leq i \leq n-1$. Whenever a tree sequence λ = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), x, REF(k+1), ..., REF(n-1)) is generated by changing the value of DIGIT(k) of the previous tree sequence, the NEXTVERTEX array is updated as follows.

Update 1: NEXTVERTEX(i) = REF(i), $k+1 \leq i \leq n-1$.

Update 2: NEXTVERTEX(k) = j, where j is the first vertex greater than k in the tree path from vertex k to vertex n.

Note that j will be known when the tree compatibility test for λ is completed.

We have implemented Char's algorithm with path compression using NEXTVERTEX array. In Table 3.3 we give the total number of comparisons made by the implementation of Char's algorithm with Heuristic 1 and the implementation with Heuristic 1 and path compression, for the ten randomly generated graphs given in Table 3.1. From Table 3.3 it is clear that the use of path compression considerably reduces the total number of comparisons.

Next we compute the number of computational steps required to create and update the NEXTVERTEX array. Note that initially NEXTVERTEX(n-1) = REF(n-1) = n. Since Update 2 sets NEXTVERTEX(n-1) to the first vertex greater than n-1 in the tree path from vertex n-1 to vertex n, it is

Table 3.3

Number of Comparisons Made

Graph	Number of Spanning trees	Number of non-tree sequences	Number of Comparisons	
			Heuristic 1	Heuristic 1 with Path Compr.
G_1	24672	14412	110374	83342
G_2	13931	5308	40711	33811
G_3	151662	66079	593753	442127
G_4	151719	65657	565449	434929
G_5	1360710	504279	5323910	3841745
G_6	12897990	7136979	64931380	48970813
G_7	1592512	528193	5599546	4080411
G_8	1820488	634183	6749935	4808779
G_9	14689650	6179924	66484047	45516982
G_{10}	26520950	9096476	102984126	69897903

clear that NEXTVERTEX(n-1) is always equal to n and so we need to update only NEXTVERTEX(i), $1 \leq i \leq n-2$. For each tree sequence of the form λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), x, REF(k+1), ..., REF(n-1)) with $x \neq$ REF(k), Update 1 requires (n-k-1) assignments and Update 2 requires exactly one assignment. Thus Update 1 and Update 2 together require (n-k) computational steps for each tree sequence of the form λ_k. The number of tree sequences of the form λ_k is given by t(k+1)-t(k), where t(i), $1 \leq i \leq n-2$, is the number of spanning trees of the graph $G_k^{(s)}$ defined in Section 3.2. Thus, the total number of computational steps required to create and update the NEXTVERTEX array is given by

$$n-1 + \sum_{k=1}^{n-2} [t(k+1)-t(k)](n-k) = 2t(n-1) + \sum_{k=2}^{n-2} t(k),$$

since t(1)=1.

$$= t(n-1) + \sum_{k=2}^{n-1} t(k)$$

$$= t\left[\frac{1}{d_{n-1}} + \sum_{k=2}^{n-1} \frac{1}{d_{n-1}d_{n-2}\cdots d_k}\right],$$

which is O(nt). Thus employing path compression in the implementation of Char's algorithm does not affect the asymptotic complexity of the algorithm.

Since the total number of comparisons is reduced when Char's algorithm is implemented with path compression, the execution time of the algorithm with path compression should also be less than the execution time of the algorithm without path compression. This can be verified from Table 3.4 where we tabulate the execution times for three implementations of Char's algorithm - Char's implementation where breadth-first search (BFS) is used to select the initial spanning tree, implementation using Heuristic 1, and implementation using Heuristic 1 and path compression - for the ten randomly generated graphs given in Table 3.1. These execution times are for a CDC Cyber 170 and these algorithms are implemented in PASCAL. From Table 3.4, it can also be seen that the reduction in the execution time achieved when Char's algorithm is implemented with path compression is not proportional to the corresponding reduction in the number of comparisons made. This is due to the additional work required to create and update the NEXTVERTEX array. However, the reduction is significant for denser graphs, for example G_9 and G_{10}. Thus it is clear that Char's algorithm with Heuristic 1 and path compression is an efficient implementation of the algorithm.

Table 3.4

Execution Time

Graph	Number of Spanning trees	Execution time in seconds		
		BFS	Heuristic 1	Heuristic 1 with Path Compr.
G_1	24672	2.037	1.924	1.767
G_2	13931	0.970	0.944	0.928
G_3	151662	12.495	10.167	9.462
G_4	151719	10.661	10.489	10.205
G_5	1360710	102.660	87.835	81.612
G_6	12897990	994.735	966.608	894.635
G_7	1592512	113.124	105.868	99.491
G_8	1820488	129.747	124.721	115.582
G_9	14689650	1193.974	1026.815	946.918
G_{10}	26520950	2264.015	1822.345	1662.247

CHAPTER 4

ANALYSIS OF CHAR'S ALGORITHM FOR SPECIAL GRAPHS

In this chapter we present an analysis of Char's algorithm for special graphs. In Chapter 3 the time complexity of this algorithm was shown to be $O(m+n+n^3t)$ for a general graph with m edges, n vertices and t spanning trees. However, for a class of graphs the algorithm requires only $O(m+n+nt)$ time and hence is as efficient as theoretically possible. In Section 4.1 we discuss the complexity of Char's algorithm for this class of graphs. For certain graphs in this class the number t_0 of non-tree subgraphs generated by the algorithm can be determined as a function of n. In Section 4.2 we present elegant expressions for the value of t_0 in the cases of complete graphs, ladders and wheels. We also obtain expressions for the total number of computational steps required in the cases of ladders and wheels. Based on these expressions we show that in these cases Char's algorithm requires, on the average, at most 4 computational steps per spanning tree. Since the value of t_0 and hence the complexity of Char's algorithm depends on the initial spanning tree, it is an interesting problem to study the minimum value of t_0 possible for a graph. In Section 4.3 we define the minimum value of t_0 over all initial spanning trees of a graph as the min-tree-number of the graph and present some

conjectures on this number.

4.1 Complexity of Char's Algorithm for a Special Class of Graphs

Let $G^{(n-1)}$ be the set of all n-vertex connected graphs which have at least one vertex of degree n-1. Any graph $G \in G^{(n-1)}$ contains a star tree as one of its spanning trees. In this section we first prove that for any graph $G \in G^{(n-1)}$, Char's algorithm requires only O(m+n+nt) time when the star tree is chosen as the initial spanning tree.

Consider a graph $G \in G^{(n-1)}$. Let the star vertex in G, which is a vertex of degree n-1, be numbered as n and the other vertices of G be numbered in any arbitrary order. Since a star tree is used as the initial spanning tree, it is clear that REF(i) = n, $1 \leq i \leq n-1$. Let λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k), REF(k+1), ..., REF(n-1)), with DIGIT(k) $\neq$ REF(k), be a non-tree sequence generated by Char's algorithm which does not have the tree compatibility property at k. Then for the non-tree subgraph $G_k = (V, E_k)$ of G corresponding to λ_k,

$$V = \{1, 2, \ldots, n\}$$

and

$$E_k = \{(1,\text{DIGIT}(1)), (2,\text{DIGIT}(2)), \ldots, (k,\text{DIGIT}(k)), (k+1,n), \ldots, (n-1,n)\}.$$

Since in Char's algorithm any edge (i,DIGIT(i)) in E_k is traversed from vertex i to vertex DIGIT(i), we can consider the edges in E_k as directed edges. Thus, from our discussion in Section 3.1, it follows that G_k contains exactly one directed circuit passing through vertex k using edges from the set {(1,DIGIT(1)), (2,DIGIT(2)), ..., (k,DIGIT(k))}.

Now we prove the following.

THEOREM 4.1.

For any graph $G \in G^{(n-1)}$, $t_0 \leq t$ if a star tree is used as the initial spanning tree.

Proof:

We prove the theorem by showing that each non-tree sequence generated by Char's algorithm when applied on G corresponds to a unique tree sequence. Let λ_i and λ_j be two distinct non-tree sequences generated by the algorithm which do not have the tree compatibility property at positions i and j respectively, and let G_a and G_b be the corresponding non-tree subgraphs of G. In G_a there is a directed circuit (i, DIGIT(i), DIGIT(DIGIT(i)), ..., x, i) passing through vertex i, and in G_b there is a directed circuit (j, DIGIT(j), DIGIT(DIGIT(j)), ..., y, j) passing through vertex j. From $G_a = (V, E_a)$ and $G_b = (V, E_b)$, let us construct the graphs $G'_a = (V, E'_a)$ and $G'_b = (V, E'_b)$ such that

and

$$E'_a = E_a - (x,i) \cup (x,n)$$
$$E'_b = E_b - (y,j) \cup (y,n).$$

It can be easily seen that both G'_a and G'_b are spanning trees of G. In fact, when considered as directed graphs, both G'_a and G'_b are directed spanning trees in which every vertex except vertex n has out-degree equal to 1.

The proof is completed by showing that G'_a and G'_b are distinct whenever G_a and G_b are distinct. Assume, on the contrary, that $G'_a = G'_b$. First we note that i = j. If not, let i < j. Then the edge (j,n) will be present in G'_a but not in G'_b, contradicting the assumption that $G'_a = G'_b$. Thus i = j.

Note that all the directed edges in E_a except (x,i) are present in E'_a, and all the directed edges in E_b except (y,j) are present in E'_b. Since i = j, $E'_a = E'_b$ and in $G'_a = G'_b$ every vertex except n has out-degree equal to 1, it follows that x = y. Thus we have $E_a = E'_a - (x,n) \cup (x,i) = E'_b - (y,n) \cup (y,j) = E_b$, contradicting that G_a and G_b are distinct. Hence the theorem. □

The above was originally proved in [14]. The proof given here is more elegant than that given in [14]. Since the time complexity of Char's algorithm is $O(m+n+n(t+t_0))$ and $t_0 \leq t$ for any graph in $G^{(n-1)}$, when a star tree is used as the initial spanning tree, we get the following.

THEOREM 4.2.

For any graph $G \in G^{(n-1)}$, Char's algorithm requires $O(m+n+nt)$ time if a star tree is used as the initial spanning tree. □

Now we prove a result more general than Theorem 4.2. Consider a graph $G \in G^{(n-1)}$ and let S_1 be any arbitrary spanning tree of G. Let S_2 be a star tree of G and v_a be the star vertex. Suppose we assign the number n to vertex v_a, and number the other vertices of G using S_1 so that in the sequence corresponding to S_1, DIGIT(i) > i, $1 \leq i \leq n-1$. In S_2, every vertex is adjacent to vertex n, and so in the corresponding tree sequence DIGIT(i) > i, $1 \leq i \leq n-1$. Thus if $t_{0,1}$ and $t_{0,2}$ are the numbers of non-tree sequences generated when S_1 and S_2 are used as the initial spanning trees, then by Theorem 3.4 $t_{0,1} = t_{0,2}$. Since these arguments are valid for any arbitrary initial spanning tree, we get the following.

THEOREM 4.3.

For any graph $G \in G^{(n-1)}$, $t_0 \leq t$ for any initial spanning tree if a vertex of degree n-1 is assigned the number n, and the other vertices of G are numbered so that in the corresponding tree sequence DIGIT(i) > i, $1 \leq i \leq n-1$. □

Since a complete graph is in $G^{(n-1)}$ and all the vertices of a complete graph have degree n-1, we get the following result from Theorem 4.3.

COROLLARY 4.3.1.

For a complete graph, $t_0 \leq t$ for any choice of the initial spanning tree. □

4.2 Char's Algorithm on Complete Graphs, Ladders and Wheels

In this section we discuss the behaviour of Char's algorithm in the cases of complete graphs, ladders and wheels and point out certain interesting properties of the algorithm in these cases. We develop elegant expressions for the number t_0 of non-tree subgraphs generated when Char's algorithm is applied on these graphs. Note that these graphs belong to $G^{(n-1)}$ and hence Char's algorithm requires O(m+n+nt) time in these cases.

4.2.1 Complete Graphs

Let K_n be an n-vertex complete graph. For every vertex i of K_n

$$\deg(i) = n-1. \qquad (4.1)$$

Let G_1 be the weighted graph obtained from K_n by assigning unit weight to each edge of K_n. Let G_i, $2 \leq i \leq n-1$, be the graph obtained from G_{i-1} by performing pivotal condensation at vertex $i-1$ in G_{i-1}. Since G_1 is a complete graph with n vertices, the graph G_i, $2 \leq i \leq n-1$, is a complete graph with $(n-i+1)$ vertices and all the edges in G_i have the same weight, say c_i. Thus we get

$$d_i = (n-i)c_i, \quad 1 \leq i \leq n-1. \tag{4.2}$$

We now prove by induction that

$$c_i = \frac{n}{n-i+1}, \quad 1 \leq i \leq n-1. \tag{4.3}$$

Since $c_1 = 1$, (4.3) is true for $i = 1$. Let (4.3) be true for all values of $i < k$. Consider $i = k > 1$. In G_{k-1}, vertex $k-1$ is adjacent to the vertices $k, k+1, \ldots, n$ and $d_{k-1} = (n-k+1)c_{k-1}$. Thus in G_k the weight c_k of each edge is given by

$$c_k = c_{k-1} + \left(\frac{c_{k-1}}{n-k+1}\right)$$

$$= \left(\frac{n-k+2}{n-k+1}\right) c_{k-1}$$

$$= \left(\frac{n-k+2}{n-k+1}\right)\left(\frac{n}{n-k+2}\right)$$

$$= \left(\frac{n}{n-k+1}\right).$$

Thus (4.3) follows. Using (4.1), (4.2), and (4.3) in Theorem 3.3 we get

$$t+t_0 = 1 + (n-2)t\left[\frac{1}{n^{n-2}} + \sum_{i=2}^{n-1} \frac{i}{n^{i-1}}\right].$$

Since $t = n^{n-2}$ for K_n, the above expression reduces to

$$t+t_0 = 2n^{n-2} - \left[\frac{n^{n-1}-1}{(n-1)^2}\right]$$

and hence we get the following.

THEOREM 4.4.

For an n-vertex complete graph,

$$t_0 = n^{n-2} - \left[\frac{n^{n-1}-1}{(n-1)^2}\right]. \qquad \square$$

From the above expression we see that $t_0 < t$ for a complete graph, which is better than the bound given in Corollary 4.3.1.

4.2.2 Ladders

The graph shown in Fig. 4.1(a) is called an n-vertex ladder. (A ladder is also known as a fan [20].) Let L_n

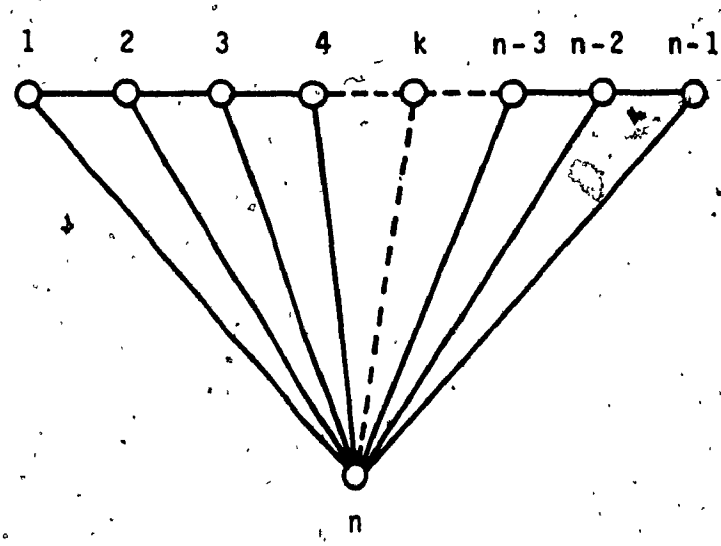


Figure 4.1(a)

n-vertex Ladder

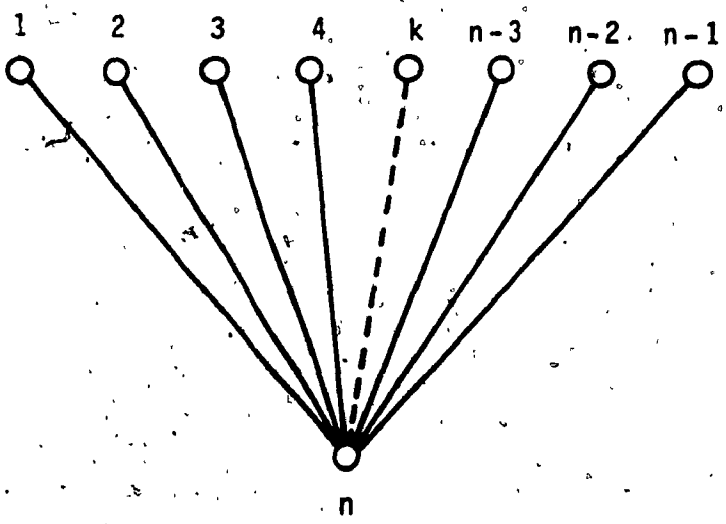


Figure 4.1(b)

Star Tree

denote the number of spanning trees of an n-vertex ladder and let L_n^0 denote the number of non-tree subgraphs generated when Char's algorithm is applied on the ladder choosing the star tree shown in Fig. 4.1(b) as the initial spanning tree. Note that a 1-vertex ladder has a single vertex and no edge and hence $L_1 = 1$, and that a 2-vertex ladder has a single edge and so $L_2 = 1$.

It is known that [21]

$$L_n = 3L_{n-1} - L_{n-2}, \quad n \geq 4. \tag{4.4}$$

From (4.4) it can be seen that $L_2, L_3, \ldots$ are alternate numbers in the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, ... with $L_2 = 1$, $L_3 = 3$, $L_4 = 8$ and so on. Let the number next to L_i in the Fibonacci sequence be denoted as $\text{NEXT}(L_i)$. Note that $\text{NEXT}(L_1) = 1$ and $\text{NEXT}(L_2) = 2$. Using the identities

$$L_1 + L_2 = 1 + L_2 = \text{NEXT}(L_2),$$
$$\text{NEXT}(L_1) + \text{NEXT}(L_2) = 1 + \text{NEXT}(L_2) = L_3,$$
$$L_j + \text{NEXT}(L_j) = L_{j+1}, \quad j \geq 2,$$
$$\text{NEXT}(L_{j-1}) + L_j = \text{NEXT}(L_j), \quad j \geq 2,$$

we can show that

$$\sum_{i=1}^{j} L_i = \text{NEXT}(L_j) \tag{4.5}$$

and

$$\sum_{i=1}^{j} \text{NEXT}(L_i) = L_{j+1}. \tag{4.6}$$

Now we compute the value of L_n^0 using Theorem 3.1. Note that for an n-vertex ladder,

$$\deg(1) = \deg(n-1) = 2 \tag{4.7}$$

and

$$\deg(i) = 3, \ 2 \leq i \leq n-2. \tag{4.8}$$

The graph $G_i^{(s)}$ obtained from an n-vertex ladder by coalescing the vertices i, i+1, ..., n is shown in Fig. 4.2. The number of spanning trees of this graph is given in the following.

LEMMA 4.1.

The number of spanning trees t(i) of the graph shown in Fig. 4.2 is given by

$$t(i) = \text{NEXT}(L_i), \ 1 \leq i \leq n-1.$$

Proof:

Let e be the edge shown in Fig. 4.2. The number t(i) of spanning trees of this graph is the sum of the number L_i of spanning trees of the graph constructed by removing e, and the number t(i-1) of spanning trees of the graph constructed by contracting e. Thus

$$t(i) = L_i + t(i-1).$$

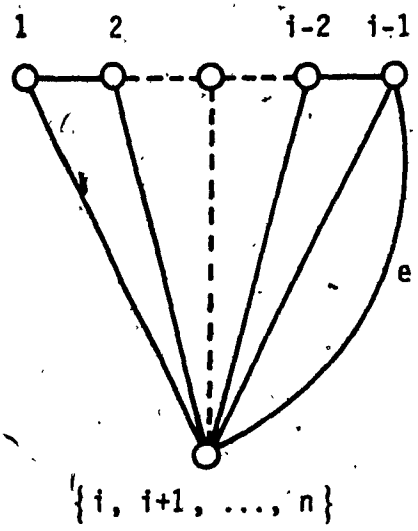


Figure 4.2

Graph $G_i^{(s)}$

Solving this recurrence relation using (4.5) we get

$$t(i) = \text{NEXT}(L_i). \qquad \square$$

Using (4.7), (4.8), and Lemma 4.1 in Theorem 3.1 we get

$$L_n+L_n^0 = 1 + \text{NEXT}(L_1) + \text{NEXT}(L_{n-1}) + 2\sum_{k=2}^{n-2} \text{NEXT}(L_k)$$

$$= 1 + \sum_{k=1}^{n-1} \text{NEXT}(L_k) + \sum_{k=2}^{n-2} \text{NEXT}(L_k).$$

Since $\text{NEXT}(L_1) = 1$ we can rewrite the above as

$$L_n+L_n^0 = \sum_{k=1}^{n-1} \text{NEXT}(L_k) + \sum_{k=1}^{n-2} \text{NEXT}(L_k).$$

Using (4.6) in the above expression we get

$$L_n+L_n^0 = L_n + n-1$$

and hence the following theorem.

THEOREM 4.5.

For an n-vertex ladder, the number L_n^0 of non-tree subgraphs generated by Char's algorithm when a star tree is used as the initial spanning tree is given by

$$L_n^0 = L_{n-1}. \qquad \square$$

This result was first stated and proved in [14]. However, the proof given here is much simpler than that reported in [14].

Solving (4.4) we get

$$L_n = \frac{1}{\sqrt{5}}\left[\left(\frac{3+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{3-\sqrt{5}}{2}\right)^{n-1}\right]. \qquad (4.9)$$

Using (4.9) and Thoerem 4.5 we can show that

$$\operatorname*{Lt}_{n\to\infty} \frac{L_n^0}{L_n} \approx 0.382.$$

Since L_{n-1}/L_n is an increasing function of n, it follows that for an n-vertex ladder Char's algorithm generates at most $0.382L_n$ non-tree sequences.

We now proceed to compute the number of computational steps required by Char's algorithm to generate all the spanning trees of an n-vertex ladder, when a star tree is used as the initial spanning tree. Note that to output a spanning tree at least (n-1) computational steps are required. In the following analysis we do not consider the computational steps required to output the spanning trees. Also we do not consider the computational steps required to determine the initial spanning tree.

We have shown in Section 3.2 that the total cost COST of Char's algorithm is the sum of

$$\text{COST1} = \sum_{k=2}^{n-1} t(k) - (n-2) \qquad (4.10)$$

and

$$\text{COST2} \leq \sum_{k=1}^{n-1} c_k^m |T_k \cup T_k'|. \qquad (4.11)$$

For an n-vertex ladder, the number t(k) of spanning trees of the graph obtained by coalescing the vertices k, k+1, ..., n is given in Lemma 4.1. Using this value for t(k) in (4.10), we can show that for an n-vertex ladder

$$\text{COST1} = \sum_{k=2}^{n-1} \text{NEXT}(L_k) - (n-2) = L_n - n + 1. \qquad (4.12)$$

Let COST2 = COST2(T)+COST2(T') where COST2(T) is the cost of Type 2 computations for generating tree sequences and COST2(T') is the corresponding cost for generating non-tree sequences. It can be easily seen that in the case of a ladder, the circuit passing through vertex k in the non-tree subgraph corresponding to a sequence in T', $2 \leq k \leq n-1$, is of the form (k, DIGIT(k), k). These non-tree subgraphs require exactly two computational steps to test

for the tree compatibility property. From this observation and the fact that Char's algorithm generates L_{n-1} non-tree subgraphs for an n-vertex ladder, when a star tree is used as the initial spanning tree, we get

$$\text{COST2}(T') = 2L_{n-1}. \tag{4.13}$$

Next we compute COST2(T). Since to verify the tree compatibility property of any sequence in T_k, we start with the edge (k,DIGIT(k)) and traverse some of the edges in the set $\{(1,\text{DIGIT}(1)), (2,\text{DIGIT}(2)), \ldots, (k-1,\text{DIGIT}(k-1))\}$, it follows that a minimum of one and a maximum of k computational steps will be required for each tree sequence in T_k. Let $T_k(i)$ be the set of tree sequences in T_k which require exactly i computational steps to test for the tree compatibility property. The following lemma gives the number of sequences in any $T_k(i)$, $1 \leq i \leq k$ and $1 \leq k \leq n-1$.

LEMMA 4.2.

For an n-vertex ladder, the number of tree sequences in T_k, $1 \leq k \leq n-1$, which require exactly i computational steps is given by

$$|T_k(i)| = \text{NEXT}(L_{k-i+1}),\ 1 \leq i \leq k,\ 1 \leq k \leq n-2,$$

and

$$|T_{n-1}(1)| = 0,$$

$$|T_{n-1}(i)| = \text{NEXT}(L_{n-i}),\ 2 \leq i \leq n-1.$$

Proof:

We first prove the lemma for i = 1. Consider any tree sequence λ in T_k, k < n-1. The spanning tree G_λ corresponding to this sequence contains the edges (k+1,n), (k+2,n), ..., (n-1,n). If, in addition, the edge (k,k+1) is also in G_λ, then the sequence λ would require only one computational step. All the spanning trees having the edge (k,k+1) should be of the form shown in Fig. 4.3(a), where $1 \leq p \leq k$. For a given value of p, the number of spanning trees of the form shown in Fig. 4.3(a) is L_p. Thus we get

$$|T_k(1)| = \sum_{p=1}^{k} L_p = \text{NEXT}(L_k). \qquad (4.14)$$

For any other value of i, $2 \leq i \leq k$, the edges (k,k-1), (k-1,k-2), ..., (k-i+2,k-i+1), (k-i+1,n) must be traversed in G_λ while testing λ for the tree compatibility property. These spanning trees should be of the form shown in Fig. 4.3(b), where $1 \leq p \leq k-i+1$. Hence we get

$$|T_k(i)| = \sum_{p=1}^{k-i+1} L_p = \text{NEXT}(L_{k-i+1}). \qquad (4.15)$$

Note that all the sequences in T_{n-1} for an n-vertex ladder require at least two computational steps. Furthermore (4.15) is valid for k = n-1 and i ≠ 1. These

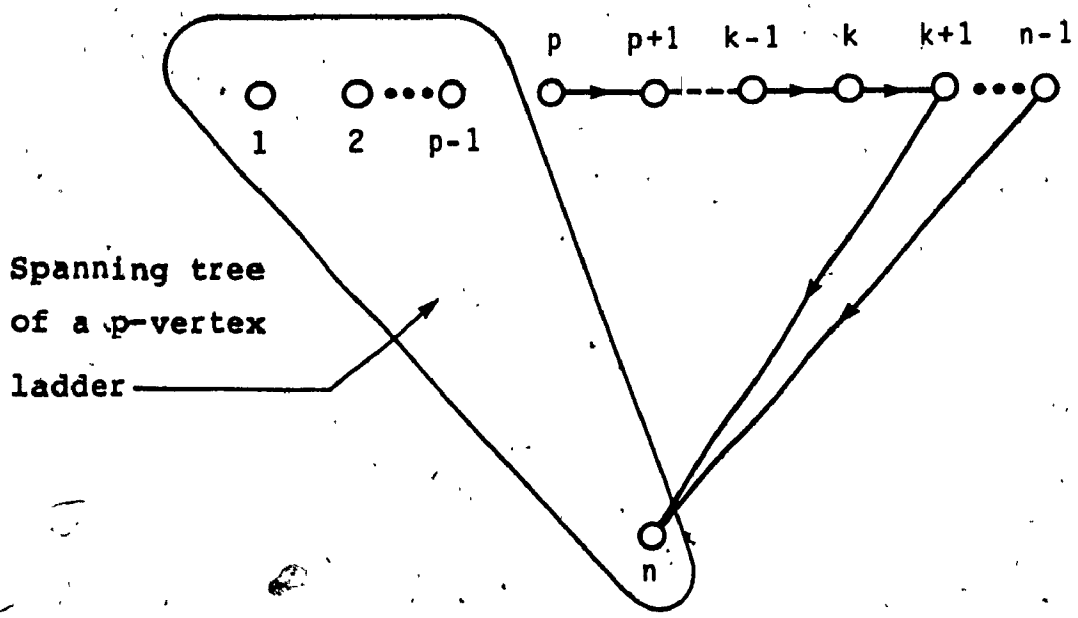


Figure 4.3(a)

Spanning Trees in $T_k(1)$

$1 \leq p \leq k$

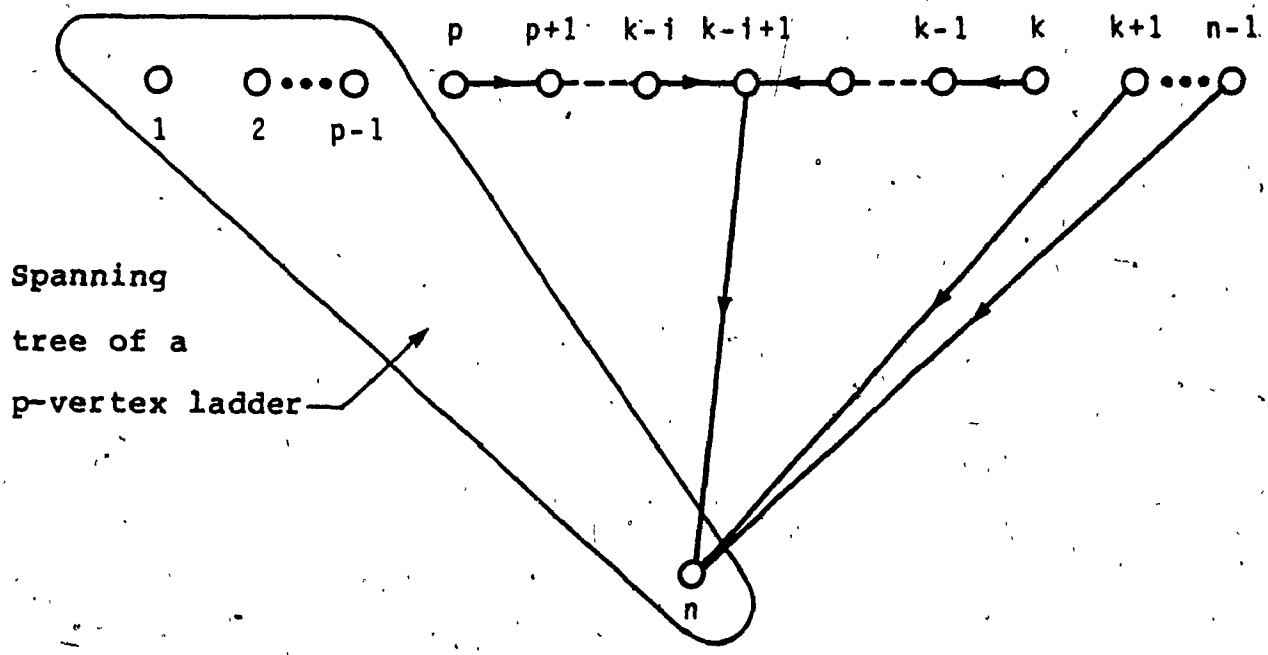


Figure 4.3.(b)

Spanning Trees in $T_k(i)$

$1 \leq p \leq k-i+1$

observations along with (4.14) and (4.15) prove the lemma.□

Using Lemma 4.2 we get

$$\text{COST2}(T) = \sum_{k=1}^{n-1} \sum_{i=1}^{k} i|T_k(i)| = 2L_n - n. \qquad (4.16)$$

From (4.12), (4.13) and (4.16), we get

$$\text{COST} = 3L_n + 2L_{n-1} - 2n + 1. \qquad (4.17)$$

Using (4.9) in (4.17) we can show that

$$\frac{\text{COST}}{L_n} < 4.$$

Thus we get the following.

THEOREM 4.6.

For an n-vertex ladder, when the star tree is used as the initial spanning tree,

(i) the total cost of Char's algorithm is given by

$$\text{COST} = 3L_n + 2L_{n-1} - 2n + 1.$$

(ii) Char's algorithm requires, on the average, at most 4 computational steps to generate a spanning tree. □

4.2.3 Wheels

The graph shown in Fig. 4.4(a) is an n-vertex wheel.

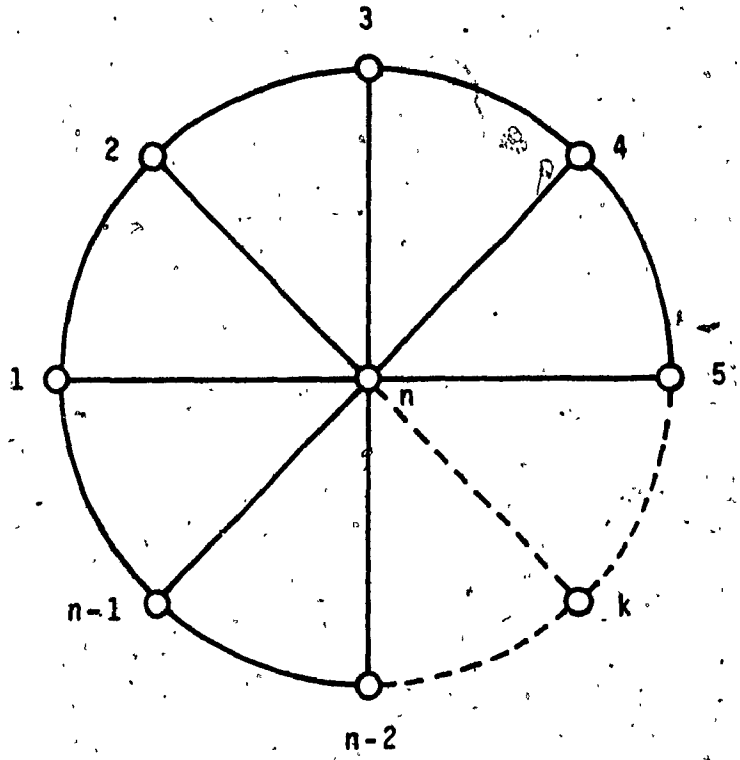


Figure 4.4(a)

n-vertex Wheel

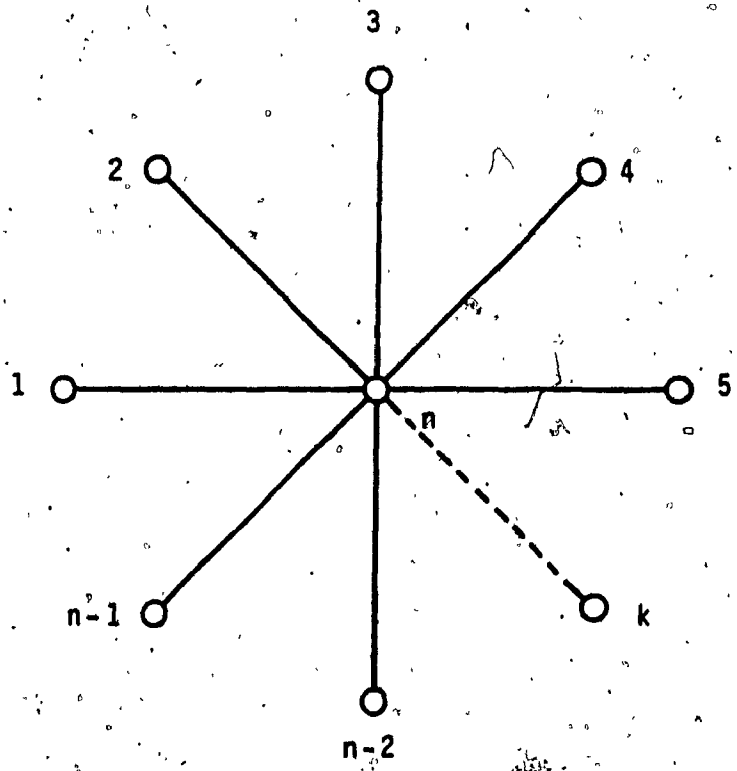


Figure 4.4(b)

Star Tree

Let W_n denote the number of spanning trees of an n-vertex wheel G and W_n^0 denote the number of non-tree subgraphs generated by Char's algorithm when the star tree shown in Fig. 4.4(b) is chosen as the initial spanning tree. Now we derive an expression for W_n^0 using Theorem 3.1.

The wheel shown in Fig. 4.4(a) can be redrawn as in Fig. 4.5(a). Note that for an n-vertex wheel,

$$\deg(i) = 3, \quad 1 \leq i \leq n-1. \tag{4.18}$$

The graph $G_i^{(s)}$ obtained from G by coalescing the vertices i, i+1, ..., n is shown in Fig. 4.5(b). The number of spanning trees t(i) of $G_i^{(s)}$ is given in the following lemma.

LEMMA 4.3.

The number t(i) of spanning trees of the graph shown in Fig.4.5(b) is given by

$$t(i) = L_{i+1}.$$

Proof:

Let e be the edge of $G_i^{(s)}$ indicated in Fig. 4.5(b). The graph $G_i^{(s)}-e$, constructed by removing e, is shown in Fig. 4.5(c) and the graph $G_i^{(s)}.e$, constructed by contracting e, is shown in Fig. 4.5(d). Note that Fig. 4.5(c) is identical to Fig. 4.2 and so $t(G_i^{(s)}-e) = \text{NEXT}(L_i)$. Also the graph $G_i^{(s)}.e$ is isomorphic to the graph $G_{i-1}^{(s)}$. Thus we get the following recurrence relation.

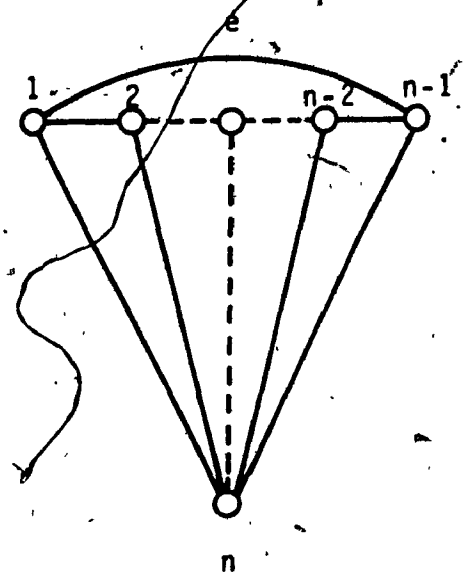
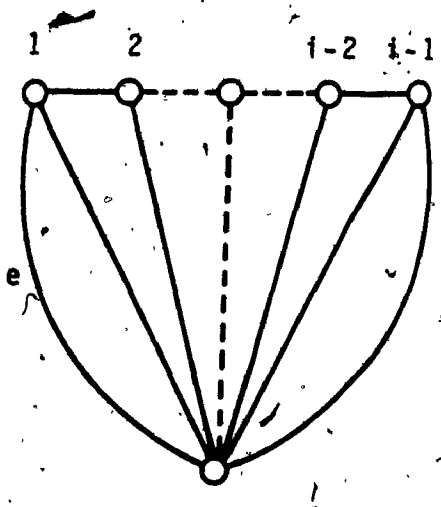


Figure 4.5(a)

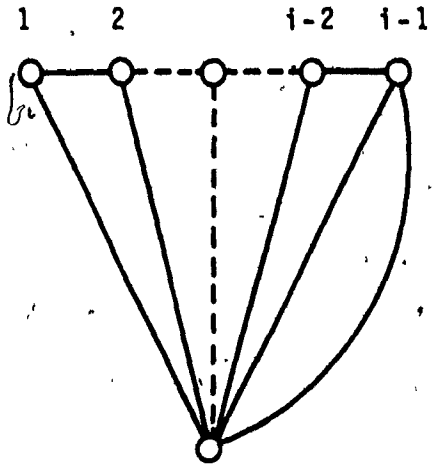
n-vertex Wheel redrawn



$\{i, i+1, \ldots, n\}$

Figure 4.5(b)

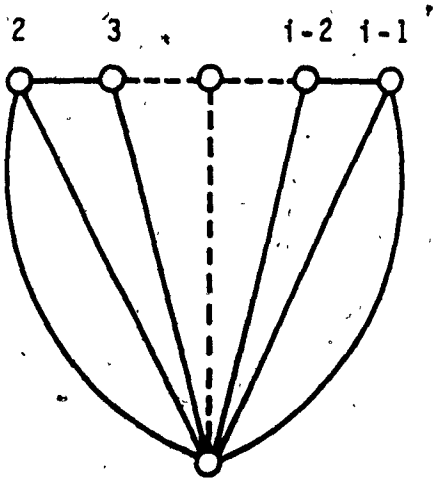
Graph $G_i^{(s)}$



$\{i, i+1, \ldots, n\}$

Figure 4.5(c)

Graph $G_i^{(s)}-e$



$\{1, i, i+1, \ldots, n\}$

Figure 4.5(d)

Graph $G_i^{(s)}.e$

$$t(i) = \text{NEXT}(L_i) + t(i-1).$$

Solving this recurrence relation using (4.6), we obtain

$$t(i) = \sum_{k=1}^{i} \text{NEXT}(L_k) = L_{i+1}. \qquad \square$$

The following lemma gives the number of spanning trees of an n-vertex wheel.

LEMMA 4.4.

The number W_n of spanning trees of an n-vertex wheel is given by

$$W_n = 2\text{NEXT}(L_n) - L_n - 2, \quad n \geq 3.$$

Proof:

Consider the n-vertex wheel G shown in Fig. 4.5(a) and let e be the edge indicated. Then the graph G-e, constructed by removing e from G, is the n-vertex ladder shown in Fig. 4.1(a) and the graph G.e, constructed by contracting e in G, is shown in Fig. 4.5(e). The number of spanning trees of the graph in Fig. 4.5(e) can be shown to be

$$W_{n-1} + t(n-2) = W_{n-1} + L_{n-1}.$$

Thus we get the following recurrence relation for the number

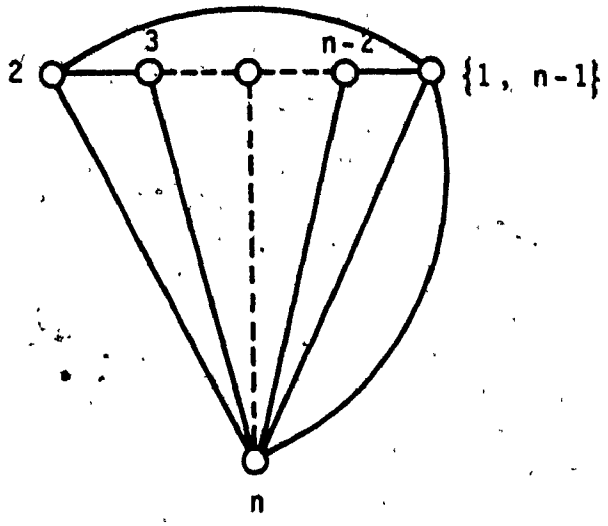


Figure 4.5(e)

Graph G.e

of spanning trees of an n-vertex wheel.

$$W_n = W_{n-1} + L_{n-1} + L_n.$$

Solving the above recurrence relation we get

$$W_n = 2\sum_{i=1}^{n} L_i - L_n - 2.$$

Using (4.5) the above expression can be reduced to

$$W_n = 2\text{NEXT}(L_n) - L_n - 2. \qquad \square$$

Using (4.18) and Lemma 4.3 in Theorem 3.1 we get

$$W_n + W_n^0 = 1 + 2\sum_{k=1}^{n-1} L_{k+1}$$

$$= 2\text{NEXT}(L_n) - 1. \qquad (4.19)$$

From (4.19) and Lemma 4.4 we get the following theorem, which was first proved in [14] using very involved arguments.

THEOREM 4.7.

For an n-vertex wheel, the number W_n^0 of non-tree subgraphs generated by Char's algorithm when a star tree is used as the initial spanning tree is given by

$$W_n^0 = 1 + L_n. \qquad \square$$

It has been shown in [22] that

$$W_n = \left(\frac{3+\sqrt{5}}{2}\right)^{n-1} + \left(\frac{3-\sqrt{5}}{2}\right)^{n-1} - 2.$$

Using this expression and Theorem 4.7, it can be shown that

$$\lim_{n\to\infty} \frac{W_n^0}{W_n} \approx 0.4472.$$

This means that for large values of n, Char's algorithm generates at most $0.4472W_n$ non-tree sequences for an n-vertex wheel. Note that W_n^0/W_n can be shown to be a decreasing function of n.

Next we compute the number of computational steps required by char's algorithm to generate all the spanning trees of an n-vertex wheel. In the case of a wheel, the graph obtained by coalescing the vertices k, k+1, ..., n is shown in Fig. 4.6. The number t(k) of spanning trees of this graph is

$$W_k + L_k = 2\text{NEXT}(L_k) - 2.$$

Thus, for an n-vertex wheel

$$\text{COST1} = \sum_{k=2}^{n-1} (2\text{NEXT}(L_k) - 2) + (n-2) = 2L_n - 3n + 4. \quad (4.20)$$

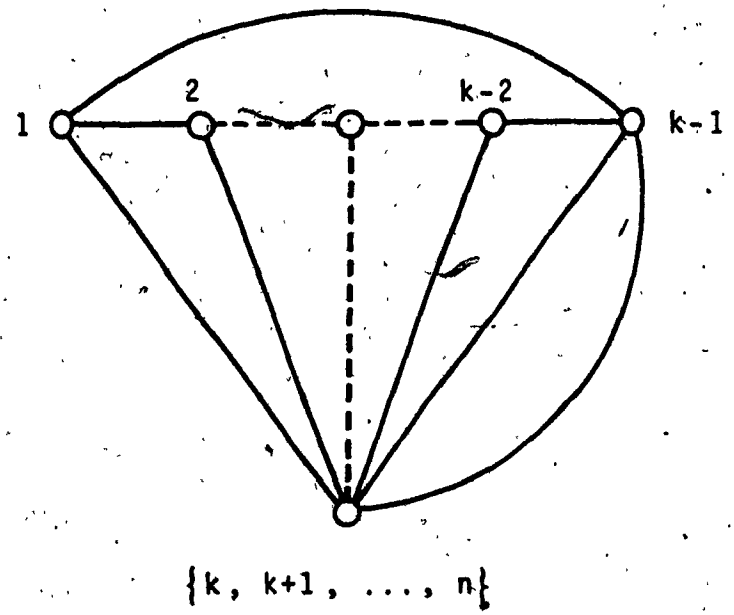


Figure 4.6

Graph $G_k^{(s)}$

Char's algorithm generates $1+L_n$ non-tree subgraphs for an n-vertex wheel. It can easily be seen that one of these non-tree subgraphs is the circuit (n-1, n-2, ..., 2, 1, n-1) and the other is the circuit (n-1, 1, 2, ..., n-2, n-1). Note that each of these two non-tree subgraphs requires exactly (n-1) computational steps to test for the tree compatibility property. Since each of the other L_n-1 non-tree subgraphs require exactly two computational steps, we get

$$COST2(T^1) = 2L_n+2n-4. \qquad (4.21)$$

Now we prove the following.

LEMMA 4.5.

For an n-vertex wheel, the number of tree sequences in T_k, $1 \leq k \leq n-1$, which require exactly k computational steps is given by

$$|T_k(k)| = 2.$$

Proof:

First consider the case $k = n-1$. The spanning tree corresponding to a sequence in $T_{n-1}(n-1)$ must contain either the edges (n-1,n-2), (n-2,n-3), ..., (2,1), (1,n) or the edges (n-1,1), (1,2), ..., (n-3,n-2), (n-2,n). Thus the lemma follows for $k = n-1$. For other values of k, $1 \leq k \leq n-2$, the spanning tree corresponding to a sequence in $T_k(k)$

must contain the edges (k+1,n), (k+2,n), ..., (n-1,n) and the edges (k,k-1), (k-1,k-2), ..., (2,1) along with either the edge (1,n) or the edge (1,n-1). Thus the lemma follows for any k, $1 \leq k \leq n-2$. Hence the lemma. □

LEMMA 4.6.

For an n-vertex wheel, the number of tree sequences in T_k which require exactly i computational steps is given by

$$|T_k(i)| = L_{k-i+2}, \quad 1 \leq i \leq k-1, \quad 2 \leq k \leq n-2,$$

and

$$|T_{n-1}(1)| = 0,$$

$$|T_{n-1}(i)| = 2L_{n-i+1}, \quad 2 \leq i \leq n-2.$$

Proof:

First we consider the case k = n-1. It can be easily seen that all the sequences in T_{n-1} require at least two computational steps and hence $|T_{n-1}(1)| = 0$. Now, consider the case k = n-1 and $2 \leq i \leq n-1$. When considered as a directed tree, the spanning tree corresponding to a sequence in $T_{n-1}(i)$, $i \neq 1$, must contain a directed path of length i from vertex n-1 to vertex n. Thus each one of these spanning trees should be of one of the three forms shown in Figs. 4.7(a), (b), and (c). The numbers of spanning trees in these three groups are, respectively,

$$\sum_{p=1}^{n-i} L_p = \text{NEXT}(L_{n-i}),$$

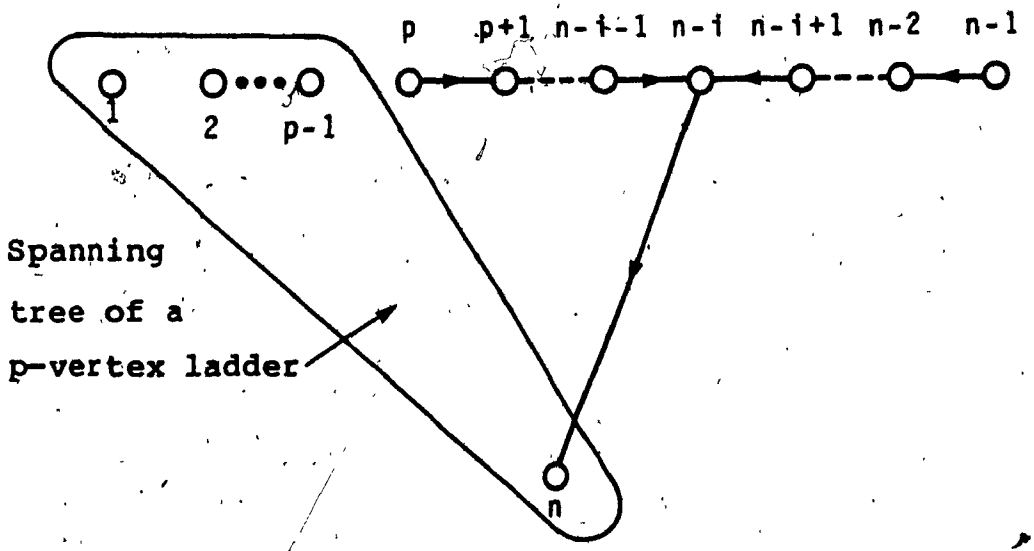


Figure 4.7(a)

Spanning Trees in $T_{n-1}(i)$

which do not contain the edge (1,n-1) or the edge (n-1,1)

$1 \leq p \leq n-i$

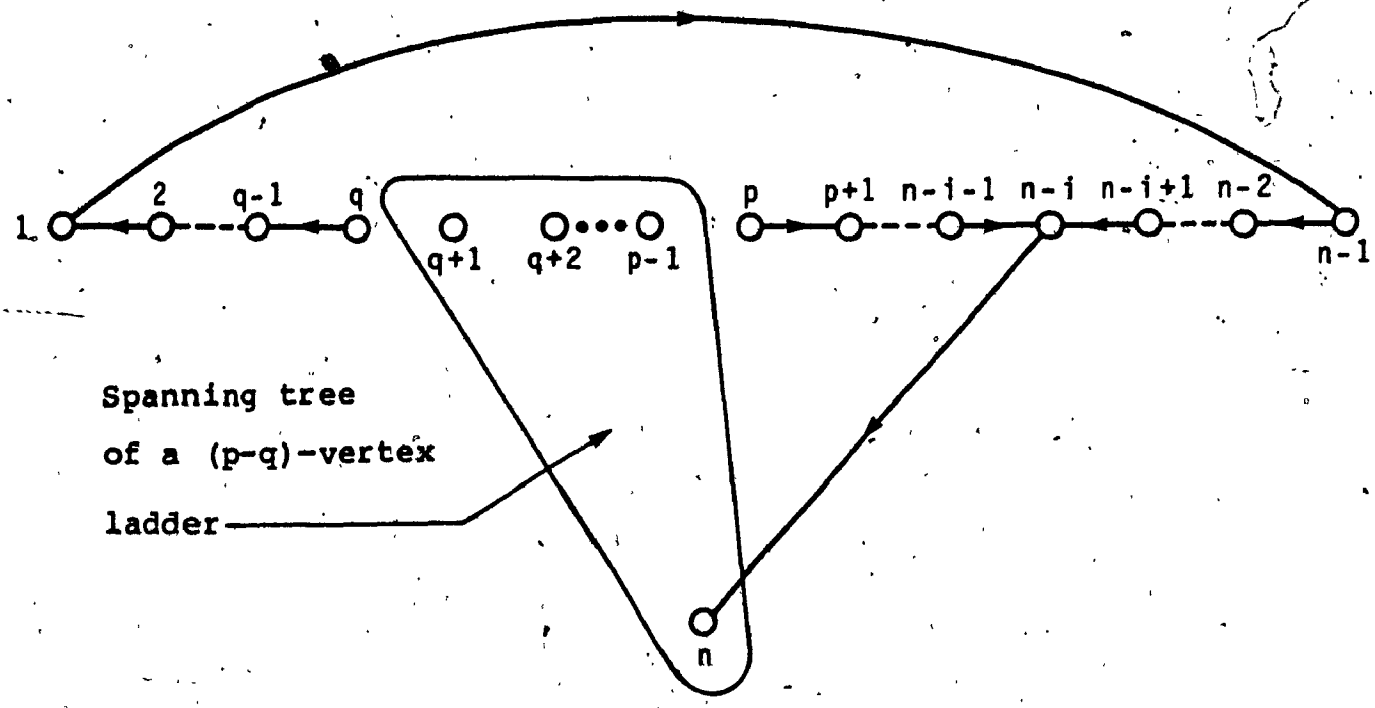


Figure 4.7(b)

Spanning Trees in $T_{n-1}(i)$

which contain edge (1,n-1)

$2 \leq p \leq n-i$

$1 \leq q \leq p-1$

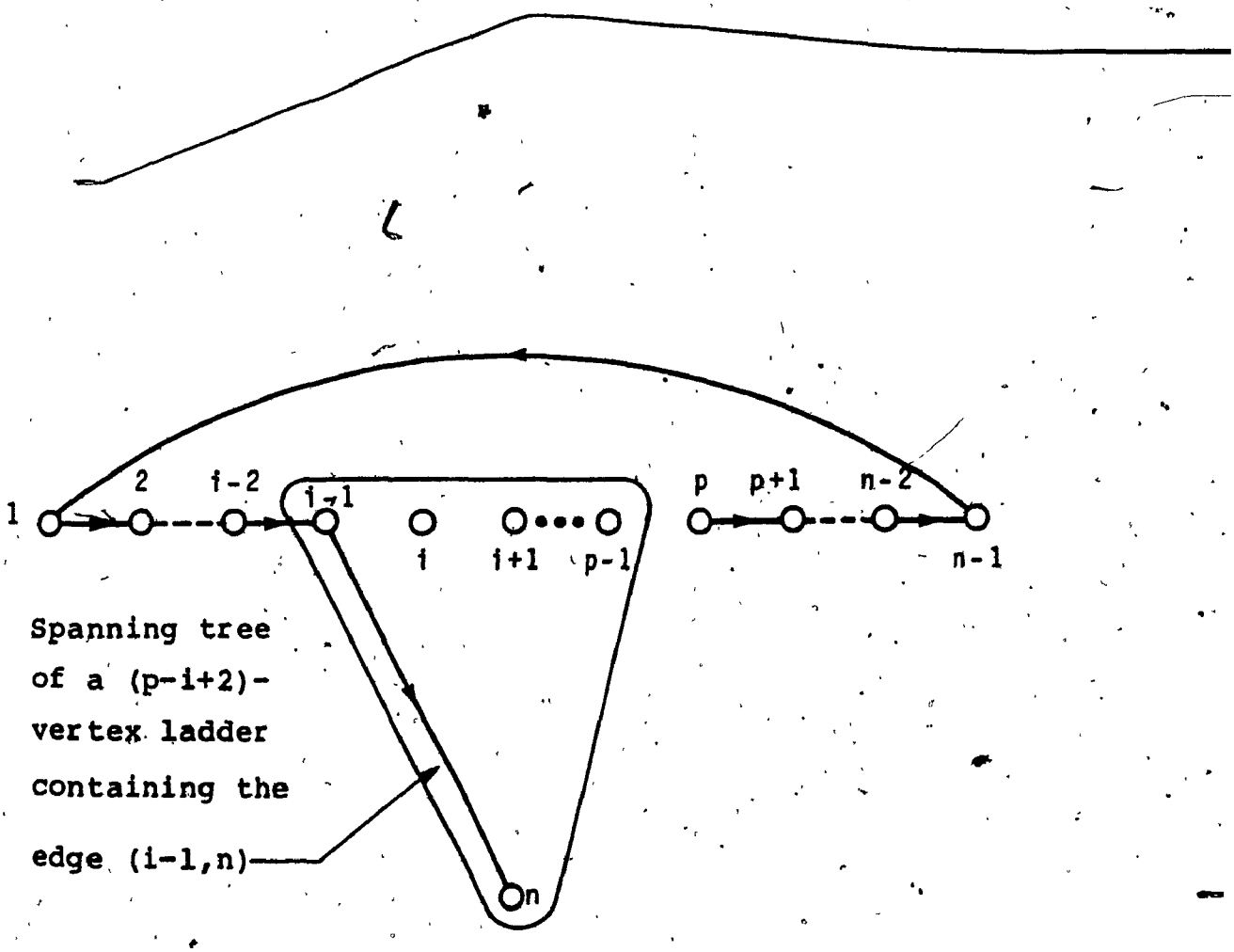


Figure 4.7(c)

Spanning Trees in $T_{n-1}(i)$

which contain edge $(n-1,1)$

$i \leq p \leq n-1$

$$\sum_{p=2}^{n-i}\sum_{q=1}^{p-1} L_{p-q} = L_{n-i},$$

$$\sum_{p=i}^{n-1} \text{NEXT}(L_{p-i+1}) = L_{n-i+1}.$$

Thus we get

$$|T_{n-1}(i)| = \text{NEXT}(L_{n-i})+L_{n-i}+L_{n-i+1}$$
$$= 2L_{n-i+1}, \quad 2 \leq i \leq n-2$$

which proves the lemma for $k = n-1$.

We next prove the lemma for other values of k, $2 \leq k \leq n-2$. First we consider the case $i = 1$. The tree sequences in $T_k(1)$ must contain the edge $(k,k+1)$ and these spanning trees should be of the form shown in Fig. 4.8. The number of spanning trees of the form shown in Fig. 4.8(a) is L_p, $1 \leq p \leq k$. The number of spanning trees of the form shown in Fig. 4.8(b) is L_{p-q}, for $2 \leq p \leq k$ and $1 \leq q \leq p-1$. Thus the total number of spanning trees in $T_k(1)$ is given by

$$|T_k(1)| = \sum_{i=1}^{k} L_p + \sum_{p=2}^{k}\sum_{q=1}^{p-1} L_{p-q} = L_{k+1}$$

and hence the lemma follows for $2 \leq k \leq n-2$ and $i = 1$.

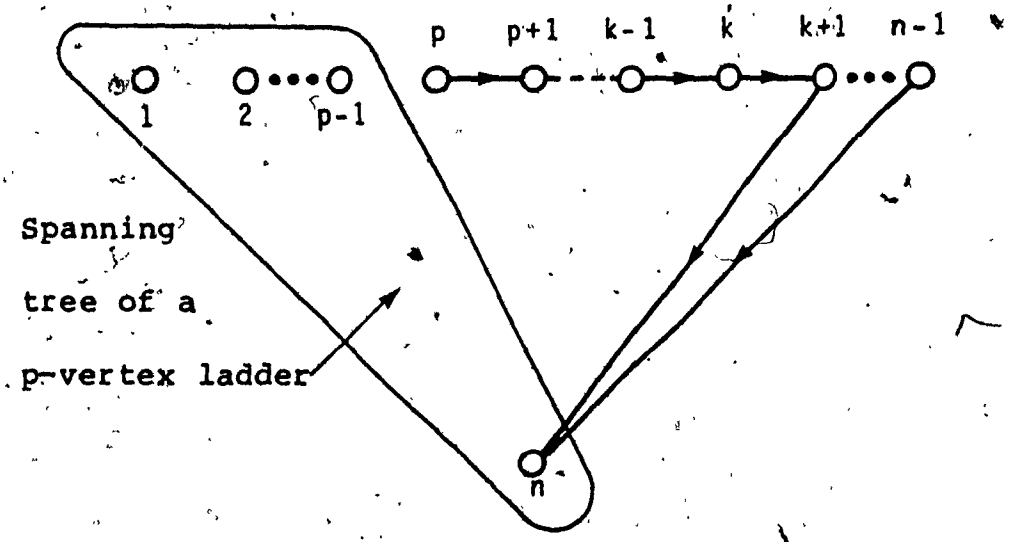


Figure 4.8(a)

Spanning Trees in $T_k(1)$

which do not contain edge (1,n-1)

$1 \leq p \leq k$

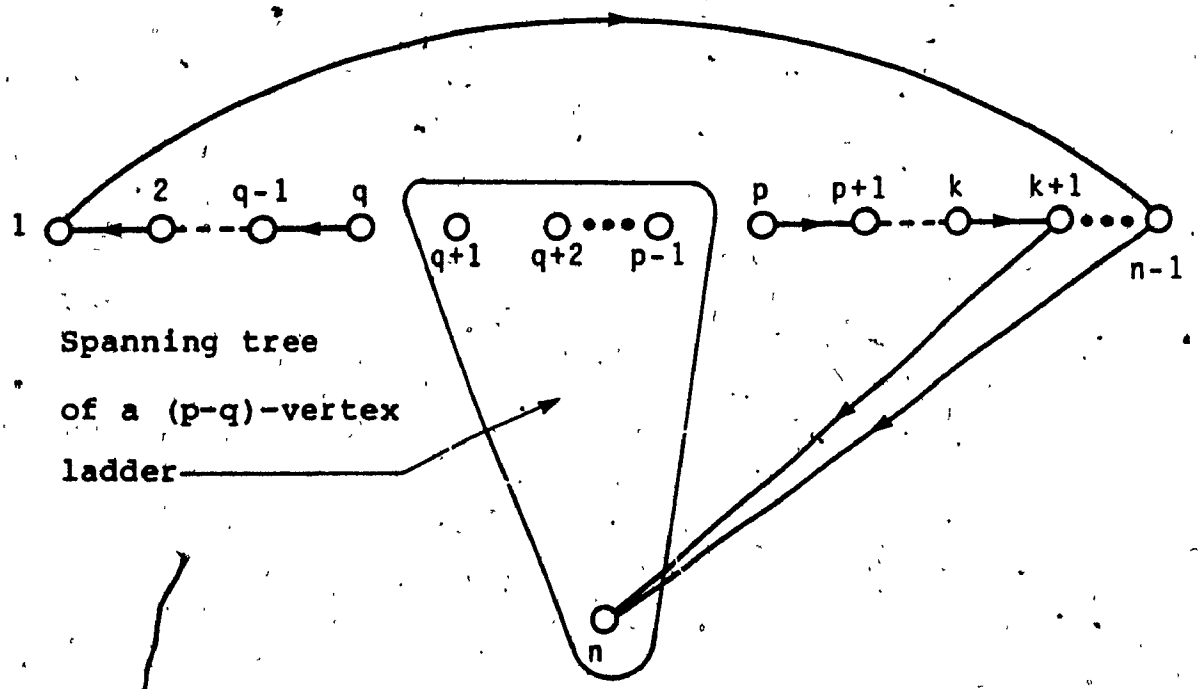


Figure 4.8(b)

Spanning Trees in $T_k(1)$

which contain edge (1,n-1)

$2 \leq p \leq k$

$1 \leq q \leq p-1$

For other values of i, $2 \leq i \leq k-1$, the tree sequences in $T_k(i)$ must contain the edges $(k,k-1)$, $(k-1,k-2)$, ..., $(k-i+2,k-i+1)$, $(k-i+1,n)$ and these spanning trees should be of the form shown in Fig.4.9. The number of spanning trees of the form in Fig.4.9(a) is L_p, $1 \leq p \leq k-i+1$ and the number of those of the form shown in Fig. 4.9(b) is L_{p-q}, $2 \leq p \leq k-i+1$ and $1 \leq q \leq p-1$. Thus the total number of spanning trees in $T_k(i)$, $2 \leq i \leq k-1$, is given by

$$|T_k(i)| = \sum_{p=1}^{k-i+1} L_p + \sum_{p=2}^{k-i+1} \sum_{q=1}^{p-1} L_{p-q} = L_{k-i+2}.$$

Hence the proof. □

Using Lemmas 4.5 and 4.6 we get

$$\begin{aligned} COST2(T) &= \sum_{k=1}^{n-1} 2k + \sum_{i=2}^{n-2} 2iL_{n-i+1} + \sum_{k=2}^{n-2} \sum_{i=1}^{k-1} iL_{k-i+2} \\ &= 3NEXT(L_n) - 3n - 4. \end{aligned} \tag{4.22}$$

From (4.20), (4.21) and (4.22), we get the total number of computational steps required by Char's algorithm to generate all the spanning trees of an n-vertex wheel, when a star tree is used as the initial spanning tree, as

$$COST = L_{n+2} + 2L_n - 4n. \tag{4.23}$$

Using the expressions for L_n and W_n, and (4.23) we can show

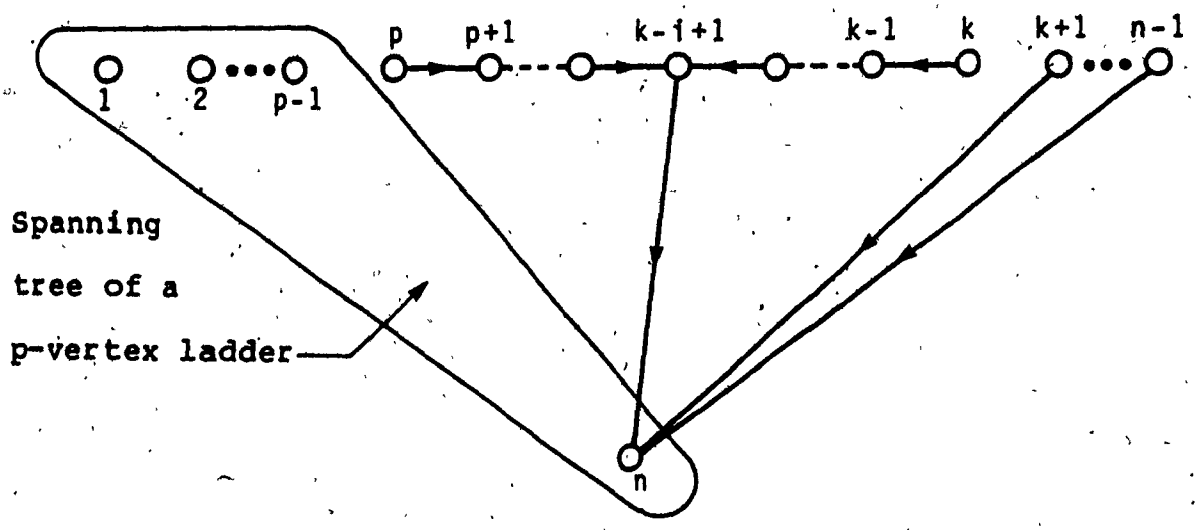


Figure 4.9(a)

Spanning Trees in $T_k(i)$

which do not contain edge (1,n-1)

$1 \leq p \leq k-i+1$

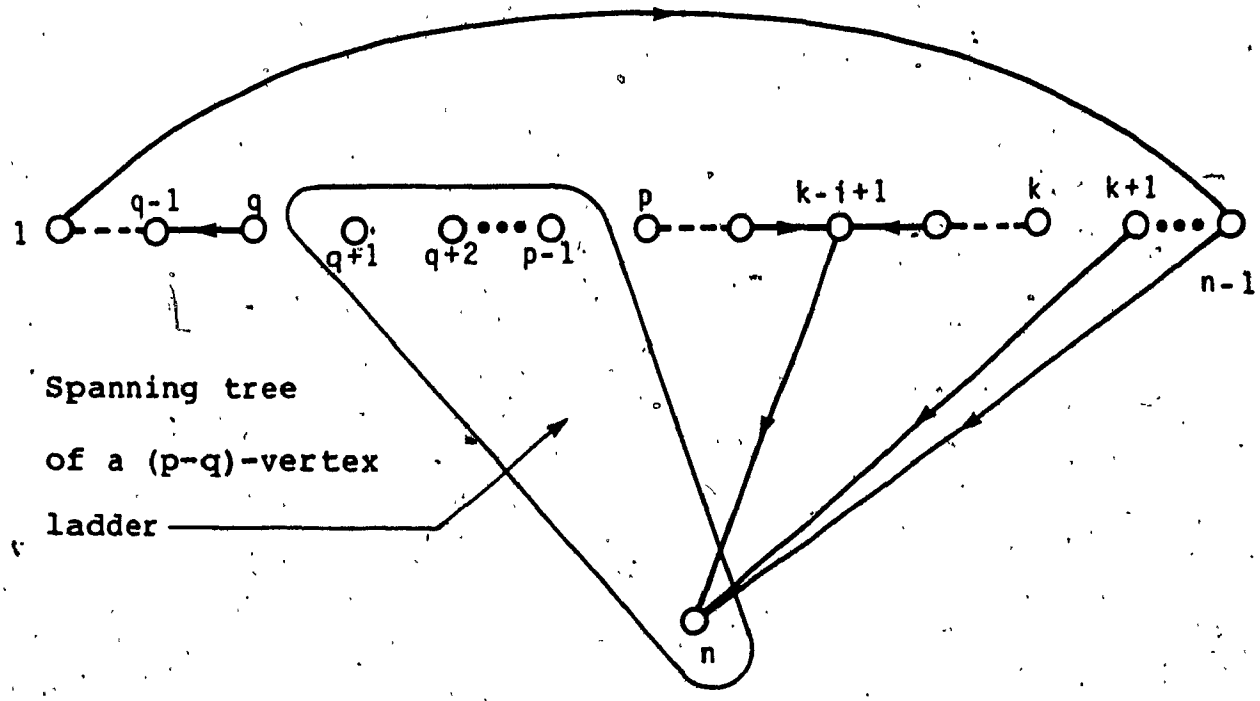


Figure 4.9(b)

Spanning Trees in $T_k(i)$

which contain edge (1,n-1)

$2 \leq p \leq k-i+1$

$1 \leq q \leq p-1$

that for an n-vertex wheel

$$\frac{\text{COST}}{W_n} < 4.$$

Thus we get the following.

THEOREM 4.8.

For an n-vertex wheel, when the star tree is used as the initial spanning tree,

(i) the total cost of Char's algorithm is given by

$$\text{COST} = L_{n+2}+2L_n-4n.$$

(ii) Char's algorithm requires, on the average, at most 4 computational steps to generate a spanning tree. □

4.3 Min-Tree-Number of a Graph and Some Conjectures

We have shown in Section 3.2 that for any graph G, the value of t_0 depends on the choice of the initial spanning tree. We now define the min-tree-number, ϵ_{min}, of G as the minimum value of t_0 over all possible choices of initial spanning trees.

Two immediate consequences of Theorem 4.1 and Corollary 4.3.1 are

THEOREM 4.9.

For any graph $G \in G^{(n-1)}$, $\epsilon_{min} \leq t$. □

THEOREM 4.10.

For a complete graph ϵ_{min} is independent of the choice of the initial spanning tree. □

Note that for a complete graph the value of ϵ_{min} is given in Theorem 4.4. In view of Theorem 4.3 and Theorem 4.9 the question arises whether for graphs in $G^{(n-1)}$, t_0 attains the minimum value ϵ_{min} when a star tree is chosen as the initial spanning tree.

We have computed the value of t_0 for a number of randomly generated graphs. For all these graphs, we have chosen the initial spanning tree by performing a breadth-first search [14]. In general, we have observed that $t_0 \leq t$, except in the case of certain sparse graphs having vertices of degree 2. We can prove that for an n-vertex circuit $t_0 = ((n-1)(n-2))/2$. Since an n-vertex circuit has n spanning trees, it follows that in this case $t_0 = O(nt)$. We observed from our computational experiences that only for n-vertex circuits $t_0 = O(nt)$. Note that a circuit is a sparse graph in which all the vertices are of degree 2. These observations lead us to believe that the following are true.

CONJECTURE 4.1.

For any biconnected graph, $\epsilon_{min} = O(nt)$. □

CONJECTURE 4.2.

For any biconnected graph with minimum degree at least 3, $\epsilon_{min} \leq t$. □

CHAPTER 5

MOD-CHAR: AN EFFICIENT IMPLEMENTATION OF CHAR'S ALGORITHM

In Chapter 3 we have shown that Char's algorithm involves two types of computations, namely the Type 1 computations and the Type 2 computations. Whereas the cost of Type 1 computations is $O(nt)$, Type 2 computations cost $O(n^3t)$ for an n-vertex graph. In this chapter we develop a new algorithm, based on the principles of Char's algorithm, which requires $O(n^2t)$ Type 2 computations only. Recall that Type 2 computations are essentially those required to test the sequences for the tree compatibility property. We call this modified algorithm as algorithm MOD-CHAR. In Section 5.1 we discuss algorithm MOD-CHAR and in Section 5.2 we present a complexity analysis of the algorithm. We discuss in Section 5.3 our computational results on algorithm MOD-CHAR and compare this algorithm with Char's and Gabow and Myers' algorithms.

5.1 Algorithm MOD-CHAR

Consider an n-vertex undirected graph $G = (V,E)$. Let the vertices of G be numbered as in Char's algorithm. Consider a tree sequence λ = (DIGIT(1), DIGIT(2), ..., DIGIT(k), REF(k+1), REF(k+2), ..., REF(n-1)) with DIGIT(k)

$\neq$ REF(k), generated by Char's algorithm when applied on G. Note that λ is a tree sequence in T_k (see Section 3.2). After generating λ, Char's algorithm proceeds to generate the tree sequences in $T_{k+1} \cup T_{k+2} \cup \ldots \cup T_{n-1}$ as well as the non-tree sequences in $T'_{k+1} \cup T'_{k+2} \cup \ldots \cup T'_{n-1}$ which have the same DIGIT(1), DIGIT(2), ..., DIGIT(k) as λ, by changing DIGIT(n-1), DIGIT(n-2), ..., DIGIT(k+1) in an appropriate order as described in Section 3.1. Then another sequence in $T_k \cup T'_k$ is generated by setting DIGIT(i) = REF(i) for $k+1 \leq i \leq n-1$, and changing DIGIT(k) in λ.

Consider now the sequences in $T_k \cup T'_k$ having the same DIGIT(1), DIGIT(2), ..., DIGIT(k-1) as λ. It is clear that these sequences are not generated immediately after λ, and generating each one of these sequences requires at most n Type 2 computations. We now show how these computations can be reduced by an appropriate implementation of Char's algorithm. We use the ideas developed in the course of the proof of Theorem 3.1.

Consider a tree sequence λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), REF(k), REF(k+1), ..., REF(n-1)). Let G'_k denote the spanning 2-tree obtained by removing the edge (k,REF(k)) from the spanning tree G_k corresponding to the tree sequence λ_k. Note that in G'_k the vertices k+1, k+2, ..., n are in one component and the vertex k is in the other component. For each vertex $x \neq$ REF(k) adjacent to vertex k,

the sequence λ''_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), x, REF(k+1), ..., REF(n-1)) will be in $T_k \cup T'_k$. This sequence can be classified as follows. If vertices k and x are in the same component of G'_k, then there is a circuit passing through vertex k in G''_k (the subgraph of G corresponding to the sequence λ''_k) and so λ''_k is a non-tree sequence in T'_k. On the other hand, if vertices k and x are in different components of G'_k, then λ''_k is a tree sequence in T_k. Thus if, for each λ_k defined above, we obtain the information whether each neighbour x of k in G is in the same component of G'_k as vertex k or not, then only one comparison is required to test each one of these sequences.

In order to determine the information about the two components of G'_k, we associate a label with each vertex of G'_k. We denote the label of a vertex i, $1 \leq i \leq n$ as LABEL(i). For each neighbour x of k, LABEL(x) is defined such that LABEL(x) = k if the vertices k and x are in the same component of G'_k; and LABEL(x) = n otherwise. In order to obtain these label values, we traverse the path in G'_k from vertex x to either vertex k or to some vertex greater than k. If this path leads to vertex k, then we set LABEL(x) = k; otherwise LABEL(x) = n. These computations are performed efficiently as follows.

Since in G'_k there is a path from each one of the vertices k+1, k+2, ..., n-1 to vertex n, we initialize

$$\text{LABEL}(i) = 0, \quad 1 \leq i \leq k-1,$$

$$\text{LABEL}(k) = k,$$

and

$$\text{LABEL}(i) = n, \quad k+1 \leq i \leq n.$$

For each neighbour x of k in G, we traverse the path in G'_k from x to some vertex y such that LABEL(y) $\neq$ 0. As soon as y is found, we traverse this path once again and set LABEL(v) = LABEL(y) for all the vertices v in this path except y. It is easy to see that this procedure determines the label values correctly. Moreover, each edge of G'_k is traversed at most twice in this procedure. More precisely, each one of the edges (1,DIGIT(1)), (2,DIGIT(2)), ..., (k-1,DIGIT(k-1)) is traversed at most twice and hence this traversal requires at most 2(k-1) computational steps. Thus the label values can be computed in O(n) time.

From the discussions thus far, it is clear that algorithm MOD-CHAR will require considerably less number of Type 2 computations than Char's algorithm. We now present a recursive version of algorithm MOD-CHAR in ALGOL-like notation.

<u>Modified Char's Algorithm to Enumerate All the Spanning Trees of a Graph</u>.

procedure MOD-CHAR;

comment procedure MOD-CHAR enumerates all the spanning trees of a connected n-vertex graph using algorithm

MOD-CHAR. The graph $\mathcal{G}$ is represented by the adjacency lists ADJ(i), $1 \leq i \leq n-1$, of its vertices.

```
procedure GENERATE(k);
comment procedure GENERATE, when called with the argument
        k, sets DIGIT(k) to generate a tree sequence.
        This procedure uses a local array LABEL.
begin
  if k = n
    then output the tree sequence
    else begin
      {Set DIGIT(i) to REF(i), k ≤ i ≤ n-1}
      DIGIT(k) := REF(k);
      GENERATE(k+1);
      {Generate all the sequences in T_k ∪ T'_k having the
      same DIGIT(1), DIGIT(2), ..., DIGIT(k-1)}
      compute LABEL(x) for each neighbour x of k;
      for x ∈ ADJ(k)-REF(k) do
        if LABEL(x) = n
          then begin
            DIGIT(k) := x;
            GENERATE(k+1)
          end
    end
end GENERATE;

begin
  find the initial tree sequence (REF(1), REF(2), ...,
```

```
    REF(n-1));
    renumber the vertices of G;
    GENERATE(1)
end MOD-CHAR;
```

5.2 Computational Complexity of Algorithm MOD-CHAR

We now study the complexity of generating all the spanning trees of a graph using algorithm MOD-CHAR. To output t spanning trees, this algorithm requires at least (n-1)t computational steps. These are not included in the following analysis. Also to find the initial spanning tree, we need O(m+n) computations where m is the number of edges in the graph and we do not include these also in our complexity analysis.

It is clear that algorithm MOD-CHAR requires the same amount of Type 1 computations as Char's algorithm. Thus from (3.4), COST1 for algorithm MOD-CHAR becomes

$$\text{COST1} = t \sum_{k=2}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k} - (n-2).$$

We can write COST1 as

$$COST1 = H_n t - (n-1)$$

where

$$H_n = \sum_{k=1}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k}.$$

Thus COST1 is $O(H_n t)$.

From our discussions in Section 5.1, we can see that algorithm MOD-CHAR requires Type 2 computations to determine the label values and to generate and test the sequences. First we compute the cost to determine the label values. It is easy to see that for a given tree sequence λ_k = (DIGIT(1), DIGIT(2), ..., DIGIT(k-1), REF(k), REF(k+1), ..., REF(n-1)), we need n computational steps to initialize LABEL(i), $1 \leq i \leq n$, and at most 2(k-1) steps to determine the necessary LABEL(x)'s. Since there are t(k) such λ_k's, where t(k) is the number of spanning trees of the graph obtained by coalescing the vertices k, k+1, ..., n in G, the cost of computing the labels is less than

$$\sum_{k=1}^{n-1} (n + 2(k-1))t(k) = t \sum_{k=1}^{n-1} \frac{n + 2(k-1)}{d_{n-1} d_{n-2} \cdots d_k},$$

which is $O(nH_n t)$.

Now we compute the cost of generating and testing the sequences. Note that algorithm MOD-CHAR requires exactly

one comparison to test a sequence and one assignment to generate a tree sequence. Thus the computational steps required to generate and test the sequences is $2t+t_0$, which is $O(nH_n t)$ since $t+t_0$ is $O(nH_n t)$ according to Theorem 3.3. Thus the total number of Type 2 computations required by algorithm MOD-CHAR is

$$COST2 = O(nH_n t).$$

From these results we get the following.

THEOREM 5.1.

The time complexity of algorithm MOD-CHAR is $O(nH_n t)$, where

$$H_n = \sum_{k=1}^{n-1} \frac{1}{d_{n-1}d_{n-2}\cdots d_k}. \qquad \square$$

Since the complexity of algorithm MOD-CHAR depends on the number H_n, we now study this number. If $S_i = \{j \mid j>i$ and vertex j is adjacent to vertex $i\}$, $1 \leq i \leq n-1$, then it can be easily seen that $d_i \geq |S_i|$. Thus, in general, $d_i \geq 1$ and so each term in H_n is less than or equal to 1. Assuming that, in the worst case, each term in H_n is 1, the computational complexity of algorithm MOD-CHAR becomes $O(n^2 t)$. Thus algorithm MOD-CHAR has a better asymptotic complexity than Char's algorithm which has $O(n^3 t)$ asymptotic

time complexity. However, the bound $H_n \leq n$ is a very crude one and in the case of a number of graphs H_n is a constant as we shall see now.

Let M denote the set of all graphs such that the vertices of each graph in M can be numbered as in Char's algorithm with the property that $|S_i| \geq 2$, $1 \leq i \leq n-2$. A numbering with this property will be called an M-numbering. Then for any graph $G \in M$, $d_i \geq |S_i| \geq 2$, for $1 \leq i \leq n-2$, and $d_{n-1} \geq 1$ and so in this case

$$H_n \leq 1 + \sum_{k=1}^{n-2} \frac{1}{2^k} < 2$$

and hence the following theorem.

THEOREM 5.2.

Algorithm MOD-CHAR is of complexity O(nt) for an n-vertex graph G whose vertices can be numbered as in Char's algorithm such that each vertex i, $1 \leq i \leq n-2$, is adjacent to at least two vertices greater than i. □

Since a complete graph is in M for any arbitrary numbering of its vertices, it follows that for a complete graph algorithm MOD-CHAR requires O(nt) time. However, in this case we can prove more interesting results. Since for an n-vertex complete graph deg(k) = n-1 for all k, we get

from Theorem 3.3

$$t+t_0 = 1 + (n-2)tH_n.$$

Since $t+t_0 < 2t$ for a complete graph (see Section 4.2.1), it follows that

$$H_n < \frac{2}{n-2}. \qquad (5.1)$$

From (5.1) and Theorem 5.1, we can see that algorithm MOD-CHAR has O(t) time complexity for a complete graph.

Now we determine an upper bound for the number of computational steps required by algorithm MOD-CHAR to generate a spanning tree of a complete graph. Note that each Type 1 computation involves setting DIGIT(k) = REF(k) for some k and hence one assignment operation. Thus the total number of assignments for all the Type 1 computations is $H_n t - (n-1)$. For a given k, to find the label values we require $n + 2(k-1) \leq 3n-4$ computational steps. Thus the label computations require at most $(3n-4)H_n t$ computational steps. Moreover, $2t+t_0 < 3t$ computational steps are required to generate and test all the sequences for a complete graph. Thus at most $3t + (3n-3)H_n t$ computational steps are required by algorithm MOD-CHAR for an n-vertex complete graph. From this observation and (5.1) we can show that algorithm MOD-CHAR requires, on the average, at most

$$9 + \frac{6}{n-2}$$

computational steps to generate a spanning tree of an n-vertex complete graph. Thus we get the following.

THEOREM 5.3.

Algorithm MOD-CHAR requires, on the average, at most 10 computational steps to generate a spanning tree of a complete graph having more than 8 vertices. □

Consider next the class of all n-vertex biconnected graphs which have maximum degree n-1. Recall that a vertex with degree n-1 is called a star vertex. Let G be any graph in this class and S be a star tree of G. Assigning number n to the star vertex in S and the number n-1 to any other vertex of S, we can obtain an M-numbering of S. If this were not possible, then there would exist a subset $X = \{x_1, x_2, \ldots, x_k\}$ of vertices such that vertices n and n-1 are not in X and each x_i is adjacent to exactly one vertex (namely, the vertex n) not in X. But then, in such a case, the vertex n would be a cut vertex of G, contradicting that G is biconnected. From this and Theorem 5.2 we get the following.

THEOREM 5.4.

For an n-vertex biconnected graph with maximum degree

n-1, algorithm MOD-CHAR is of complexity O(nt). □

Finally, if an n-vertex biconnected graph G has a $(1+\log_2 n)$-vertex connected subgraph G' which permits an M-numbering of the vertices of G', then for the graph G

$$nH_n = n \sum_{k=1}^{n-1} \frac{1}{d_{n-1}d_{n-2}\cdots d_k}$$

$$< n \sum_{k=0}^{\log_2 n} \frac{1}{2^k} + n \sum_{k=1}^{n-2-\log_2 n} \frac{1}{n2^k}$$

$$< 2n + 1$$

$$< 3n$$

Thus we get the following theorem.

THEOREM 5.5.

Algorithm MOD-CHAR is of complexity O(nt) in the case of an n-vertex biconnected graph G, if G has a $(1+\log_2 n)$-vertex connected subgraph which permits an M-numbering. □

5.3 Computational Experiences

The complexity analysis of algorithm MOD-CHAR presented

in Section 5.2 brings out the fact that this algorithm is of time complexity O(nt) for certain classes of n-vertex graphs. In this section we present our computational experiences on algorithm MOD-CHAR

In Table 5.1 we give the execution times required by Char's algorithm, algorithm MOD-CHAR and Gabow and Myers' algorithm when applied on the ten randomly generated graphs listed in Table 3.1. All the algorithms are implemented in PASCAL and the execution times are for a CDC Cyber 170. In the case of Char's algorithm and algorithm MOD-CHAR, the initial spanning tree has been chosen by performing a breadth-first search.

From Table 5.1 we can see the following.

(i) Even though algorithm MOD-CHAR has a better asymptotic complexity than Char's algorithm, it requires about twice as much execution time as Char's. This is due to the additional computations required to compute the label values in algorithm MOD-CHAR.

(ii) Char's algorithm seems to be the fastest of the three algorithms. In fact Table 5.1 shows that Char's algorithm takes less than one-tenth of the time required by Gabow and Myers' algorithm. This is perhaps due to the simplicity of the algorithm. The only operations required in Char's algorithm are assignments and comparisons and this algorithm does not require any complicated data structure mani-

Table 5.1

Execution Time

Graph	Vertices	Spanning trees	Execution Time in Seconds		
			CHAR	MOD-CHAR	Gabow and Myers
G_1	9	24672	2.037	3.812	27.575
G_2	10	13931	0.970	1.938	17.333
G_3	10	151662	12.495	26.189	208.422
G_4	11	151719	10.661	23.144	225.576
G_5	11	1360710	102.660	188.759	1458.505
G_6	11	12897990	994.735	1958.547	NA
G_7	12	1592512	113.124	242.613	2490.788
G_8	12	1820488	129.747	247.238	2618.466
G_9	12	14689650	1193.974	2049.267	NA
G_{10}	13	26520950	2264.815	NA	NA

NOTE: NA means that the execution time is more than 3000 seconds and is not available.

pulations.

The computational experiences and the complexity analysis presented in this chapter lead us to believe that Char's algorithm is the fastest algorithm reported so far to enumerate all the spanning trees of a graph. To conclusively establish this, further study of the number H_n is required.

CHAPTER 6

A COMPARATIVE EVALUATION OF CHAR'S ALGORITHM

Complexity analysis of Char's algorithm presented in Section 3.2 has shown that this algorithm requires $O(n^3t)$ time. In Chapter 5 we developed an efficient implementation of Char's algorithm, namely algorithm MOD-CHAR, which needs only $O(n^2t)$ time. Even though algorithm MOD-CHAR has a better asymptotic complexity than Char's, the computational results presented in Section 5.3 seem to imply that the latter algorithm is twice as fast as the former. Moreover, Table 5.1 suggests that Char's algorithm might be the fastest of all the spanning tree enumeration algorithms reported so far.

Although the execution times required by different algorithms help us compare their relative efficiencies, this alone may not provide an accurate measure of the efficiencies. This is because the execution time of an implemented algorithm depends on many factors which have little or nothing to do with the algorithm proper. These factors include the programmer, the programming language used and the computer on which the program is run, the implementation, and the data structures used in the implementation.

To obtain an evaluation which is independent of these factors, we may first determine the basic operations required by the concerned algorithms and then determine the numbers of these basic operations performed during the execution of these algorithms. As suggested by Chase [8], we may assign weights to these basic operations so that the costs computed using this approach reflect the efficiencies of these algorithms more accurately.

Using the above approach we present in this chapter a computational evaluation of Char's algorithm when compared with algorithm MOD-CHAR, and Gabow and Myers' algorithm. In Section 6.1 we identify the basic operations performed by these algorithms. In Section 6.2 we present our experimental results and make a few comments on the efficiencies of these algorithms.

6.1 Basic Operations of the Algorithms

In this section we identify the basic operations performed by Char's algorithm, algorithm MOD-CHAR, and Gabow and Myers' algorithm. In the following we will not consider the computations required to output the spanning trees.

From our discussions in Section 3.1, it is easy to see that Char's algorithm uses the adjacency lists of the graph

and the sequences (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) and (REF(1), REF(2), ..., REF(n-1)) only. Recall that DIGIT(i), $1 \leq i \leq n-1$, in fact corresponds to the edge (i,DIGIT(i)) and REF(i), $1 \leq i \leq n-1$, corresponds to the edge (i,REF(i)). Thus Char's algorithm can be considered as using only the edges of the graph during its execution. So we consider edge access as a basic operation performed by Char's algorithm. Note that the test for tree compatibility property basically requires traversing the edges in the subgraph. Also determining the initial spanning tree requires traversing the edges of the graph. Thus the algorithm requires only edge accesses. Now we present a version of Char's algorithm in which the different statements involving edge accesses are identified. Note that this version of the algorithm is the same as that presented in Section 3.1.

Char's algorithm to Enumerate All the Spanning Trees of a Graph.

```
procedure CHAR;
comment procedure CHAR enumerates all the spanning trees of
        a connected n-vertex graph G represented by the
        adjacency lists of its vertices.
begin
  select an initial spanning tree of G;
  perform a depth-first search or a breadth-first search on
  the initial spanning tree and renumber the vertices as n,
```

```
n-1, ..., 1 in the order in which they are visited during
the search;
find FATHER(i), 1 ≤ i ≤ n-1;
{All the above operations can be performed during a single
search. They involve edge accesses}
for i := 1 to n-1 do
  begin
    REF(i) := FATHER(i);
    DIGIT(i) := REF(i)
                                        {2(n-1) edge accesses}
  end;
output the initial tree sequence (REF(1), REF(2), ...,
REF(n-1));
i := n-1;
while i ≠ 0 do
  begin
    if SUCC(DIGIT(i)) ≠ nil
                                        {one edge access}
      then begin
        DIGIT(i) := SUCC(DIGIT(i));
                                        {one edge access}
        if (DIGIT(1), DIGIT(2), ..., DIGIT(n-1)) is a tree
        sequence
                                        {edge accesses}
          then begin
            output the tree sequence;
            i := n-1
```

```
            end
        end
        else begin
            DIGIT(i) := REF(i);
                                        {one edge access}
            i := i-1
        end
    end
end CHAR;
```

Next we consider algorithm MOD-CHAR. Note that this algorithm involves the same basic operations as Char's algorithm except for the computation of the label values. Since to compute the label values, we traverse the paths in a spanning 2-tree (see Section 5.1), again only edge accesses are required for this computation. Thus in the case of algorithm MOD-CHAR also we identify edge accesses as the basic operations performed. In the following we present algorithm MOD-CHAR in which the different edge accesses are clearly identified.

Modified Char's Algorithm to Enumerate All the Spanning Trees of a Graph.

```
procedure MOD-CHAR;
comment procedure MOD-CHAR enumerates all the spanning trees
        of a connected n-vertex graph G using algorithm
        MOD-CHAR. The graph is represented by the adjacency
```

lists ADJ(i), $1 \leq i \leq n-1$, of its vertices.

```
procedure GENERATE(k);
comment procedure GENERATE, when called with the argument
        k, sets DIGIT(k) to generate a tree sequence.
        This procedure uses a local array LABEL.
begin
  if k = n
    then output the tree sequence
    else begin
      {Set DIGIT(i) to REF(i), k ≤ i ≤ n-1}
      DIGIT(i) := REF(i);
                                          {one edge access}
      GENERATE(k+1);
      {Generate all the sequences in T_k ∪ T_k^r having the
      same DIGIT(1), DIGIT(2), ..., DIGIT(k-1)}
      compute LABEL(x) for each neighbour x of k;
                                          {edge accesses}
      for x ∈ ADJ(k)-REF(k) do
        if LABEL(x) = n
                                          {one edge access}
          then begin
            DIGIT(k) := x;
                                          {one edge access}
            GENERATE(k+1)
          end
    end
end GENERATE;
```

```
begin
  find the initial tree sequence (REF(1), REF(2), ...,
  REF(n-1));
  renumber the vertices of G;
  {The above operations can be performed during the same
  search. They involve edges accesses}
  GENERATE(1)
end MOD-CHAR;
```

Finally, we consider Gabow and Myers' algorithm to generate all the spanning trees of a graph [12]. Since we have not presented this algorithm so far, a discussion of this algorithm is now in order. As we have stated in Chapter 2, Gabow and Myers' algorithm is based on the following principle. If e is an edge of a graph, then the spanning trees of G can be classified into those which contain e and those which do not contain e.

Thus Gabow and Myers' approach involves finding recursively all the spanning trees of the graph G containing a subtree T (which is a single vertex to start with). To do this, they choose an edge e_1 connecting a vertex in T and a vertex not in T; find all the spanning trees containing $T \cup e_1$; then delete e_1 from the graph. Next choose an edge e_2 connecting T to a vertex not in T; find all the spanning trees (in the modified graph) containing $T \cup e_2$; then delete

e_2. To continue, they repeatedly choose an edge e_i connecting T to a vertex not in T; find all the spanning trees (in the modified graph) containing $T \cup e_i$; then delete e_i. This process is stopped when the edge e_k that has just been processed is a bridge of the modified graph. At this point each spanning tree containing T has been found exactly once, because if a spanning tree does not contain any e_j, $j < k$, it must contain the bridge e_k.

In order to detect, in the above procedure, the edge e_k which is a bridge, Gabow and Myers grow the tree T depth-first. Suppose all the spanning trees containing $T \cup e$ have been found, and we want to check if e is a bridge. Let L be the last spanning tree found that contains $T \cup e$, and let $e = (u,v)$. It has been shown in [12] that edge e is a bridge when no edge (besides e) goes from a nondescendant of v (in L) to v. It can be easily seen that all the above operations involve only edge accesses.

To grow T depth-first, Gabow and Myers' algorithm uses F, a list of all edges connecting vertices in T to vertices not in T. Besides F, the algorithm uses lists FF. Each recursive invocation has a local FF list. It is used to reconstruct the original F list. Manipulating these two lists involves stack operations. Thus Gabow and Myers' algorithm requires <u>list accesses</u> to maintain the lists in addition to the edge accesses. Now we present the

algorithm, in ALGOL-like notation, in which we identify the different list accesses as well as the edge accesses.

Gabow and Myers' Algorithm to Enumerate All the Spanning Trees of a Graph.

```
procedure GABOW_MYERS;
comment procedure GABOW_MYERS finds all the spanning trees
        of a connected n-vertex graph G.

  procedure GROW;
  comment procedure GROW finds all the spanning trees
          containing T.
  begin
    if T has n vertices
      then begin
        L ← T;
                                          {n-1 edge accesses}
        output L
      end
      else begin
        make FF an empty list local to GROW;
        repeat
          pop an edge e from F; let e go from T to a vertex
          v not in T;
                                          {one list access}
          add e to T;
                                          {one edge access}
          push each edge (v,w), w ∉ T, onto F;
```

```
                                {edge accesses and list accesses}
    remove each edge (w,v), w ∈ T, from F;
                                {edge accesses and list accesses}
    GROW;
    pop each edge (v,w), w ∉ T, from F;
                                {edge accesses and list accesses}
    restore each edge (w,v), w ∈ T, in F;
                                {edge accesses and list accesses}
    remove e from T and from G;
                                                {two edge accesses}
    add e to FF;
                                                {one list access}
    if there is an edge (w,v), where w is not a
    descendant of v in L
                                                {edge accesses}
      then bridge ← false
      else bridge ← true
  until bridge;
  pop each edge e from FF;
                                                {list accesses}
  push e onto F;
                                                {list accesses}
  add e to G
                                                {edge accesses}
 end
end GROW;
```

```
begin
  initialize T to contain vertex 1;
  initialize F to contain all the edges (1,v);
                          {edge accesses and list accesses}
  GROW
end GABOW_MYERS;
```

6.2 The Computational Evaluation

From our discussions in the previous section, it is clear that while Char's algorithm and algorithm MOD-CHAR require only edge accesses as their basic operations, Gabow and Myers' algorithm involves both edge accesses and list accesses. Thus the total computational work required by the last algorithm is the sum of the edge accesses and list accesses. In this section we present our experimental results on the total computational effort required by these algorithms.

Gabow and Myers [12] discuss an efficient implementation of their algorithm in which the list F is managed as a doubly linked list. We have implemeted this algorithm as suggested by them. In Table 6.2 we show the average number of computational steps required by the three algorithms to generate a spanning tree when applied on several test graphs. The number of vertices, the number of

edges and the number of spanning trees of these test graphs are shown in Table 6.1.

Table 6.2 substantiates our observation in Chapter 5 that Char's algorithm might be the fastest. We can see that in most cases this algorithm requires about one-fifth as much computational effort as Gabow and Myers' algorithm. It is interesting to note that Char's algorithm requires comparatively more number of computations for the graphs G_{12} and G_{13} which are simple circuits on 10 and 20 vertices respectively. This may be due to the fact that Char's algorithm generates O(nt) non-tree subgraphs when applied on an n-vertex circuit. However, as can be seen in Table 6.2, Gabow and Myers' algorithm also requires comparatively more number of computations and is inferior to Char's in these cases too.

6.3 Conclusion

Our objective in this part of the thesis has been to study Char's algorithm and evaluate its performance in comparison to Gabow and Myers'. Our analysis has shown that this algorithm can be implemented with complexity $O(nH_n t)$, which is $O(n^2 t)$ in the worst case. Note that Gabow and Myers' algorithm has complexity O(nt). However, we believe that this poor complexity of Char's algorithm in relation to

Table 6.1

Test Graphs

Graph	Number of Vertices	Number of Edges	Number of Spanning Trees
G_1	8	14	497
G_2	8	17	3465
G_3	8	20	16968
G_4	8	23	49392
G_5	8	25	100352
G_6	8	28	262144
G_7	11	30	1360710
G_8	15	25	15764
G_9	15	30	921456
G_{10}	20	30	66448
G_{11}	25	35	34368
G_{12}	10	10	10
G_{13}	20	20	20

Table 6.2

Average Number of Computational Steps

Graph	Number of Spanning Trees	Average Number of Computational Steps per Spanning Tree		
		CHAR	MOD-CHAR	Gabow and Myers
G_1	497	5.5	11.0	27.0
G_2	3465	5.3	11.0	24.0
G_3	16968	5.7	10.0	24.0
G_4	49392	5.4	8.7	22.0
G_5	100352	5.5	8.9	22.0
G_6	262144	6.0	11.0	23.0
G_7	1360710	5.3	11.0	24.0
G_8	15764	6.9	12.0	37.0
G_9	921456	5.0	8.9	32.0
G_{10}	66448	9.4	13.0	49.0
G_{11}	34368	6.8	13.9	64.0
G_{12}	10	22.0	75.0	60.0
G_{13}	20	47.0	250.0	120.0

Gabow and Myers' is mainly due to our inability to obtain a bound for H_n tighter than the one, namely $H_n \leq n$, which we have used. The extreme simplicity of Char's algorithm along with the theoretical and experimental results presented in this part of the thesis suggest that this algorithm might be superior to all the other spanning tree enumeration algorithms. So we conclude this part of the thesis with the conjecture that Char's algorithm implemented with one of our heuristics to select the initial spanning tree and path compression to reduce the number of comparisons made is the best of all the spanning tree enumeration algorithms reported so far. To prove this conjecture, further study of H_n is required.

PART II

PLANAR EMBEDDING AND MAXIMAL PLANARIZATION

CHAPTER 7

PLANARITY TESTING AND PQ-TREES

A graph G is planar if there exists a one-to-one mapping of its vertices and edges into the plane such that

(i) each vertex v is mapped into a distinct point in the plane;

(ii) each edge (v,w) is mapped onto a simple curve with the vertices v and w mapped onto the endpoints of the curve, and

(iii) the mappings of distinct edges have in common only the mappings of their common end vertices.

A mapping of G which satisfies the above conditions is called a planar embedding of G. Testing a graph for planarity and embedding a planar graph in the plane have several applications. For example, the design of integrated circuits and the layout of printed circuit boards require testing whether a circuit can be embedded in the plane without crossovers. Determining isomorphism of chemical structures is simplified if the structures are known to be planar [23], [24]. A maximum cut in a graph can be determined efficiently if the graph is planar [25], whereas the problem is NP-complete for an arbitrary graph [26].

7.1 Planarity Testing Algorithms

Because of its great practical interest, the problem of testing planarity of a graph has been widely studied. The earliest characterization of planar graphs was given by Kuratowski [27]. He proved that a graph is planar if and only if it does not contain a subgraph which, upon removal of degree two vertices, is isomorphic either to K_5, the complete graph on five vertices, or to $K_{3,3}$, the complete bipartite graph on six vertices. Mei and Gibbs [28] have given an algorithm, based on Kuratowski's characterization, to test a graph for planarity. Their algorithm first finds all circuits of length five or greater in the graph. It then processes two circuits at a time and checks whether the union of the two circuits is one of Kuratowski's "forbidden" subgraphs. This algorithm, however, is not efficient. In fact, Kuratowski's characterization, although mathematically elegant, is not useful as a practical test for planarity, because testing for the presence of Kuratowski's subgraphs may require an amount of time proportional to at least n^3, where n is the number of vertices in the graph.

Another characterization of planar graphs is due to Whitney [29] who proved that a graph is planar if and only if it has a dual. Later, MacLane [30] showed that a graph is planar if and only if it contains a set of fundamental circuits such that no edge appears in more than two of these

circuits. However, these characterizations also have not yielded any efficient algorithm to test a graph for planarity.

The most successfull approach so far for testing the planarity of a graph seems to be an attempt to construct a representation of a planar embedding of the graph. If such a representation can be obtained, then the graph is planar; if not, the graph is nonplanar. All the planarity testing algorithms based on this idea can be grouped into two categories as follows,

(i) Path Addition Algorithms: The algorithms in this category first find a cycle in the graph. When this cycle is removed, the remaining edges of the graph would form several connected components. Each of these components is then embedded in the plane along with the original cycle and the embeddings of the components are combined, if possible, to give an embedding of the entire graph.

The first such algorithm was proposed by Auslander and Parter [31]. This algorithm embeds the connected components by calling itself recursively. Unfortunately the algorithm was not correct; the proposed method may loop indefinitely. Goldstein [32] correctly formulated Auslander and Parter's algorithm using iteration instead of recursion. Shirey [33], in his thesis, gave an implementation of

Goldstein's algorithm, using a list structure representation of graphs. He also proved that his implementation has an $O(n^3)$ time bound. Later, Hopcroft and Tarjan [34] devised a variant of Goldstein's algorithm with a time bound of O(nlogn) using depth-first search.

In 1974 Hopcroft and Tarjan [35] proposed a linear time algorithm for testing planarity of a graph. This algorithm finds a cycle in the graph using depth-first search. This cycle is then embedded in the plane, thereby dividing the plane into two faces - one inside the cycle and the other outside the cycle. The connected components of the graph, obtained after removing the cycle, are then successively embedded either in the inside face or in the outside face. During the embedding of any component, if necessary, all the components which are already embedded in the inside face may be moved to the outside face, and all those already embedded in the outside face may be moved to the inside face. All these rearrangements are done in an efficient manner without actually drawing the embedding, so that the algorithm has O(n) time bound. An excellent exposition of Hopcroft and Tarjan's algorithm may be found in [36].

Earlier, Demoucron, Malgrange and Pertuiset [37] had given an algorithm similar to Hopcroft and Tarjan's, which avoids the rearrangement of already embedded components by choosing the components, for embedding at each stage, in an

appropriate manner. Rubin [38] developed an $O(n^2)$ space and $O(n^2)$ time implementation of this algorithm and showed that, for all practical purposes, his implementation behaves as an $O(n)$ algorithm. It is interesting to note that Rubin's implementation is, on the average, about twice as fast as Hopcroft and Tarjan's implementation of their algorithm.

A novel path addition algorithm for testing the planarity of a graph was proposed by Fisher and Wing [39]. This algorithm works directly on the incidence matrix of the graph. If the graph is nonplanar, this algorithm systematically identifies a set of edges whose deletion yields a subgraph that is planar. However, this algorithm is not computationally efficient, nor any algorithm which uses the incidence matrix.

(ii) Vertex Addition Algorithms: The algorithms in this category use an alternate approach to embed a graph in the plane. These algorithms start with a single embedded vertex and add all the edges incident on that vertex. The other end vertices of these edges are not embedded. They then embed an unembedded vertex and add all edges incident on it in the same way. This process of embedding is continued until the entire graph is constructed. For these algorithms to work correctly, the vertices must be embedded in a special order.

Hopcroft and Tarjan [35] refer to one such algorithm due to Mondshein [40] which requires $O(n^2)$ time. Another vertex addition algorithm was proposed by Lempel, Even, and Cederbaum [41]. In this algorithm, at any time during the embedding process, the subgraph embedded upto that time is represented by certain formulas which are then manipulated, by applying certain transformations, to check whether the next vertex can be embedded. Lempel, Even, and Cederbaum did not give any implementation or time bound for their algorithm; however, an implementation of this algorithm due to Tarjan, requiring $O(n)$ space and $O(n^2)$ time is referred to in [35]. The best implementation of Lempel, Even, and Cederbaum's algorithm was reported by Booth and Lueker [42]. They developed a data structure called PQ-tree to represent the permutations of a set in which elements of certain subsets of the given set are required to occur consecutively, and presented efficient algorithms to manipulate the PQ-tree. Using PQ-trees, Booth and Lueker developed an $O(n)$ time implementation of Lempel, Even, and Cederbaum's algorithm.

An interesting algorithm to test the planarity of a graph, which does not fall into any of the above two categories of algorithms, was proposed by Bruno, Steiglitz, and Weinberg [43]. Their algorithm is based on some of Tutte's results on triconnected graphs. Instead of embedding a graph in the plane, they reduce it to simpler

and simpler graphs until a wheel is obtained. Then the original graph is reconstructed from the wheel. During this reconstruction, a planar embedding of the graph is obtained if the graph is planar. Although they gave no explicit time bound, their algorithm does not compare favorably with those mentioned above.

In this part of the thesis, we develop, using Lempel, Even, and Cederbaum's algorithm along with PQ-trees, efficient algorithms for the planar embedding and maximal planarization problems. In order to make the thesis self-contained, we present a discussion of Lempel, Even, and Cederbaum's algorithm in the following section. In Section 7.3, we describe PQ-tree and explain how the use of PQ-trees leads to an efficient implementation of Lempel, Even, and Cederbaum's algorithm.

7.2 Lempel, Even, and Cederbaum's Planarity Testing Algorithm

In this section we discuss the vertex addition algorithm due to Lempel, Even, and Cederbaum to test the planarity of a graph. Hereafter we refer to this algorithm as the LEC algorithm. Since a graph is planar if and only if its biconnected components are planar, we consider only simple biconnected graphs. A complete discussion of this

algorithm may be found in [44].

Let $G = (V,E)$ be a simple biconnected graph with $n = |V|$ vertices and $m = |E|$ edges. For any edge (s,t) of G, Lempel, Even, and Cederbaum define an st-numbering of G as a one-to-one function $g: V \rightarrow \{1, 2, \ldots, n\}$ satisfying the following conditions:

(i) $g(s) = 1$,

(ii) $g(t) = n$,

(iii) for every vertex $v \in V-\{s, t\}$, there are adjacent vertices u and w such that $g(u) < g(v) < g(w)$.

They also showed that for every biconnected graph G, there exists an st-numbering for any edge (s,t) of G. Their proof suggested an $O(mn)$ time algorithm to compute such an st-numbering. Later, using depth-first search, Even and Tarjan [45] presented an $O(m+n)$ time algorithm to compute an st-numbering. Recently, Ebert [46] presented an algorithm which uses less space and time than Even and Tarjan's.

The LEC algorithm first renumbers the vertices of G as $1, 2, \ldots, n$ using an st-numbering. The vertices of G are thereafter referred to by their st-numbers and they are processed in that order for embedding. The st-numbering is essential for the algorithm to work correctly. The graph G, with its vertices labeled according to an st-numbering is called an st-graph. Clearly the edges $(1,2)$, $(n-1,n)$, and

(1,n) are present in any st-graph G. Furthermore, if each edge is oriented from its lower vertex to its higher vertex, then G may be viewed as a directed graph in which the edges are directed from lower to higher vertices. The following observations follow easily from the definition of st-numbering.

Observation 1: In G, the in-degree of vertex 1 is zero, the out-degree of vertex n is zero, and for every other vertex v, $2 \leq v \leq n-1$, the in-degree and out-degree are nonzero.

Observation 2: For any vertex v, $2 \leq v \leq n$, there exists a path in G from vertex 1 to v such that all the internal vertices on the path are less than v.

The above observations may be verified for the st-graph G shown in Fig. 7.1.

For any st-graph G let G_k, $1 \leq k \leq n$, denote the subgraph of G induced by the vertex set $V_k = \{1, 2, \ldots, k\}$. Thus G_k consists of all those edges of G whose end vertices are both in V_k. Now we define the graph B_k as follows. G_k is a subgraph of B_k. In addition to G_k, B_k contains all the edges of G which emanate from vertices of V_k and enter, in G, vertices of $V-V_k$. These edges are called virtual edges and the vertices they enter in $V-V_k$ are called virtual vertices. These vertices are labeled as their counterparts in G; but kept separate. Thus, in B_k there may be several virtual vertices with the same label, each with

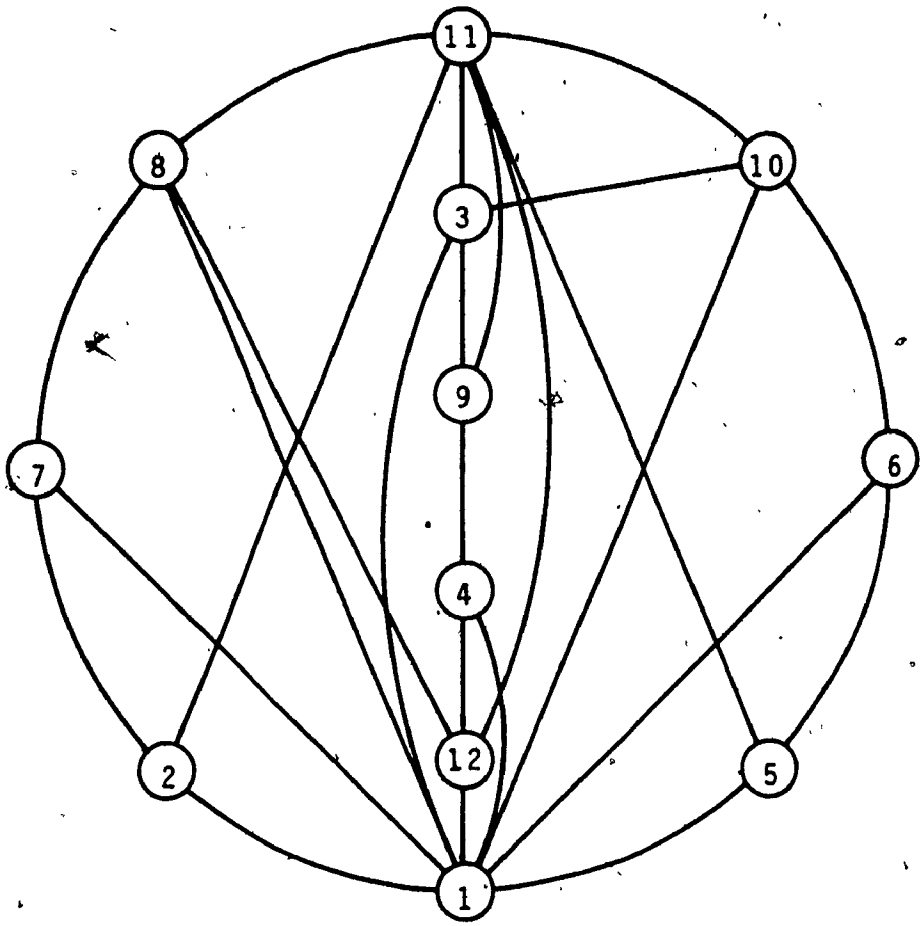


Figure 7.1

st-graph G

exactly one entering edge. For example, Fig. 7.2 shows the graph B_9 of the st-graph shown in Fig. 7.1.

If the st-graph G is planar, then there exists a planar embedding $\hat{G}$ of G. Note that $\hat{G}$ contains a planar embedding $\hat{G}_k$ of G_k, $1 \leq k \leq n$. Using Observation 1, the following lemma can be proved.

LEMMA 7.1.

If $\hat{G}_k$ is a planar embedding of G_k contained in a planar embedding $\hat{G}$ of an st-graph G, then all the edges and vertices of $\hat{G}-\hat{G}_k$ are drawn on one face of $\hat{G}_k$. □

Thus we can assume, without loss of generality, that if G is a planar graph then there exists a planar embedding of B_k in which all the virtual edges are drawn in the outside face. Since the edge (1,n) is a virtual edge in every B_k, $1 \leq k \leq n-1$, it follows that vertex 1 is on the outside face of every G_k. Thus we can draw the graph B_k in the following form. Vertex 1 is drawn at the bottom level. All the virtual vertices appear at the highest level on one horizontal line. The remaining vertices of G_k are drawn in such a way that vertices with higher labels are drawn higher. Such a realization of B_k is called the bush form of B_k. For example, the bush form of the graph B_9 is shown in Fig. 7.3. Since B_k and its bush form are isomorphic, hereafter we shall refer to the bush form of B_k also by B_k.

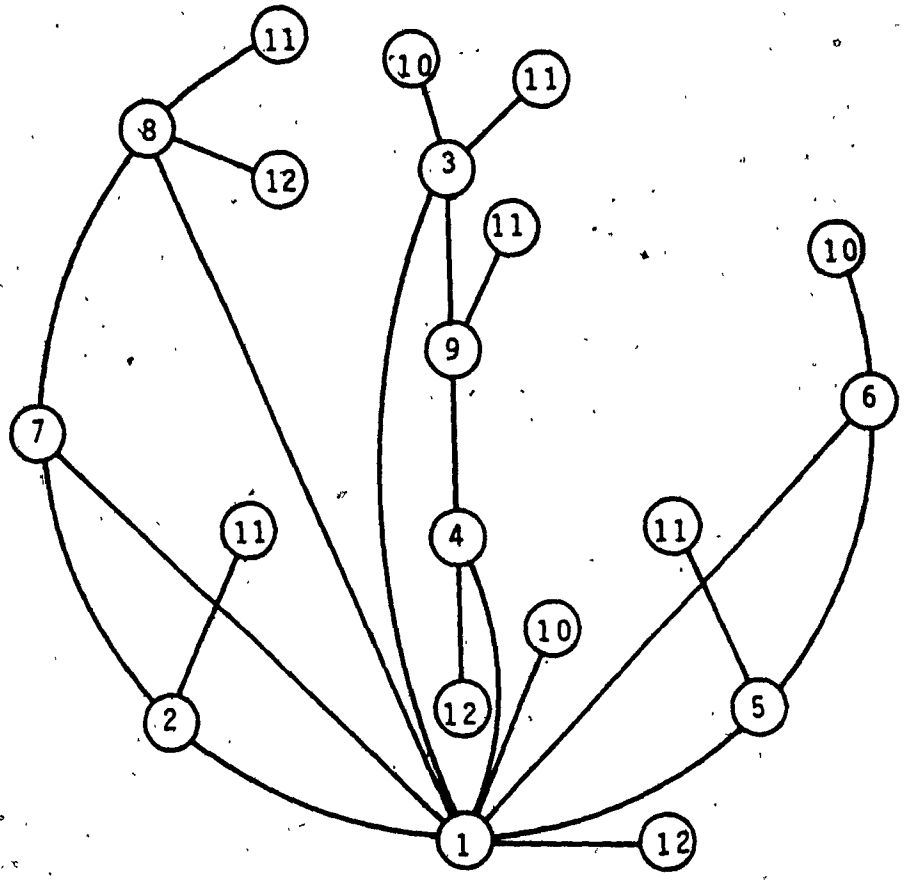


Figure 7.2

Graph B_9

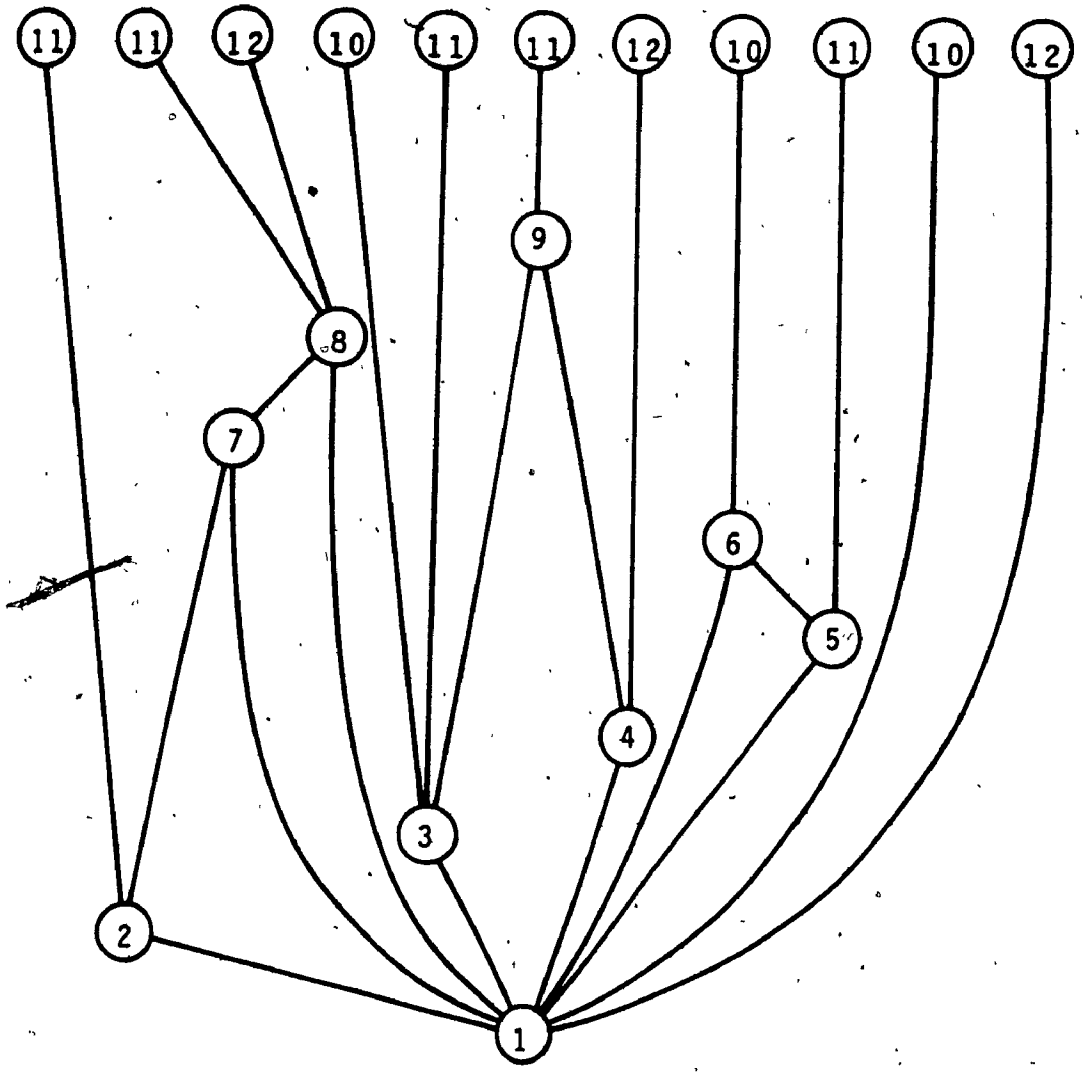


Figure 7.3

Bush Form B_9

Note that in the bush form B_k, the virtual vertices are labeled k+1 or higher, and the st-numbering ensures that there exists at least one virtual vertex labeled k+1. Lempel, Even, and Cederbaum proved that if G is planar, then there exists a bush form of B_k in which all the virtual vertices with labels k+1 appear next to each other on the horizontal line. Let B'_k be such a bush form isomorphic to B_k. For example, the bush form B'_9 corresponding to B_9 is shown in Fig. 7.4. If for a given B_k a corresponding B'_k exists, then the bush form B_{k+1} can be constructed from B'_k as follows. Merge all the virtual vertices labeled k+1 into one vertex and pull it down from the horizontal line. Add all the edges of G which emanate from vertex k+1 as virtual edges. Now vertex k+1 is considered embedded. Thus, if for each B_k, $1 \leq k \leq n-2$, the corresponding B'_k exists, then we can construct the bush forms B_2, B_3, ..., B_{n-1} starting with B_1. Note that $B'_{n-1} = B_{n-1}$ and applying the above procedure to B_{n-1} will give a planar embedding of G. Thus, if for each B_k, $1 \leq k \leq n-2$, the corresponding B'_k exists, then G is planar.

Using the above ideas, Lempel, Even, and Cederbaum formulated a planarity testing algorithm. Their vertex addition algorithm is presented below in ALGOL-like notation.

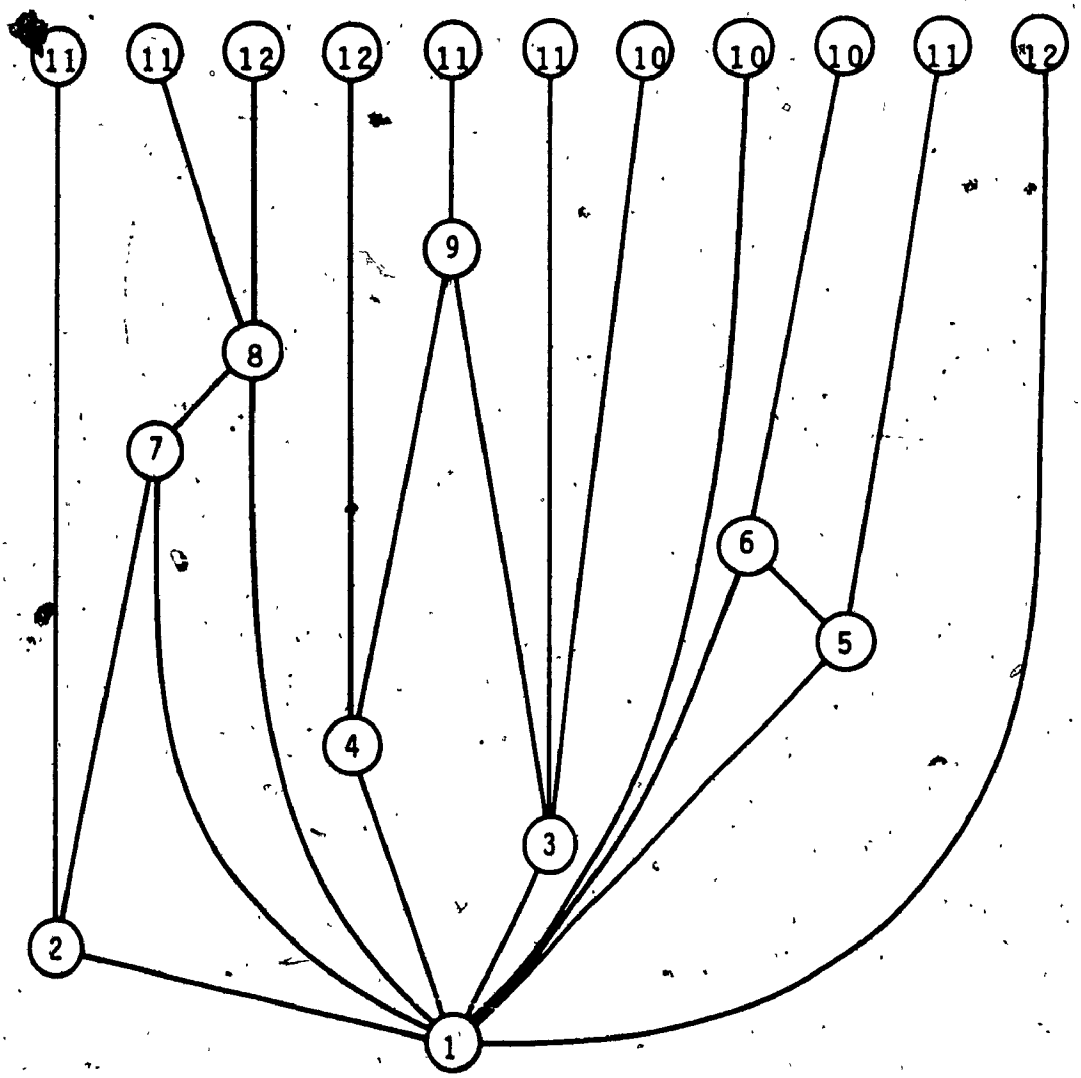


Figure 7.4

Bush Form B_9'

Lempel, Even, and Cederbaum's Vertex Addition Algorithm to Test Planarity of a Graph.

```
boolean function PLANAR(G);
comment function PLANAR tests the planarity of a simple
        biconnected graph G. It returns the value true if G
        is planar; false otherwise;
begin
  find an st-numbering for G;
  renumber the vertices of G according to the st-numbering
  and obtain the st-graph G;
  {Bush form B_1 consists of the vertex 1 and all the edges
  in G incident out of vertex 1 as virtual edges.}
  construct the bush form B_1;
  for k := 1 to n-2 do
    {B'_k is a bush form isomorphic to B_k in which all the
    virtual vertices labeled k+1 appear next to each other.}
    if B'_k exists
      then
        construct B_{k+1} from B'_k
        {B_{k+1} is constructed from B'_k by merging all the
        virtual vertices labeled k+1 into a single vertex
        and adding all the edges in G incident out of vertex
        k+1 as virtual edges.}
      else
        {G is nonplanar.}
        return false
  {G is planar.}
```

```
return true
end PLANAR;
```

We now illustrate the above algorithm on the st-graph G shown in Fig. 7.1. Various bush forms of this st-graph are shown in Figs. 7.5 to 7.15 and a planar embedding of G is shown in Fig. 7.16.

The crucial step in the LEC algorithm is the construction of B_k' from B_k for every $1 \leq k \leq n-2$. Such bush forms would exist if the given graph G is planar. We now state two lemmas which form the basis of an algorithm for constructing B_k' from B_k. The proof of these lemmas use Observation 2 and Lemma 7.1, and may be found in [44].

LEMMA 7.2.

Let v be a cut vertex of B_k. If $v > 1$, then exactly one component of B_k, with respect to v, contains vertices lower than v. □

LEMMA 7.3.

Let H be a maximal biconnected component of B_k and y_1, y_2, ..., y_q be the vertices of H which are also end vertices of the edges of B_k-H. In every bush form isomorphic to B_k, y_1, y_2, ..., y_q are on the outside window of H and in the same order, except that the orientation may be reversed. □

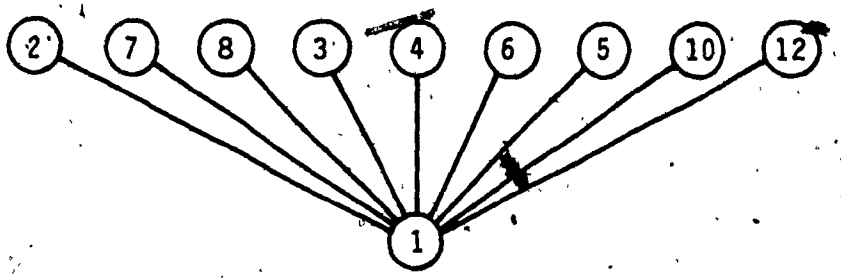


Figure 7.5

Bush Form $B_1 = B_1'$

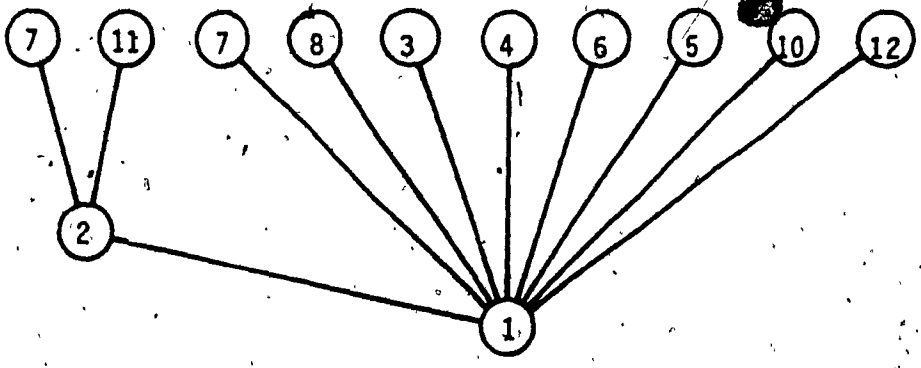


Figure 7.6

Bush Form $B_2 = B_2'$

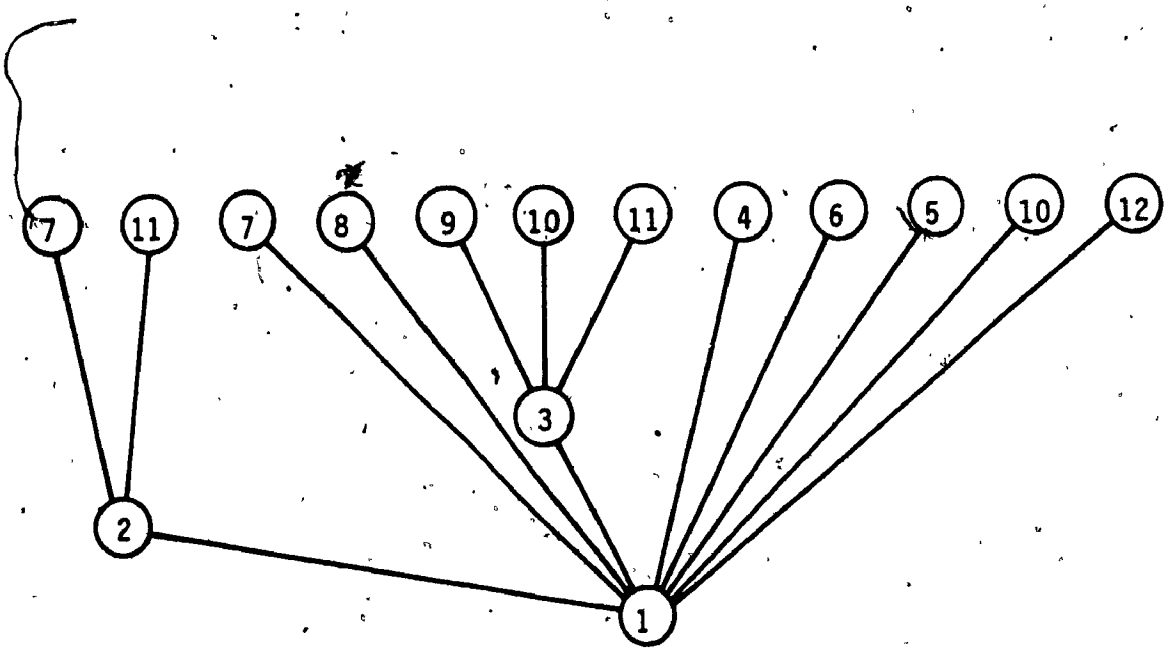


Figure 7.7

Bush Form $B_3 = B'_3$

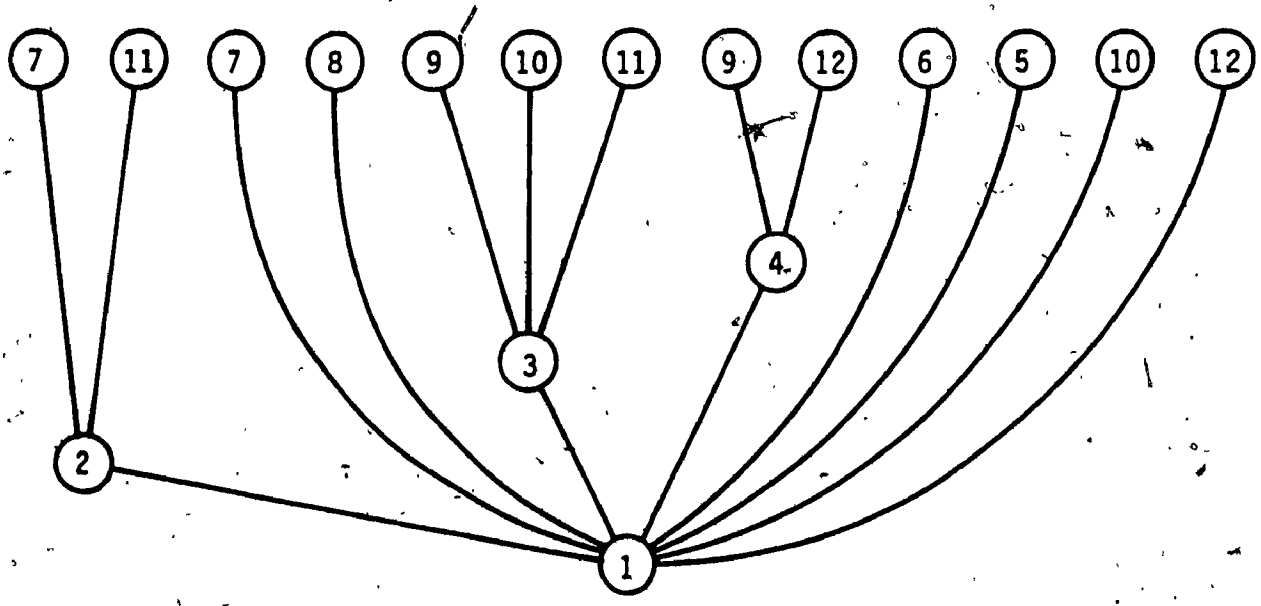


Figure 7.8

Bush Form $B_4 = B'_4$

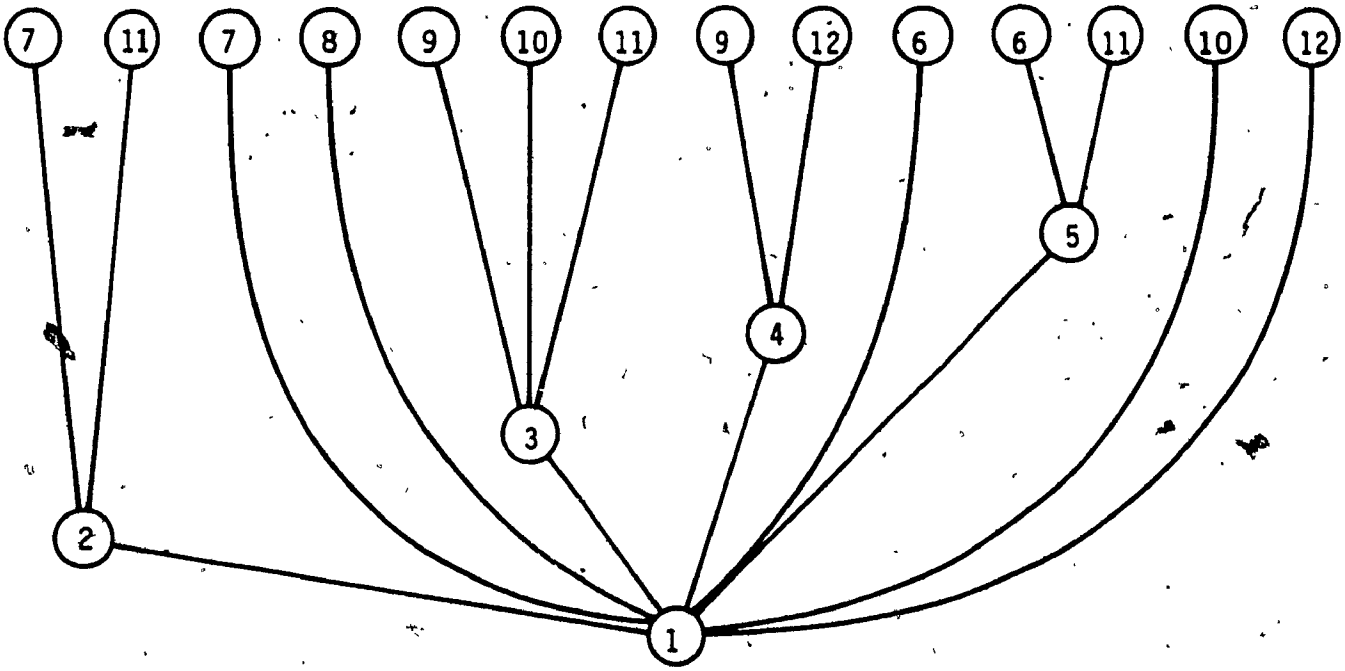


Figure 7.9

Bush Form $B_5 = B_5'$

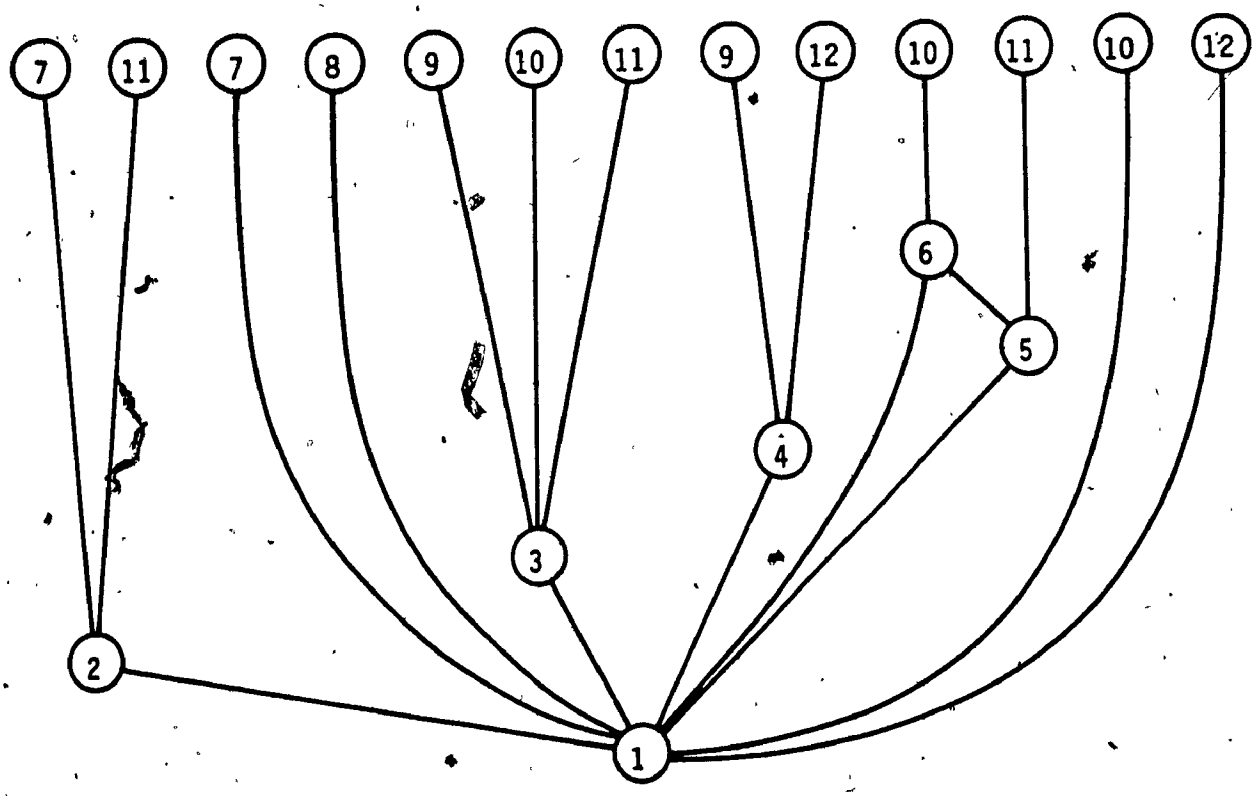


Figure 7.10(a)

Bush Form B_6

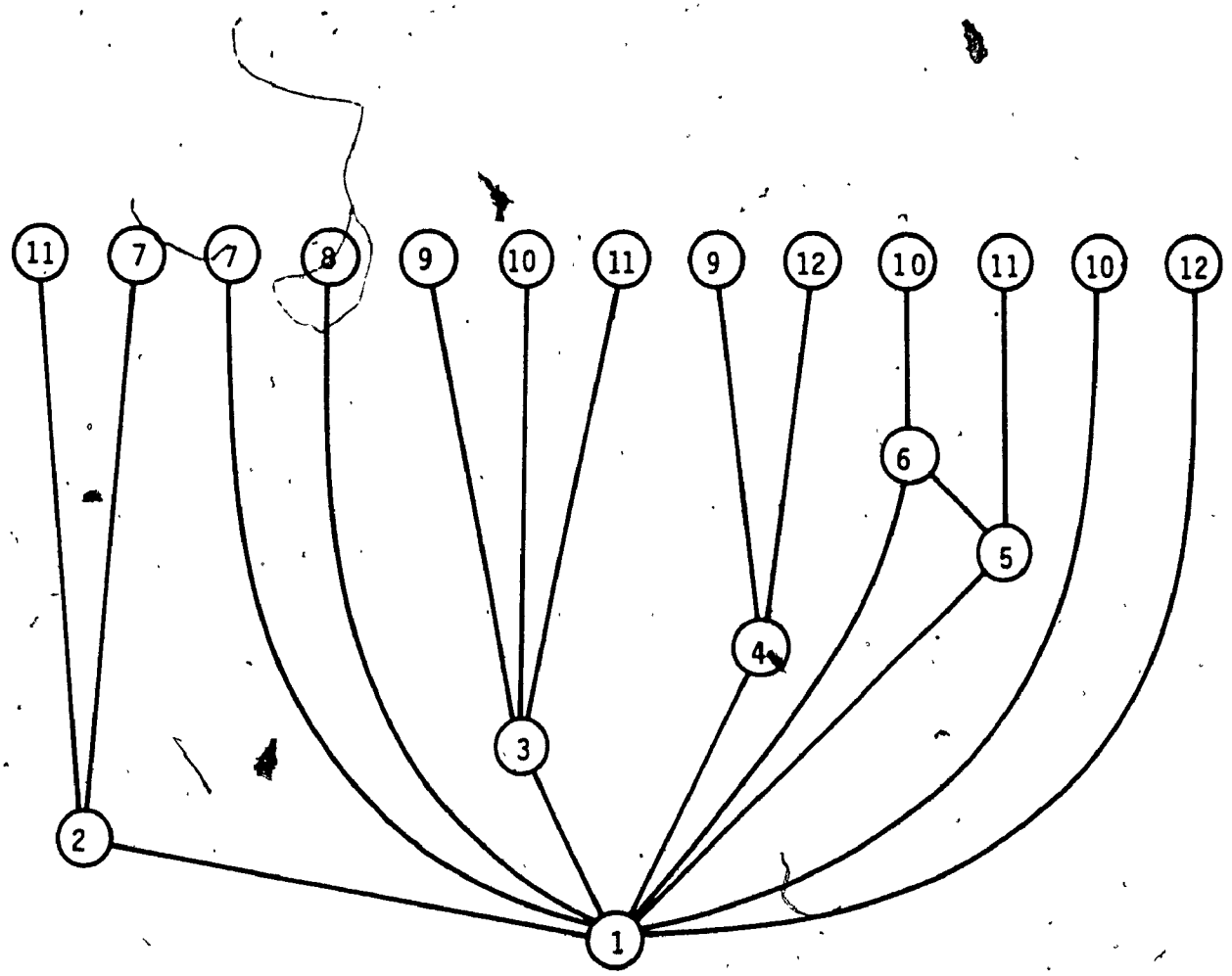


Figure 7.10(b)

Bush Form B_6'

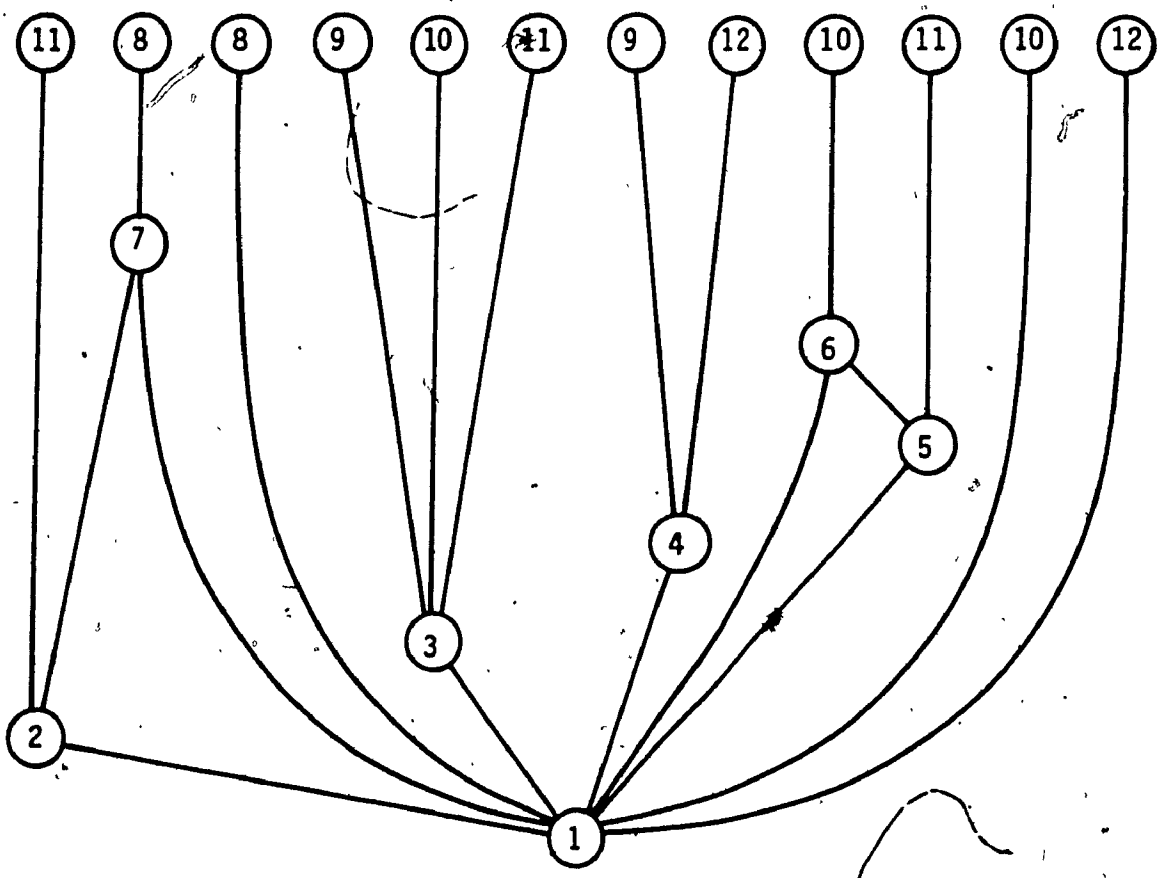


Figure 7.11

Bush Form $B_7 = B_7'$

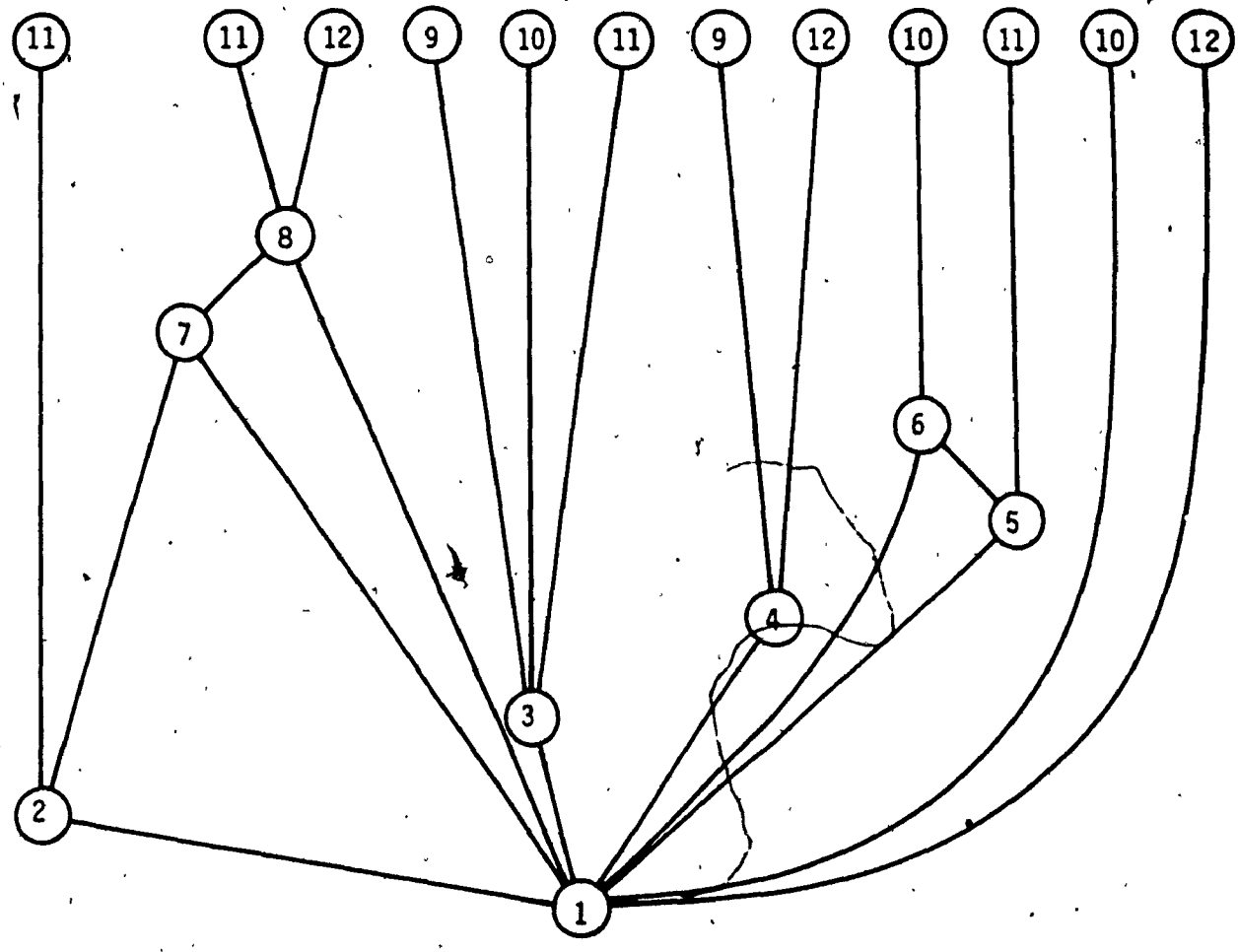


Figure 7.12(a)

Bush Form B_8

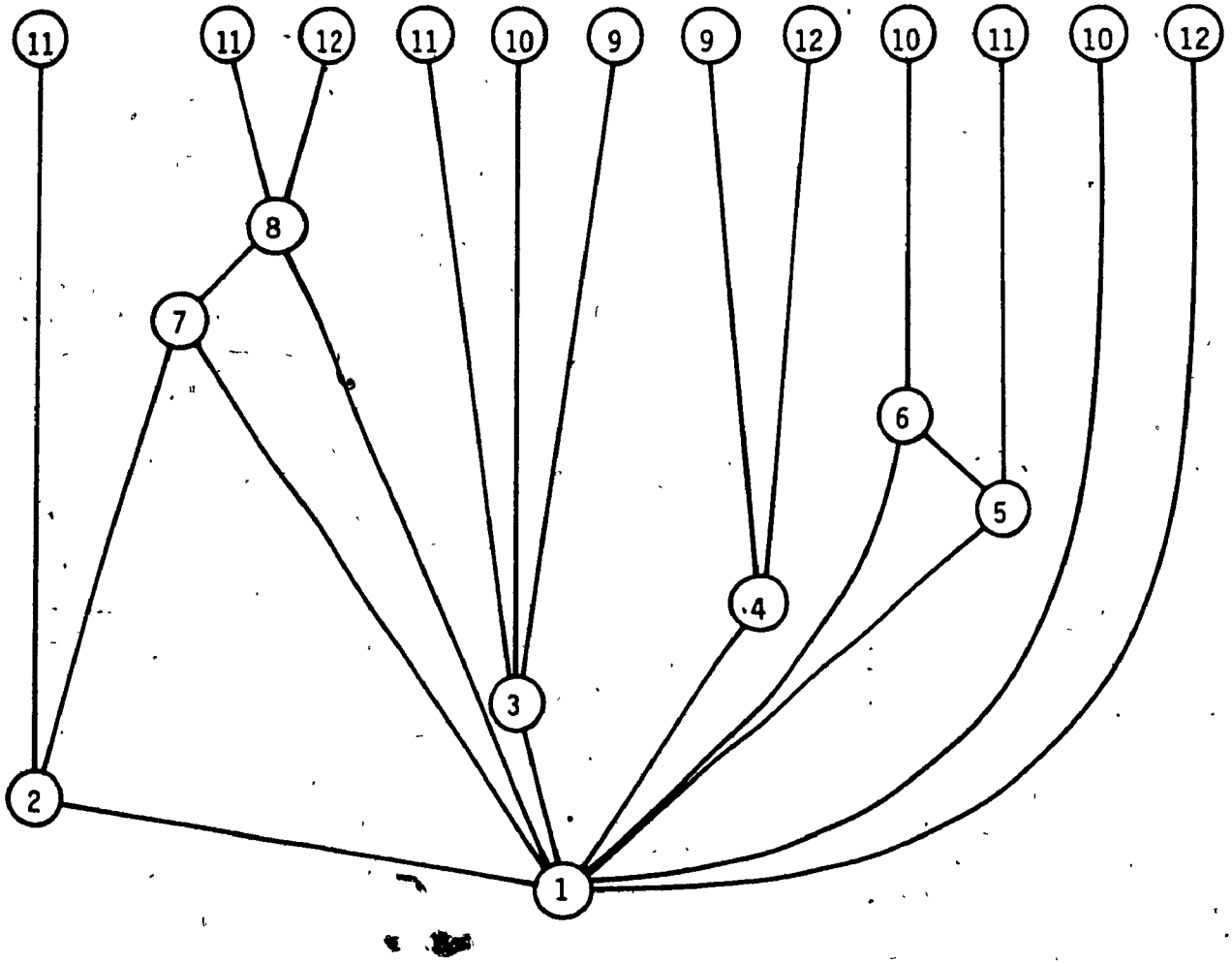


Figure 7.12(b)

Bush Form B_8^1

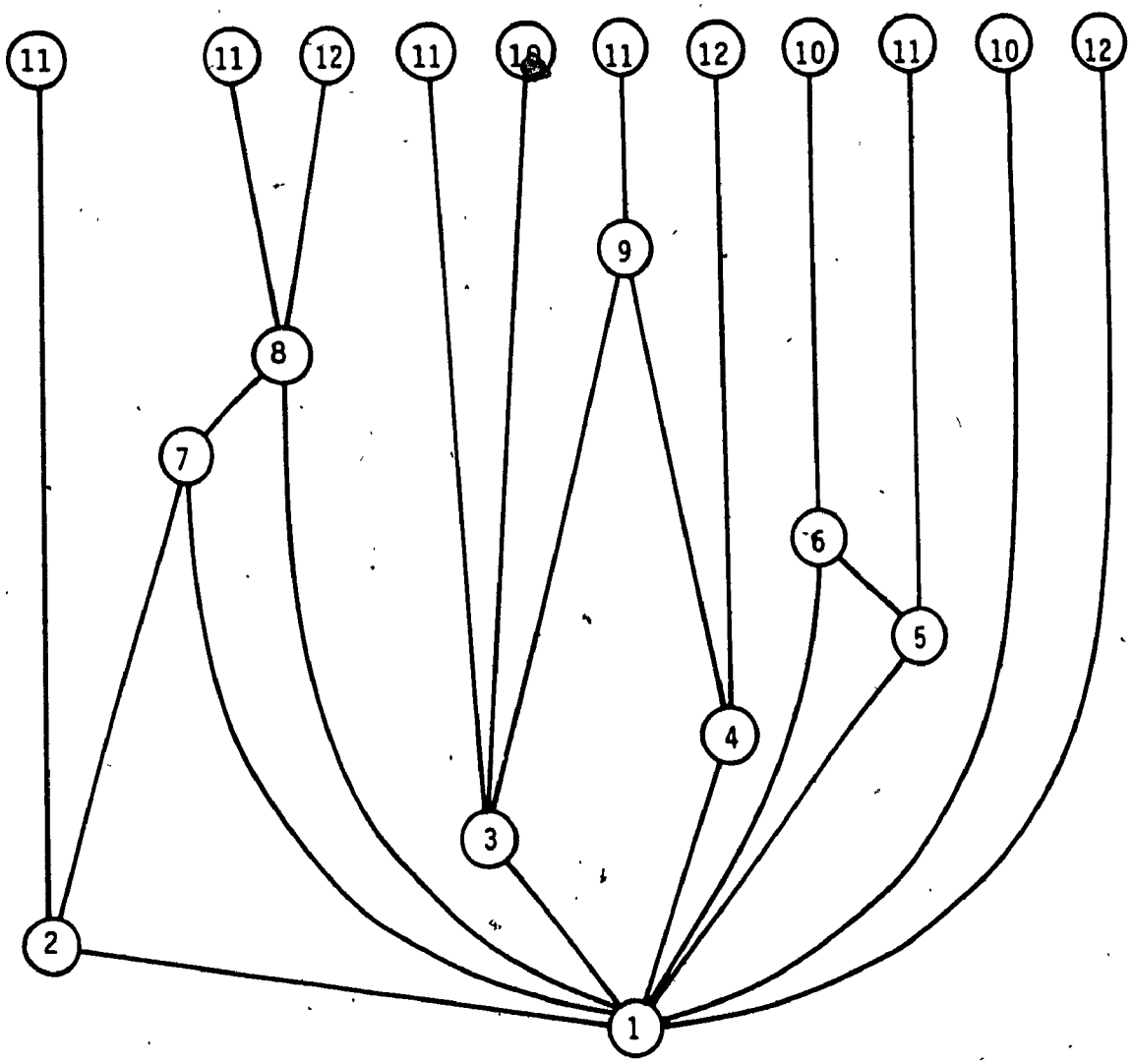


Figure 7.13(a)

Bush Form B_9

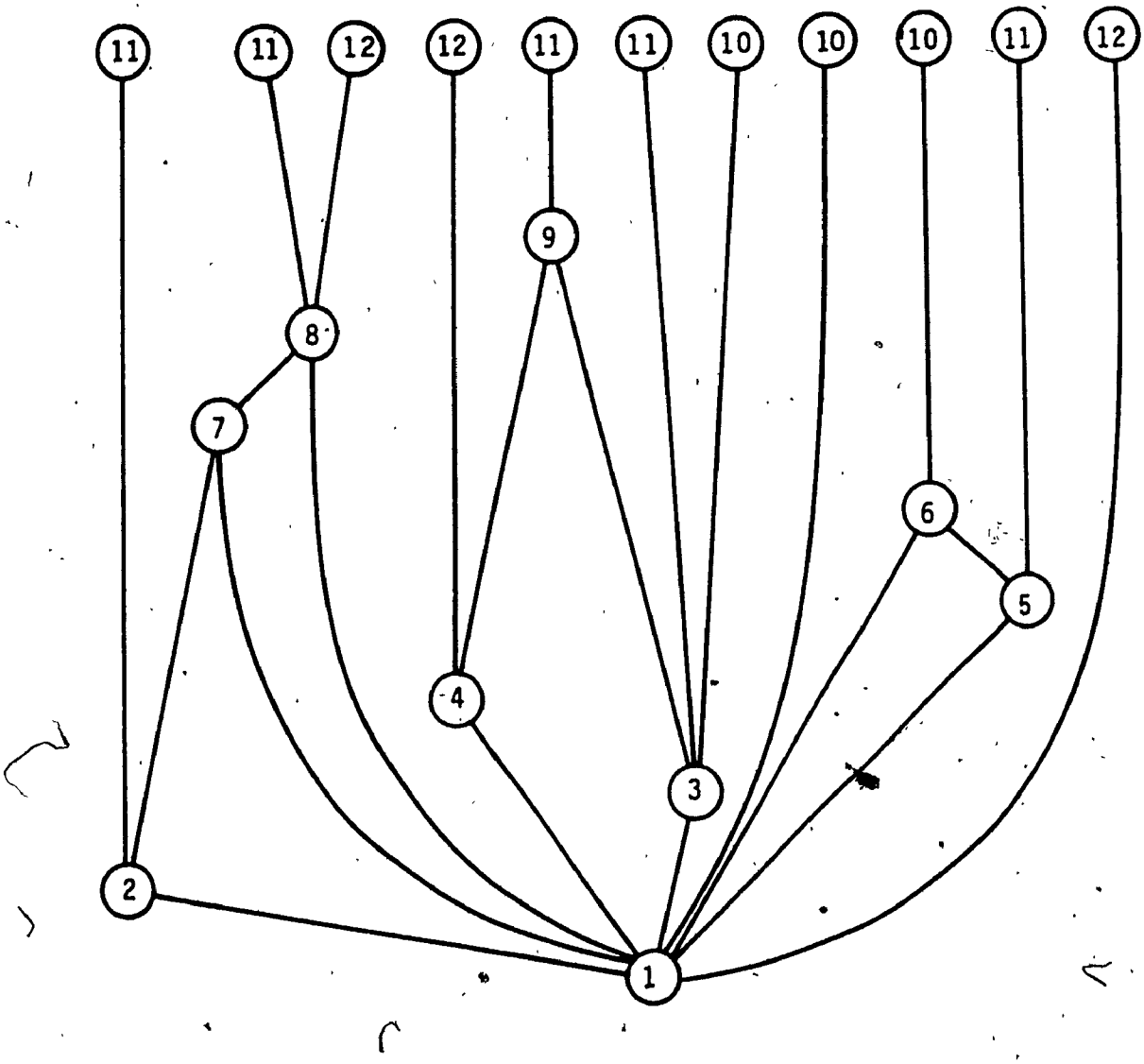


Figure 7.13(b)

Bush Form B_9'

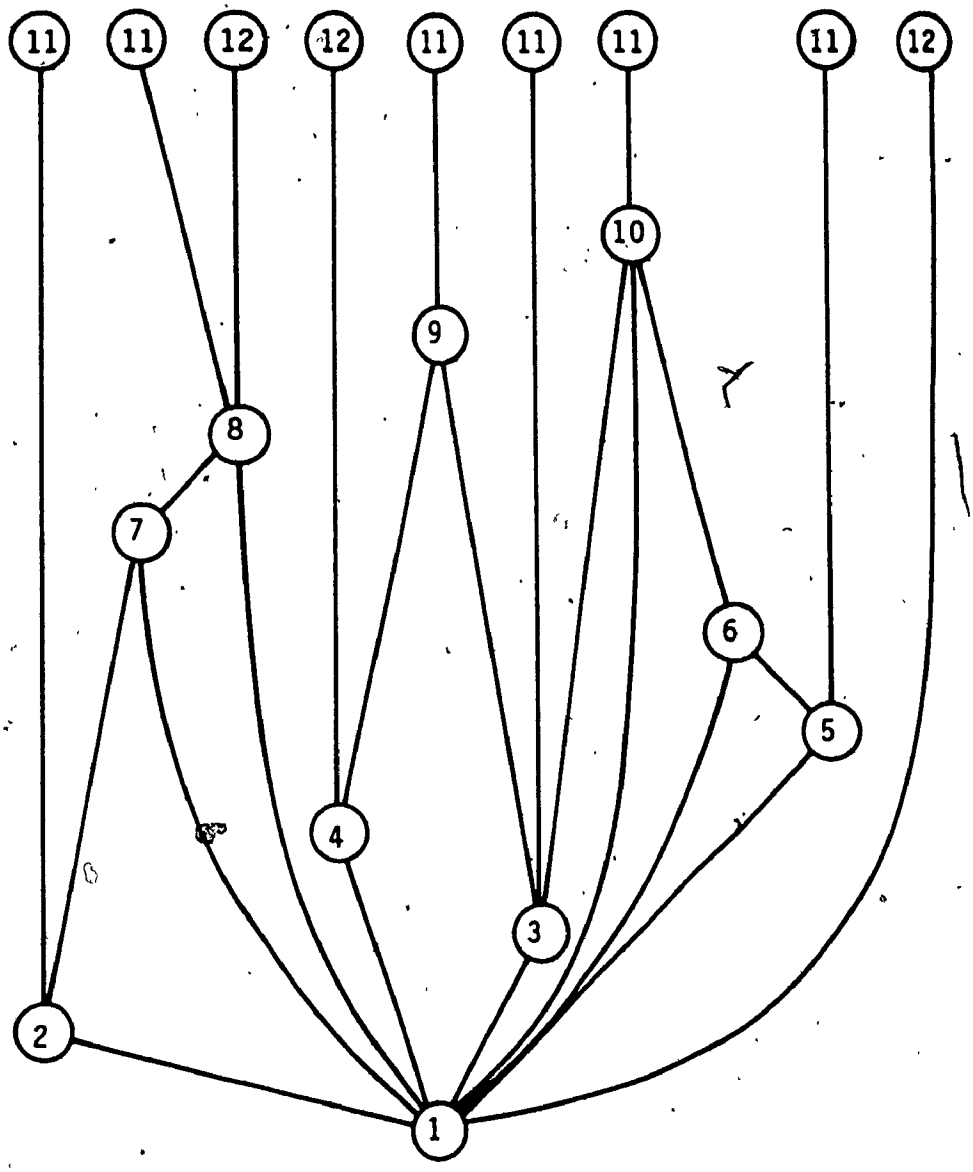


Figure 7.14(a)

Bush Form B_{10}

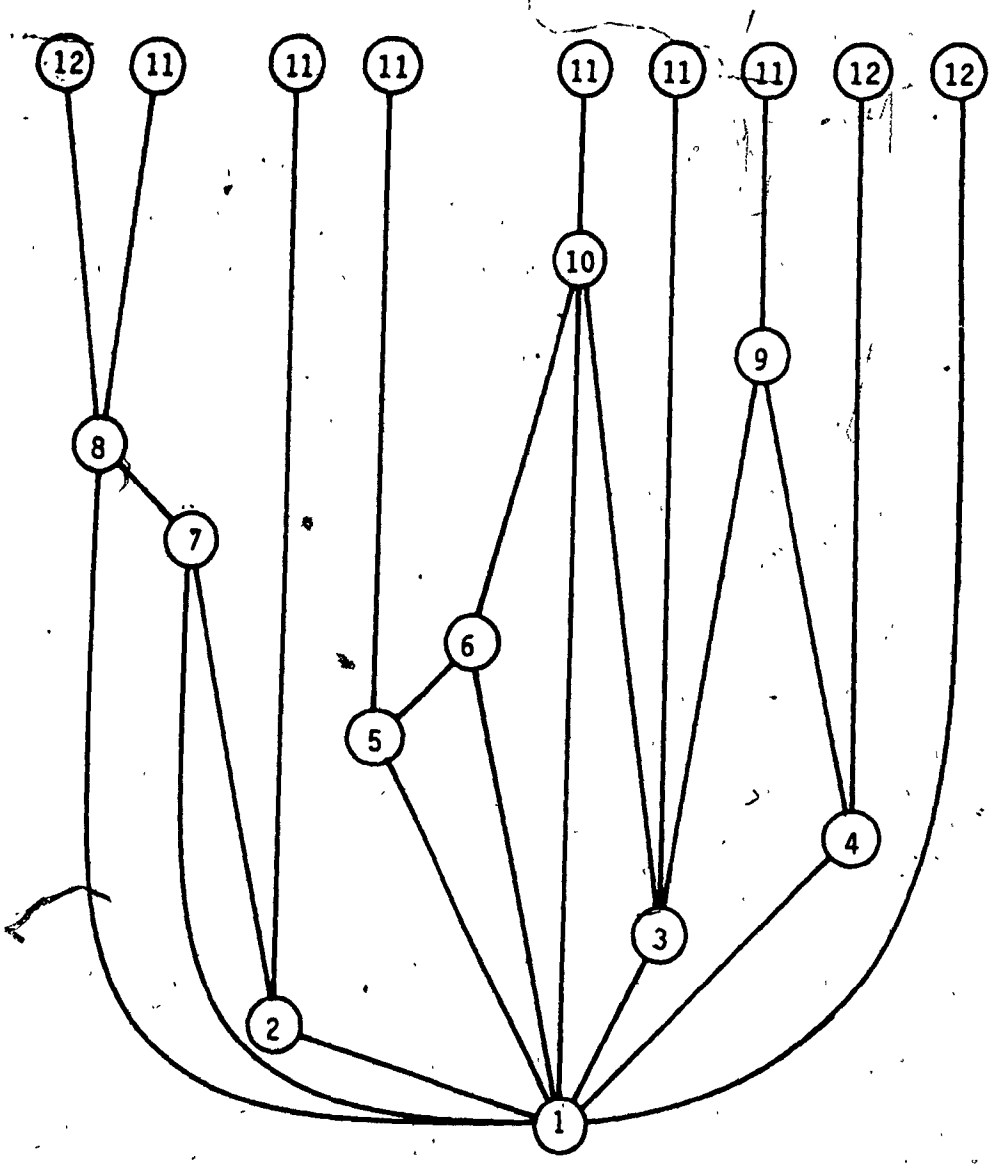


Figure 7.14(b)

Bush Form B'_{10}

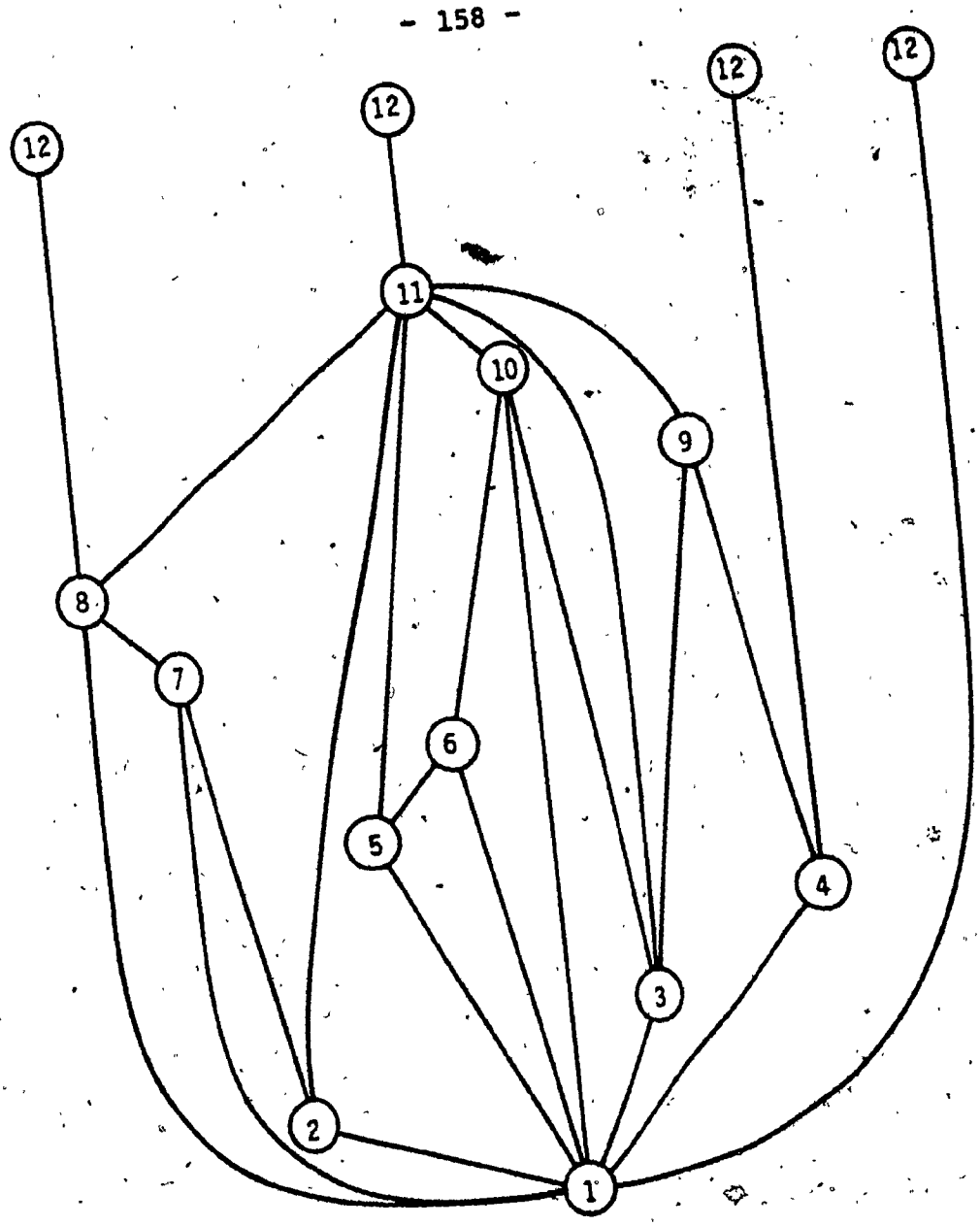


Figure 7.15

Bush Form $B_{11} = B'_{11}$

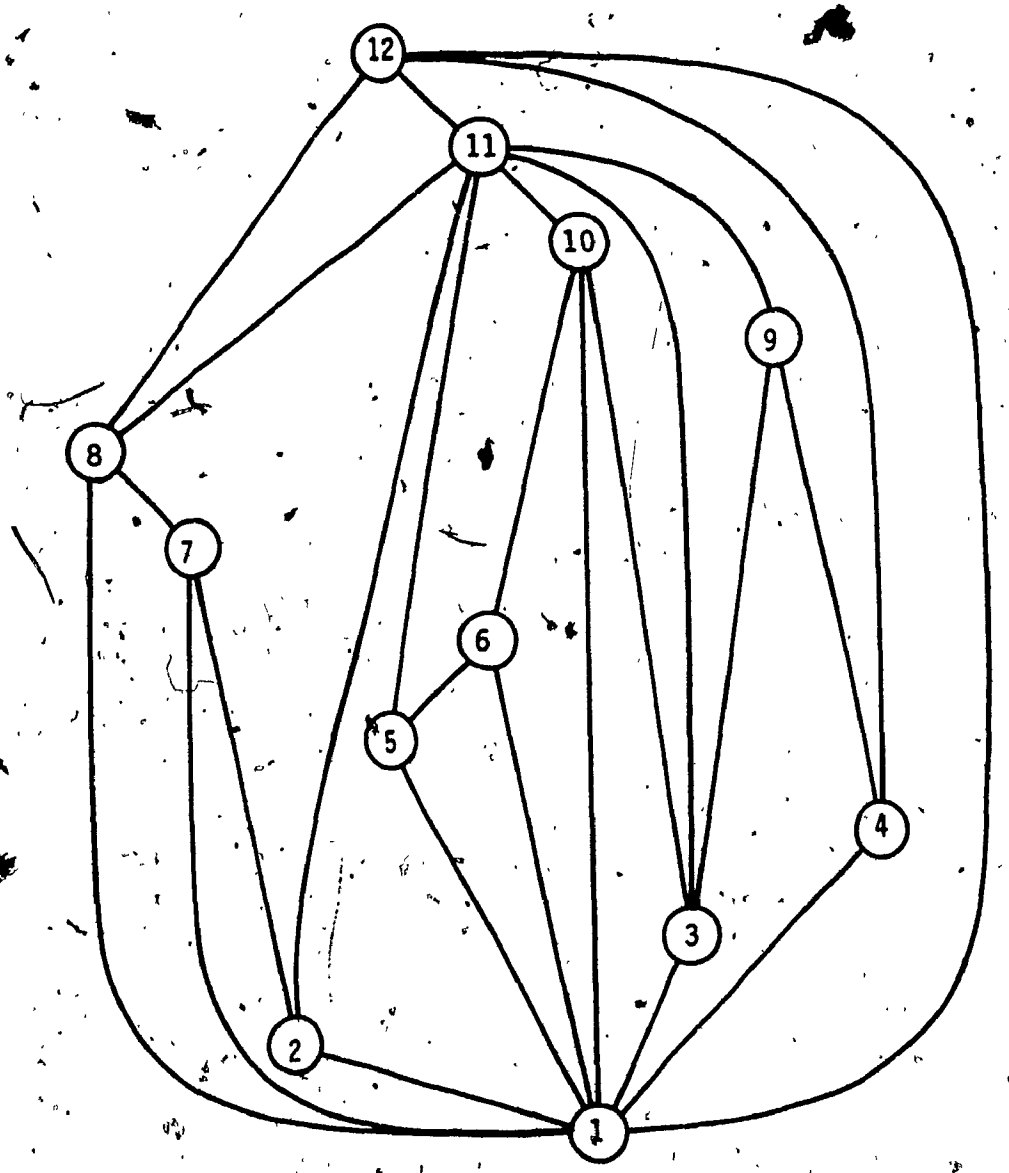


Figure 7.16

Plane Realization of G

From Lemma 7.3 it follows that a bush form isomorphic to B_k can be obtained by flipping a maximal biconnected component. Lemma 7.2 implies that a cut vertex v of B_k is the lowest vertex in each of the components, except the one which contains vertex 1, if v > 1. Each of these components has the same structure as a bush form, except that its lowest vertex is v rather than 1, and so we call it a subbush. If there are p such components of B_k with respect to v, then these subbushes can be permuted around v in any of the p! permutations to obtain a bush form isomorphic to B_k. Also each of these subbushes can be flipped over. These transformations, namely permutation and flipping, maintain the bush form. In fact Lempel, Even, and Cederbaum proved the following [44].

THEOREM 7.1.

If $\hat{B}_k^1$ and $\hat{B}_k^2$ are bush forms of the same B_k, then there exists a sequence of permutations and flippings which transforms $\hat{B}_k^1$ into $\hat{B}_k^3$ such that in $\hat{B}_k^2$ and $\hat{B}_k^3$ the virtual vertices appear in the same order. □

The above theorem implies that each bush form B_k can be transformed into a bush form B_k' in which all the virtual vertices labeled k+1 appear next to each other. For example, the bush form B_9' shown in Fig. 7.4 is obtained from the bush form B_9 of Fig. 7.3 by flipping the biconnected component containing the set of vertices {1, 3, 4, 9} and

permuting the subbushes around the cut vertex 1. Now the problem is to find, from among all possible permutations and flippings, an appropriate sequence of permutations and flippings which will transform B_k into B'_k. Moreover, we would like to do these transformations efficiently, without drawing the actual bush forms. Lempel, Even, and Cederbaum [41] represented the information about a bush form using certain expressions. They developed different methods to manipulate these expressions, which would reflect the effect of permutations and flippings of the subbushes. However, their method did not result in an efficient implementation of the algorithm. In the next section we describe a data structure called PQ-tree. We shall discuss how it could be used to represent the information pertaining to a bush form as well as to obtain the bush form B_{k+1} from the given B_k. We also show that using PQ-trees, the LEC algorithm can be implemented with $O(m+n)$ time bound.

7.3 PQ-trees to Represent Bush Forms

Given a set U and a collection $\{S_1, S_2, \ldots\}$ of subsets of U. Booth and Lueker [42] introduced a data structure to represent the class of possible permutations of the elements of U in which all the elements in each subset S_i appear consecutively. If U_i, $1 \leq i \leq n-1$, is the set of the virtual edges in the bush form B_i of a graph G and S_i is

the set of the virtual edges entering vertex i+1 in B_i, then the LEC algorithm implies that G is planar if and only if for each i there exists a permutation of the edges of U_i in which all the edges in S_i appear consecutively. Based on this Booth and Lueker showed how PQ-trees could be used to implement the LEC algorithm in O(m+n) time. In this section we discuss PQ-trees in the context of the planarity testing problem. A more general description of PQ-trees may be found in [42]. We describe how to represent any bush form B_k, $1 \leq k \leq n-1$, of G using a PQ-tree. We also discuss methods of manipulating a PQ-tree representing B_k to obtain the PQ-tree representing B_{k+1}.

7.3.1 PQ-tree Representation of a Bush Form

Consider a bush form B_k, $1 \leq k \leq n-1$, of an st-graph G. The first step in applying the LEC algorithm is to transform B_k, if possible, to an equivalent bush form B'_k in which all the virtual vertices labeled k+1 appear consecutively. As we noted in the previous section, such a B'_k, whenever it exists, can be obtained by performing a sequence of transformations, namely flippings of maximal biconnected components of B_k and permutations around cut vertices of the subbushes of B_k. Thus while applying the LEC algorithm for testing the planarity of G the following are of interest.

(i) the virtual vertices (and virtual edges) in B_k,

(ii) the cut vertices in B_k and the maximal biconnected

components of B_k, and

(iii) the cut vertices $y_1, y_2, \ldots, y_q$ appearing in that order on the outside window of any maximal biconnected component of B_k.

Let T_k denote the PQ-tree corresponding to B_k. Then, in T_k, the above pieces of information are represented by different types of nodes as described below.

(i) Leaf: Leaves in a PQ-tree represent virtual vertices in the corresponding bush form. Since each virtual vertex is the end vertex of a virtual edge, a leaf also represents a virtual edge. Leaves are indicated by squares in our figures. A leaf has the same label as the virtual edge it represents.

(ii) P-node: P-nodes represent cut vertices in the bush form. P-nodes are indicated by circles in our figures. A P-node is labeled as the cut vertex it represents.

(iii) Q-node: Q-nodes represent the maximal biconnected components in a bush form. Let $y_1, y_2, \ldots, y_q$ be the cut vertices (except the lowest vertex), appearing in that order, on the outside window of a maximal biconnected component. Then this component is represented by a Q-node whose children are the P-nodes corresponding to $y_1, y_2, \ldots, y_q$. Furthermore, these children appear in the same left-to-

right order as the order of the corresponding cut vertices on the outside window of the maximal biconnected component. The P-nodes corresponding to y_1 and y_q are called endmost children of the Q-node and the other P-nodes are called internal children. Q-nodes are shown as rectangles in the figures.

We now describe the procedure to construct T_k. Consider a cut vertex v in the bush form B_k, $1 \leq k \leq n-1$. Let $C_{k(1)}, C_{k(2)}, \ldots, C_{k(i)}$ be the components* of B_k with respect to v. Any component $C_{k(j)}$, $1 \leq j \leq i$, may be of one of the following two types.

(i) $C_{k(j)}$ has only one edge (v,x) incident out of v in G. In this case the node corresponding to vertex x is made a child of the P-node corresponding to v. Note that the node in T_k corresponding to x may be a P-node or a leaf depending on whether x is a cut vertex or a virtual vertex in B_k.

(ii) $C_{k(j)}$ has more than one edge incident out of v in G. In this case $C_{k(j)}$ is represented by a Q-node whose children are the P-nodes corresponding to the cut vertices other than v appearing on the outside window of $C_{k(j)}$. This Q-node is then made a child of the P-node corresponding to v.

---

*Note that only those biconnected components in which vertex v is the lowest vertex are of interest to us.

Repeating the above procedure for each component of every cut vertex in B_k, we can construct the PQ-tree T_k corresponding to B_k. As an example, the PQ-tree T_9 corresponding to the bush form B_9 of Fig. 7.3 is shown in Fig. 7.17. Note that the PQ-tree is drawn with the P-node corresponding to vertex 1 at the top because it is customary to draw rooted trees with the root at the top.

Suppose a node Y in T_k has only one child Z. Let X be the parent of Y in T_k. Then, (X,Y) and (Y,Z) are series edges in T_k, and replacing these series edges by the edge (X,Z) will not affect the essential features of the bush form B_k, which are required for testing the planarity of G. So, if any node Y has only one child Z, then we delete Y from T_k and make Z a child of X. Thus we assume, without loss of generality, that all the nodes in a PQ-tree have at least two children.

As we noted before, whenever the st-graph G is planar, a bush form B_k of G can be converted into an equivalent bush form B_k', in which all the virtual vertices labeled k+1 appear together, using a sequence of one or more of two types of operations, namely flipping a biconnected component and permuting the subbushes around a cut vertex. Clearly the corresponding operations on a PQ-tree are, respectively,

(i) reversing the order of the children of a Q-node, and

(ii) permuting the children of a P-node.

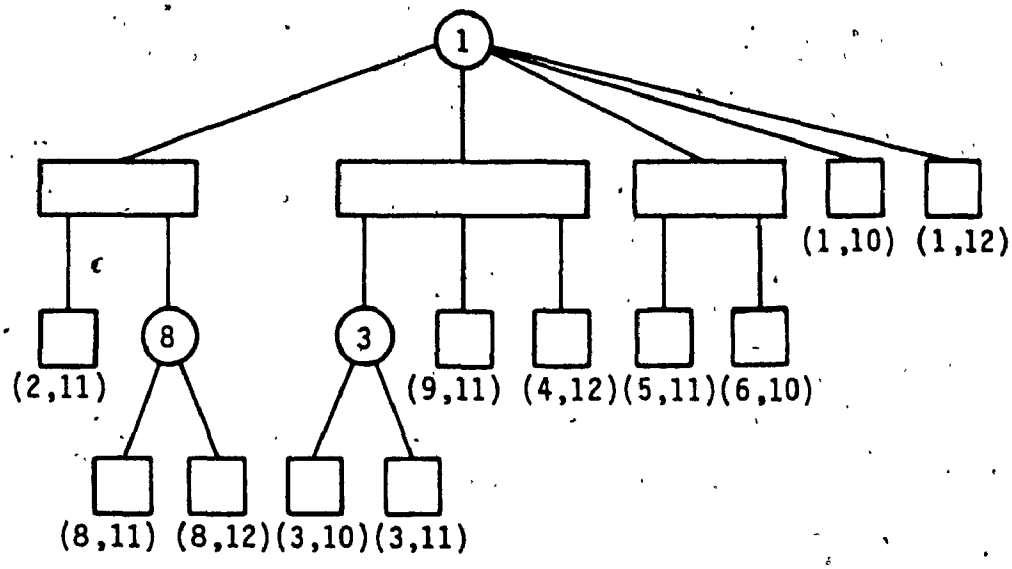


Figure 7.17

PQ-tree T_9 corresponding to B_9

Thus we consider two PQ-trees to be equivalent if we can transform one into the other using a sequence of one or more of the above two operations.

7.3.2 Template Matching

Given the PQ-tree T_k representing a bush form B_k, $1 \leq k \leq n-1$, we now describe an algorithm for constructing T_{k+1} from T_k. We wish to achieve this without drawing B_k or B_{k+1}. First we need a few definitions.

Let S(k+1) denote the set of leaves in T_k which correspond to the virtual vertex k+1. A node X in T_k is said to be <u>full</u> if all its descendant leaves are in S(k+1); X is said to be <u>empty</u> if none of its descendant leaves are in S(k+1). In our figures we indicate full nodes by shading them and empty nodes are left unshaded. If some but not all of the descendant leaves of X are in S(k+1) then X is said to be <u>partial</u>. Partial nodes are shown partially shaded. A node which is either full or partial is referred as a <u>pertinent</u> node. We define the <u>frontier</u> of T_k as the sequence of all the leaves in T_k read from left to right. For example, the frontier of T_9 shown in Fig. 7.17 is 11, 11, 12, 10, 11, 11, 12, 11, 10, 10, 12. The <u>pertinent subtree</u> of T_k with respect to S(k+1) is the subtree of minimum height whose frontier contains all the

leaves in S(k+1). The pertinent subtree and its root are unique. The root of the pertinent subtree is not necessarily the root of T_k. The pruned pertinent subtree of T_k with respect to S(k+1) is the smallest connected subgraph of T_k which contains all the pertinent nodes. For example, for the PQ-tree T_9, the pruned pertinent subtree, with respect to the set of leaves corresponding to virtual vertex 10, is shown in Fig. 7.18. Note that in this case T_9 itself is the pertinent subtree. In Fig. 7.19, we have shown this pertinent subtree with the leaves corresponding to virtual vertex 10 marked full. Finally let T(i), $1 \leq i \leq n-1$, denote a PQ-tree having one P-node labeled i and as many leaves as the number of edges incident out of vertex i in G. These leaves are children of the P-node and are labeled as their corresponding edges in G. Note that $T(1) = T_1$.

To construct T_{k+1} from T_k, we first construct a PQ-tree T_k^* in which all the full leaves of T_k appear consecutively as the children of a Q-node. Of course, if there is only one full leaf in T_k, then T_k^* will be the same as T_k. For example, the PQ-tree T_9^* corresponding to the PQ-tree T_9 of Fig. 7.17 is shown in Fig. 7.20. Now replacing the leaves corresponding to the virtual vertex k+1 by T(k+1) we obtain the PQ-tree T_{k+1} representing the bush form B_{k+1}.

We now describe a procedure to transform T_k into T_k^*. This procedure for reducing T_k into T_k^* involves processing

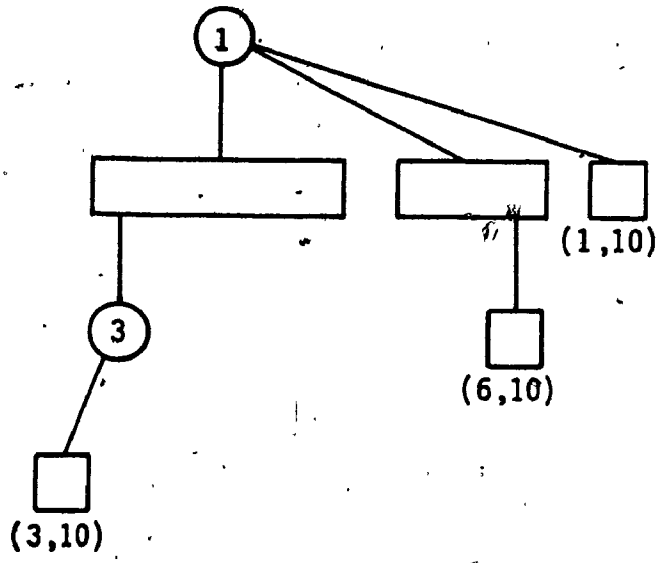


Figure 7.18

Pruned Pertinent Subtree of T_9

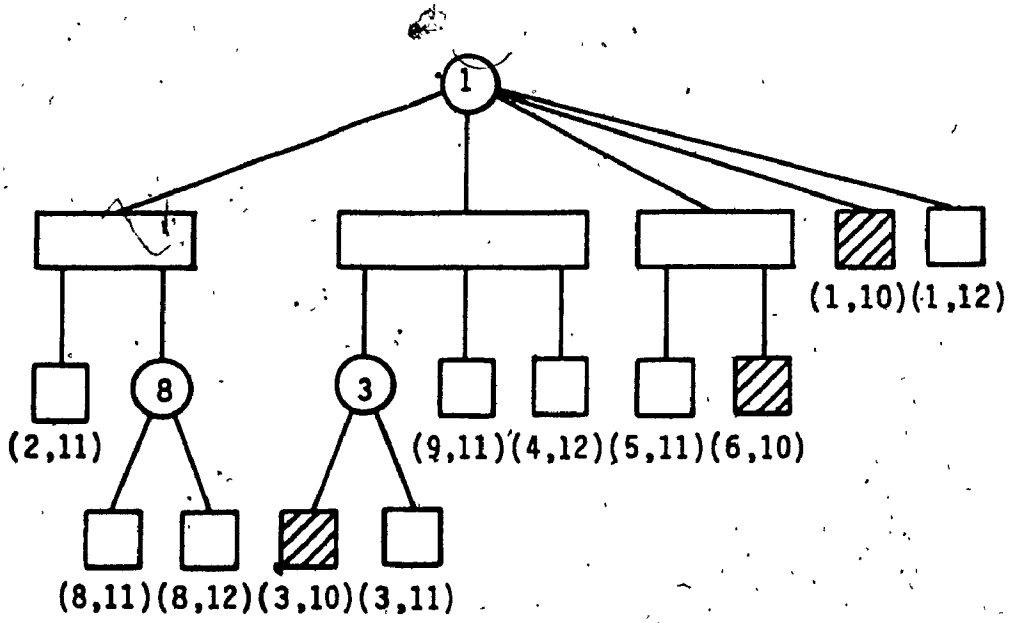


Figure 7.19

Pertinent Subtree of T_9

Pertinent Leaves are marked Full

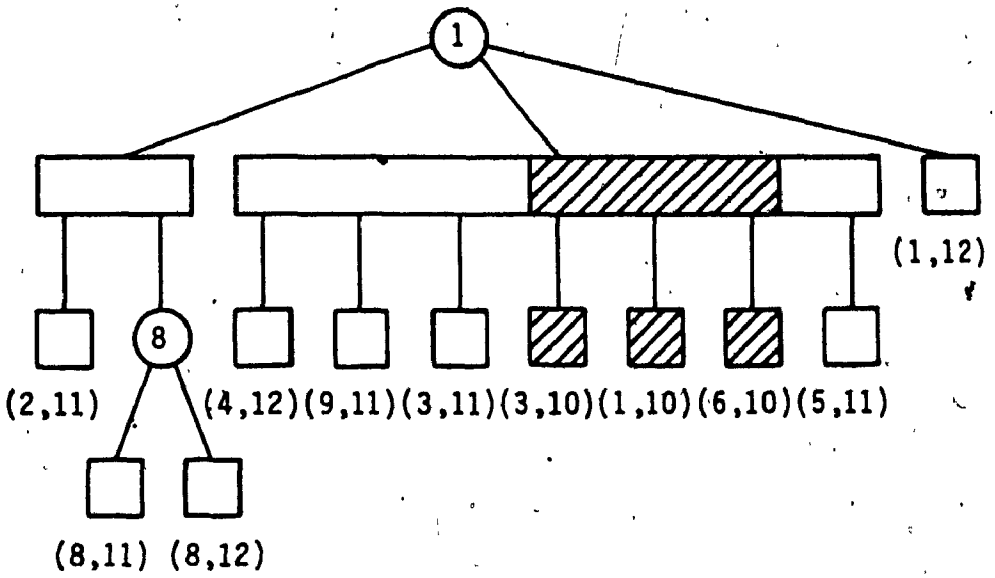


Figure 7.20

PQ-tree T_9^*

(in an appropriate manner to be described below) the pertinent subtree of T_k with respect to the leaves in S(k+1). The processing is carried out bottom-up. That is, a node of the pertinent subtree is processed only after all its pertinent children are processed. When a node is processed, the node and its children are compared with a sequence of templates. Each template has a pattern and a replacement. During the template matching, if necessary, the children of a P-node may be arbitrarily permuted, and if any of the children is a Q-node, then the children of this Q-node may be reversed so as to match the pattern of a template. If a node and its children match a template's pattern, then the pattern is replaced within the tree by the template's replacement. Thus, each template specifies a local change within the PQ-tree and the tree obtained after the replacement is also a PQ-tree. This template matching is repeated until the root of the pertinent subtree is processed. The bottom-up strategy is used to ensure that the subtrees rooted at the pertinent children of a node have already been processed when the node itself is considered for template matching.

To begin the template matching, all the pertinent leaves in T_k (that is, the leaves in S(k+1)) are marked full and all the other leaves are marked empty. When any internal node is processed, our aim is to ensure that after replacement, all the pertinent leaves in the frontier of the

subtree rooted at that node occur as a consecutive subsequence of the frontier. Moreover, we want to do the template matching in such a way that all the leaves in S(k+1) are made children of a single node in T_k^*. Note that in T_k^*, this node, which is the parent of all the leaves in S(k+1), will be a Q-node if $|S(k+1)| > 1$.

Now we describe the sequence of templates which are needed to achieve the above goals. In the figures which follow, a triangle represents a subtree. Our discussion of template matching is in the context of reducing T_k into T_k^*. So, each pertinent leaf represents a virtual vertex labeled k+1 as well as a virtual edge (i,k+1), for some i, incident into the vertex k+1. Furthermore, during the reduction of T_k into T_k^*, each Q-node will represent either a biconnected component of B_k or the biconnected component which will result if we coalesce in B_k all the virtual vertices which are represented as children of the Q-node.

During the template matching the following different cases occur, where X denotes the node being processed.

Case 1: X is a P-node.

(i) If all the children of X are empty, then no change is necessary.

(ii) Template P1 (Fig. 7.21): In this case all the

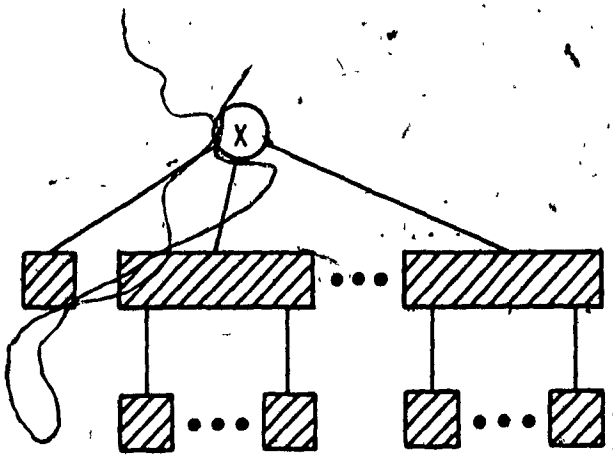
children of X are full. To bring all the pertinent leaves as children of the same node, we replace X by the replacement shown in Fig. 7.21.

(iii) Template P2 (Fig. 7.22): In this case X is partial and is the root of the pertinent subtree. Thus the reduction process will stop after processing X. So, we make all pertinent leaves as children of the same node by the replacement shown in Fig. 7.22.

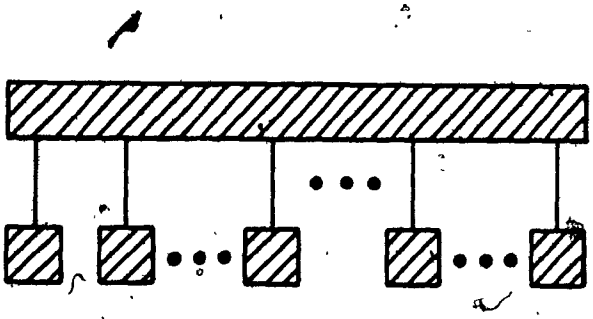
(iv) Template P3 (Fig. 7.23): Now X is partial and is not the root of the pertinent subtree. Thus there is at least one more pertinent node to be processed which is not a descendant of X. So, after the reduction X will be on the outside window of some biconnected component. This is reflected in the replacement shown in Fig. 7.23.

(v) Template P4 (Fig. 7.24): In this case X is partial; it is the root of the pertinent subtree and has exactly one partial Q-node among its children. If $y_1, y_2, \ldots, y_q$ are the cut vertices on the outside window of the biconnected component corresponding to the partial Q-node, then in B_{k+1}, this biconnected component will have the cut vertices $y_1, y_2, \ldots, y_q$, k+1 on its outside window. From this observation, the replacement in Fig. 7.24 follows.

(vi) Template P5 (Fig. 7.25): Now X is a partial node; it is not the root of the pertinent subtree and has exactly one partial Q-node among its children. Let $y_1, y_2, \ldots, y_q$ be



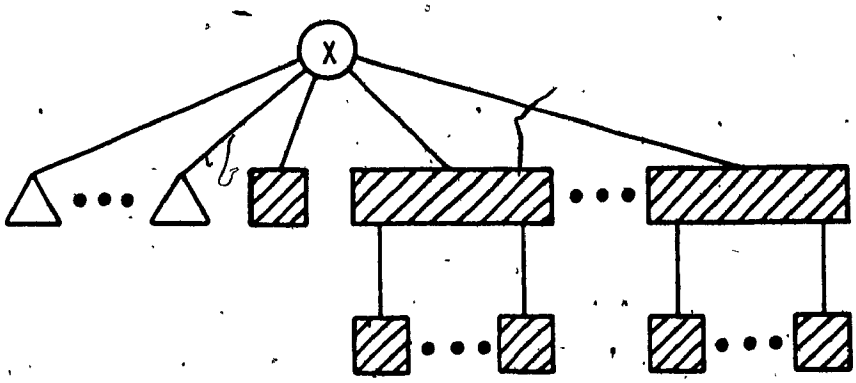
Pattern



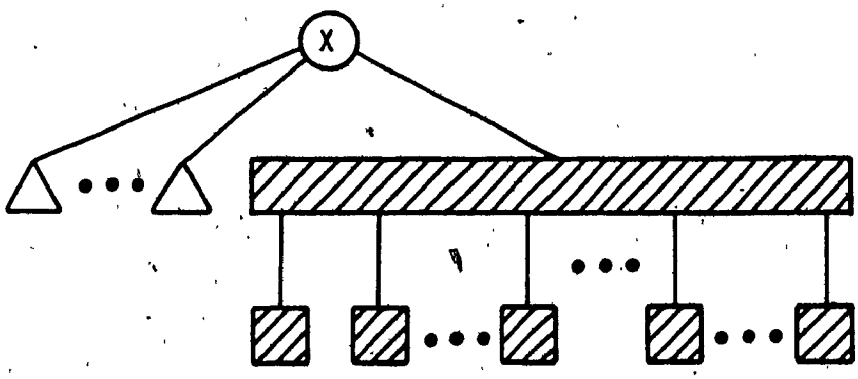
Replacement

Figure 7.21

Template P1



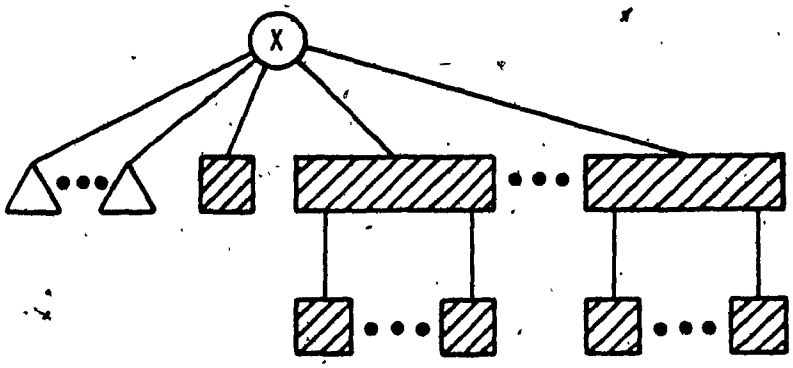
Pattern



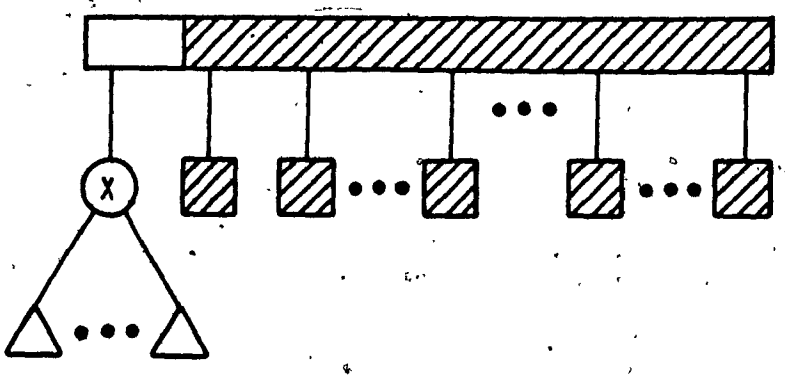
Replacement

Figure 7.22

Template P2



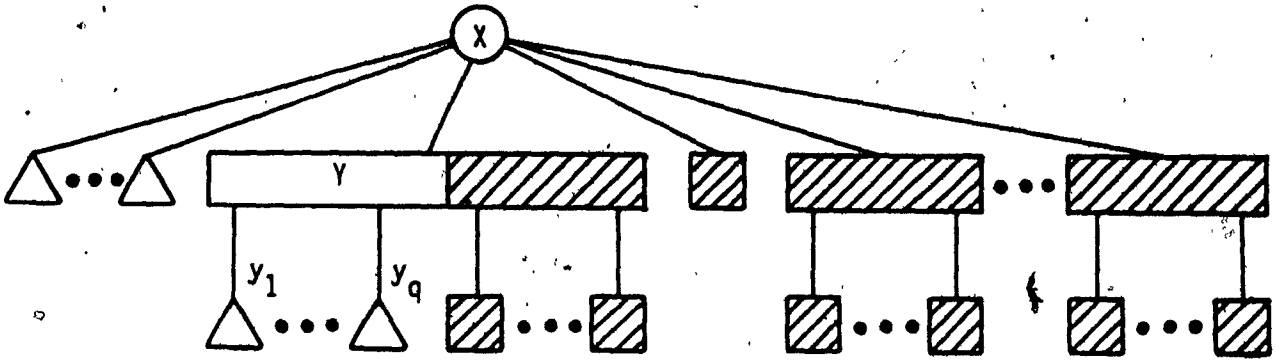
Pattern



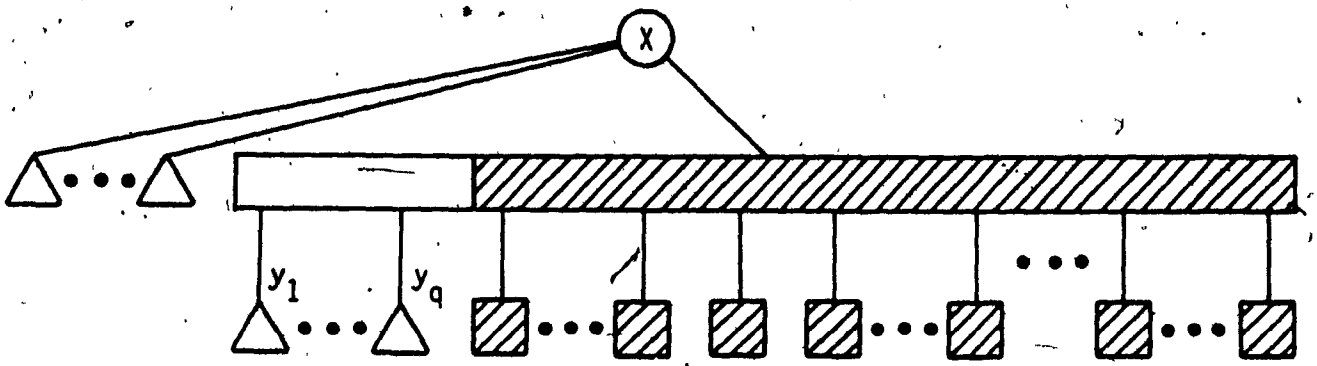
Replacement

Figure ~~7.~~23

Template P3



Pattern



Replacement

Figure 7.24

Template P4

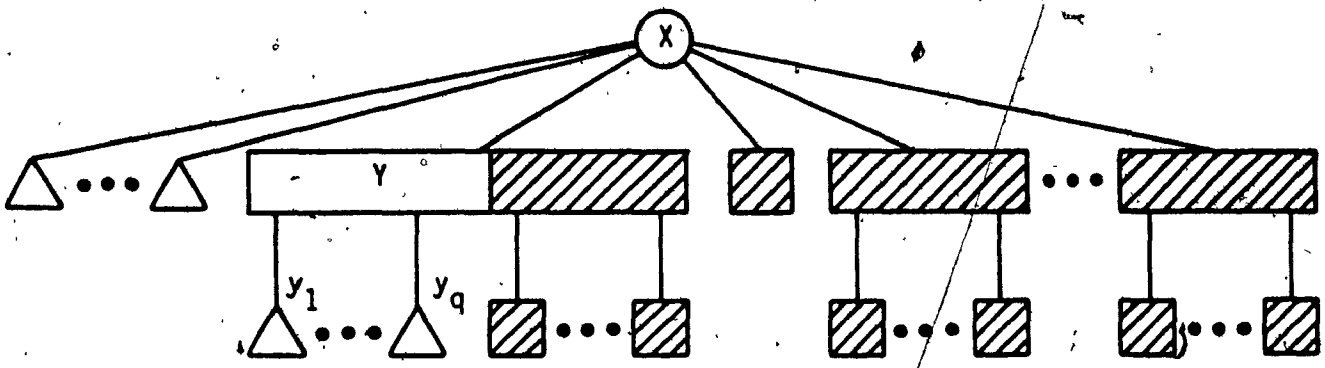
the cut vertices on the outside window of the biconnected component corresponding to the partial Q-node. Since X is not the root of the pertinent subtree, in B_{k+1} there will be a biconnected component having the vertices X, y_1, y_2, ..., y_q, $k+1$, ... on its outside window. Thus the replacement in Fig. 7.25 follows.

(vii) Template P6 (Fig. 7.26): In this case X has two partial Q-nodes, say Y and Z, among its children. Note that X must be the root of the pertinent subtree, for otherwise the tree T_k cannot be reduced. Let y_1, y_2, ..., y_i and z_1, z_2, ..., z_j be the order of the cut vertices on the outside windows of the biconnected components corresponding to the two partial Q-nodes. Then B_{k+1} will have a biconnected component which has the cut vertices y_1, y_2, ..., y_i, k+1, z_1, z_2, ..., z_j appearing in that order on its outside window. To obtain the T_{k+1} corresponding to this B_{k+1}, we use the replacement shown in Fig. 7.26.

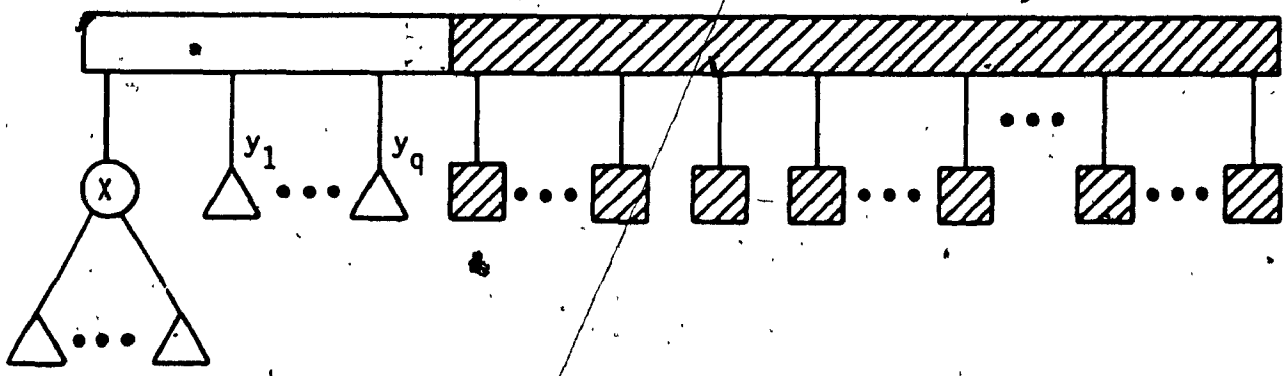
It is easy to see that if a P-node has more than two partial Q-nodes as its children, then the PQ-tree cannot be reduced. Thus if a PQ-tree has any P-node which does not match any of the above templates, then the tree is not reducible and so the graph G is not planar.

Case 2: X is a Q-node.

(i) If all the children of X are empty, then no change



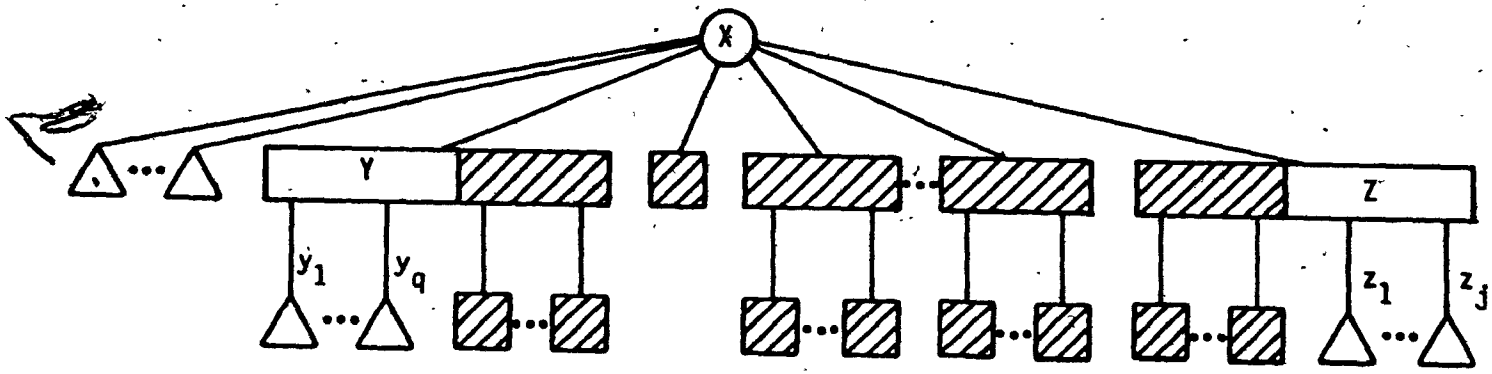
Pattern



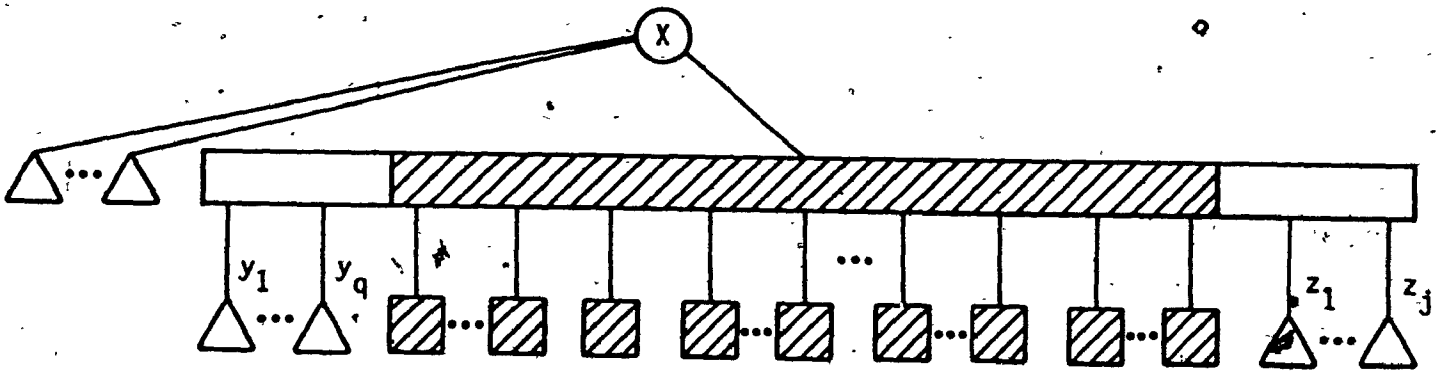
Replacement

Figure 7.25

Template P5



Pattern



Replacement

Figure 7.26

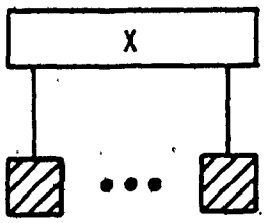
Template P6

is necessary.

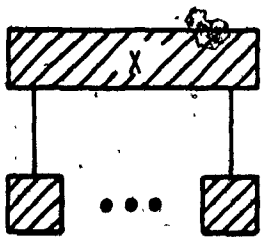
(ii) Template Q1 (Fig. 7.27): In this case all the children of X are full. So, no change is necessary except to shade X, thereby indicating that it is now full.

(iii) Template Q2 (Fig. 7.28): In this case X has exactly one partial Q-node, say Y, among its children. Let the biconnected component of B_k corresponding to X have the cut vertices $x_1, x_2, \ldots, x_i, \ldots$ on its outside window. Suppose the biconnected component corresponding to Y have the cut vertices $y_1, y_2, \ldots, y_j$ on its outside window. Then in B_{k+1} there will be a biconnected component having the vertices $x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j, k+1, \ldots$ on its outside window. Thus we use the replacement shown in Fig. 7.28 for this template.

(iv) Template Q3 (Fig. 7.29): Now X has exactly two partial Q-nodes among its children. Note that in this case X must be the root of the pertinent subtree; for otherwise the tree cannot be reduced. Let the partial Q-nodes Y and Z represent the set of cut vertices $\{y_1, y_2, \ldots, y_r\}$, and the set of cut vertices $\{z_1, z_2, \ldots, z_s\}$ respectively. Also let X represent the set of cut vertices $\{x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_j\}$. Then in B_{k+1}, there will be a biconnected component having the cut vertices $x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_r, k+1, z_1, z_2, \ldots, z_s, x_{i+1}, \ldots, x_j$ on its outside window. The replacement shown in Fig. 7.29 reflects this situation.



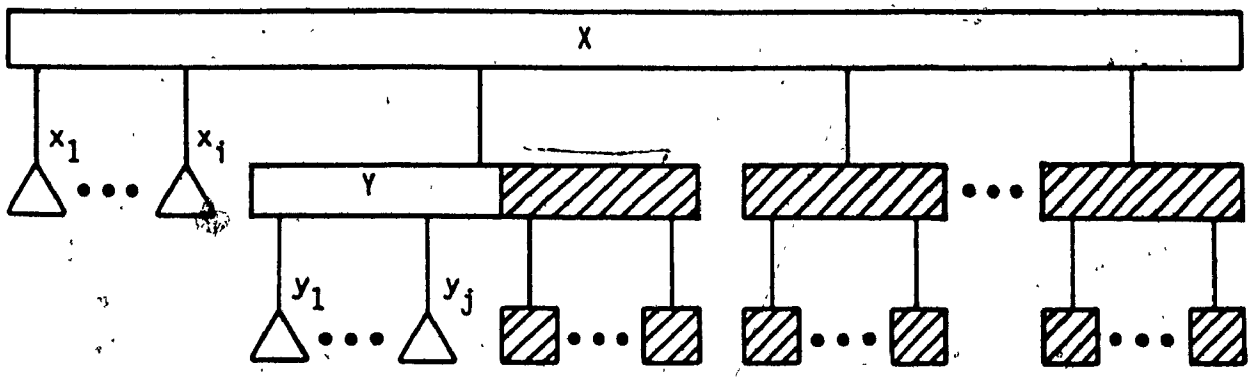
Pattern



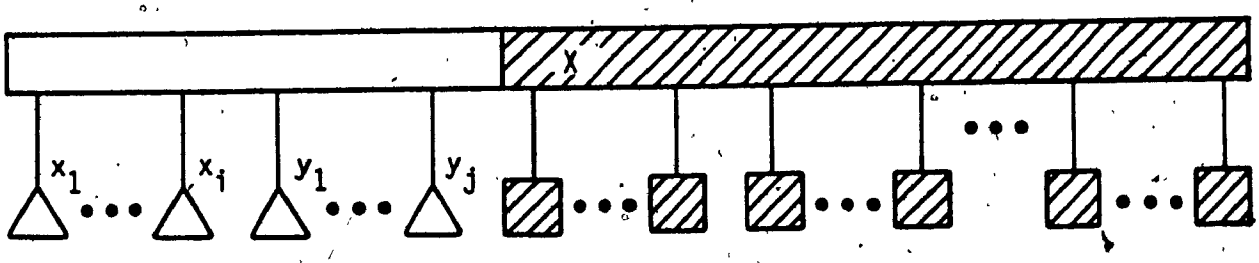
Replacement

Figure 7.27

Template Q1



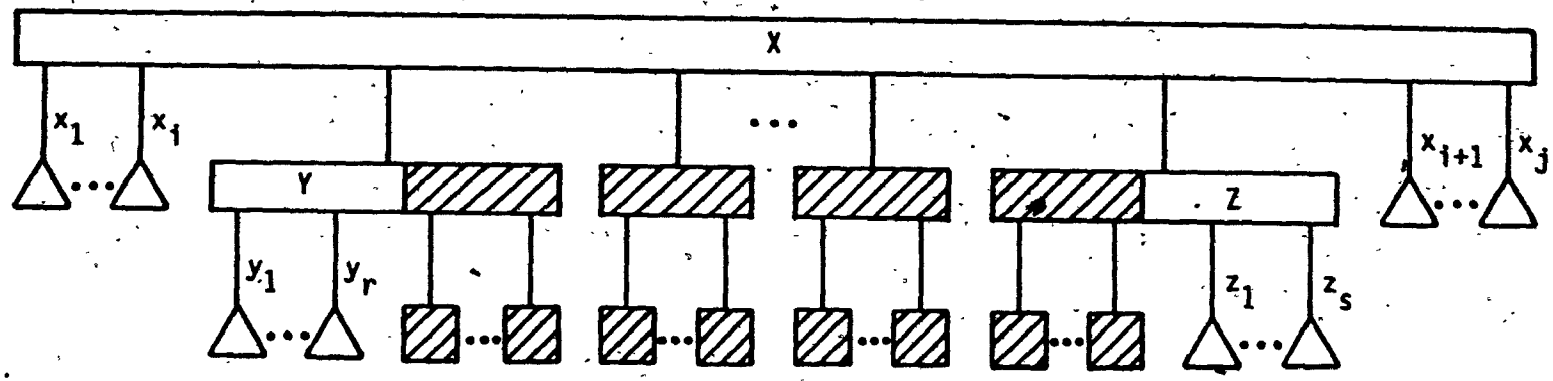
Pattern



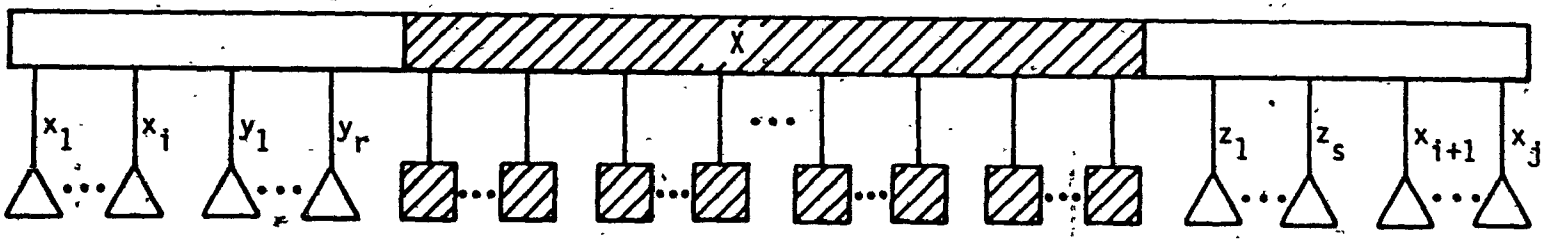
Replacement

Figure 7.28

Template Q2



Pattern



Replacement

Figure 7.29

Template Q3

If a Q-node does not match any of the above templates, then the tree cannot be reduced. The templates explained above are the only templates that can occur in the case of a planar graph. So, if any node in the PQ-tree T_k, $1 \leq k \leq n-2$, does not match any one of the above templates, then T_k cannot be reduced to T_k^* and in such a case we can conclude that G is nonplanar.

We illustrate in Fig. 7.30 the reduction of the PQ-tree T_9 into T_9^*. Starting with T_9 in which all the pertinent leaves are marked full and all the other nodes are marked empty (Fig. 7.30(a)), we obtain the PQ-tree in Fig. 7.30(b) after applying Template P3 to node A. The PQ-tree shown in Fig. 7.30(c) is obtained by applying Template Q2 to node B, and Fig. 7.30(d) results after applying Template Q2 to node C. Finally applying Template P6 to node D gives the PQ-tree shown in Fig. 7.30(e), which is the PQ-tree T_9^* shown in Fig. 7.20.

Thus, to test a graph for planarity, we start with the PQ-tree T_1 corresponding to the bush form B_1. At any point we reduce a PQ-tree T_k, $1 \leq k \leq n-2$, into the corresponding T_k^* and then construct the PQ-tree T_{k+1} from T_k^*. If all the PQ-trees $T_2, T_3, \ldots, T_{n-1}$ can be obtained in this way, then G is planar; otherwise G is nonplanar. In Figs. 7.31 to 7.41 we give the PQ-trees corresponding to the bush forms of the st-graph G of Fig. 7.1. Since all the required PQ-trees

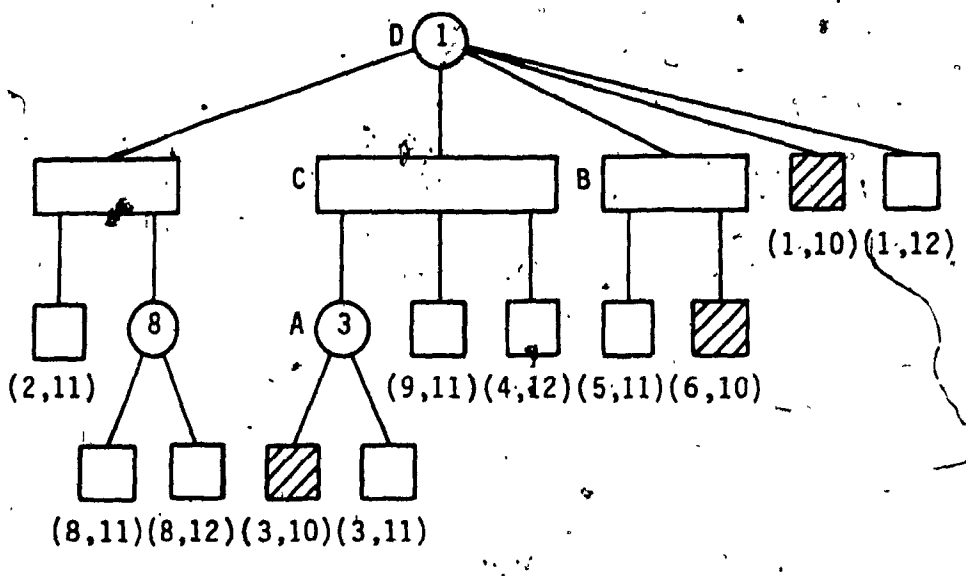


Figure 7.30(a)

PQ-tree T_9

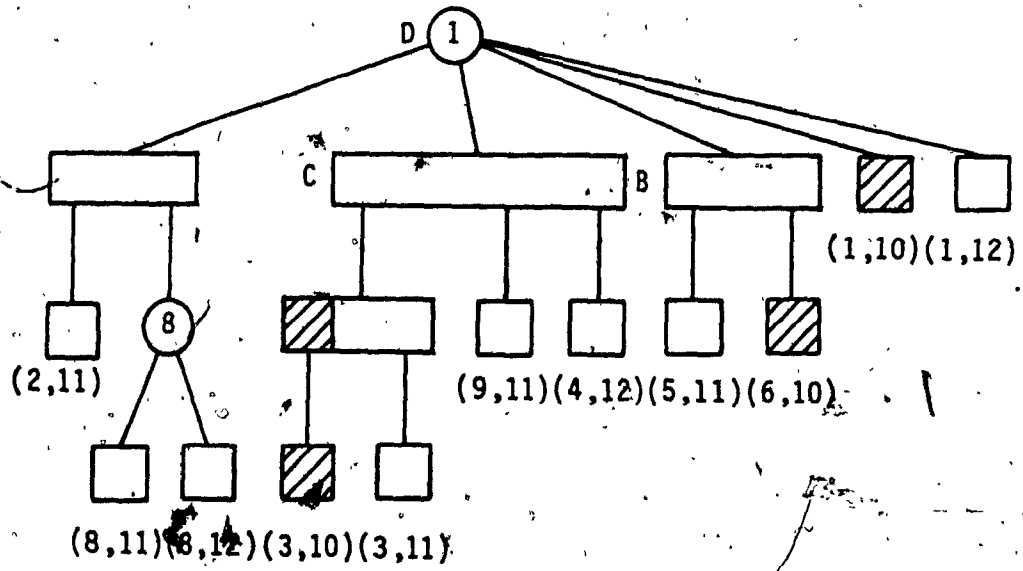


Figure 7.30(b)

PQ-tree after applying Template P3 to A

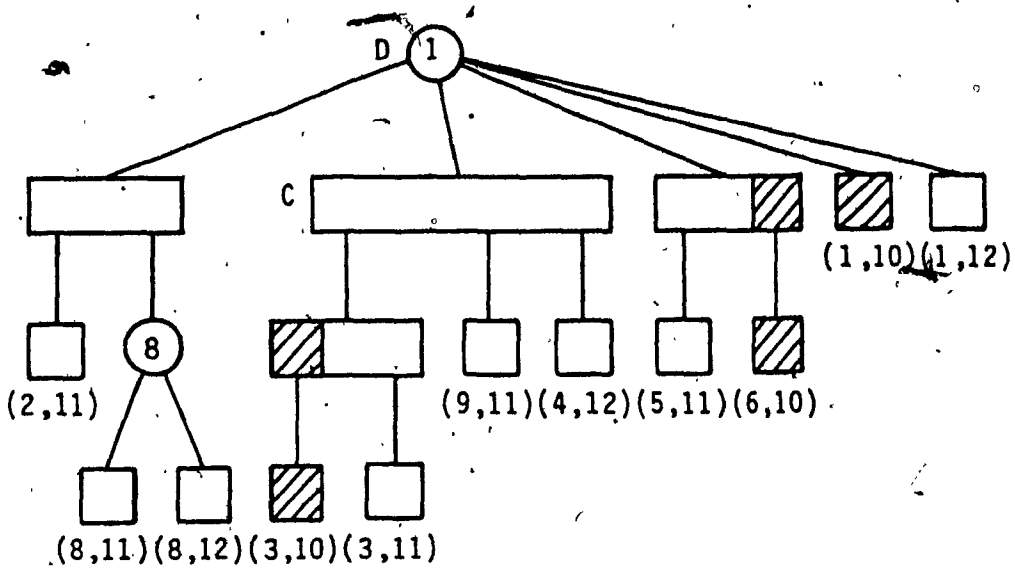


Figure 7.30(c)

PQ-tree after applying Template Q2 to B

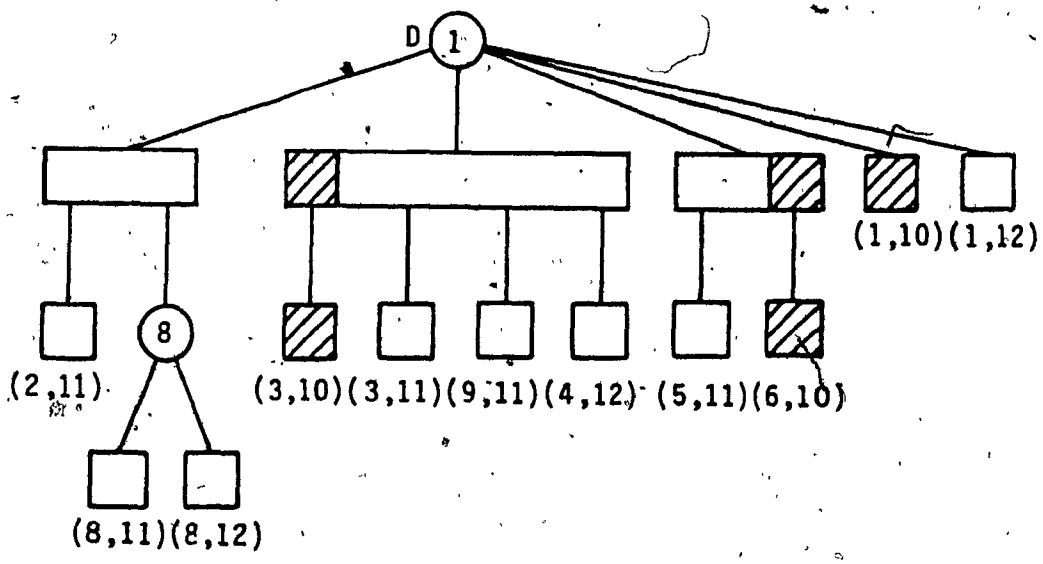


Figure 7.30(d)

PQ-tree after applying Template Q2 to C

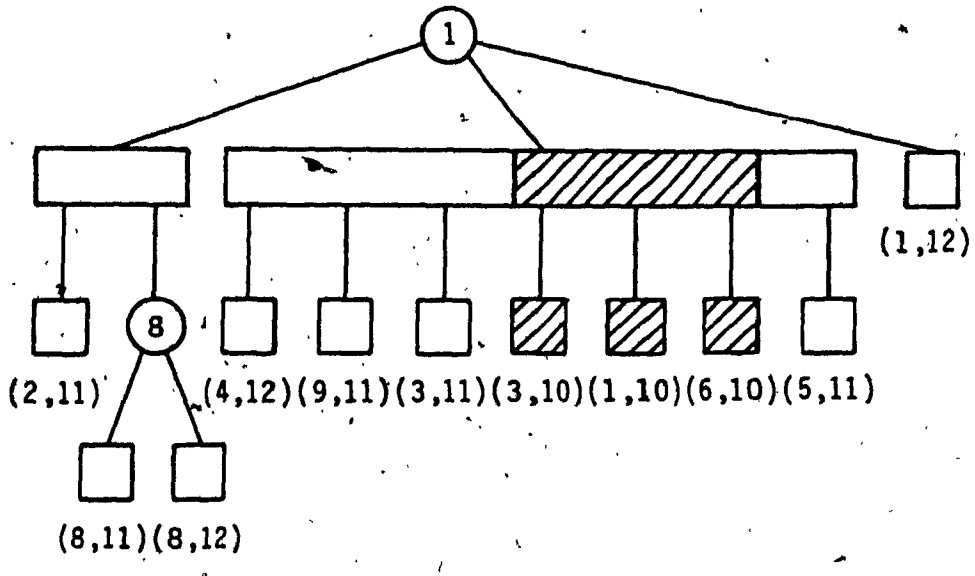


Figure 7.30(e)

PQ-tree after applying Template P6 to D

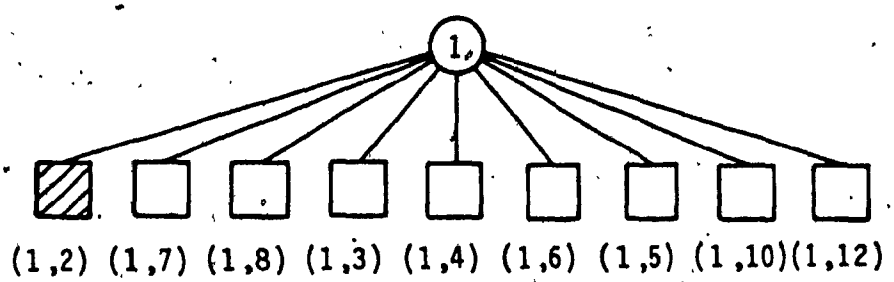


Figure 7.31

PQ-tree $T_1 = T_1^*$

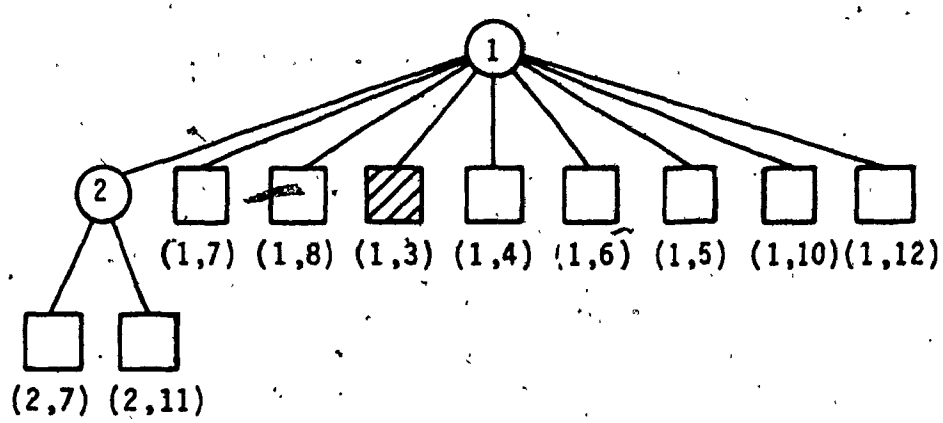


Figure 7.32

PQ-tree $T_2 = T_2^*$

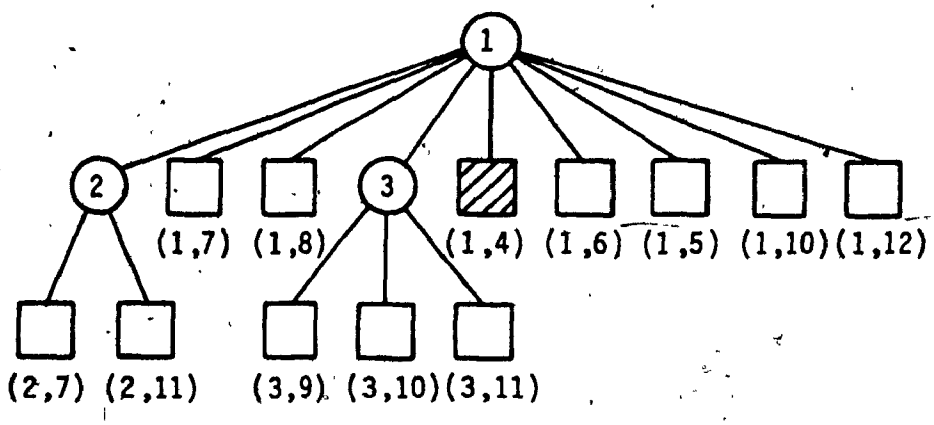


Figure 7.33

PQ-tree $T_3 = T_3^*$

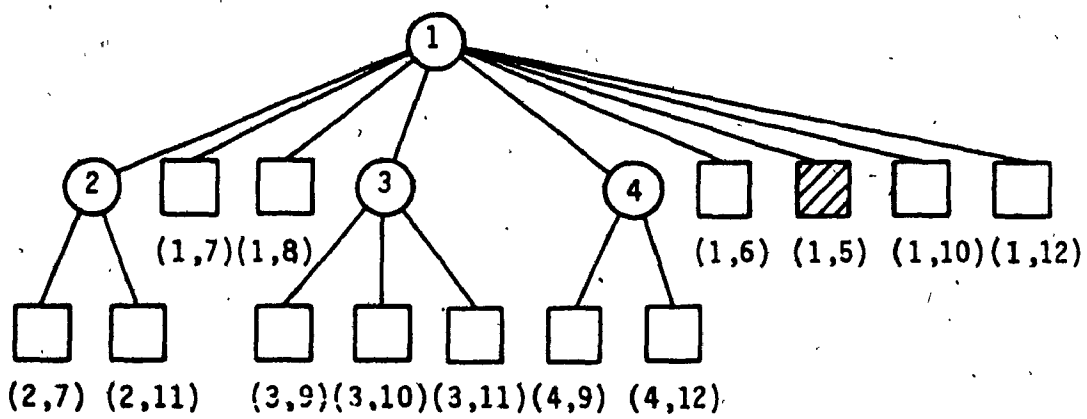


Figure 7.34

PQ-tree $T_4 = T_4^*$

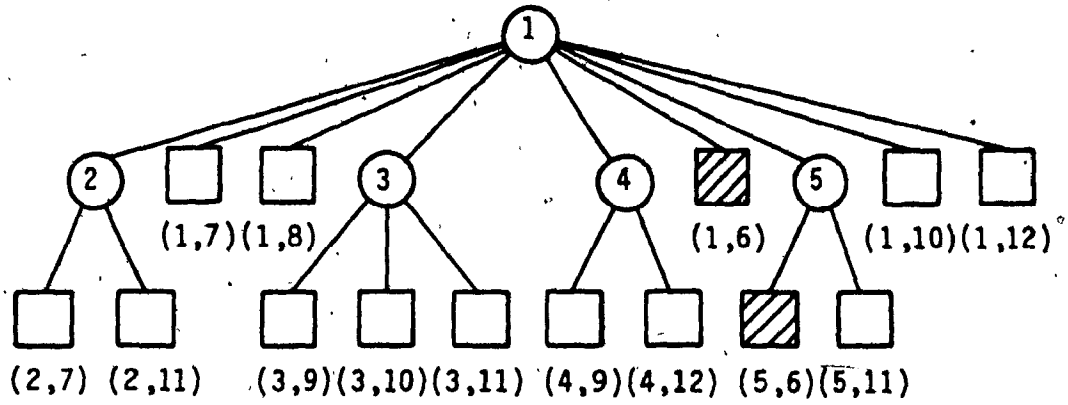


Figure 7.35(a)

PQ-tree T_5

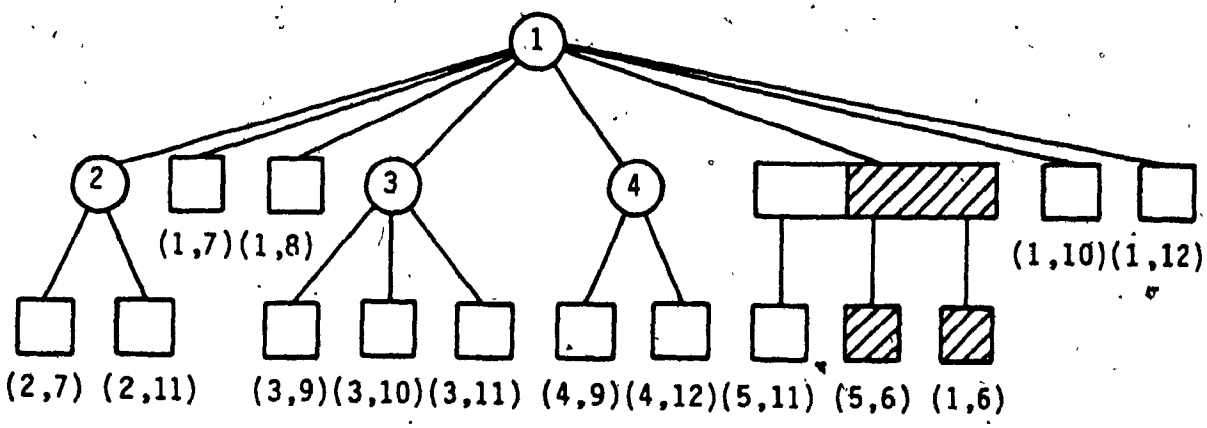


Figure 7.35(b)

PQ-tree T_5^*

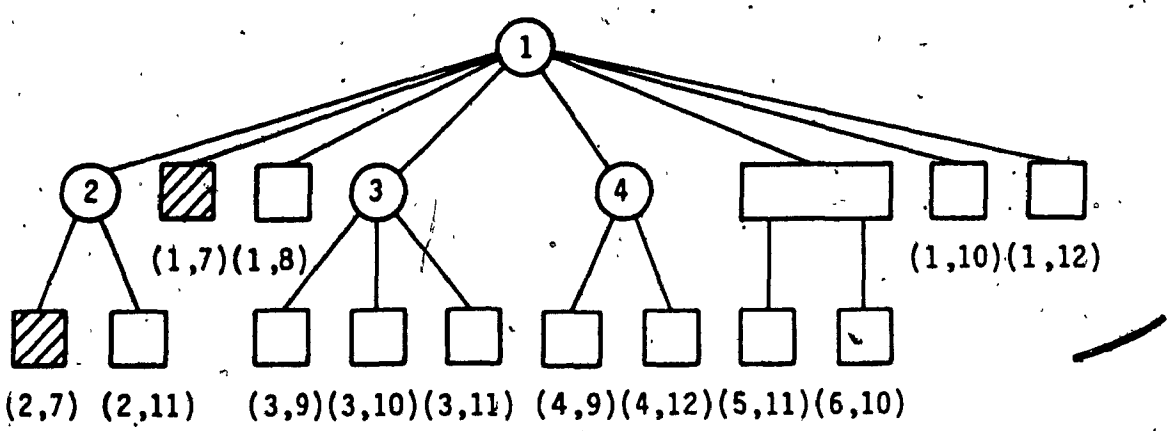


Figure 7.36(a)

PQ-tree T_6

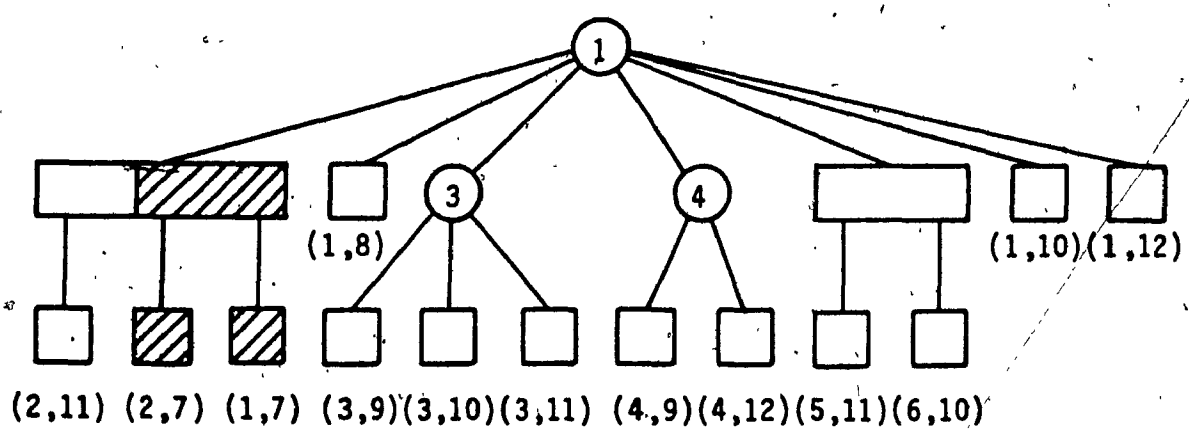


Figure 7.36(b)

PQ-tree T_6^*

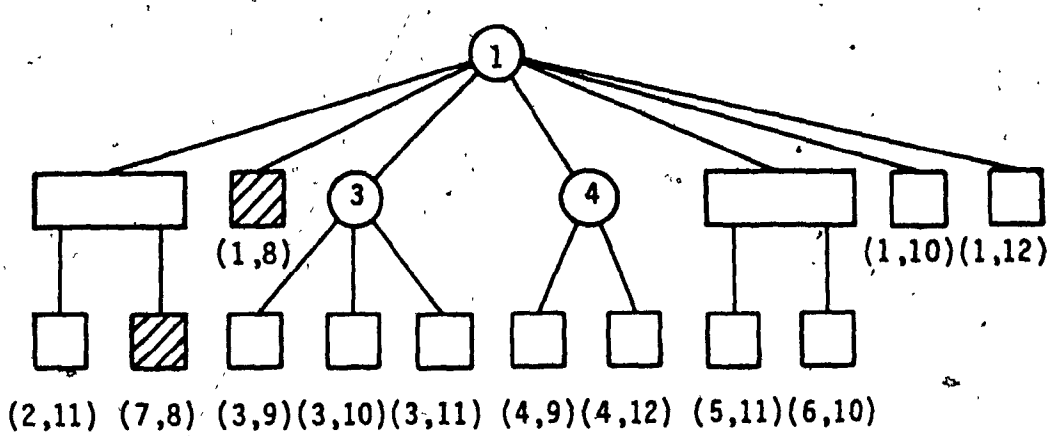


Figure 7.37(a)

PQ-tree T_7

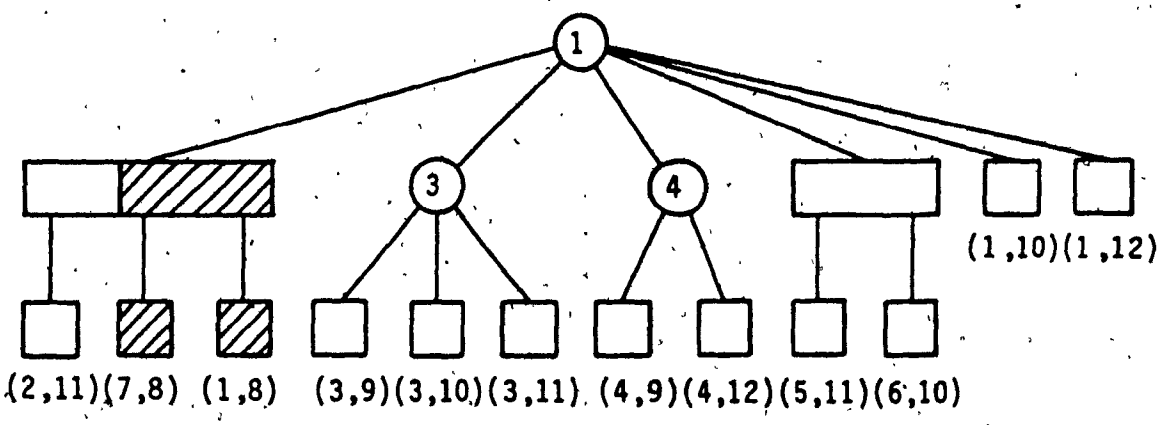


Figure 7.37(b)

PQ-tree T_7^*

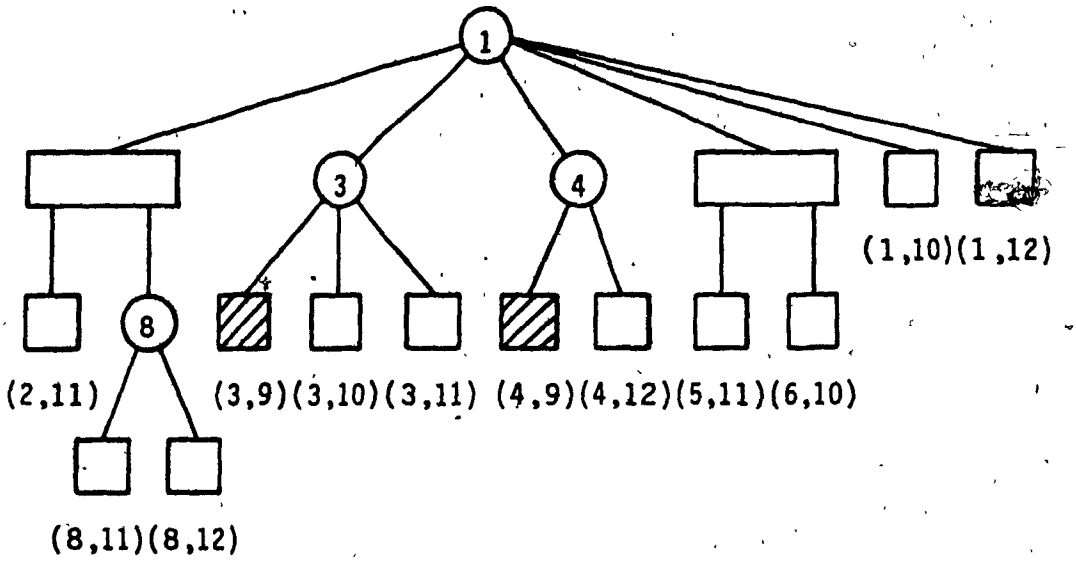


Figure 7.38(a)

PQ-tree T_8

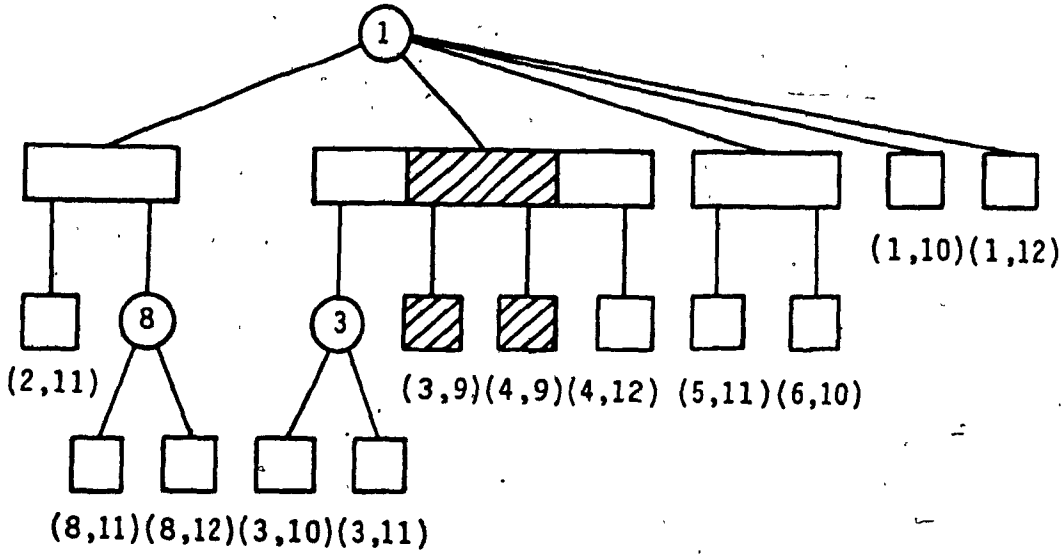


Figure 7.38(b)

PQ-tree T_8^*

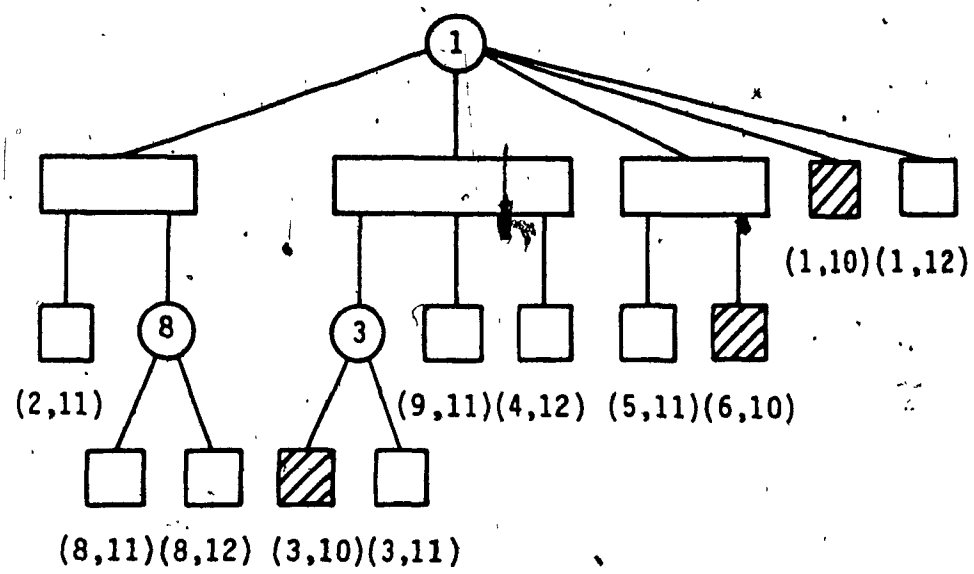


Figure 7.39(a)

PQ-tree T_9

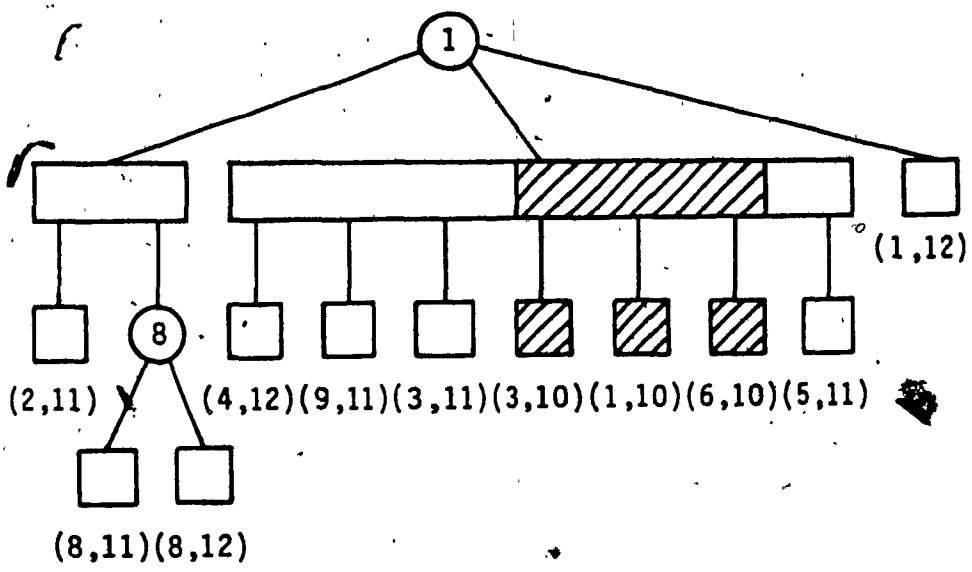


Figure 7.39(b)

PQ-tree T_9^*

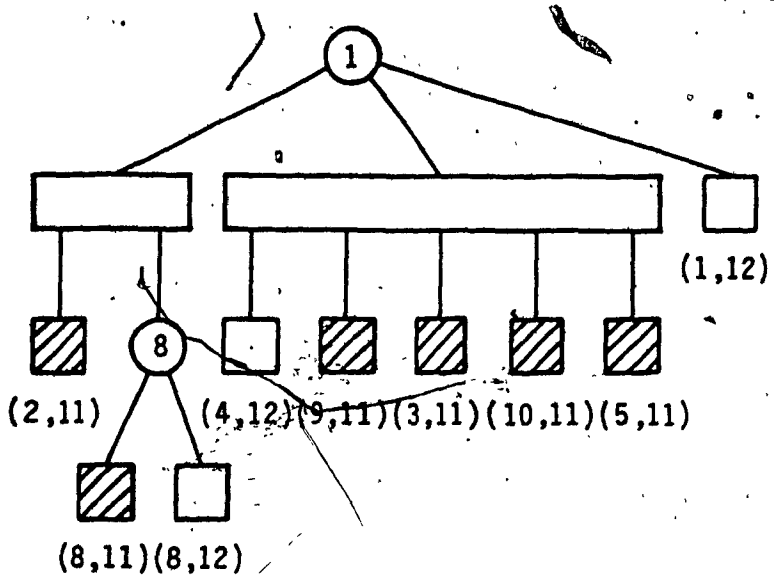


Figure 7.40(a)

PQ-tree T_{10}

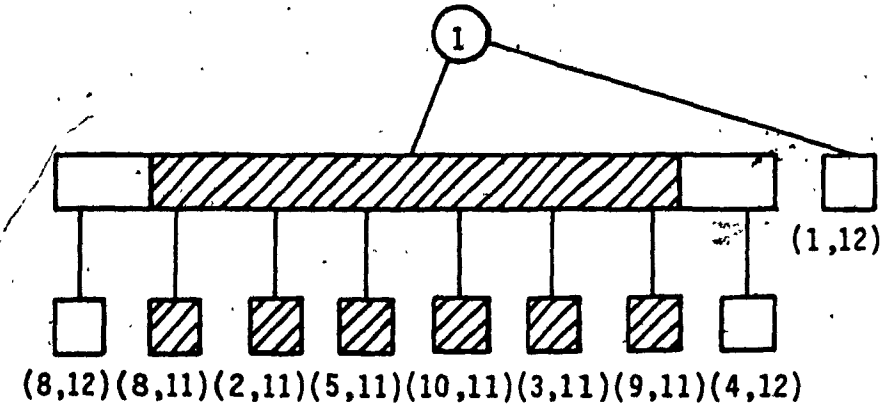


Figure 7.40(b)

PQ-tree T^*_{10}

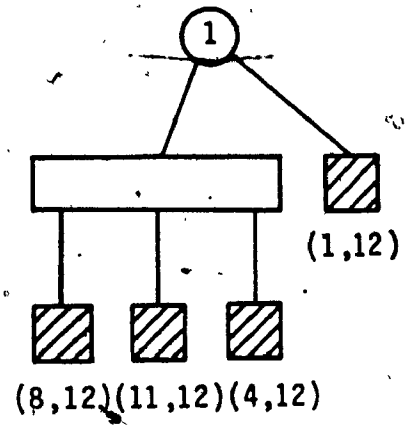


Figure 7.41

PQ-tree T_{11}

are obtained, G is a planar graph. From these trees we can observe that in order to test for planarity, we need not keep all the pertinent leaves consecutive in any PQ-tree. If all the pertinent leaves in T_k can be made consecutive, then the position for the P-node k+1 is what we need to construct T_{k+1}. Using this observation, we can simplify the templates. But we prefer to retain the templates as they are, for reasons which will become clear in later chapters.

Booth and Lueker [42] implemented the above algorithm in such a way that only the nodes in the pruned pertinent subtree are processed during the template matching process. They perform the reduction in a reduction phase. It is clear that to perform the reduction process, all the pertinent nodes in the tree should be known in advance. This information is obtained during a separate pass of the algorithm called the bubble-up phase. Moreover, in order to obtain an efficient implementation, Booth and Lueker keep parent pointers for all the children of a P-node; but only endmost children of Q-nodes are given valid parent pointers and all the children of Q-nodes are provided with sibling pointers. If, during the reduction process, any internal child of a Q-node becomes pertinent, then it should be provided with a valid parent pointer for template matching. This parent pointer assignment is also performed during the bubble-up phase. Moreover, in certain cases non-reducibility of a PQ-tree can be detected during the bubble-up

phase itself. We will not pursue these details any further. A complete discussion is available in [42].

It is easy to observe that the reduction of a PQ-tree requires time proportional to the number of pertinent nodes in it. Using this observation, Booth and Lueker [42] proved that, when implemented using PQ-trees, the LEC algorithm requires $O(m+n)$ time. Since for any planar graph $m = O(n)$, the time complexity of this algorithm is $O(n)$ for planar graphs.

As we have stated at the beginning of this section, the data structure PQ-tree was invented to represent the class of all the permutations of a set in which all the elements in certain subsets of the set appear together. Using the PQ-trees, Booth and Lueker [42] developed efficient algorithms to test for the consecutive ones property of matrices, and to test for interval graphs in addition to the implementation of the LEC algorithm discussed above. Recently, PQ-trees have been used in solving a wide variety of graph problems. Fujishige [47] used the ideas of PQ-trees to solve a graph realization problem. Ohtsuki and Mori [48] used PQ-tree algorithms to obtain an interval graph from a given graph by adding a minimum number of edges to it. Ozawa and Takahashi [49] developed an algorithm using PQ-trees to obtain a planar subgraph, which contains as many edges as possible, of a nonplanar graph. (However,

we show in Chapter 9 that this algorithm may not find a maximal planar subgraph in some cases.) In the graph representation of electronic circuits which contain integrated circuit components, certain vertices (representing the pins in the integrated circuits) must appear in a specified order. Testing planarity of such graphs is called constrained planarity testing. Masuda, Kashiwabara and Fujisawa [50] and Nakajima and Sun [51] extended the ideas of PQ-trees and introduced PQR-trees and PQS-trees respectively to develop a linear time algorithm for the constrained planarity testing problem.

In the following chapters, we develop efficient algorithms using PQ-trees to

(i) obtain a planar embedding of a planar graph, and

(ii) obtain a maximal planar subgraph of a nonplanar graph.

CHAPTER 8

A O(n) VERTEX-EDGE ORDERING ALGORITHM FOR PLANAR EMBEDDING

This chapter is concerned with the problem of obtaining a planar embedding of a planar graph.

One of the earliest algorithms to construct a planar embedding of a planar triconnected graph G was proposed by Tutte [52]. His "barycentric" embedding algorithm first finds a cycle C, called the peripheral polygon, in G and embeds this peripheral polygon as a regular convex polygon. Then a planar embedding of G is constructed by forming and solving systems of simultaneous linear equations which express the position of each vertex not in C as the centroid of its neighbours. The formulation and solution of these systems requires $O(n^3)$ time and $O(n^2)$ space in general. Once the coordinates of the vertices are determined, the edges may be embedded as straight-line segments.

Later, Woo [53] presented another algorithm to obtain a planar embedding of a planar graph in which all the edges are drawn as straight-line segments. Although his algorithm drew all planar graphs with upto 22 cycles, it failed for larger graphs. In the event of failure of his algorithm, Woo suggested a heuristic procedure involving man-machine communication to obtain the planar embedding. Koppe [54]

developed a completely automatic algorithm to obtain a planar embedding in which the edges are drawn as straight-line segments.

Wing [55] and Fisher and Wing [39] presented an algorithm to construct a planar embedding when the positions of the vertices in the plane are arbitrarily specified. However, in this algorithm it may be necessary to redraw some of the previously embedded subgraphs in such a way that a cut vertex appears on the outside window. Recently, Maly [56] developed another algorithm to obtain a planar embedding of a planar graph whose vertices are already placed in the plane. A computer program to draw electronic circuit diagrams in the plane has been reported by Hope [57].

As discussed in Chapter 7, there are two efficient O(n) time algorithms to test the planarity of a graph G with n vertices, namely Hopcroft and Tarjan's path addition algorithm and Lempel, Even, and Cederbaum's vertex addition algorithm (in short, the LEC algorithm). These algorithms test G for planarity by trying to construct an embedding of G in the plane. Tarjan [58] shows that his planarity testing algorithm can be used to obtain a planar embedding and gives the details of how to do this "by hand". He calls the planar embedding which his planarity testing algorithm constructs as a "standard embedding". Recently Williamson

[59] presented a procedure to construct a planar embedding of a planar graph based on the ideas of Hopcroft and Tarjan's planarity testing algorithm. However, no work has yet been reported on constructing a planar embedding using the LEC algorithm.

Brehaut [60] proposed an algorithm to find a planar mesh of a planar graph G in O(n) time and space using the ideas of Hopcroft and Tarjan's planarity testing algorithm. He also pointed out that using this mesh as the peripheral polygon in Tutte's barycentric mapping algorithm, a planar embedding of G can be obtained if G is triconnected. Later, Brehaut [61] presented an algorithm to find the coordinates of all the vertices of G in a planar embedding in $O(n^2)$ time and O(n) space. Unfortunately, one of the steps in this algorithm is computationally difficult and Brehaut suggested a heuristic to implement this step.

In this chapter we discuss the problem of obtaining a planar embedding of a planar graph G using the LEC algorithm. We develop an O(n) time algorithm to determine the positions of the vertices in a planar embedding of G. We also develop another O(n) time algorithm to determine the order in which the edges should be drawn around a vertex so that an intersection-free drawing of the edges can be achieved. Finally, we describe a procedure to obtain a planar embedding "by hand".

8.1 Bush Forms and τ-order

In this section we first discuss the principle underlying our procedure for drawing a planar embedding of a planar graph G using the different bush forms constructed by the LEC algorithm. We then draw attention to certain problems that will be encountered in a straightforward implementation of this procedure. In the subsequent sections we shall develop algorithms to overcome these problems.

Let $G = (V,E)$ be an n-vertex planar st-graph. Since replacing the edges incident on a vertex of degree two by a single edge does not affect the planarity of G, we assume, without loss of generality, that every vertex in G has degree at least three. We may recall that the st-graph G can be considered as a directed graph in which each edge is oriented from its lower numbered end vertex to the higher numbered end vertex. For any vertex i, $2 \leq i \leq n$, let $\Gamma^+(i)$ be the set of lower numbered neighbours of i. Let $B_1 = B_1'$, B_2, B_2', ..., B_i, B_i', ..., B_{n-1} be the sequence of bush forms generated by the LEC algorithm. Recall that in the bush form B_i, the virtual vertices labeled (i+1) may not be appearing consecutively whereas in B_i' these virtual vertices appear consecutively. Let T_i be the PQ-tree representing B_i. Note that the PQ-tree implementation of the LEC algorithm does not explicitly construct the PQ-tree

corresponding to B'_i. Rather, starting with T_i, it constructs a PQ-tree T^*_i from which T_{i+1} can easily be obtained.

Consider now the virtual edges entering vertex i in B'_{i-1}. The left-to-right order of these edges imposes an anticlockwise order around i among the vertices in $\Gamma^+(i)$. We call this order as the τ-order in B_i for vertex i. In general, the τ-order for vertex i in a planar embedding of G will refer to the anticlockwise order around i of the edges entering i from lower numbered vertices as well as to the corresponding order of the lower numbered vertices. The τ-order for vertex i in B_i will be denoted by $\tau(i)$. Note that in the PQ-tree T^*_{i-1}, the pertinent leaves corresponding to the virtual edges entering vertex i in B'_{i-1} appear consecutively as children of the pertinent root in the same left-to-right order as the virtual edges appear in B'_{i-1}. Note also that the pertinent root is a Q-node provided $|\tau(i)| > 1$. So, if (v_1,i), (v_2,i), ..., (v_j,i), $j \geq 1$, is the left-to-right order of these pertinent leaves in T^*_{i-1}, then $\tau(i) = (v_1, v_2, \ldots, v_j)$. Thus $\tau(i)$ for each i can be constructed from the corresponding T^*_{i-1}. For example, from the PQ-tree T^*_8 shown in Fig. 7.38(b), we get $\tau(9) = (3,4)$.

In T^*_{n-1}, the leaf corresponding to the virtual edge (1,n) is a child of the P-node labeled 1. Since each vertex of G has degree at least three, all the other children of

this P-node are Q-nodes. These Q-nodes can be merged into a single Q-node because all of them are full Q-nodes. The order of all the edges incident into vertex n, except the edge (1,n), is determined by the left-to-right order of their appearance as children of this new Q-node. Let (v_1,n), (v_2,n), ..., (v_j,n), $j \geq 1$, be this order. The edge (1,n) has the freedom to appear either on the left or on the right of this sequence of edges. Moreover, vertex 1 will appear in the τ-order of some other vertex less than n. So we decide to omit vertex 1 from $\tau(n)$, and write $\tau(n)$ as $\tau(n) = (v_1, v_2, \ldots, v_j)$. For example, from the PQ-tree T^*_{11} shown in Fig. 7.41 we obtain $\tau(12) = (8, 11, 4)$.

Note that the bush form B_{i-1}, $2 \leq i \leq n$, contains a planar embedding of G_{i-1}, the subgraph of G induced by the vertices 1, 2, ..., i-1. In this planar embedding the vertices 1, 2, ..., i-1 are placed at different vertical levels such that vertices with higher labels appear at higher levels. Also, all the edges incident into vertex i in this planar embedding enter from below and $\tau(i)$ specifies the anticlockwise order around vertex i of these edges. Using these observations we can draw a planar embedding of G from the τ-orders of its vertices as follows.

We start the embedding by placing vertex 1 at the lowest vertical level, say Level 1. This represents a planar embedding of G_1 and we now call vertex 1 as

"embedded". We then place vertex 2 at Level 2 higher than Level 1. Since $\tau(2) = 1$, we draw an edge between vertex 1 and vertex 2 and obtain a planar embedding of G_2. In general, suppose we have embedded the vertices 1, 2, ..., i-1. Then we can embed vertex i as follows. First we need to obtain B'_{i-1} (and hence $\tau(i)$) from the bush form B_{i-1}. This could be achieved by using a sequence of flippings and permutations of the maximal biconnected components in B_{i-1}. Let $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(j)}$ be the maximal biconnected components in B_{i-1} other than the virtual edges. We call these maximal biconnected components as <u>blocks</u>. Since the planar embedding of G_{i-1} contains planar embeddings of the blocks $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(j)}$, it follows that if we flip and/or permute a set of blocks in B_{i-1} to obtain B'_{i-1}, then the same flippings and/or permutations are performed on the planar embeddings of these blocks in G_{i-1} also. Clearly, the resulting drawing is also a planar embedding of G_{i-1} and the vertices in $\Gamma^+(i)$ get arranged around i as in $\tau(i)$. Thus we can embed vertex i by placing it at Level i higher than Level i-1 and drawing all the edges entering vertex i in the anticlockwise order specified by $\tau(i)$ such that these edges do not intersect any of the edges already drawn.

Consider, for example, the planar embedding of G_9 shown in Fig. 8.1. This is obtained from the bush form B_9 shown in Fig. 7.3. This planar embedding has three blocks C_1, C_2 and C_3 induced by the vertex sets {1, 2, 7, 8}, {1, 3, 4, 9}

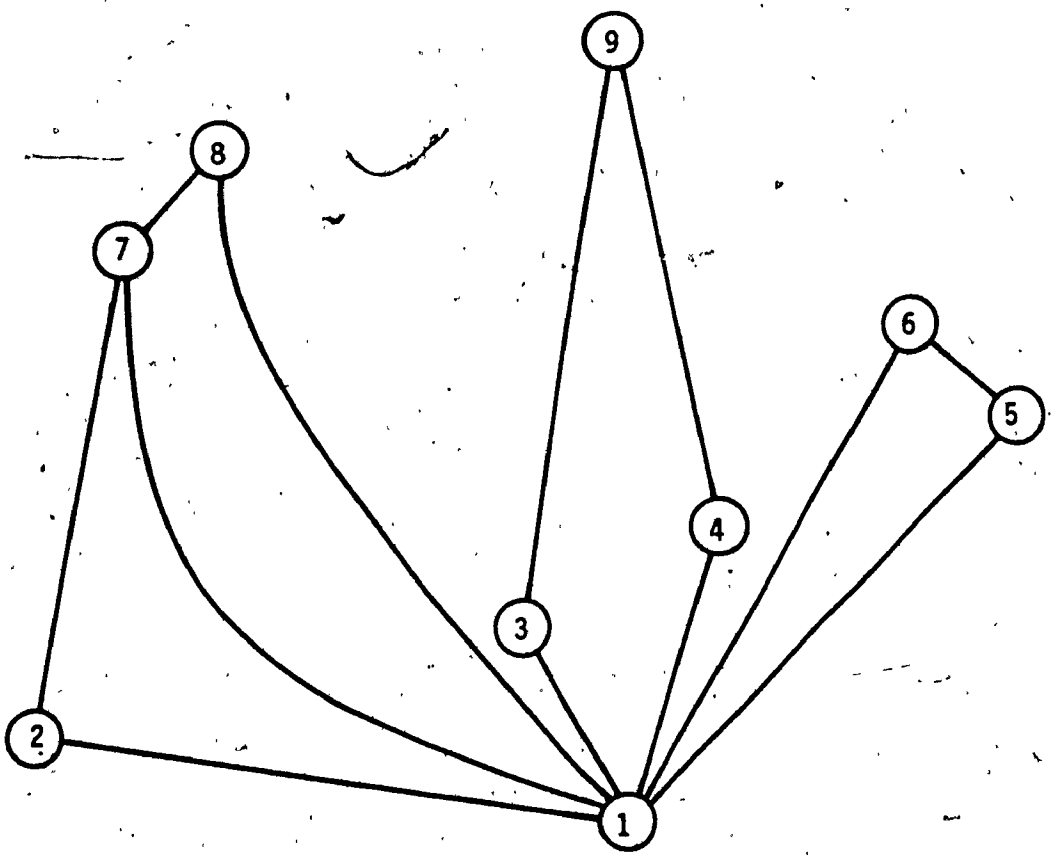


Figure 8.1

Planar Embedding of G_9 in B_9

and {1, 5, 6} respectively. We obtain B_9' (see Fig. 7.4) by flipping C_2 in B_9 and so we flip the planar embedding of C_2 in that of G_9 also. This new planar embedding is shown in Fig. 8.2. From this new planar embedding of G_9 we obtain the planar embedding of G_{10} by drawing the edge (3,10) first and then the edges (1,10) and (6,10) in that order since $\tau(10) = (3, 1, 6)$. This planar embedding of G_{10} is shown in Fig. 8.3. Note that this is the planar embedding of G_{10} contained in the bush form B_{10} shown in Fig. 7.14(a).

Embedding the vertices 2, 3, ..., n in that order as described above, we can eventually obtain a planar embedding of G. However, the above procedure is not elegant nor is it efficient. First of all when we embed vertex i, we may have to redraw some portions of G_{i-1} corresponding to the blocks which are flipped and/or permuted. Thus we may have to redraw certain portions many times before we obtain a planar embedding of G. Moreover, for larger graphs this redrawing is a very cumbersome process. Note that when a block $C_{i(k)}$ in G_{i-1} is involved in a permutation, its position in the final embedding relative to other blocks is affected. Also, when $C_{i(k)}$ is flipped, the τ-orders of all the vertices in $C_{i(k)}$ are reversed. Furthermore, $C_{i(k)}$ will be involved in several permutations and/or flippings before the final embedding of G is obtained. So, if we wish to avoid the redrawings required by the above straightforward procedure, then we should not attempt drawing until all the bush forms

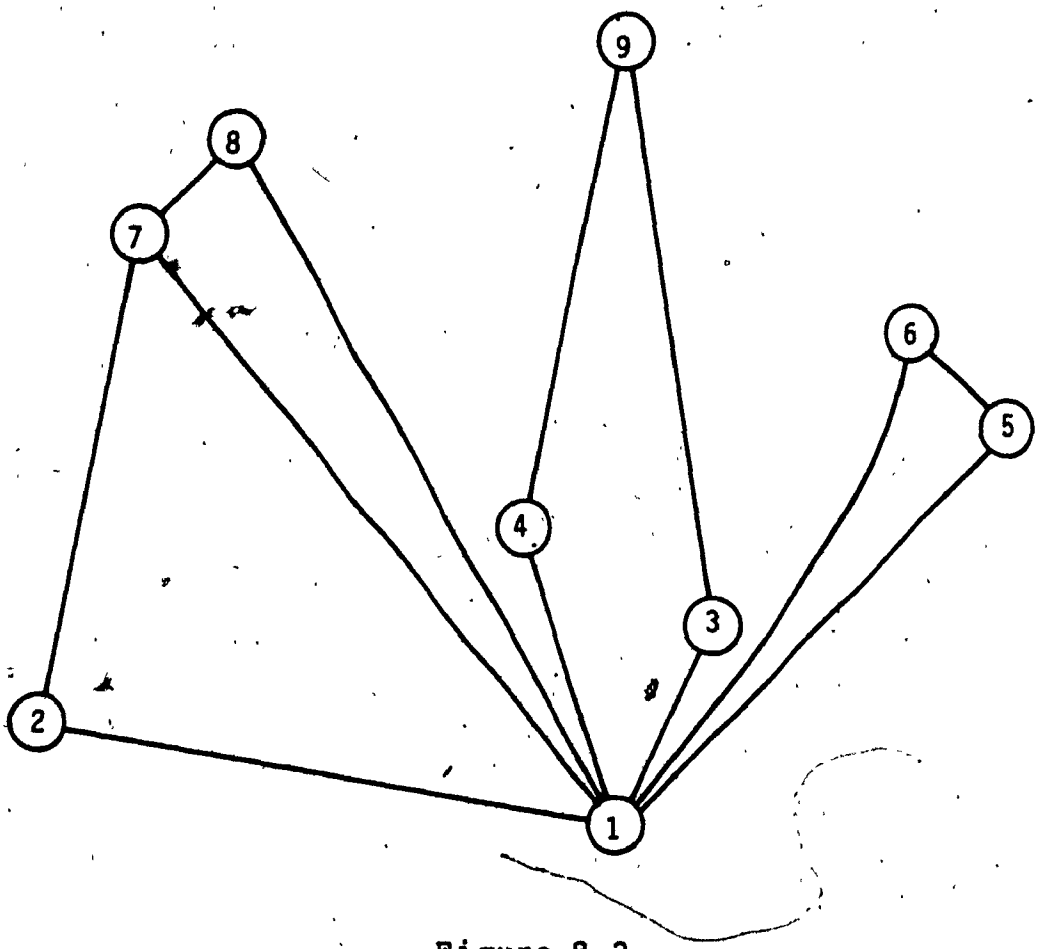


Figure 8.2

Planar Embedding of G_9 after Flipping

the Block Containing Vertices

1, 3, 4, and 9

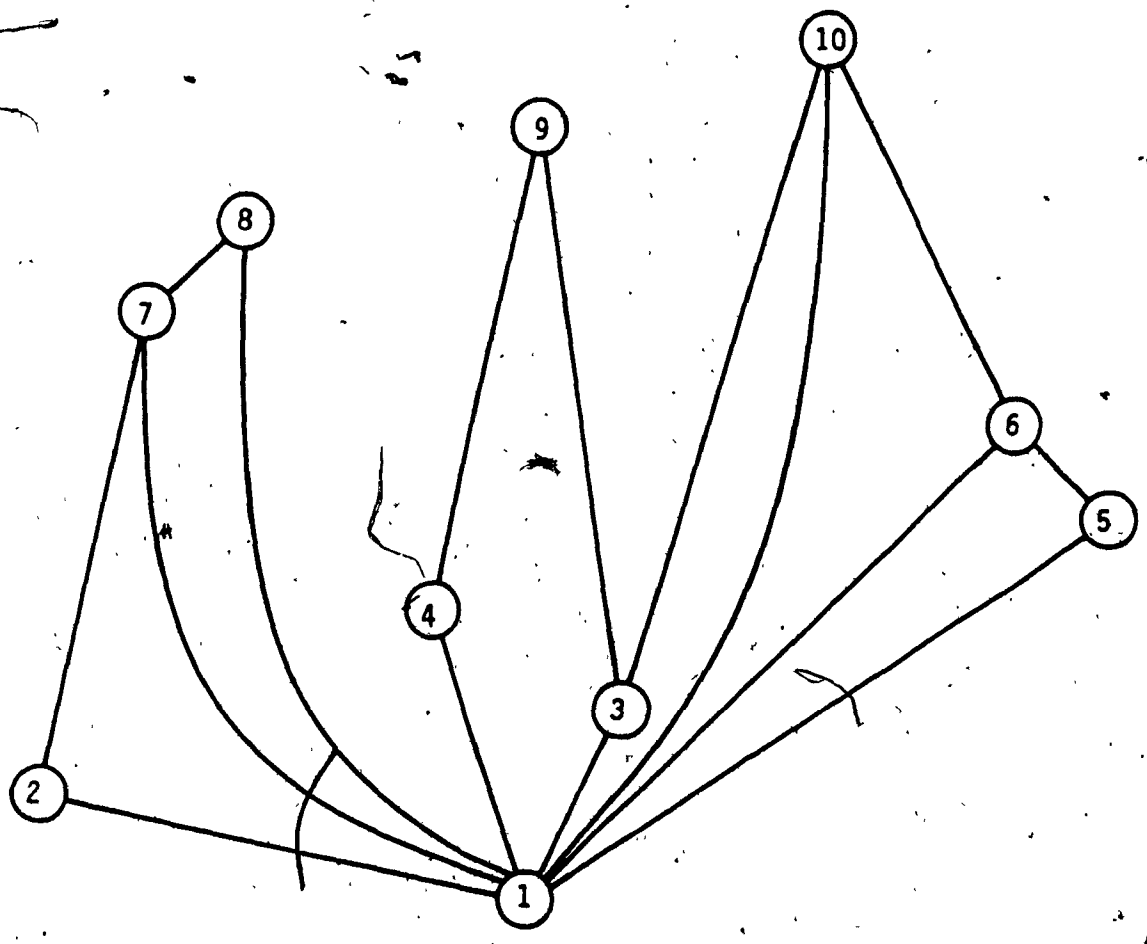


Figure 8.3

Planar Embedding of G_{10} obtained from that of G_9

are obtained. As we construct these bush forms, we should extract adequate information to enable us to obtain the relative locations of all the vertices in the final embedding of G.

As pointed out earlier, the τ-order for a vertex i gets reversed whenever a block containing i is flipped while embedding vertices greater than i. Thus the τ-order of vertex i in the final embedding of G may not be the same as $\tau(i)$. In Section 8.2 we develop an algorithm to obtain the τ-orders for all the vertices in a planar embedding of G. In our discussion thus far; we have assumed that the vertices appear at different vertical levels in the final embedding. Without loss of generality, let us also assume that no two distinct vertices of G appear on the same horizontal level. Then by scanning such an embedding left-to-right we can also obtain a horizontal order of the vertices of G. Let us call this horizontal order the <u>vertex order</u>. In Section 8.3 we develop a procedure to obtain a vertex order from the τ-orders of all the vertices in the final planar embedding of G.

Finally, let us consider the way the edges entering vertex i should be drawn. While $\tau(i)$ specifies the anticlockwise order around vertex i in which the edges entering vertex i should be drawn, unfortunately, this condition alone will not result in an intersection-free

drawing. For example, consider Fig. 8.4. Here $\pi(6) = (1, 5)$. So the edges (1,6) and (5,6) have to be drawn in that order. If these edges were drawn as shown in Fig. 8.4, then when vertex 9 is embedded at a later time, there is no way the edge (4,9) can be drawn without intersecting some of the edges already drawn. To avoid this problem, we should have drawn the edges (1,6) and (5,6) as shown in Fig. 8.5. This example shows that to obtain an intersection-free drawing, the edges should also be drawn in an appropriate way if we wish to avoid redrawing any of the edges already drawn. In Section 8.3 we study this question further and present a procedure to draw the edges of G.

8.2 Block Graph and τ-order

As discussed in Section 8.1, before we start drawing a planar embedding of a planar graph G, we would like to determine the τ-order of every vertex in the final embedding of G. This would help us in obtaining a planar embedding without redrawing any portion already embedded. In this section we first discuss how the blocks are formed during the bush growing process and then develop an O(n) time algorithm to determine the final τ-order of each vertex using the information obtained during the bush growing process.

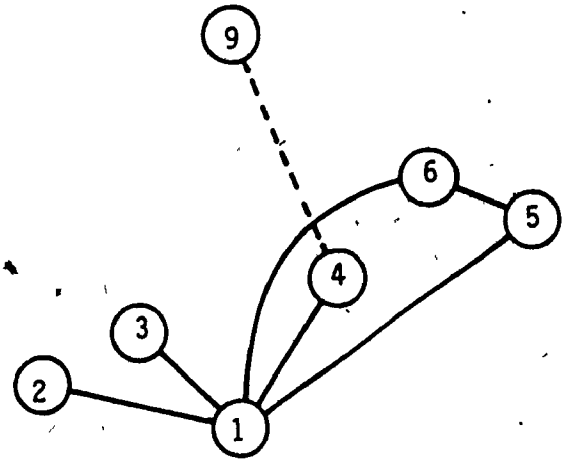


Figure 8.4

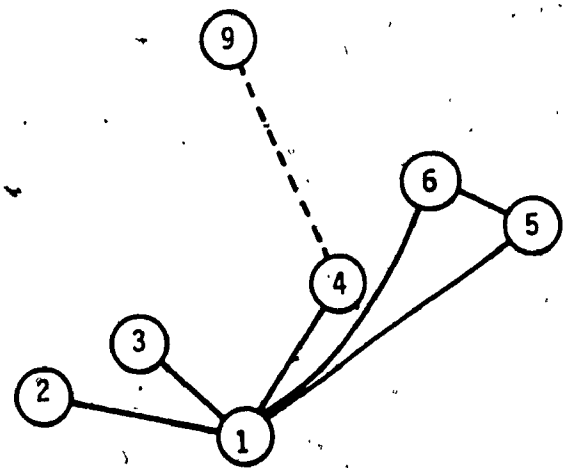


Figure 8.5

Consider the bush form B'_{i-1}. We know that the virtual edges entering vertex i emerge from vertices on the outside window of the maximal biconnected components, or blocks, of B_{i-1}. Let $C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$ be the blocks from which these virtual edges emanate. The τ-order $\tau(i)$ induces an anticlockwise order around i among these blocks. For any two virtual edges (x,i) and (y,i) emanating from distinct blocks $C_{i(r)}$ and $C_{i(s)}$ respectively, let us assume that $r < s$ if x is to the left of y in $\tau(i)$ so that $C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$ is the anticlockwise order around i of these blocks in the bush form B'_{i-1}.

Let $v_1^j, v_2^j, \ldots, v_p^j, v_1^j$ be the sequence of vertices on the outside window of block $C_{i(j)}$ in B'_{i-1} when the outside window is traversed in the clockwise direction from the lowest vertex v_1^j of $C_{i(j)}$. Let (v_α^1, i) and (v_β^k, i) be respectively the first and the last virtual edges entering vertex i in B'_{i-1}. When B_i is formed by merging the virtual edges entering vertex i in B'_{i-1}, a new block is formed. In this new block the vertices $v_1^1, v_2^1, \ldots, v_\alpha^1, i, v_\beta^k, v_{\beta+1}^k, \ldots, v_r^k$ will appear in that order on the outside window. Since i is the highest vertex in this new block, we number it as block i and denote it by C_i.

During the bush growing process, several blocks of B_{i-1} may merge to form C_i. These blocks are precisely those represented by the Q-nodes in the pertinent subtree of T_{i-1}.

Such blocks will be considered to be enclosed by C_i. Clearly, C_i encloses $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$. For example, the planar embedding of G_9 shown in Fig. 8.2 consists of the blocks C_6, C_8 and C_9. In Fig. 8.3, the block C_{10} is obtained by merging the blocks C_6 and C_9. Thus the block C_{10} encloses C_6 and C_9. Now we prove the following.

THEOREM 8.1.

If C_i encloses the blocks $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$, then $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$ will not be blocks in the bush forms B_i, B_{i+1}, ..., B_{n-1}.

Proof:

Clearly C_i is a block in B_i and $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$ are all subgraphs of C_i. Since a block is a maximal biconnected component, it follows that $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$ will not be present as blocks in the bush forms B_i, B_{i+1}, ..., B_{n-1}. □

The above theorem implies that G_{i-1} will contain at most (i-2) blocks.

While growing the bushes, a block C_i may be involved in several permutations and flippings. Permutations do not affect the τ-order of any vertex in C_i. On the other hand, flipping a block C_i reverses the τ-order of i. Furthermore, if C_i encloses C_j, then the τ-order of j will also be reversed whenever C_i is flipped. Our interest is to

determine the τ-order of each vertex in the final embedding of G. In other words, we wish to determine the status of a block, namely reversed or not, in the final embedding.

Let $\tau'(i)$ denote the τ-order of vertex i in the final embedding of G. If $\tau_{rev}(i)$ denotes the list obtained by reversing the list $\tau(i)$, then it can be seen that $\tau'(i)$ is equal to either $\tau(i)$ or $\tau_{rev}(i)$. We now develop an efficient algorithm to determine the τ'-order for each vertex.

First we discuss a way to represent the different blocks. If C_i is a block with only one edge, then $\tau(i)$ will have only one vertex in it. As a result, flipping C_i will have no effect on $\tau(i)$. In other words, for a block C_i with only one edge, $\tau'(i) = \tau(i)$. So in the following discussion we will be considering only those blocks which have at least three edges. Such blocks will be referred to as <u>nontrivial blocks</u> and the others will be called <u>trivial blocks</u>.

We represent the nontrivial blocks and the enclosing relation among them by a directed graph called a <u>block graph</u>. The vertices of the block graph represent the nontrivial blocks in the various bush forms. We denote the vertex representing block C_i as c_i, and with each vertex we associate a label. The label of vertex c_i is R if block C_i is reversed when the first block enclosing C_i is formed;

otherwise the label is NR. If block C_i encloses the blocks $C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$, then in the block graph we draw edges directed from vertex c_i to each one of the vertices $c_{i(1)}, c_{i(2)}, \ldots, c_{i(k)}$.

The block graph can be constructed as follows. Note that in a PQ-tree representing a bush form, a nontrivial block is represented by a Q-node. If $|T(i)| > 1$ for any i, $2 \leq i \leq n-1$, then in the PQ-tree T^*_{i-1} the pertinent root will be a Q-node and in later reductions this Q-node will represent the block C_i and so we assign the block number i to this Q-node. Thus in a PQ-tree (representing a bush form) each Q-node is assigned a block number, which is the number of the highest numbered vertex in the block. In the following we refer the reduction process transforming the tree T_{i-1} into T^*_{i-1} as reduction (i-1).

Suppose that the blocks $C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$ in the bush form B_{i-1} are merged to form the block C_i. Then the corresponding Q-nodes $Q_{i(1)}, Q_{i(2)}, \ldots, Q_{i(k)}$ will all be present in the pertinent subtree of T_{i-1}. During reduction (i-1) these nodes will be processed and merged into a single Q-node whose block number is i. Thus we can construct the block graph by adding an edge directed from vertex c_i to vertex c_j if the Q-node Q_j is processed during reduction (i-1). For example, the block graph of the planar st-graph G shown in Fig. 7.1 is given in Fig. 8.6. It can easily be

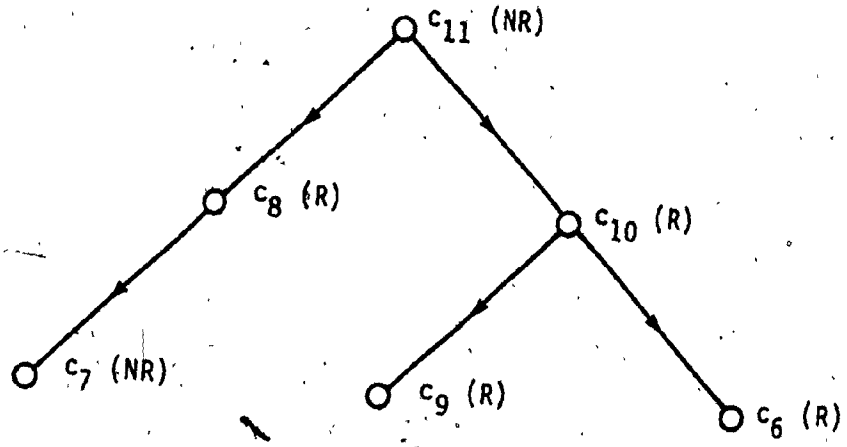


Figure 8.6

Block Graph

seen that the block graph is a rooted directed forest.

It now remains to determine the label of each vertex in the block graph. Consider now the reduction (i-1). Let C_i enclose the blocks $C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$. The label of each one of these blocks which C_i encloses indicates the status of that block, namely, whether the block is reversed or not in the embedding, when C_i is formed. To determine these labels, we should keep track of the flippings which the blocks suffer as the reduction (i-1) progresses. For this purpose, we construct what we call the i-th intermediate block graph which will be denoted by IBG(i). To start with, we add to IBG(i) nodes to correspond to the Q-nodes in the pertinent subtree of T_{i-1} and associate with each one of these nodes the label NR.

Suppose a node, say X, in T_{i-1} is being processed and that $Q_j, Q_k, \ldots, Q_\ell$ are the Q-nodes which are pertinent children of X. Consider the case that X is a P-node. Let Q_X be the Q-node created after the processing of X is completed. Now we add to IBG(i) a node, say q_X, to correspond to Q_X and add an edge directed from q_X to each one of the nodes representing $Q_j, Q_k, \ldots, Q_\ell$. On the other hand, if X is a Q-node, then IBG(i) will contain a node, say q_X, corresponding to X. Now, as before, we add to IBG(i) an edge directed from q_X to each one of the nodes corresponding to $Q_j, Q_k, \ldots, Q_\ell$. If any of the nodes $Q_j, Q_k, \ldots, Q_\ell$ is

reversed while processing X, then we change the label of the corresponding node in IBG(i) to R. It can be seen that IBG(i) is essentially a directed tree in which each leaf corresponds to a Q-node in T_{i-1} representing a block of G_{i-1}. The root of IBG(i) will represent the block C_i.

To determine the label of the vertices in the block graph representing the blocks $C_{i(1)}$, $C_{i(2)}$, ..., $C_{i(k)}$ enclosed by C_i, we proceed as follows. We traverse IBG(i) depth-first starting at its root. Suppose, during this traversal, we are at vertex y. If the label of y in IBG(i) is R, then we switch the labels of all the children of y in IBG(i). (By switching the label we mean setting the label to R if its current value is NR, or setting the label to NR if its current value is R.) At the end of the traversal of IBG(i), the label of a node will tell us whether the corresponding block enclosed by C_i is flipped in the embedding of C_i. These labels are then given to the corresponding vertices in the block graph.

The procedure for constructing IBG(i) and determining the final labels of its nodes can be formally presented as follows.

```
procedure FIND_LABEL_IBG(i);
comment procedure FIND_LABEL_IBG constructs the inter-
        mediate block graph IBG(i) during reduction (i-1).
```

```
        It also determines the labels of the blocks enclosed
        by C_i.

    procedure SET_LABEL(u);
    comment procedure SET_LABEL determines the labels of all
            the children of vertex u in IBG(i).
    begin
      for each child v of u in IBG(i) do
        begin
          if label of u is R
            then switch label of v;
          SET_LABEL(v)
        end
    end SET_LABEL;

begin
  {Construct IBG(i)}
  initialize IBG(i) to contain vertices corresponding to the
  Q-nodes in the pertinent subtree of T_{i-1};
  for each node X processed during reduction (i-1) which is
  not a leaf do
    begin
      if X is a P-node
        then add a new vertex q_X to IBG(i);
      {q_X is the vertex in IBG(i) representing node X}
      {Let Q_j, Q_k, ..., Q_l be the Q-nodes which are children
      of X in the pertinent subtree of T_{i-1}}
      draw edges directed from q_X to each one of the
```

```
        vertices corresponding to Q_j, Q_k, ..., Q_l in IBG(i);
        for each Q-node Q_r among Q_j, Q_k, ..., Q_l do
          if Q_r is reversed when node X is processed
            then label of Q_r := R
      end;
      {Determine the label of each node in IBG(i)}
      SET_LABEL(root of IBG(i))
end FIND_LABEL_IBG;
```

In the following theorem we present the complexity of the above procedure.

THEOREM 8.2.

Cost of procedure FIND_LABEL_IBG(i) is $O(N_i)$, where N_i is the number of pertinent nodes in T_{i-1}.

Proof:

It can be easily seen that the number of vertices in IBG(i) is no more than N_i, the number of pertinent nodes in T_{i-1}. Since IBG(i) is a directed tree, it has $O(N_i)$ edges. So the cost of constructing IBG(i) and the cost of traversal of IBG(i) to determine the labels of its vertices are both $O(N_i)$. The theorem follows since the procedure FIND_LABEL_IBG involves only these two costs. □

This completes the discussion of our procedure to construct the block graph. Note that the procedure involves

executing procedure FIND_LABEL_IBG for all i. In Fig. 8.6 we have shown within parentheses the label of each vertex in the block graph.

We now give a formal presentation in ALGOL-like notation of our procedure to construct the block graph. In this procedure, the labels of the vertices of the block graph are stored in the array STATUS.

```
procedure BLOCK_GRAPH;
comment procedure BLOCK_GRAPH constructs the block graph and
        stores the status information of each block during
        the PQ-tree reduction process. STATUS(i) represents
        the status of block C_i.
begin
  for i:= 2 to n-1 do
    begin
      {Construct the block graph and determine the status of
      the blocks}
      FIND_LABEL_IBG(i);
      for each pertinent Q-node in T_{i-1} do
        begin
          draw a directed edge from c_i to c_j, where j is the
          block number of the Q-node;
          STATUS(j) := label of the Q-node
        end;
      {Create the block C_i}
```

obtain T_i;
assign the block number i to the Q-node which is the pertinent root in T^*_{i-1};
STATUS(i) := NR
end
end BLOCK_GRAPH;

The following theorem shows that the above computations can be performed in O(n) time.

THEOREM 8.3.

Procedure BLOCK_GRAPH correctly constructs the block graph and determines the status information of each block in O(n) time.

Proof:

Correctness of the procedure follows from our discussion so far. To find the complexity, note that the cost of procedure BLOCK_GRAPH during reduction (i-1), exclusive of the cost for procedure FIND_LABEL_IBG(i), is proportional to the number of pertinent nodes in T_{i-1}. From Theorem 8.2 the cost of procedure FIND_LABEL_IBG(i) is proportional to the number of pertinent nodes in T_{i-1}. Hence the overall cost of procedure BLOCK_GRAPH is proportional to the number of pertinent nodes in T_{i-1}. Thus the complexity of procedure BLOCK_GRAPH is O(m+n) which is O(n) for a planar graph. □

Having obtained the block graph and the status of each block in it, we now proceed to find whether a block will be reversed in the final embedding of G or not. This will determine the τ'-order for each vertex. Note that block C_n will not be present in the block graph because it is not processed during any reduction. Also block C_i, $2 \leq i \leq n-1$ will be present in the block graph if and only if $|\tau(i)| > 1$. We determine the τ'-order by traversing the block graph in a depth-first way. Suppose we are at a vertex, say c_i, of the block graph. If the status of the block C_i is R, then all the blocks enclosed by C_i require flippings. These blocks are represented in the block graph by the children of c_i and so we update their status by switching their labels. No updating of the labels is required if the status of C_i is NR.

The following procedure FIND_STATUS determines the status of each block in the final embedding of G and $\tau'(i)$, $2 \leq i \leq n$. We begin the procedure by initializing all the blocks "not processed" and repeat the procedure until all the blocks are processed.

```
procedure FIND_STATUS;
comment procedure FIND_STATUS traverses the block graph in a
        depth-first way and obtains the status of each of
        the blocks in the final embedding of G. It also
        finds τ'(i), 2 ≤ i ≤ n.
```

```
procedure UPDATE_STATUS(i);
comment procedure  UPDATE_STATUS  determines the status of
        the blocks enclosed by block C_i and finds τ'(i).
begin
  set block C_i processed;
  for each child c_j of c_i in the block graph do
    begin
      if STATUS(i) = R
        then switch the status of block C_j;
      UPDATE_STATUS(j)
    end;
  if STATUS(i) = R
    then τ'(i) := reversed τ(i)
    else τ'(i) := τ(i)
end UPDATE_STATUS;

begin
  initialize all blocks "not processed";
  τ'(n) := τ(n);
  for i:= n-1 downto 2 do
    if |τ(i)| = 1
      then τ'(i) := τ(i)
      else if C_i is not processed
        then UPDATE_STATUS(i)
end FIND_STATUS;
```

As an example, in Fig. 8.7 we give the final status of

each block in the block graph shown in Fig. 8.6 and the τ'-orders for all the vertices in G obtained using the above procedure. The following theorem gives the complexity of procedure FIND_STATUS.

THEOREM 8.4.

Procedure FIND_STATUS determines $\tau'(i)$, $2 \leq i \leq n$, correctly in $O(n)$ time.

Proof:

Correctness of the procedure is easy to see. To find the complexity, note that the block graph is a forest and so the cost of procedure FIND_STATUS is proportional to the number of vertices in the block graph. The number of vertices in the block graph is at most n, the number of vertices in G and so procedure FIND_STATUS is of complexity $O(n)$. □

It can be easily seen that procedure BLOCK_GRAPH can be implemented along with the PQ-tree reduction procedure. Once the block graph is constructed and the status of the blocks are determined, procedure FIND_STATUS can be applied to the block graph to obtain the τ'-orders $\tau'(i)$, $2 \leq i \leq n$. In the next section we use these τ'-orders to obtain the vertex order.

Block	Status	τ-order	τ'-order
C_{12}	NR	(8,11,4)	(8,11,4)
C_{11}	NR	(8,2,5,10,3,9)	(8,2,5,10,3,9)
C_{10}	R	(3,1,6)	(6,1,3)
C_9	R NR	(3,4)	(3,4)
C_8	R	(7,1)	(1,7)
C_7	NR R	(2,1)	(1,2)
C_6	R NR	(5,1)	(5,1)
C_5	—	(1)	(1)
C_4	—	(1)	(1)
C_3	—	(1)	(1)
C_2	—	(1)	(1)

Figure 8.7

τ'-orders Obtained From Status Information

8.3 Vertex Order and Planar Embedding

In Section 8.2 we developed an O(n) time algorithm to determine the τ'-orders of the vertices of a planar graph G. In this section we discuss an efficient procedure to construct a planar embedding of G using these τ'-orders. The embedding scheme discussed in Section 8.1 places the vertices of G in the plane at different horizontal and vertical levels such that no two distinct vertices are placed in the same vertical or horizontal levels. Recall that the left-to-right order of the vertices of G in such a placement is called the vertex order. We shall denote it by μ. Note that the vertex order μ is to be such that if the vertices are placed at different horizontal levels as specified by it, then the edges from vertices in $\tau'(i)$, $2 \leq i \leq n$, entering vertex i can be drawn around i, entering i from below in the anticlockwise order specified by $\tau'(i)$. Now we develop an efficient algorithm to determine such a vertex order and discuss a method to draw a planar embedding of G.

We embed G in the plane by embedding the vertices 2, 3, ..., n in that order. By "embedding vertex i" we mean connecting i to its lower numbered neighbours using the order specified by $\tau'(i)$. Thus when vertex i is to be embedded, the lower numbered vertices 1, 2, ..., i-1 are already embedded. Some of these embedded vertices may be

adjacent to vertices greater than i in G. We shall call these vertices as Type 2 vertices relative to i. All the other vertices will be called Type 1 vertices relative to i. In the following we shall refer to these vertices as simply Type 2 and Type 1 vertices, respectively, if the context makes it clear that they indeed have these properties relative to vertex i.

We represent the vertex order μ as a doubly linked list. To start with μ contains the vertex n and we add the vertices in $\tau'(n)$, $\tau'(n-1)$, ..., $\tau'(2)$ to μ in that order. Whenever a vertex is placed in μ, we store the address of the element in μ corresponding to that vertex so that we can access any vertex in μ in constant time. When we add the vertices in $\tau'(i)$ to μ, vertex i should be already present in μ since i should be in $\tau'(j)$ for some $j > i$. Moreover, at this stage all the Type 2 vertices in $\tau'(i)$ will also be present in μ. Thus we can check whether a vertex is Type 2 or not by simply testing for its presence in μ. Furthermore, since all the Type 2 vertices in $\tau'(i)$ are already in μ, we need to add to μ only the Type 1 vertices in $\tau'(i)$.

Consider reduction (i-1) in which the PQ-tree T_{i-1} is transformed into the PQ-tree T^*_{i-1}. We know that when the pertinent root in T_{i-1} is processed, it can have at most two partial children (which are partial Q-nodes) but any number

of full children (some of which may be pertinent leaves). Just before the pertinent children of the pertinent root are merged to obtain T^*_{i-1}, one of the partial children should have its full children at its right end and the other should have its full children at its left end. We shall call these partial children as the Left Child and the Right Child respectively. All the other pertinent children of the pertinent root will be called Center Children. As mentioned before all the Center Children will be full. For example, for the planar graph G of Fig. 7.1, we have shown in Fig. 8.8 the PQ-tree T''_9 at the time the pertinent root of the PQ-tree T_9 is being processed. In this figure we have indicated the Left Child, Right Child and the Center Child of the pertinent root.

It is easy to see that in $\tau(i)$ the vertices corresponding to the pertinent leaves of the Left Child should appear consecutively and we denote this portion of $\tau(i)$ as $\tau_L(i)$. Similarly, the vertices in $\tau(i)$ corresponding to the Center Children and Right Child should appear consecutively and we denote these portions of $\tau(i)$ as $\tau_C(i)$ and $\tau_R(i)$ respectively. Thus $\tau(i) = (\tau_L(i), \tau_C(i), \tau_R(i))$ and at least one of $\tau_L(i)$, $\tau_C(i)$ and $\tau_R(i)$ is not empty for any i, $2 \leq i \leq n$. For example, from Fig. 8.8 we can see that $\tau_L(10) = (3)$, $\tau_L(10) = (1)$, $\tau_R(10) = (6)$ and so $\tau(10) = (\tau_L(10), \tau_C(10), \tau_R(10)) = (3, 1, 6)$. Note that $\tau_L(i)$, $\tau_C(i)$ and $\tau_R(i)$, $2 \leq i \leq n$, can easily be obtained during

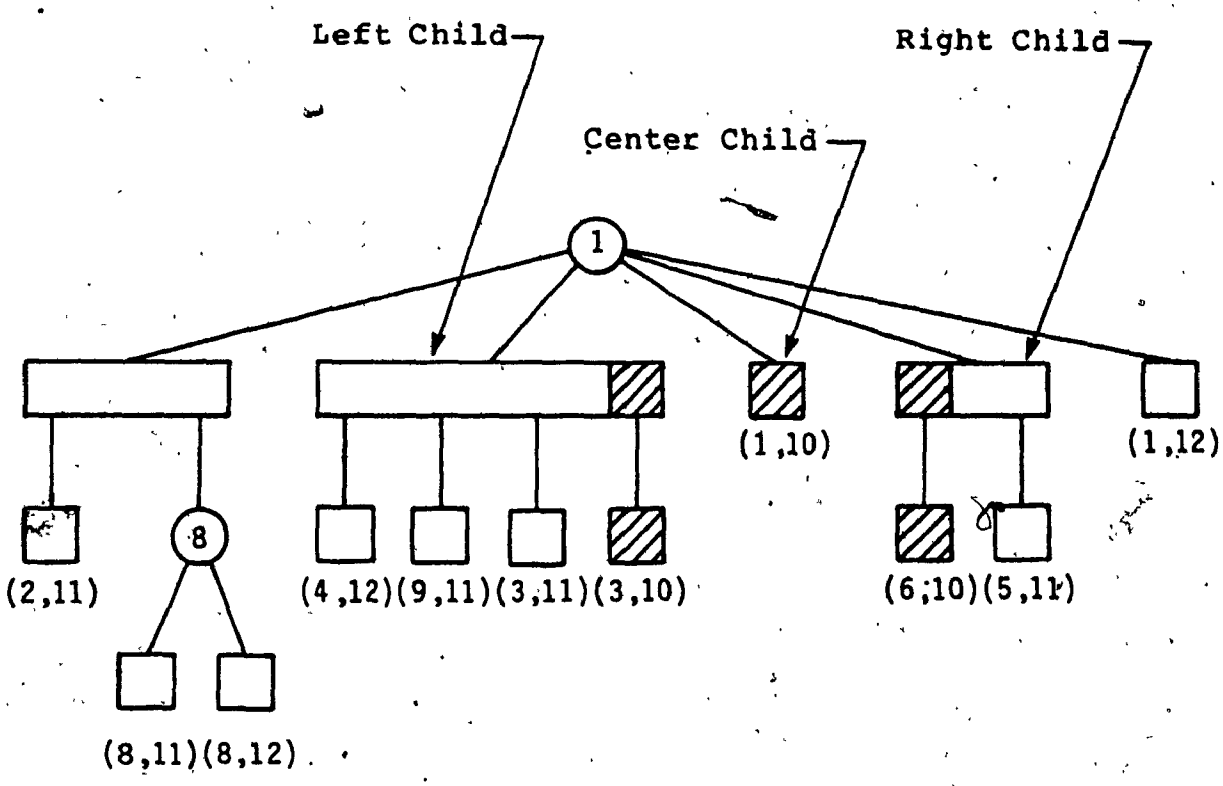


Figure 8.8

PQ-tree T_9''

$\tau_L(10) = (3)$, $\tau_C(10) = (1)$, $\tau_R(10) = (6)$

the PQ-tree reduction without increasing the computational complexity of the reduction procedure. Furthermore, if $\tau(i)$ is reversed to obtain the final τ-order $\tau'(i)$, then $\tau'_L(i)$, $\tau'_C(i)$ and $\tau'_R(i)$ will simply be the reversals of $\tau_R(i)$, $\tau_C(i)$ and $\tau_L(i)$, respectively. Hence $\tau'_L(i)$, $\tau'_C(i)$ and $\tau'_R(i)$ can be obtained in O(n) time using the algorithm discussed in Section 8.2. In our example, since the block C_{10} is reversed in the final embedding of G, $\tau'_L(10) = (6)$, $\tau'_C(10) = (1)$ and $\tau'_R(10) = (3)$. In Fig. 8.9 we show $\tau'_L(i)$, $\tau'_C(i)$ and $\tau'_R(i)$ for all the vertices i, $2 \leq i \leq n$, of the planar graph shown in Fig. 7.1.

We want to construct the vertex order μ such that for any vertex i, $2 \leq i \leq n$, all the vertices in $\tau'_L(i)$ will appear to the left of i in μ, and all the vertices in $\tau'_R(i)$ will appear to the right of i in μ. If the vertices are placed according to such a μ, then in the final embedding the blocks containing the vertices in $\tau'_L(i)$ will be on the left side of i and those containing the vertices in $\tau'_R(i)$ will be on the right side of i. A vertex order with this property would aid us in obtaining an elegant planar embedding, as we will discuss later. To construct such a μ, we place the Type 1 vertices in $\tau'_L(i)$ to the immediate left of vertex i, and the Type 1 vertices in $\tau'_R(i)$ to the immediate right of i as described in the following procedures.

Vertex i	$\tau'(i)$	$\tau'_L(i)$	$\tau'_C(i)$	$\tau'_R(i)$
12	(8,11,4)	–	(8,11,4)	–
11	(8,2,5,10,3,9)	(8,2)	–	(5,10,3,9)
10	(6,1,3)	(6)	(1)	(3)
9	(3,4)	(3)	–	(4)
8	(1,7)	–	(1)	(7)
7	(1,2)	–	(1)	(2)
6	(5,1)	(5)	(1)	–
5	(1)	–	(1)	–
4	(1)	–	(1)	–
3	(1)	–	(1)	–
2	(1)	–	(1)	–

Figure 8.9

τ'_L, τ'_C, τ'_R orders

```
procedure PLACE_LEFT(i);
comment procedure  PLACE_LEFT  places the Type 1 vertices in
        τ'_L(i) = (j_L.1, j_L.2, ..., j_L.p)  to  the  left  of
        vertex i in μ.
begin
  recently_placed_vertex := i;
  for x := |τ'_L(i)| downto 1 do
    if j_L.x is Type 1
      then begin
        place j_L.x to the immediate left of
        recently_placed_vertex;
        recently_placed_vertex := j_L.x
      end
end PLACE_LEFT;

procedure PLACE_RIGHT(i);
comment procedure  PLACE_RIGHT places the Type 1 vertices in
        τ'_R(i) = (j_R.1, j_R.2, ..., j_R.q)  to  the  right  of
        vertex i in μ.
begin
  recently_placed_vertex := i;
  for x := 1 to |τ'_R(i)| do
    if j_R.x is Type 1
      then begin
        place j_R.x to the immediate right of
        recently_placed_vertex;
        recently_placed_vertex := j_R.x
```

end

end PLACE_RIGHT;

After placing the vertices in $\tau'_L(i)$ and $\tau'_R(i)$, we place the Type 1 vertices in $\tau'_C(i)$ around vertex i in μ. We split these Type 1 vertices into two halves and place the first half to the left of vertex i and the second half to the right of vertex i in μ such that the left-to-right order of these vertices in μ is the same as in $\tau'_C(i)$. This is described in the following procedure.

procedure PLACE_CENTER(i);
comment procedure PLACE_CENTER places all the Type 1 vertices in $\tau'_C(i)$ around vertex i in μ so that in μ vertex i appears in the center of these Type 1 vertices.
begin
place all the Type 1 vertices in $\tau'_C(i)$ around vertex i in μ such that in μ i appears in the center of these vertices
end PLACE_CENTER;

Thus we can obtain the vertex order μ using the following procedure VERTEX_ORDER.

procedure VERTEX_ORDER;
comment procedure VERTEX_ORDER determines the vertex order from $\tau'(i) = (\tau'_L(i), \tau'_C(i), \tau'_R(i))$, $2 \leq i \leq n$.

```
begin
  initialize μ to contain the vertex n;
  for i:= n downto 2 do
    begin
      if τ'L(i) is not empty
        then PLACE_LEFT(i);
      if τ'R(i) is not empty
        then PLACE_RIGHT(i);
      if τ'C(i) is not empty
        then PLACE_CENTER(i)
    end
end VERTEX_ORDER;
```

We will illustrate in Fig. 8.10 the above procedure to find the vertex order for the graph of Fig. 7.1. In this figure we show the progressive growth of the vertex order as we add the vertices in $\tau'(i)$, $n \geq i \geq 2$.

We now prove that the vertex order constructed by procedure VERTEX_ORDER has the desired property.

THEOREM 8.5.

In the vertex order constructed by procedure VERTEX_ORDER, the vertices in $\tau'_L(i)$ will appear to the left of vertex i for any i, $2 \leq i \leq n$, and the vertices in $\tau'_R(i)$ will appear to the right of vertex i.

$\tau'(i)$ placed in μ	Vertex order μ
Initial	12
$\tau'(12)$	8,12,11,4
$\tau'(11)$	8,12,2,11,5,10,3,9,4
$\tau'(10)$	8,12,2,11,5,6,10,1,3,9,4
$\tau'(9)$	8,12,2,11,5,6,10,1,3,9,4
$\tau'(8)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(7)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(6)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(5)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(4)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(3)$	8,7,12,2,11,5,6,10,1,3,9,4
$\tau'(2)$	8,7,12,2,11,5,6,10,1,3,9,4

Figure 8.10

Finding Vertex Order

Proof:

If $|\tau'(i)| = 1$, then the only vertex in $\tau'(i)$ will be in $\tau'_C(i)$. Thus we need to consider only the case $|\tau'(i)| > 1$. Note that procedure PLACE_LEFT places all the Type 1 vertices in $\tau'_L(i)$ to the left of i in the vertex order μ in the same left-to-right order as in $\tau'_L(i)$. Also procedure PLACE_RIGHT places all the Type 1 vertices in $\tau'_R(i)$ to the right of i in μ in the same left-to-right order as in $\tau'_R(i)$. So it only remains to prove that all the Type 2 vertices in $\tau'_L(i)$ will appear to the left of i in μ and all such vertices in $\tau'_R(i)$ will appear to the right of i in μ.

For any vertex v, let first(v) be the highest numbered neighbour of v. This means that v is in $\tau'(\text{first}(v))$ and it is placed in μ when we add the vertices in $\tau'(\text{first}(v))$. Also v is a Type 1 vertex in $\tau'(\text{first}(v))$. Hence procedure VERTEX_ORDER will place v around first(v) and no Type 2 vertex in $\tau'(\text{first}(v))$ will appear between v and first(v) in μ. Now let j be a Type 2 vertex in $\tau'_L(i)$. From the PQ-tree reduction procedure it should be clear that in T_i, the node corresponding to vertex j will appear to the left of the node corresponding to vertex i. Both these nodes will be children of a Q-node.

Let i, first(i), first(first(i)), ..., x and j, first(j), first(first(j)), ..., y be the sequences of vertices such that first(x) = first(y) = k. Suppose we

carry out the PQ-tree reduction procedure making sure that at each step the Q-nodes representing the different blocks of a bush form give rise to the τ'-orders, then no reversal of these nodes will be required. So, in such T_{k-1}, the nodes corresponding to the vertices x and y should appear as children of a Q-node with the node corresponding to vertex y appearing to the left of the node corresponding to vertex x. Since both x and y are Type 1 vertices in $\tau'(k)$, procedure VERTEX_ORDER will place y to the left of x in μ. This along with the fact that any vertex in the sequence i, first(i), first(first(i)), ..., x and in the sequence j, first(j), first(first(j)), ..., y is placed around its successor in the sequence in μ implies that j will be placed to the left of i in μ. Thus all the Type 2 vertices in $\tau'_L(i)$ will be placed to the left of vertex i in μ. Similarly we can prove that all the Type 2 vertices in $\tau'_R(i)$ will be placed to the right of vertex i in μ. □

The following theorem establishes the complexity of procedure VERTEX_ORDER.

THEOREM 8.6.

Procedure VERTEX_ORDER determines the vertex order in O(n) time.

Proof:

It is easy to see that for a given i, the costs of execution of procedures PLACE_LEFT, PLACE_CENTER and

PLACE_RIGHT are $|\tau'_L(i)|$, $|\tau'_C(i)|$ and $|\tau'_R(i)|$, respectively. Thus the cost of execution of procedure VERTEX_ORDER is $|\tau'_L(i)| + |\tau'_C(i)| + |\tau'_R(i)|$, which is the in-degree of vertex i in the st-graph G. Summing up these costs over all i, $2 \leq i \leq n$, we get the execution time of procedure VERTEX_ORDER as O(n) for a planar graph. □

Having obtained the vertex order, we now describe our drawing procedure to obtain a planar embedding. We place the vertices of G in the plane at different horizontal and vertical levels. In the following, the horizontal line at vertical level r will be denoted by X_r and the vertical line at horizontal level r will be denoted by Y_r. Whereas the vertical level of a vertex in the placement is dictated by its st-number, the horizontal level is dictated by the position of the vertex in the vertex order μ. Thus if vertex i occurs at the j-th position in μ, then it will be placed at the i-th vertical level and j-th horizontal level. In such a placement no two vertices will appear in the same horizontal or vertical level. We then construct a planar embedding of G by constructing planar embeddings of the vertex induced subgraphs $G_2, G_3, \ldots, G_n = G$, successively. At each step of the embedding process, we have to ensure that the corresponding Type 2 vertices appear on the outside window. Clearly this requirement is satisfied by G_2.

Suppose we have embedded G_{i-1} such that all the

vertices connected to vertices numbered i or higher are on the outside window of G_{i-1}. When we embed vertex i, clearly it will appear on the outside window of G_i. However, the edges connecting i to vertices in $\tau'(i)$ should be drawn so that in G_i all the Type 2 vertices appear on the outside window. Let $\tau'(i) = (j_1, j_2, \ldots, j_k)$. Connecting vertex i to the vertices j_1 and j_k forms a circuit, say χ_i, in G_i. In addition to the edges (j_1,i) and (j_k,i), this circuit will contain the path from j_1 to j_k traced along the outside window of G_{i-1}. Now recall that in μ all the Type 1 vertices in $\tau'(i)$ are placed around vertex i. Thus in μ no Type 2 vertex appears between i and a Type 1 vertex. Also, in μ all the vertices in $\tau'_L(i)$ appear to the left of i and those in $\tau'_R(i)$ appear to the right of i. Furthermore, $\tau'(i)$ can have at most two Type 2 vertices from each block of G_{i-1} and these Type 2 vertices are necessarily cut vertices in G_{i-1}. These observations imply that the region bounded by χ_i will enclose no Type 2 vertices provided the edges (j_1,i) and (j_k,i) are drawn so that all the Type 2 vertices placed to the left (right) of i in μ lie left (right) of the edge connecting i to j_1 (j_k). Also the edge connecting i to j_1 can be drawn within the region bounded by the lines X_{j_1}, X_i, Y_{j_1} and Y_i. Similarly the edge connecting i to j_k can be drawn within the region bounded by the lines X_{j_k}, X_i, Y_{j_k} and Y_i.

Thus to embed vertex i, we first draw the edge (j_1,i)

within the region bounded by X_{j_1}, X_i, Y_{j_1} and Y_i such that all the Type 2 vertices placed in this region appear above this edge. Next we draw the edges (j_2,i), (j_3,i), ..., (j_k,i) entering vertex i from below in such a way that any edge enters vertex i to the immediate right of its predecessor in the sequence. Note that the edge (j_k,i) has to be drawn so that all the Type 2 vertices in the region bounded by X_{j_k}, X_i, Y_{j_k} and Y_i lie above this edge. Embedding vertex i this way we obtain a planar embedding of G_i. Repeating this procedure we can obtain planar embeddings of G_{i+1}, G_{i+2}, ..., $G_n = G$. In Fig. 8.11 we show a planar embedding of the planar graph G shown in Fig. 7.1 obtained using the above procedure.

Even though the vertex order can be computed in O(n) time, the embedding procedure described above has to be implemented manually. However, this is a systematic procedure in the sense that the regions in which the edges should be drawn can be determined easily and so the planar embedding can be obtained without any difficulty. The vertex order helps us to construct the planar embeddings of G_2, G_3, ..., G_n in such a way that the edges can be drawn as smooth line segments without awkward bends. Thus this procedure will construct a nice planar embedding.

From Fig. 8.11 we can see that many of the edges in the planar embedding can be drawn as straight-line segments. It

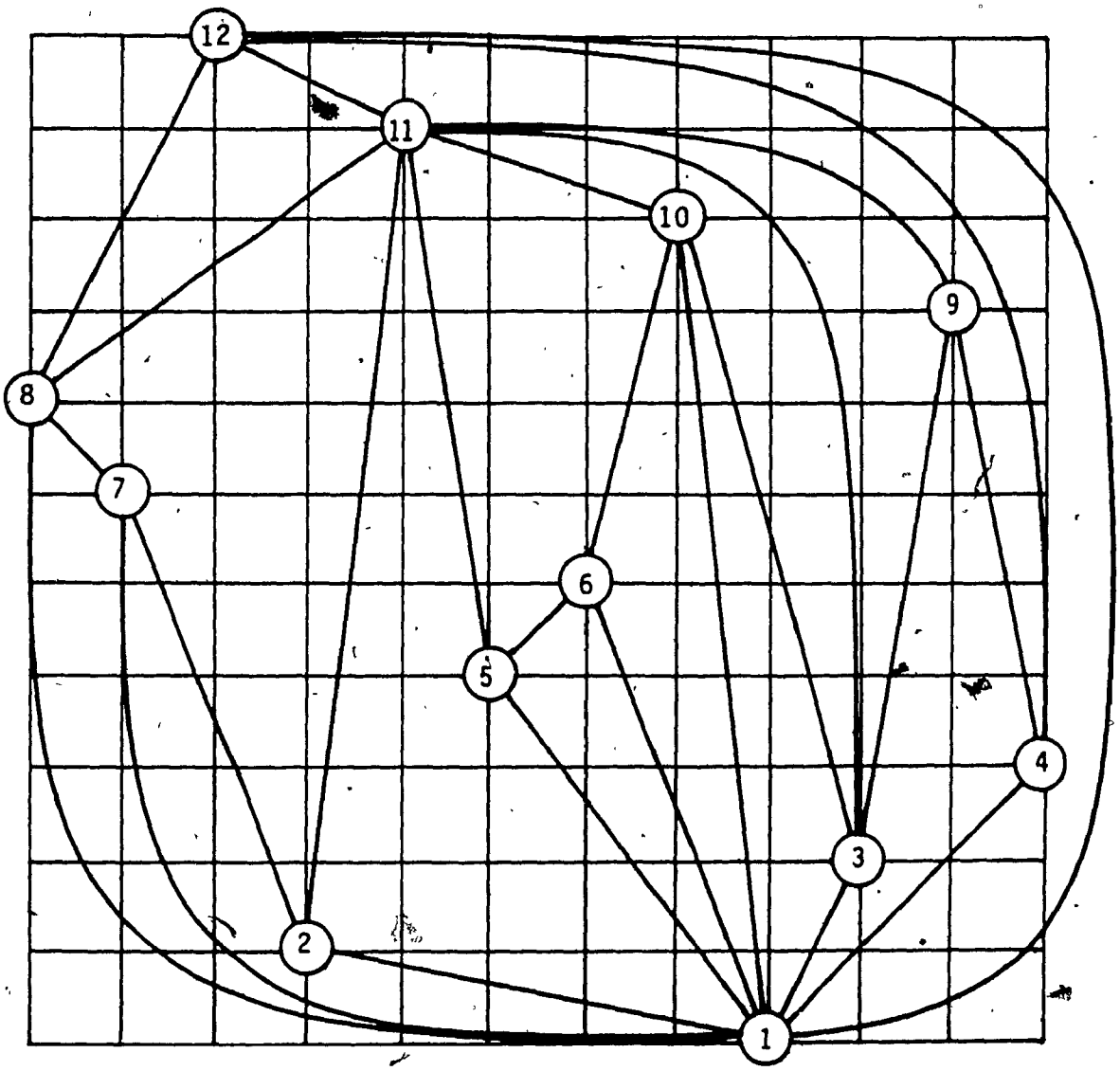


Figure 8.11

Planar Embedding

is well known that any simple planar graph can be embedded in the plane in such a way that all the edges are straight-line segments. Intuitively it appears that by properly shifting the vertices and adjusting their positions in the planar embedding obtained by our procedure, it should be possible to draw all the edges as straight-line segments. However, the way in which the vertices are to be adjusted is not very obvious.

CHAPTER 9

A $O(n^2)$ ALGORITHM FOR MAXIMAL PLANARIZATION OF NONPLANAR GRAPHS

A subgraph G' of a nonplanar graph G is a maximal planar subgraph of G if G' is planar and adding to G' any edge not present in G' results in a nonplanar subgraph of G. The process of removing a set of edges from a nonplanar graph to obtain a maximal planar subgraph is known as maximal planarization. Maximal planarization of a nonplanar graph is an important problem encountered in the automated design of printed circuit boards. If an electronic circuit cannot be wired on a single layer of a printed circuit board, then we would like to determine the minimum number of layers necessary to wire the circuit. Since only a planar circuit can be wired on a single layer board, we would like to decompose the nonplanar circuit into a minimum number of maximal planar circuits. In general, for a nonplanar graph, neither the set of edges to be removed to maximally planarize it nor the number of these edges is unique.

Determining the minimum number of edges whose removal from a nonplanar graph will yield a maximal planar subgraph is an NP-complete problem [26]. However, a few algorithms which attempt to produce maximal planar subgraphs having the largest possible number of edges have been reported. One of the earliest algorithms to planarize a nonplanar graph is

due to Fisher and Wing [39]. Their planarity testing algorithm identifies a set of edges whose removal makes a nonplanar graph planar. However, the planar subgraph obtained may not be maximally planar. Later Pasedach [62] suggested an algorithm to obtain a maximal planar subgraph of a nonplanar graph using Fisher and Wing's planarity testing algorithm. However, this algorithm works on the incidence matrix of the graph and so it is not very efficient. Another algorithm to planarize a triconnected nonplanar graph was proposed by Marek-Sadowska [63]. This algorithm works on the circuit matrix of the nonplanar graph and hence it is also not very efficient.

Recently, Chiba, Nishioka, and Shirakawa [64] modified Hopcroft and Tarjan's planarity testing algorithm to maximally planarize a nonplanar graph. Their algorithm needs $O(mn)$ time and $O(mn)$ space for a nonplanar graph having n vertices and m edges. Ozawa and Takahashi [49] proposed another $O(mn)$ time and $O(m+n)$ space algorithm to planarize a nonplanar graph using the PQ-tree implementation of the LEC algorithm. They expected their algorithm to find a maximal planar subgraph when applied on a complete graph. However, for a general graph this algorithm may not determine a maximal planar subgraph.

In this chapter, we present an efficient $O(n^2)$ time and $O(m+n)$ space algorithm to determine a maximal planar

subgraph of a nonplanar graph. We attempt to include as many edges as possible in the maximal planar subgraph. Our algorithm is also based on the LEC algorithm. We present the basic principles of the planarization algorithm in Section 9.1. In Section 9.2 we discuss Ozawa and Takahashi's algorithm and point out that this algorithm may not determine a maximal planar subgraph of a nonplanar graph. However, we show that this algorithm determines a maximal planar subgraph when applied on a complete graph. In Section 9.3 we develop a $O(n^2)$ algorithm to determine a spanning planar subgraph of a nonplanar graph. In Section 9.4 we present a $O(n^2)$ algorithm which maximally planarizes the spanning planar subgraph with respect to the given nonplanar graph.

9.1 Principle of the Planarization Algorithm

Consider a simple biconnected st-graph G. Let T_1, T_2, ..., T_{n-1} be the PQ-trees corresponding to the bush forms of G. For any node X in T_i, recall that, the frontier of X is the left-to-right order of appearance of the leaves in the subtree of T_i rooted at X. Ozawa and Takahashi [49] classify the nodes of any PQ-tree according to their frontier as follows.

Type W: A node is said to be Type W if its frontier consists of only non-pertinent leaves.

Type B: A node is said to be Type B if its frontier consists of only pertinent leaves.

Type H: A node X is said to be Type H if the subtree rooted at X can be rearranged such that all the descendant pertinent leaves of X appear consecutively at either the left or the right end of the frontier. Note that at least one non-pertinent leaf will appear at the other end of the frontier.

Type A: A node X is said to be Type A if the subtree rooted at X can be rearranged such that all the descendant pertinent leaves of X appear consecutively in the middle of the frontier with at least one non-pertinent leaf appearing at each end of the frontier.

The following theorem is the central concept of the planarization algorithm.

THEOREM 9.1.

An n-vertex graph G is planar if and only if the pertinent roots in all the PQ-trees T_2, T_3, ..., T_{n-1} of G are Type B, H or A.

Proof:

Since the pertinent leaves in any T_i, $2 \leq i \leq n-1$, are all descendants of the pertinent root, it follows that the pertinent root cannot be Type W. If the pertinent root in

T_i is Type B, H or A, then T_i can be successfully reduced to T_i^* and the next PQ-tree T_{i+1} can be constructed. Thus the sufficiency of the theorem follows. On the other hand, if the pertinent root in a PQ-tree is not Type B, H or A, then the pertinent leaves in that tree cannot be made consecutive and hence that tree cannot be reduced. Thus the graph will be nonplanar if the pertinent root of any PQ-tree is not Type B, H or A. □

We call a PQ-tree reducible if its pertinent root is Type B, H or A; otherwise it is irreducible. Theorem 9.1 implies that the graph G is planar if and only if all the T_i's are reducible. If any T_i is irreducible, we can make it reducible by appropriately deleting some of the leaves from it. Of course, we would like to delete a minimum number of leaves while trying to make T_i reducible. If we make all the T_i's reducible this way, then a planar subgraph can be obtained by removing from the nonplanar graph the edges corresponding to the leaves that are deleted.

It is easy to see that the PQ-tree T_{n-1} is always reducible because its root is Type B. The tree T_1 is also reducible because it has only one pertinent leaf - the leaf corresponding to the edge (1,2). Consider now an irreducible PQ-tree T_i of an n-vertex nonplanar graph. For a node X in T_i, let w, b, h, and a be the minimum number of descendant leaves of X which should be deleted from T_i so

that X becomes Type W, B, H, and A respectively. We denote these numbers of a node as [w,b,h,a]. Any node in T_i may be made Type W, B, H, or A by appropriately deciding the types of its children. So the [w,b,h,a] number of any node can be computed from that of its children. Thus to make T_i reducible, we first traverse it bottom-up from the leaves to the pertinent root and compute the [w,b,h,a] number for every node in T_i. Once the [w,b,h,a] number of the pertinent root is computed, we make the pertinent root Type B, H, or A depending on which one of the numbers b, h, and a of the root is the smallest. After determining the type of the pertinent root, we traverse T_i top-down from the pertinent root to the leaves and decide the type of each node in the pertinent subtree of T_i. Note that the type of a node uniquely determines the types of its children and so the types of all the leaves in T_i can be determined by this top-down traversal. This information would help us decide the nodes to be deleted from T_i in order to make it reducible. After deleting these nodes from T_i, we can apply the reduction procedure to obtain T_i^*. Note that deletion of leaves corresponds to removal of the corresponding edges from the nonplanar graph.

Repeating the above procedure for each irreducible T_i, we can obtain a planar subgraph of the nonplanar graph. It is easy to see that if the minimum of b, h, and a for the pertinent root in a PQ-tree T_i is zero, then T_i is

reducible. Thus we can determine whether a T_i is reducible or not from the [w,b,h,a] number of its pertinent root. In the following we summarize the above procedure.

```
procedure GRAPH_PLANARIZE(G);
comment procedure GRAPH_PLANARIZE determines a planar
        subgraph of an n-vertex nonplanar graph G by
        removing a minimum number of edges from G.
begin
  construct the initial PQ-tree T1 = T1*;
  for i := 2 to n-2 do
    begin
      construct the PQ-tree Ti from Ti-1*;
      compute the [w,b,h,a] number of each node in the
      pertinent subtree of Ti by traversing it bottom-up;
      if min{b,h,a} for the pertinent root is not zero
        then begin
          {Ti is irreducible}
          make the pertinent root Type B, H, or A depending
          on the minimum of b, h, and a;
          determine the type of each node in Ti by
          traversing it top-down;
          delete the necessary nodes from Ti and make it
          reducible;
          remove from G the edges corresponding to the
          leaves that are deleted from Ti
        end;
```

```
        {T_i is now reducible}
        reduce T_i to obtain T_i*
    end
end GRAPH_PLANARIZE;
```

Note that the above algorithm may not determine a maximal planar subgraph. This can be explained as follows. Suppose we delete certain leaves from T_i to make it reducible. In a later reduction step some of the leaves which caused the irreducibility of T_i may themselves be deleted. In such a case, we may be able to return to G a subset of the edges which were removed while making T_i reducible. Hence the planar subgraph obtained by procedure GRAPH_PLANARIZE may not be maximally planar.

Ozawa and Takahashi [49] have presented formulas to compute [w,b,h,a] numbers for the nodes in a PQ-tree. Using these formulas in procedure GRAPH_PLANARIZE we can determine a planar subgraph of a nonplanar graph. In the next section we discuss their approach and highlight some of its drawbacks.

9.2 Ozawa and Takahashi's Planarization Algorithm

In this section we discuss Ozawa and Takahashi's approach to planarize a nonplanar graph G. Consider an

irreducible PQ-tree T_i, $3 \leq i \leq n-2$, of G. The pertinent root of T_i has both pertinent leaves and non-pertinent leaves as its descendants. Ozawa and Takahashi make T_i reducible by deleting a minimum number of these leaves, some of which may not be pertinent, from T_i. We now present the formulas they developed to compute the [w,b,h,a] number of a node in T_i.

Consider a node X in a PQ-tree T_i. Let d be the number of descendant leaves of X. Let the children of X be numbered as 1, 2, ..., p. Also, let the w, b, h, and a numbers of child i of X be denoted as w_i, b_i, h_i, a_i respectively. To make node X Type W, we have to delete from T_i all the descendant pertinent leaves of X. Thus for the node X the value of w is equal to the number of its descendant pertinent leaves. Similarly to make X Type B, all the descendant non-pertinent leaves of X should be deleted and hence the value of b is equal to the number of such leaves of X. Based on these observations, the following formulas can be derived.

(i) <u>X is a leaf</u>.

$$w = \begin{cases} 1, & \text{if X is a pertinent leaf,} \\ 0, & \text{if X is a non-pertinent leaf.} \end{cases}$$

$b = d-w.$

$h = 0.$

$a = 0.$

(ii) <u>X is a P-node</u>.

$$w = \sum_{i=1}^{p} w_i .$$

$$b = \sum_{i=1}^{p} b_i = d-w.$$

We can make X Type H by making one of its children Type H and all the other children either Type W or Type B. Since h denotes the minimum number of leaves to be deleted to make X Type H, we get

$$h = \sum_{i=1}^{p} \min\{w_i,b_i\} - \max_{1\le i\le p} \{(\min\{w_i,b_i\}-h_i)\}.$$

The node X can be made Type A in two different ways. One way is to make two of its children Type H and all the other children either Type W or Type B. For this case, the minimum number of leaves to be deleted is given by

$$\alpha_1 = \sum_{i=1}^{p} \min\{w_i,b_i\} - \beta$$

where

$$\beta = \max_{1\le i\ne j\le p} \{(\min\{w_i,b_i\}-h_i + \min\{w_j,b_j\}-h_j)\}.$$

The other possibility is to make one of the children of X Type A and all the other children Type W. For this case, the minimum number of leaves to be deleted is given by

$$\alpha_2 = \sum_{i=1}^{p} w_i - \max_{1\leq i\leq p} \{(w_i - a_i)\}.$$

Thus the value of a for the node X when it is a P-node is given by

$$a = \min\{\alpha_1, \alpha_2\}.$$

(iii) <u>X is a Q-node</u>.

$$w = \sum_{i=1}^{p} w_i.$$

$$b = \sum_{i=1}^{p} b_i = d-w.$$

We can make X Type H by letting one of its children Type H, all the siblings of that child on one side (either left or right) Type B and all the siblings on the other side Type W. Thus the value of h for X when it is a Q-node is given by

$$h = \min_{1\leq k\leq p} \{(h_k + y_k)\}$$

where

$$y_k = \min\left\{\sum_{i=1}^{k-1}(w_i-b_i)-b_k+\sum_{i=1}^{p}b_i,\ \sum_{i=1}^{k-1}(b_i-w_i)-w_k+\sum_{i=1}^{p}w_i\right\}.$$

X can be made Type A in two different ways. The first is to make two of its children Type H, all the siblings in between these two Type H children Type B, and all the other children Type W. In this case the minimum number of leaves to be deleted is given by

$$\alpha_1 = \sum_{i=1}^{p} b_i - \max_{1\le j<k\le p}\{(y_j+z_k)\}$$

where

$$y_j = \sum_{i=1}^{j-1}(b_i-w_i)+b_j-h_j$$

and

$$z_k = \sum_{i=k+1}^{p}(b_i-w_i)+b_k-h_k.$$

The second method is to make one child Type A and all the other children Type W. For this case

$$\alpha_2 = \sum_{i=1}^{p} w_i - \max_{1\le i\le p}\{(w_i-a_i)\}.$$

Thus the value of a for the node X when it is a Q-node is given by

$$a = \min\{\alpha_1, \alpha_2\}.$$

Ozawa and Takahashi [49] presented algorithms to compute the [w,b,h,a] numbers for the nodes in a PQ-tree using the above formulas in O(n(m+n)) time. The PQ-trees are stored in O(m+n) space and so their algorithm requires O(m+n) space. As we have already stated, Ozawa and Takahashi's algorithm may result in deleting both pertinent and non-pertinent leaves from T_i in order to make it reducible. In some cases we may be able to make T_i reducible by deleting either a pertinent leaf or a non-pertinent leaf. In such cases Ozawa and Takahashi prefer to delete the non-pertinent leaf. Note that in T_i, the pertinent leaves correspond to the edges entering vertex (i+1) in the st-graph G and the non-pertinent leaves correspond to the edges entering vertices greater than (i+1). Since a PQ-tree T_i with only one pertinent leaf is always reducible, in the planar subgraph obtained after reducing such a T_i, there will be a path from vertex 1 to vertex i+1. Since Ozawa and Takahashi permit deletion of non-pertinent leaves also, it may so happen that as the algorithm proceeds, all the edges entering a vertex k > (i+1) may get removed from G and thus vertex k and some of other vertices may not be present in the resulting planar subgraph.

We illustrate this situation for the nonplanar st-graph shown in Fig. 9.1. In Figs. 9.2 to 9.8 we show the PQ-trees T_1 to T_7 for the graph in Fig. 9.1. In these PQ-trees, the [w,b,h,a] number for a node which is not a leaf is shown adjacent to it. Note that T_5 is the first irreducible PQ-tree and the algorithm removes the edge (2,6) from the graph to make T_5 reducible. Similarly the edges (4,7) and (5,7) are removed to make T_6 reducible, and the edges (5,9), (4,9) and (6,10) are removed to make T_7 reducible. Note that (4,9) and (5,9) are the only edges entering vertex 9 in the st-graph and hence after removing these two edges, vertex 9 will not be represented in the PQ-tree T_8. Thus the planar subgraph of the nonplanar graph of Fig. 9.1, obtained by Ozawa and Takahashi's algorithm, will not contain vertex 9.

From our discussions so far, it should be clear that the main drawback of Ozawa and Takahashi's algorithm, apart from the fact that it may not determine a maximal planar subgraph, is that the planar subgraph it determines may not even be a spanning subgraph of the given nonplanar graph. This is because the algorithm permits deletion of both pertinent and non-pertinent leaves. In the next section we show that by appropriately deleting only pertinent leaves, it is possible to obtain a spanning planar subgraph.

In the case of a complete graph, Ozawa and

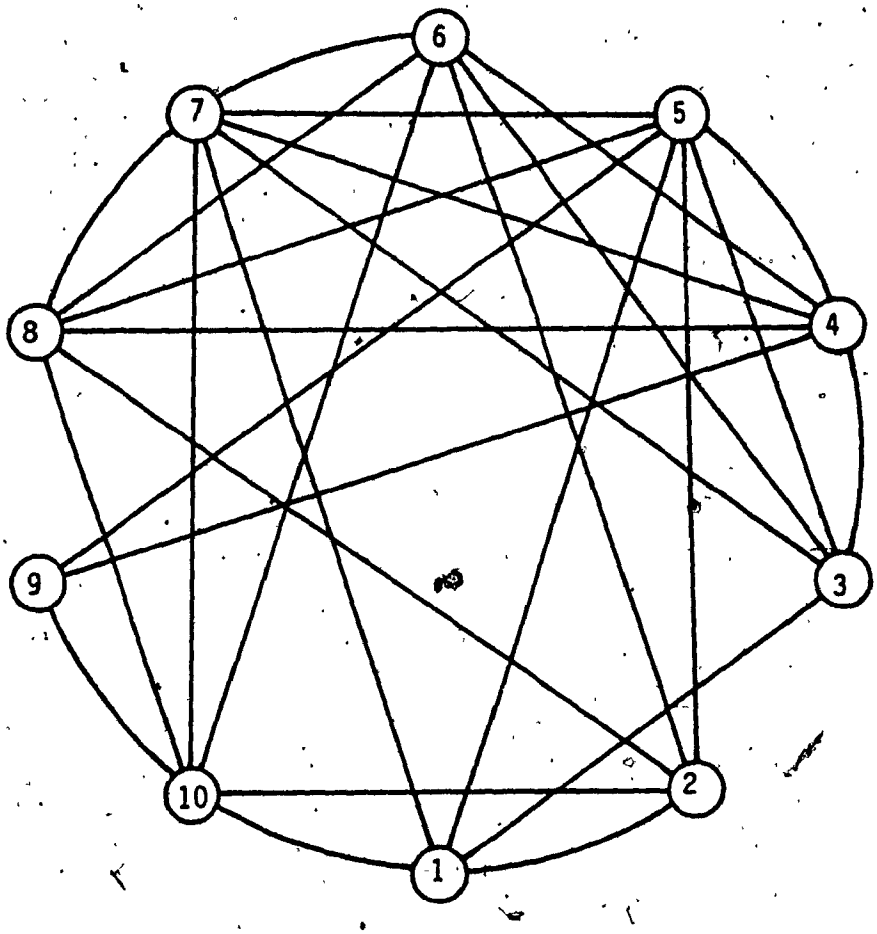


Figure 9.1

Nonplanar Graph G

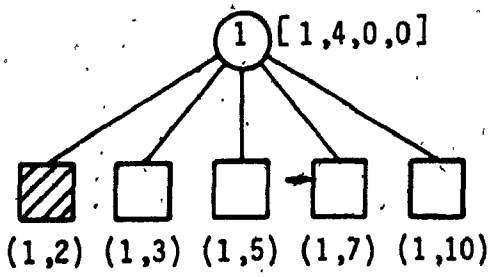


Figure 9.2

PQ-tree $T_1 = T_1^*$

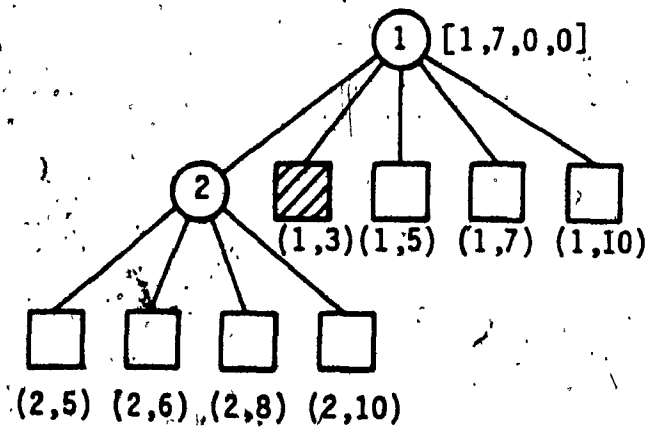


Figure 9.3

PQ-tree $T_2 = T_2^*$

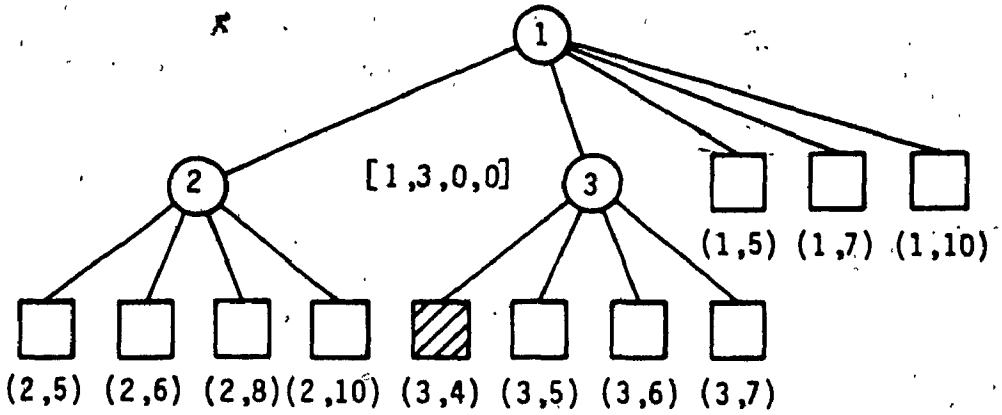


Figure 9.4

PQ-tree $T_3 = T_3^*$

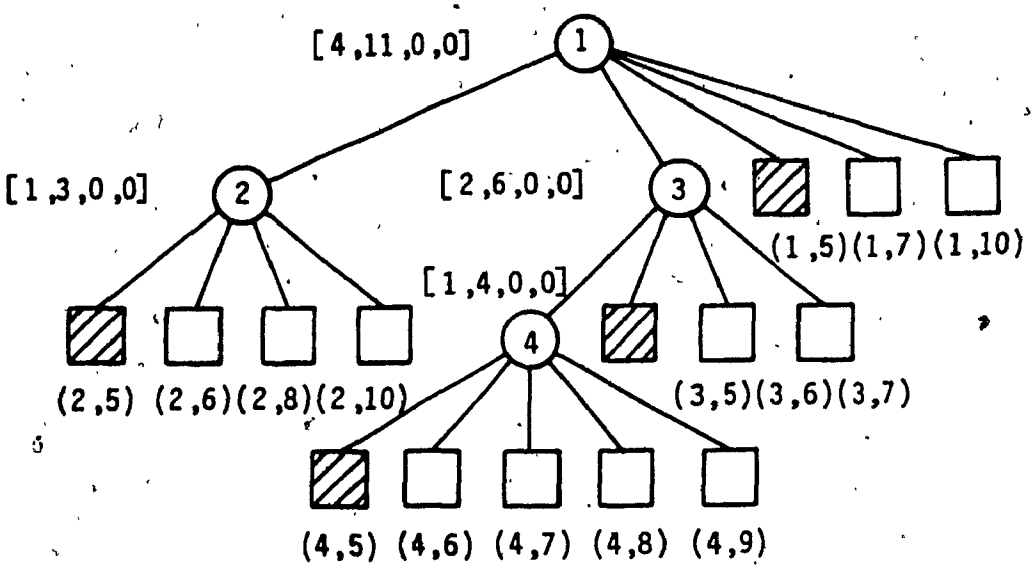


Figure 9.5(a)

PQ-tree T_4

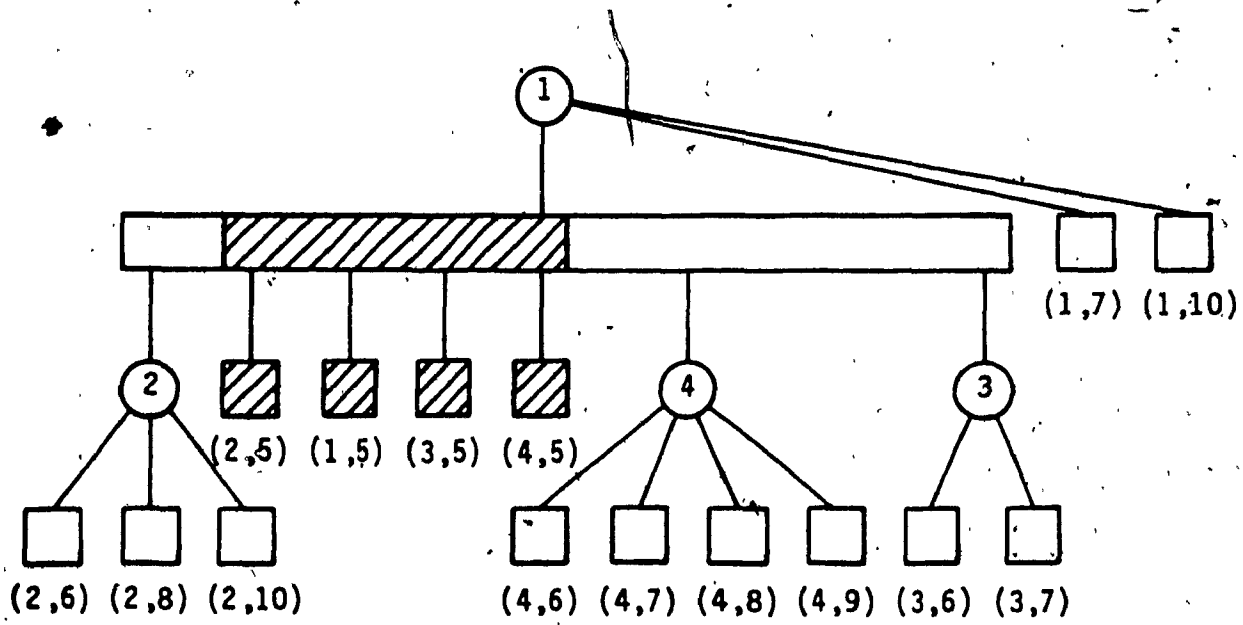


Figure 9.5(b)

PQ-tree T_4^*

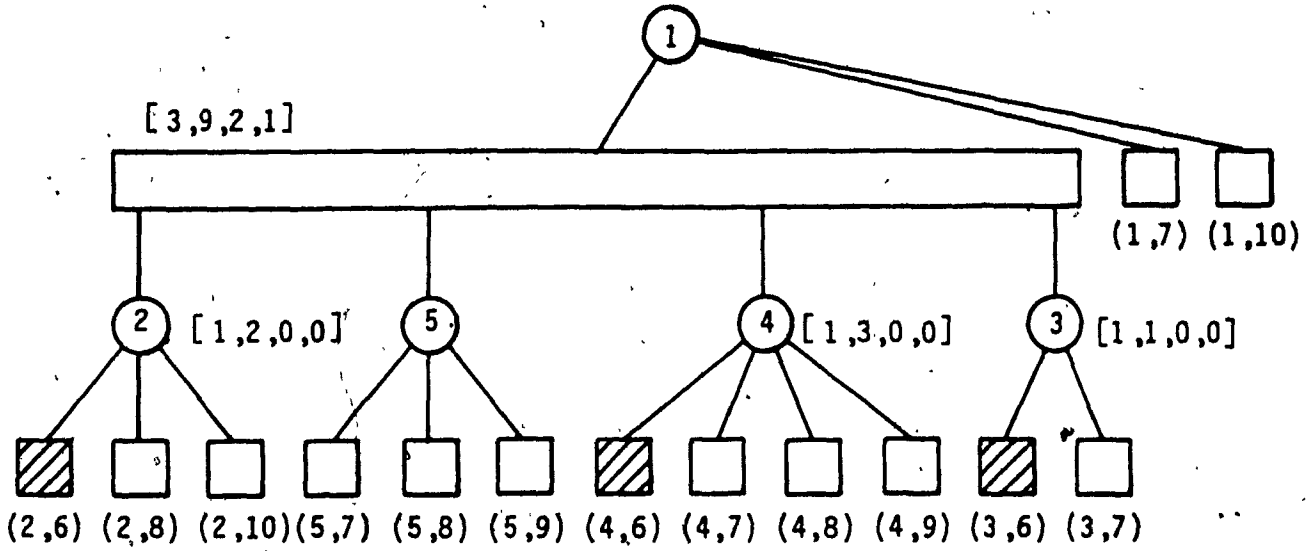


Figure 9.6(a)

PQ-tree T_5

Edge (2,6) is removed

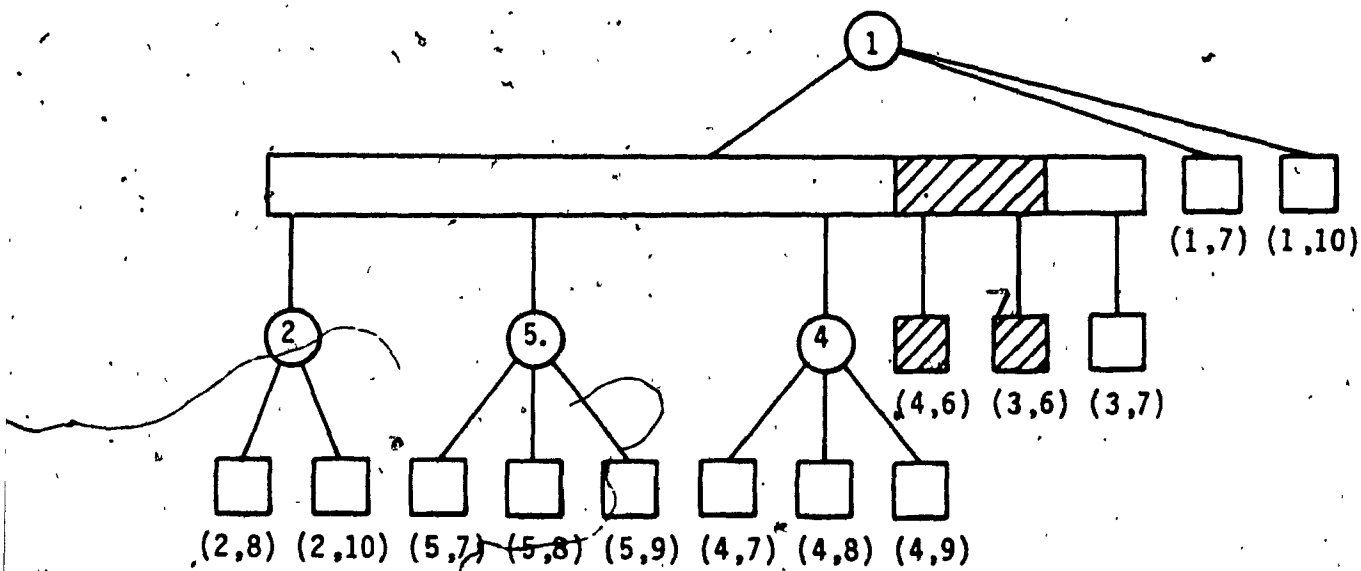


Figure 9.6(b)

PQ-tree T_5^*

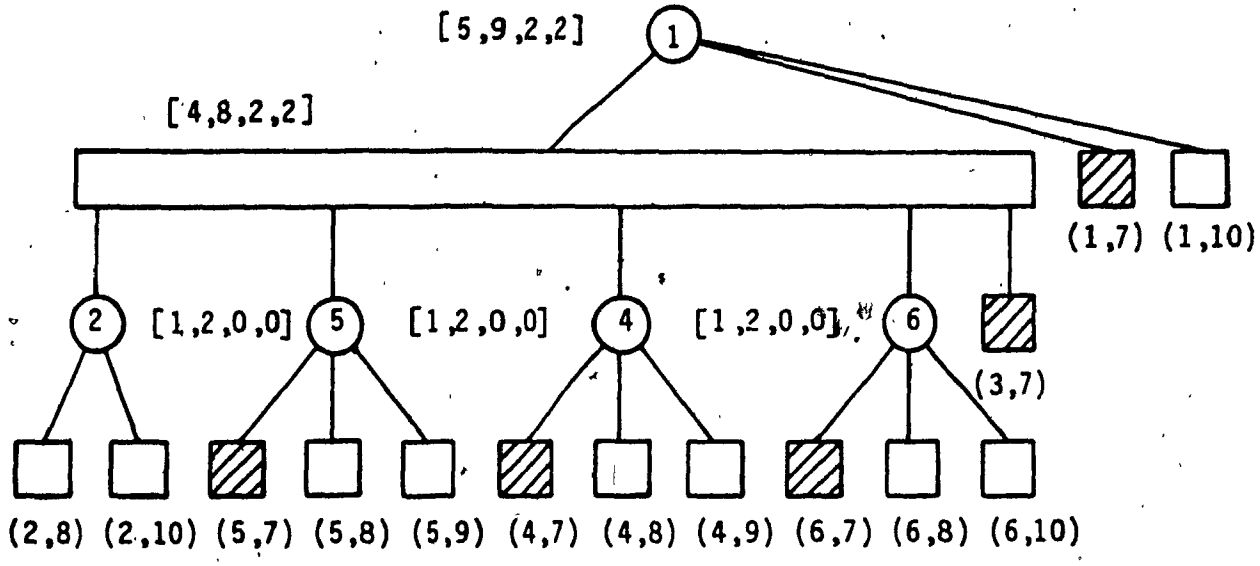


Figure 9.7(a)

PQ-tree T_6

Edges (4,7) and (5,7) are removed

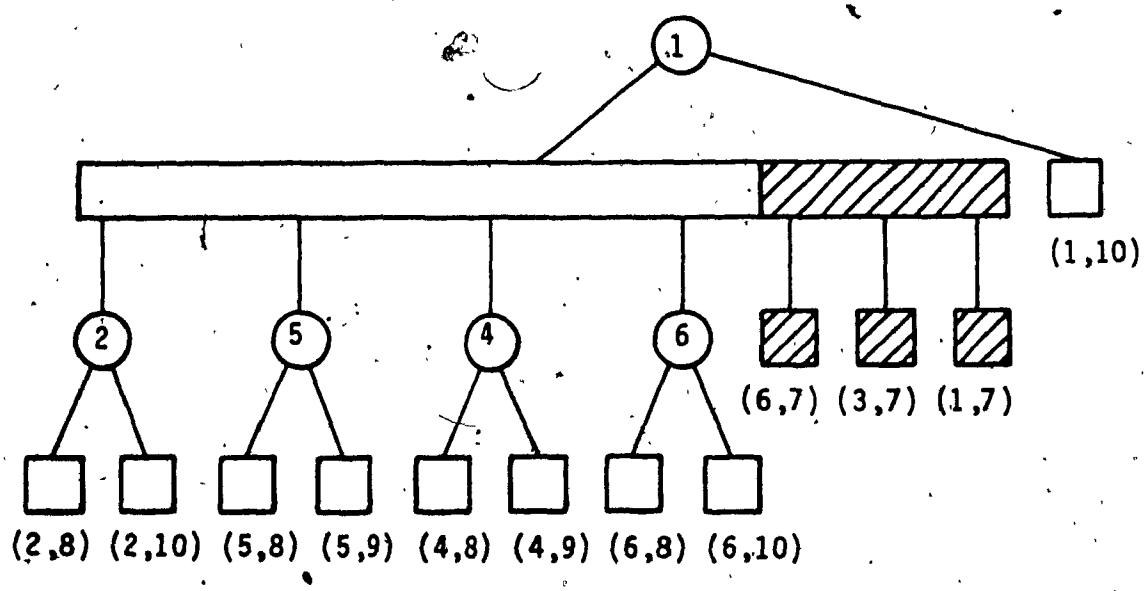


Figure 9.7(b)

PQ-tree T_6^*

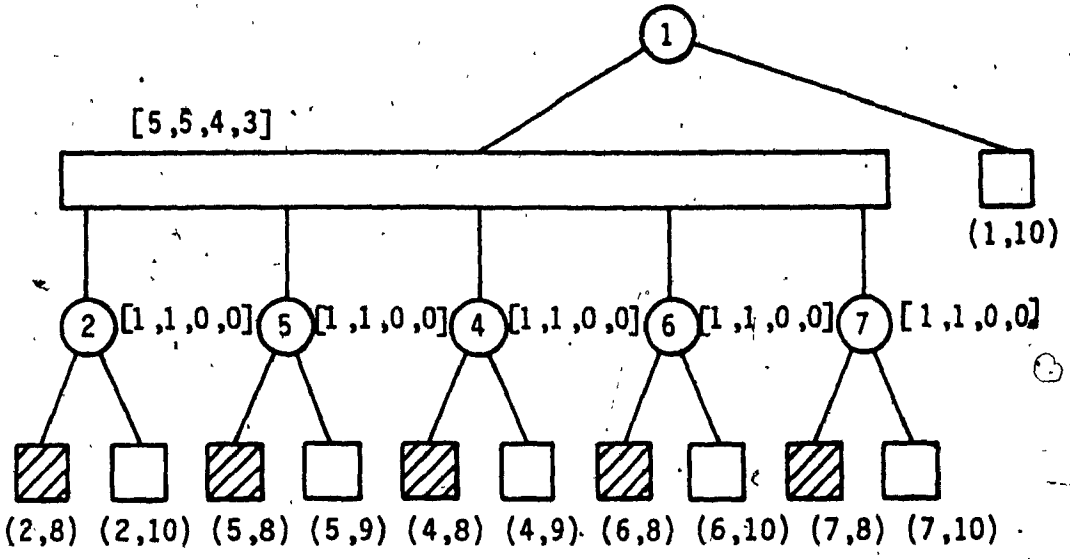


Figure 9.8(a)

PQ-tree T_7

Edges (5,9), (4,9) and (6,10) are removed

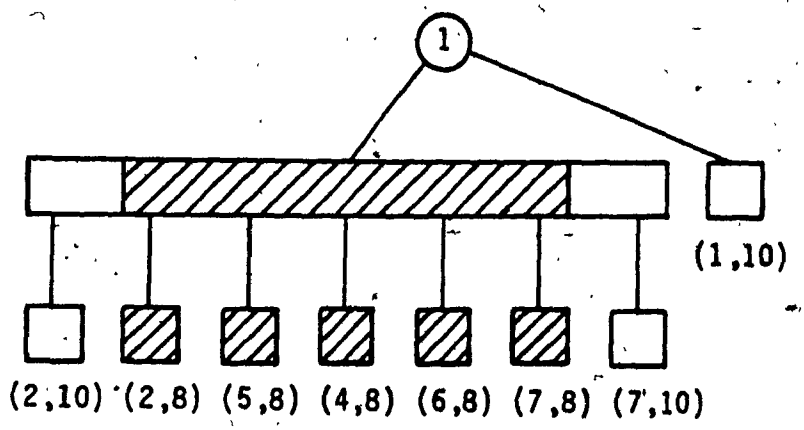


Figure 9.8(b)

PQ-tree T_7^*

Takahashi [49] expected their algorithm to determine a maximal planar subgraph. We conclude this section by proving their assertion.

THEOREM 9.2.

In the case of a complete graph, Ozawa and Takahashi's algorithm determines a maximal planar subgraph.

Proof:

We prove the theorem by showing that the planar subgraph obtained by Ozawa and Takahashi's algorithm when applied on an n-vertex complete graph will have n vertices and 3n-6 edges.

Note that for any graph the PQ-trees T_2 and T_{n-1} are always reducible and so no leaves need be deleted from these trees. For any i, $3 \leq i \leq n-2$, the PQ-tree T_i of an n-vertex complete graph is of the form shown in Fig. 9.9. The [w,b,h,a] numbers of the nodes in T_i can be easily computed as follows.

(i) For the P-nodes labeled 2, 3, ..., i

w = 1.

b = n-i-1.

h = 0.

a = 0.

(ii) For the only Q-node

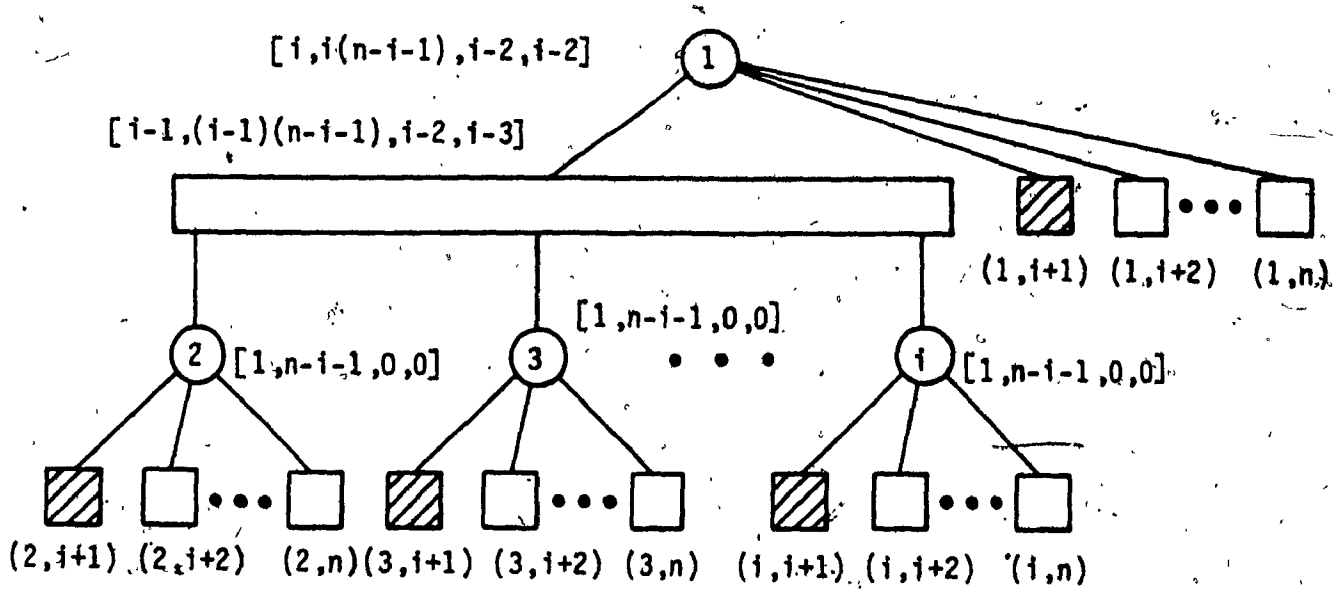


Figure 9.9

PQ-tree T_i

for an n-vertex complete graph

$w = i-1.$

$b = (i-1)(n-i-1).$

$h = i-2.$

$a = i-3.$

(iii) For the pertinent root (the P-node labeled 1)

$w = i.$

$b = i(n-i-1).$

$h = i-2.$

$a = i-2.$

Thus from each T_i, $3 \leq i \leq n-2$, $(i-2)$ leaves are removed to make it reducible. Hence the total number of edges removed is given by

$$\sum_{i=3}^{n-2} (i-2) = \frac{(n-3)(n-4)}{2}.$$

Since an n-vertex complete graph has $n(n-1)/2$ edges, the number of edges in the planar graph determined by Ozawa and Takahashi's algorithm is given by

$$\frac{n(n-1)}{2} - \frac{(n-3)(n-4)}{2} = 3n-6.$$

As can be seen from Fig. 9.9, for an n-vertex complete graph, minimum leaf deletion in the case of each T_i, $3 \leq i \leq n-3$, necessarily results in deletion of only pertinent leaves. Since from each T_i, $3 \leq i \leq n-3$, only $(i-2)$ leaves

are removed, it follows that in each such reducible T_i, there will be exactly two pertinent leaves. On the other hand, in the case of T_{n-2} minimum leaf deletion can be achieved by deleting either (n-4) pertinent leaves or (n-4) non-pertinent leaves. However, even in this case, irrespective of the choice made, there will be at least two pertinent leaves in the reducible T_{n-2}. Since the edges (1,n) and (1,2) are not removed, it follows that in the planar subgraph obtained by Ozawa and Takahashi's algorithm, each vertex will be connected to at least one lower numbered vertex and so this subgraph will be connected and will have n vertices. Hence the theorem. □

9.3 A New Graph-Planarization Algorithm

As a first step towards designing an algorithm (to be discussed in Section 9.4) to obtain a maximal planar subgraph of a nonplanar graph G, we develop in this section an efficient algorithm to determine a spanning planar subgraph of G. The planarization approach discussed in Section 9.1 will form the basis of this algorithm. As pointed out in the previous section, Ozawa and Takahashi's algorithm may not result in a spanning planar subgraph. The reason for this is that while making a PQ-tree T_i reducible, non-pertinent leaves may be deleted. We modify this approach so that deletion of only pertinent leaves is

permitted. We first prove that with this modification, the approach of Section 9.1 will result in a spanning planar subgraph of G.

THEOREM 9.3.

The planarization algorithm of Section 9.1 will determine a spanning planar subgraph of a biconnected n-vertex nonplanar graph, if only pertinent leaves are deleted while making any PQ-tree T_i, $3 \leq i \leq n-2$, reducible.

Proof:

Note that a PQ-tree with only one pertinent leaf is always reducible. So it follows that from no PQ-tree all the pertinent leaves will be deleted, if only pertinent leaves are to be chosen for deletion. This means that in the subgraph that results at the end of the application of the algorithm, each vertex will be connected to at least one lower numbered vertex. Thus the subgraph determined will be a spanning subgraph of the given nonplanar graph. □

Let G be a nonplanar st-graph. Let E_i, $2 \leq i \leq n$, be the set of edges entering vertex i in G. We determine a planar subgraph of G by removing a sequence $E'_4, E'_5, \ldots, E'_{n-1}$ $(E'_i \subset E_i)$ of edges such that for each i the subgraph of G obtained by removing the edges in $E'_4, E'_5, \ldots, E'_i$ contains a planar subgraph induced by the vertex set $\{1, 2, \ldots, i\}$. Thus after removing the edges in $E'_4, E'_5, \ldots,$

E'_{n-1}, we obtain a planar subgraph of G. It is easy to see that the edges in E'_{i+1}, $3 \leq i \leq n-2$, correspond to the pertinent leaves in the PQ-tree T_i which should be deleted to make T_i reducible. Thus E'_{i+1} can be determined while making T_i reducible.

In order to make a PQ-tree T_i reducible, we first compute the [w,b,h,a] number for each node in T_i. Recall that a node in T_i is full if the number of leaves in the pertinent subtree rooted at the node is equal to the number of pertinent leaves. Note that during the processing to make T_i reducible, a full node and all its descendants may be made Type W, or they will remain Type B. On the other hand partial nodes may be made Type W, H, or A; but never Type B because we delete only pertinent leaves from T_i. Thus any pertinent node in T_i may be made Type W, H, or A only. So we need to compute only the w, h, and a numbers for the pertinent nodes in T_i. We denote these numbers as [w,h,a].

Now we develop formulas to compute the [w,h,a] number for each pertinent node in T_i. We process T_i bottom-up from the pertinent leaves to the pertinent root. So when a pertinent node X is processed, the [w,h,a] numbers of all its pertinent children should have already been computed. Thus we can compute the [w,h,a] number for X from the numbers of its pertinent children. In the following, P(X)

denotes the set of pertinent children of X and Par(X) denotes the set of partial children of X. Along with the [w,h,a] number for each pertinent node, we also determine, for each pertinent node which is not a leaf, three children called h_child1(X), h_child2(X) and a_child(X) which will be used later to decide the type of each pertinent child of X in the reducible T_i.

(i) <u>X is a pertinent leaf</u>.

In this case

$$w = 1,$$
$$h = 0,$$
$$a = 0.$$

(ii) <u>X is a full node</u>.

In this case

$$w = \sum_{i \in P(X)} w_i,$$
$$h = 0,$$
$$a = 0.$$

(iii) <u>X is a partial P-node</u>.

To make X Type W, all its pertinent children should be made Type W. Thus

$$w = \sum_{i \in P(X)} w_i.$$

We can make X Type H by making all its full children Type B, one partial child Type H and all other partial children Type W. Thus the h number of X is given by

$$h = \sum_{i \in Par(X)} w_i - \max_{i \in Par(X)} \{(w_i - h_i)\}.$$

In this case the partial child which is made Type H will be called h_child1(X).

We can make X Type A in two different ways. We can make one partial child of X Type A and all other pertinent children Type W. In this case

$$\alpha_1 = \sum_{i \in P(X)} w_i - \max_{i \in Par(X)} \{(w_i - a_i)\}$$

descendant pertinent leaves of X will have to be deleted. The partial child which is made Type A will be called a_child(X). On the other hand, if we make two partial children Type H, all full children Type B and all other pertinent children Type W, then

$$\alpha_2 = \sum_{i \in Par(X)} w_i - \max1_{i \in Par(X)} \{(w_i - h_i)\} - \max2_{i \in Par(X)} \{(w_i - h_i)\}$$

descendant pertinent leaves will have to be deleted from T_i to make X Type A, where max1 is the first maximum and max2 is the second maximum. The partial child having

max1$\{(w_i-h_i)\}$ will be called h_child1(X) and the partial child having max2$\{(w_i-h_i)\}$ will be called h_child2(X). Thus the P-node X can be made Type A by deleting

$$a = \min\{\alpha_1,\alpha_2\}$$

pertinent leaves from T_i. If the value of a is different from α_1, then we make a_child(X) empty.

(iv) X is a partial Q-node.

To make X Type W, all its pertinent children should be made Type W. Thus for X

$$w_x = \sum_{i \in P(X)} w_i.$$

To compute the h number of X, first note that X can be made Type H only if either its leftmost child or its rightmost child is pertinent. Suppose that the leftmost child of X is pertinent. Let us traverse the children of X from left to right and find $P_L(X)$, the maximal consecutive sequence of pertinent children such that only the rightmost node in $P_L(X)$ may be partial. If the leftmost child of X is not pertinent, then $P_L(X)$ will be empty. Suppose, on the other hand, that the rightmost child of X is pertinent. As we traverse the children of X from right to left, let $P_R(X)$ be the maximal consecutive sequence of pertinent children such that only the leftmost node in $P_R(X)$ may be partial. If the rightmost child of X is not pertinent, then $P_R(X)$ is

empty. We can easily see that X can be made Type H by deleting

$$h = \sum_{i \in P(X)} w_i - \max\left\{ \sum_{i \in P_L(X)} (w_i - h_i), \sum_{i \in P_R(X)} (w_i - h_i) \right\}$$

pertinent leaves from T_i. We call as h_child1(X) the leftmost node in $P_L(X)$ or the leftmost node in $P_R(X)$ depending on which one has the maximum $\sum(w_i - h_i)$ sum in the above formula for h.

X can be made Type A in two different ways. We can make one of the pertinent children of X Type A and all the other pertinent children Type W. This can be achieved by deleting

$$\alpha_1 = \sum_{i \in P(X)} w_i - \max_{i \in P(X)} \{(w_i - a_i)\}$$

pertinent leaves from T_i. In this case the pertinent child having $\max\{(w_i - a_i)\}$ will be called a_child(X).

Let $P_A(X)$ be a maximal consecutive sequence of pertinent children of X such that all the nodes in $P_A(X)$ except the leftmost and the rightmost ones are full. The endmost nodes may be full or partial. Then we can make X Type A by making all the full nodes in $P_A(X)$ Type B, the partial nodes in $P_A(X)$ Type H and all the other pertinent

children of X Type W. Note that there may be more than one $P_A(X)$. Thus we can make X Type A by deleting

$$\alpha_2 = \sum_{i \in P(X)} w_i - \max_{P_A(X)} \left\{ \sum_{i \in P_A(X)} (w_i - h_i) \right\}$$

pertinent leaves from T_i. In this case we call the leftmost node in the $P_A(X)$ selected as h_child2(X). Thus node X can be made Type A with the deletion of

$$a = \min\{\alpha_1, \alpha_2\}$$

pertinent leaves from T_i. If the value of a is different from α_1, then we make a_child(X) empty.

Traversing T_i bottom-up we can compute the [w,h,a] number for each pertinent node in T_i using the above formulas. This is described in the following procedure.

```
procedure COMPUTE1(T_i);
comment procedure COMPUTE1 computes the [w,h,a] number for
        each pertinent node in T_i. For each pertinent node
        X which is not a leaf, h_child1(X), h_child2(X) and
        a_child(X) are also determined.
begin
  for each pertinent leaf X in T_i do
   begin
     put X into the queue;
     initialize w := 1, h := 0, and a := 0 for X
```

```
  end;
ROOT_PROCESSED := false;
while the queue is not empty and not ROOT_PROCESSED do
  begin
  remove a node X from the queue;
    if X is the pertinent root
      then
        ROOT_PROCESSED := true;
```

$$w := \sum_{i \in P(X)} w_i;$$

```
    if X is full
      then begin
        h := 0;
        a := 0
      end
      else
        if X is a P-node
          then begin
            {Traverse the pertinent children of X}
            find h_child1(X) having max1 {(w_i-h_i)};
                                 i∈Par(X)
            find h_child2(X) having max2 {(w_i-h_i)};
                                 i∈Par(X)
            find a_child(X) having max {(w_i-a_i)};
                                 i∈Par(X)
```

$$h := \sum_{i \in Par(X)} w_i - (w_i - h_i)\Big|_{i=h\_child1(X)};$$

$\alpha_1 := w - (w_i - a_i)\Big|_{i=a\_child(X)}$;

$\alpha_2 := h - (w_i - h_i)\Big|_{i=h\_child2(X)}$;

$a := \min\{\alpha_1, \alpha_2\}$;

if $a \neq \alpha_1$

then

a_child(X) := nil

end

else begin

{X is a Q-node. Traverse the children of X from left to right}

determine $P_L(X)$, $P_R(X)$ and different $P_A(X)$'s;

find h_child1(X) corresponding to

$$h_1 := \max\left\{ \sum_{i \in P_L(X)} (w_i - h_i), \sum_{i \in P_R(X)} (w_i - h_i) \right\};$$

find h_child2(X) corresponding to

$$h_2 := \max_{P_A(X)} \left\{ \sum_{i \in P_A(X)} (w_i - h_i) \right\};$$

find a_child(X) corresponding to

$$a := \max_{i \in P(X)} \{(w_i - a_i)\};$$

$h := w - h_1$;

$\alpha_1 := w - a$;

$\alpha_2 := w - h_2$;

$a := \min\{\alpha_1, \alpha_2\}$;

if $a \neq \alpha_1$

```
            then
                a_child(X) := nil
        end;
    mark X "processed";
    {PARENT(X) denotes the parent of node X in T_i} .
    Y := PARENT(X);
    increment the number of children of Y processed;
    if all pertinent children of Y are processed
      then
        put Y into the queue
  end
end COMPUTE1;
```

Cost of procedure COMPUTE1 is established in the following lemma.

LEMMA 9.1.

Procedure COMPUTE1 correctly computes the [w,h,a] numbers for all the pertinent nodes in $O(n^2)$ time.

Proof:

Proof of correctness follows from our discussions so far.

As regards the complexity, note that for a Q-node procedure COMPUTE1 traverses all the children of the node. Thus the amount of work done for the Q-nodes in a T_i is proportional to the number of children of all the Q-nodes in

T_i. The children of a Q-node corresponding to a block represent vertices, except the lowest, on the outside window of the block. Moreover, any vertex in G which is represented as a child of a Q-node in T_i can appear on the outside window of only one block. Thus the total number of children of all the Q-nodes in T_i is less than or equal to n, the number of vertices in G.

For a P-node, the work done by procedure COMPUTE1 is proportional to the number of its pertinent children. A pertinent child of a P-node is either a P-node or a Q-node or a leaf. Since a Q-node represents a block, there are no more than n Q-nodes in any T_i. Also the number of pertinent leaves in T_i is in-deg(i+1), where in-deg(i+1) is the number of edges entering vertex i+1 in G. Furthermore the number of P-nodes in T_i is at most i. Thus the amount of work for all the P-nodes in T_i is $O(n + \text{in-deg}(i+1))$.

It follows from the above that the amount of work done by procedure COMPUTE1 for all the Q-nodes and P-nodes is $O(n + \text{in-deg}(i+1))$. Summing up the work done for all T_i's, we get the complexity of procedure COMPUTE1 as $O(m+n^2) = O(n^2)$. □

After computing the [w,h,a] number for the pertinent root of T_i, we can determine whether $\bar{T}_i$ is reducible or not. If the minimum of h and a is zero for the pertinent root of

T_i, then T_i is reducible. If T_i is not reducible, then we make the pertinent root of T_i Type H or A depending on which one of h and a is minimum, and make T_i reducible by deleting the necessary pertinent leaves from T_i. Now we need to determine the type of each pertinent node in T_i to obtain a reducible T_i. Note that T_i may have certain full nodes. If we decide to keep any such full node, then we mark it Type B.

Consider now a pertinent node X in T_i whose type has been determined. To start with X is the pertinent root. We can determine the types of all the pertinent children of X uniquely from the type of X as follows.

If X is Type B, then it is a full node and we would like to keep X as well as all its descendants in T_i. So no action needs to be taken in this case.

On the other hand, if X is not Type B, then we traverse the pertinent descendants of X to determine their type. An easy case is when X is a leaf. Then it should be Type W and so we have to delete it from T_i. We also have to remove the edge corresponding to X from G. Thus the edge corresponding to X should be included in E'_{i+1} in this case. If X is not a leaf, then we have the following different cases to consider.

Suppose X is Type W. Then all its pertinent children should be made Type W. Moreover, if any of these pertinent children is a full node, then the entire subtree of T_i rooted at that full child should be deleted from T_i.

If X is Type H and a P-node, then we make the partial child h_child1(X) Type H, all the full children Type B and all other partial children Type W. If X is Type H, but a Q-node, then we traverse the children of X from h_child1(X) towards the rightmost child and determine the maximal consecutive sequence of pertinent children $P_L(X)$ or $P_R(X)$. We then make all the nodes in this sequence Type B; the rightmost node in $P_L(X)$ or the leftmost node in $P_R(X)$ are made Type H and all other pertinent children of X are made Type W.

Suppose X is Type A and a P-node. Then we process the pertinent children of X as follows. If a_child(X) is not empty, then we make a_child(X) Type A and all other pertinent children Type W. On the other hand, if a_child(X) is empty, then we make the partial children h_child1(X) and h_child2(X) Type H, all full children of X Type B and all other partial children of X Type W. If X is Type A and a Q-node, then we should process its pertinent children as follows. If a_child(X) is not empty, then we make a_child(X) Type A and all other pertinent children Type W. If a_child(X) is empty, then we traverse the children of X

from h_child2(X) towards the rightmost child and find the maximal consecutive sequence $P_A(X)$ of pertinent children of X. Then we make all nodes in $P_A(X)$ Type B, the endmost nodes in $P_A(X)$, if they are partial, Type H and all other pertinent children Type W.

From the above discussions it should be clear that the type of any pertinent node in T_i uniquely determines the types of its pertinent children. Hence we process the PQ-tree T_i top-down from the pertinent root using the following procedure DELETE_NODES. During this processing we determine the set of edges E'_{i+1} and delete from T_i the nodes which are full and marked Type W. Since certain pertinent leaves are deleted from T_i, we have to update, if necessary, for each node the number of descendant leaves. Procedure DELETE_NODES performs this update also.

procedure DELETE_NODES(T_i);
comment procedure DELETE_NODES determines the type of each pertinent node in T_i. It also determines the set E'_{i+1} of edges to be removed from the nonplanar graph G and makes T_i reducible.

procedure DELETE(X);
comment procedure DELETE determines the type of each pertinent child of X. It updates the number of descendant leaves of node X and deletes X from T_i

```
        if X is full and marked Type W.
begin
  {FLAG is a Boolean variable which is set to true if X is
  to be deleted; and false otherwise}
  FLAG := false;
  if X is not Type B.
    then begin
      if X is a leaf
        then begin
          delete X from T_i;
          add the edge corresponding to X to E'_{i+1}
        end
        else begin
          case type of node X of
          Type W:
            begin
              mark all pertinent children of X Type W;
              if X is full
                then FLAG := true;
              {DESCENDANT_LEAVES(X) refers to the number
              of descendant leaves of X}
              DESCENDANT_LEAVES(X) :=
                DESCENDANT_LEAVES(X) - w
            end;
          Type H:
            begin
              if X is a P-node
```

```
then begin
  mark h_child1(X) Type H;
  mark all full children of X Type B;
  mark all other pertinent children of X
  Type W
end
else begin
  {X is a Q-node}
  determine P_L(X) or P_R(X) from
  h_child1(X);
  mark all children in P_L(X) or P_R(X)
  Type B;
  if P_L(X) is not empty
    then
      if the rightmost node in P_L(X) is
      partial
        then mark the rightmost node in
        P_L(X) Type H;
    else
      if the leftmost node in P_R(X) is
      partial
        then mark the leftmost node in
        P_R(X) type H;
  mark all other pertinent children of X
  Type W
end
DESCENDANT_LEAVES(X) :=
```

```
        DESCENDANT_LEAVES(X) - h
    end;
Type A:
    begin
      if a_child(X) ≠ nil
        then begin
          mark a_child(X) Type A;
          mark all other pertinent children of X
          Type W
        end
        else
          if X is a P-node
            then begin
              mark h_child1(X) and h_child2(X)
              Type H;
              mark all full children of X Type B;
              mark all other partial children of X
              Type W
            end
            else begin
              {X is a Q-node}
              determine P_A(X) from h_child2(X);
              mark all nodes in P_A(X) Type B;
              if the leftmost node in P_A(X) is
              partial
                then mark the leftmost node in
                P_A(X) Type H;
```

```
                    if the rightmost node in P_A(X) is
                    partial
                      then mark the rightmost node in
                      P_A(X) Type H;
                    mark all other pertinent children of
                    X Type W
                  end;
                DESCENDANT_LEAVES(X) :=
                  DESCENDANT_LEAVES(X) - a
              end
              end case
              for each pertinent child Y of X do
                DELETE(Y);
              if FLAG
                then delete X from T_i
            end
          end
        end DELETE;

begin
  DELETE(pertinent root of T_i)
end DELETE_NODES;
```

The following lemma shows that the edges in E'_{i+1} can be determined and removed from the nonplanar graph G in $O(n^2)$ time.

LEMMA 9.2.

Cost of procedure DELETE_NODES is $O(n^2)$.

Proof:

Note that for each node X procedure DELETE_NODES traverses the pertinent children if X is a P-node, and all the children if X is a Q-node. Thus it follows from the proof of Lemma 9.1 that the cost of procedure DELETE_NODES is $O(n^2)$. □

Having made T_i reducible, we can now reduce it to obtain T_i^* using Booth and Lueker's PQ-tree reduction algorithm. We can then obtain the next PQ-tree T_{i+1} and repeat our procedures to make T_{i+1} reducible. Note that the reduction of all the reducible PQ-trees can be performed in $O(m+n)$ time if we keep the parent pointers for all children of P-nodes and for the endmost children of Q-nodes. Thus in Booth and Lueker's algorithm, interior children of Q-nodes in any T_i are not assigned valid parent pointers and if any such interior child becomes pertinent, then its parent pointer will be determined during the bubble-up phase. In our discussions so far, we have assumed that the correct parent pointer for every pertinent node is available. So we have to determine the parent pointers of all the pertinent nodes in T_i before processing it. Booth and Lueker's planarity testing algorithm stops when it detects during the bubble-up phase that certain pertinent nodes cannot be

assigned parent pointers, for that would imply nonplanarity of the given graph. However, in our case we would like to proceed further to find parent pointers of all the pertinent nodes since our aim is to planarize the nonplanar graph. As a result our bubble-up algorithm described below is different from Booth and Lueker's.

Let X be a pertinent node in T_i. If X is a child of a P-node or one of the endmost children of a Q-node, then it has a valid parent pointer. On the other hand, if X is an interior child of a Q-node, then its parent pointer will be empty. To find the correct parent pointer for X, we traverse the siblings of X from X towards the rightmost child and obtain the parent pointer for X from that of the rightmost child. Let Y be the parent of X in T_i. If at a later time another child Z of Y is processed to find its parent pointer, then the above procedure would require traversing the children of Y upto the rightmost child and may result in visiting certain nodes several times. To avoid these unnecessary visits, when we traverse the children of Y from X to the rightmost child, we assign the parent pointer of the rightmost child to all the nodes traversed and store these nodes in a queue called interior_queue. So when a child Z of Y is processed, if its parent pointer is empty, then we traverse the siblings of Z until we find a node with a non-empty parent pointer. Though this path compression technique makes our bubble-up

procedure efficient, many non-pertinent children of Q-nodes may be assigned parent pointer. In order to make the parent pointer of such non-pertinent nodes empty, we process the interior_queue at the end of the bubble-up. If any node in this queue is not pertinent, then its parent pointer is made empty.

The efficiencies of our procedures COMPUTE1 and DELETE_NODES arise from the fact that we process only the pertinent children of any P-node. In a PQ-tree the pertinent children of a P-node may appear in any arbitrary order and so we may have to traverse all the children of a P-node to find the pertinent children. In order to avoid this, we split the children of each pertinent P-node into two groups - one group consisting of pertinent children only and the other consisting of only non-pertinent children. We now present our procedure BUBBLE_UP which finds the parent pointer for all the pertinent nodes in a PQ-tree and groups the pertinent children of P-nodes together. This procedure also computes the number of pertinent children as well as the number of descendant pertinent leaves of each pertinent node in the PQ-tree T_i.

procedure BUBBLE_UP(T_i);

comment procedure BUBBLE_UP determines the parent pointers for all pertinent nodes in T_i and groups together the pertinent children of each pertinent P-node. It

```
        also computes the number of pertinent children and
        the number of pertinent leaves of each pertinent
        node in T_i.
begin
  {PERTINENT_LEAVES(X) denotes the number of descendant
  pertinent leaves of node X}
  for the leaf X corresponding to an edge in E_{i+1} do
    begin
      mark X a pertinent node;
      PERTINENT_LEAVES(X) := 1;
      put X into pertinent_queue
    end;
  initialize interior_queue empty;
  ROOT_PROCESSED := false;
  while pertinent_queue is not empty and not ROOT_PROCESSED
  do
    begin
      remove a node X from the pertinent_queue;
      if PERTINENT_LEAVES(X) = |E_{i+1}|
        then begin
          {X is the pertinent root of T_i}
          PERTINENT_ROOT := X;
          ROOT_PROCESSED := true
        end
        else begin
          {PARENT(X) denotes the parent of node X in T_i}
          if PARENT(X) = nil
```

```
then begin
  {X is an interior child of a Q-node}
  traverse the siblings of X towards the
  rightmost child and find the sequence X, X1,
  X2, ..., Xk of nodes such that PARENT(Xj) =
  nil, 1 ≤ j < k, and PARENT(Xk) ≠ nil;
  for j := k-1 downto 1 do
    begin
      PARENT(Xj) := PARENT(Xk);
      put Xj into the interior_queue
    end;
  PARENT(X) := PARENT(Xk)
end
else
  if PARENT(X) is a P-node
    then begin
      remove X from the group of non-pertinent
      children of PARENT(X);
      put X into the group of pertinent children
      of PARENT(X)
    end;
{PERTINENT_CHILDREN(X) denotes the number of
pertinent children of node X}
PERTINENT_CHILDREN(PARENT(X)) :=
  PERTINENT_CHILDREN(PARENT(X)) + 1;
PERTINENT_LEAVES(PARENT(X)) :=
  PERTINENT_LEAVES(PARENT(X)) +
```

```
            PERTINENT_LEAVES(X);
          if PARENT(X) is not queued
            then begin
              mark PARENT(X) a pertinent node;
              put PARENT(X) into the pertinent_queue
            end
          end
        end;
      while interior_queue is not empty do
        begin
          remove a node X from interior_queue;
          if X is not marked pertinent
            then PARENT(X) := nil
        end
    end BUBBLE_UP;
```

The following lemma shows that procedure BUBBLE_UP has the same time complexity as the other procedures developed so far.

LEMMA 9.3.

Procedure BUBBLE_UP requires $O(n^2)$ time.

Proof:

For a PQ-tree T_i, the computational work done by procedure BUBBLE_UP for nodes which are children of Q-nodes is proportional to the number of children of all the Q-nodes in T_i, which is $O(n)$. The computational work done for nodes

which are children of P-nodes is proportional to the number of pertinent nodes in T_i, which is $O(n + \text{in-deg}(i+1))$. Thus the total work required for any T_i is $O(n + \text{in-deg}(i+1))$. Summing up this for all the PQ-trees T_i, $2 \leq i \leq n-2$, we get the time complexity of procedure BUBBLE_UP as $O(m+n) = O(n^2)$. □

Procedure COMPUTE1 and DELETE_NODES require that we should be able to determine whether a pertinent node in T_i is full or partial. A pertinent node is full if the number of descendant pertinent leaves of the node is equal to the number of its descendant leaves; otherwise it is partial. Procedure BUBBLE_UP determines the number of descendant pertinent leaves of every pertinent node in T_i. Now we should find a way of determining the number of descendant leaves of every pertinent node in T_i. Clearly each leaf has one descendant leaf. In T_1, the only node which is not a leaf is the P-node corresponding to vertex 1. Thus the number of descendant leaves of this P-node is the number of edges incident out of vertex 1 in G. We determine the number of descendant leaves of any node in T_i, $2 \leq i \leq n-2$, from the tree T_{i-1} as follows.

Assume that the number of descendant leaves of each node in T_{i-1} is known. During the processing of T_{i-1} we may delete some leaves from it to make it reducible. Note that procedure DELETE_NODES updates the number of descendant

leaves of the nodes in T_{i-1}. Thus in T^*_{i-1} also the correct number of descendant leaves for each node is known. Let $E_i = \{(j_1,i), (j_2,i), \ldots, (j_k,i)\}$ be the set of edges entering vertex i in the planar subgraph obtained from G. In T^*_{i-1} the leaves corresponding to the edges in E_i appear as children of the same node, say X. Since these leaves are removed from T^*_{i-1} to form T_i, the number of descendant leaves of the nodes corresponding to the vertices j_1, j_2, ..., j_k, if they are present in T_i, should be decreased by one and the number of descendant leaves of node X and its ancestors in T_i should be decreased by in-deg(i). Moreover, we construct T_i from T^*_{i-1} by adding a P-node corresponding to vertex i with leaves corresponding to the edges incident out of vertex i in G as its children. Clearly the number of descendant leaves of this P-node is equal to out-deg(i) in G. Since this node is made a child of node X, the number of descendant leaves of node X and all its ancestors in T_i should be increased by out-deg(i). Thus for node X and for each one of its ancestors in T_i, the net increase in the number of descendant leaves is (out-deg(i) - in-deg(i)). The following procedure performs this updating.

```
procedure UPDATE_DESCENDANTS(T_i);
comment procedure UPDATE_DESCENDANTS updates the number of
        descendant leaves of each node in T_i.
begin
  {Let E_i = {(j_1,i), (j_2,i), ..., (j_k,i)} be the set of
```

```
edges corresponding to the pertinent leaves in T*_{i-1}}
for p := j_1 to j_k do
  if there exists the empty P-node X corresponding to
  vertex p in T_i
    {DESCENDANT_LEAVES(X) denotes the number of descendant
    leaves of node X}
    then DESCENDANT_LEAVES(X) := DESCENDANT_LEAVES(X) - 1;
let X be the leaf corresponding to the edge (j_k,i);
repeat
  if PARENT(X) = nil
    then traverse the siblings of X towards the rightmost
    child until the rightmost child and find PARENT(X);
  X := PARENT(X);
  DESCENDANT_LEAVES(X) :=
     DESCENDANT_LEAVES(X) + out-deg(i) - in-deg(i)
until X is the P-node corresponding to vertex 1
end UPDATE_DESCENDANTS;
```

The following lemma shows the complexity of procedure UPDATE_DESCENDANTS.

LEMMA 9.4.

Procedure UPDATE_DESCENDANTS requires $O(n^2)$ computational work.

Proof:

For a T_i, $2 \leq i \leq n-2$, the updates for the nodes

corresponding to the vertices j_1, j_2, ..., j_k require O(in-deg(i)) time. The updates for the other nodes may, in the worst case, result in traversing all the nodes which are not leaves in T_i. This would require O(n) time. The total computational work required by procedure UPDATE_DESCENDANTS for all T_i's is therefore $O(m+n^2) = O(n^2)$. □

Now we present our planarization algorithm which uses the procedures developed so far. This procedure determines a spanning planar subgraph G_p of the nonplanar graph G and the sets E'_3, E'_4, ..., E'_{n-1} of edges to be removed from G to obtain G_p.

```
procedure PLANARIZE(G);
comment procedure PLANARIZE determines the set of edges E'3 =
        φ, E'4, ..., E'n-1 to be removed from a nonplanar
        graph G to obtain a spanning planar subgraph Gp.
begin
  {DESCENDANT_LEAVES(X) denotes the number of descendant
  leaves of node X}
  construct the initial PQ-tree T1 = T1*;
  DESCENDANT_LEAVES(1) := out-deg(1);
  for each leaf X corresponding to an edge in E2 do
    DESCENDANT_LEAVES(X) := 1;
  for i := 2 to n-2 do
    begin
      initialize E'i+1 to be empty;
```

```
      construct the PQ-tree T_i from T*_{i-1};
      UPDATE_DESCENDANTS(T_i);
      for the P-node X corresponding to vertex i do
        DESCENDANT_LEAVES(X) := out-deg(i);
      for each leaf X corresponding to an edge in E_{i+1} do
        DESCENDANT_LEAVES(X) := 1;
      BUBBLE_UP(T_i);
      COMPUTE1(T_i);
      if min{h,a} for the pertinent root is not zero
        then begin
          make the pertinent root Type H or A corresponding
          to the minimum of h and a;
          DELETE_NODES(T_i)
        end;
      reduce T_i to obtain T*_i .
    end
end PLANARIZE;
```

The complexity of procedure PLANARIZE is stated in the following.

THEOREM 9.4.

Procedure PLANARIZE determines a spanning planar subgraph of a nonplanar graph G in $O(n^2)$ time and $O(m+n)$ space.

Proof:

The fact that procedure PLANARIZE determines a spanning

planar subgraph of the nonplanar graph follows from our discussions and Theorem 9.3.

All the procedures used in procedure PLANARIZE are of time complexity $O(m+n^2)$. The PQ-tree reduction procedure is of time complexity $O(m+n)$. Thus procedure PLANARIZE is of time complexity $O(m+n^2) = O(n^2)$.

The space required by the procedure is bounded by the space required to store the PQ-trees, which is $O(m+n)$. Hence the theorem. □

We illustrate our graph-planarization algorithm on the nonplanar graph G shown in Fig. 9.10. In Figs. 9.11 to 9.19 we show the different PQ-trees T_1 to T_9. The [w,h,a] numbers of the pertinent nodes in these trees are shown within brackets adjacent to the nodes in these figures. Our algorithm determines $E'_6 = \{(2,6)\}$, $E'_8 = \{(2,8)\}$, and $E'_9 = \{(2,9), (3,9)\}$ as the sets of edges to be removed from G to planarize it and the spanning planar subgraph G_p is shown in Fig. 9.20. In Fig. 9.21 we show a planar embedding of G_p constructed using our planar embedding algorithm. From Fig. 9.21 we can easily see that the planar subgraph obtained is not maximally planar, since the edge (2,8) in E'_8 can be added to this embedding without affecting the planarity of the resultant graph. Thus the spanning subgraph determined by procedure PLANARIZE may not be

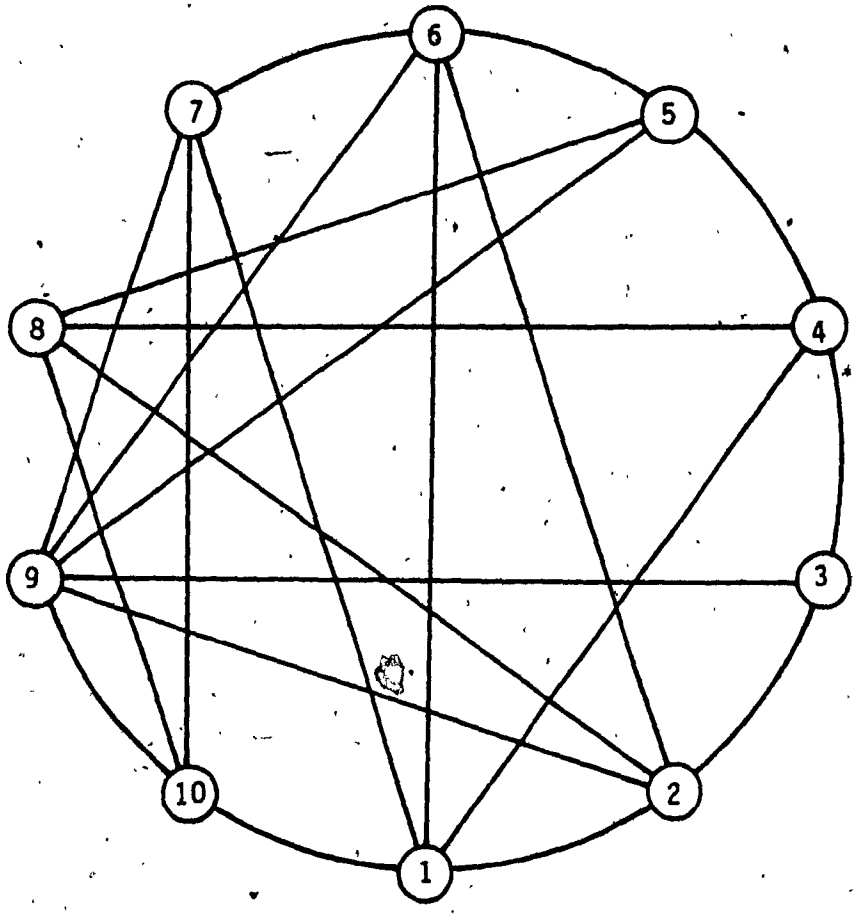


Figure 9.10

Nonplanar Graph G

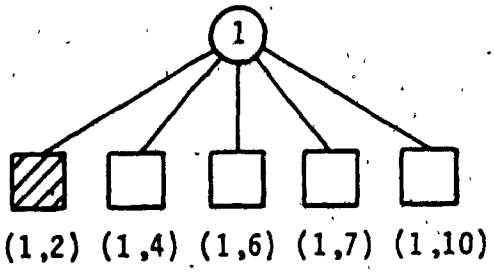


Figure 9.11

PQ-tree $T_1 = T_1^*$

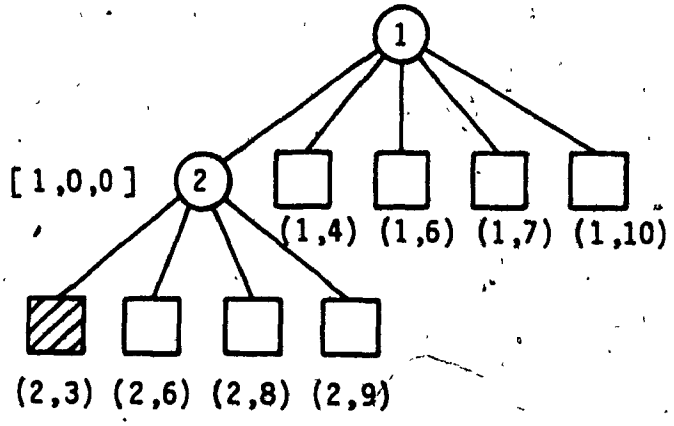


Figure 9.12

PQ-tree $T_2 = T_2^*$

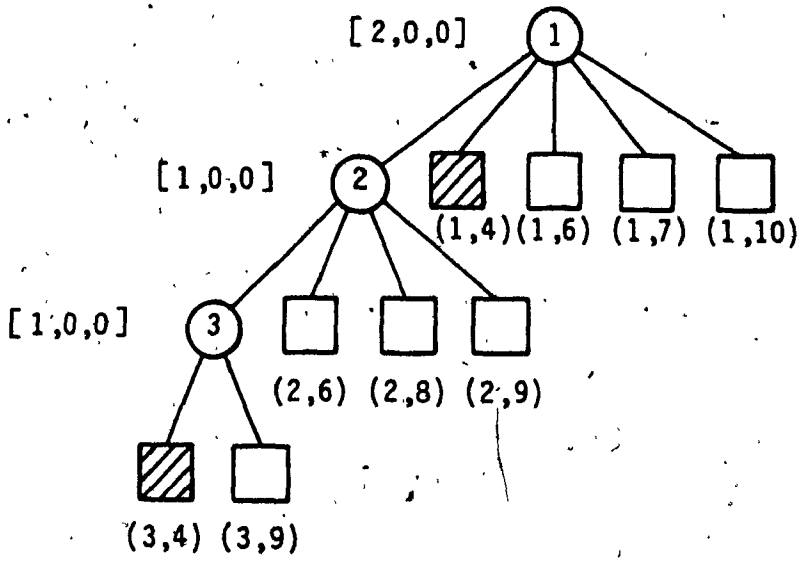


Figure 9.13(a)

PQ-tree T_3

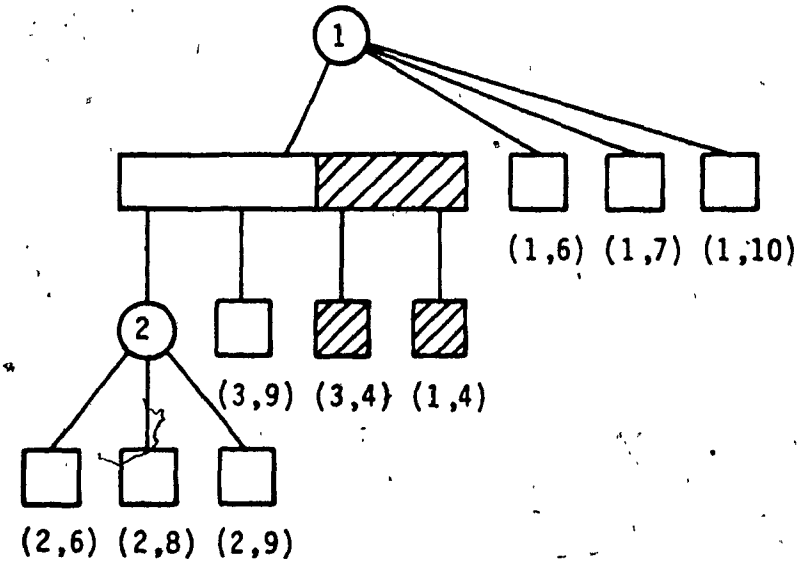


Figure 9.13(b)

PQ-tree T_3^*

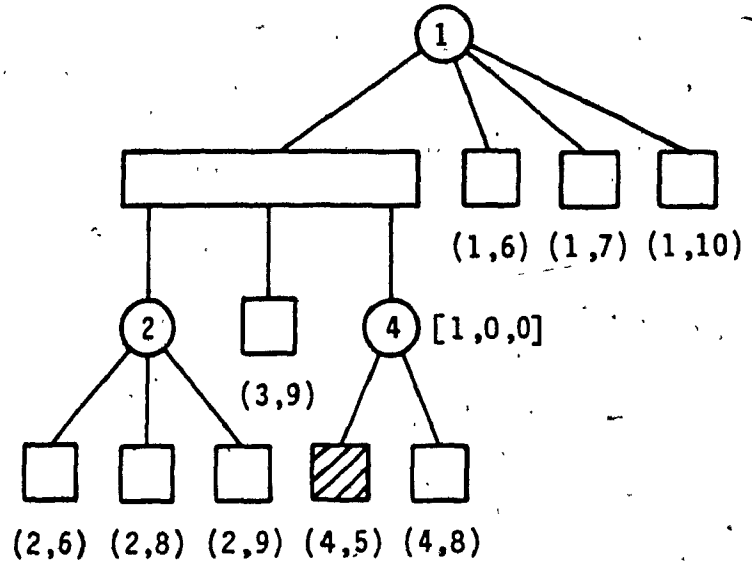


Figure 9.14

PQ-tree $T_4 = T_4^*$

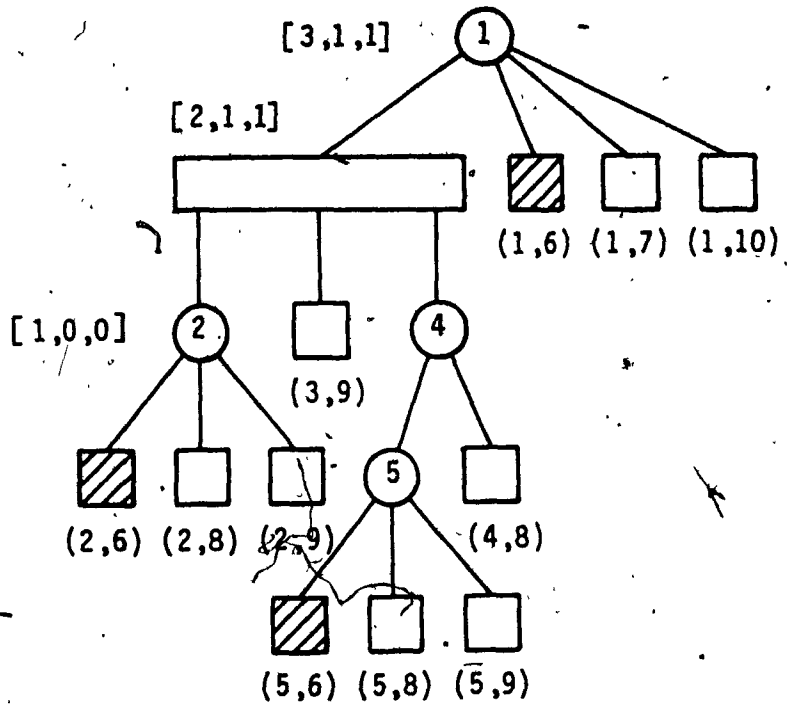


Figure 9.15(a)

PQ-tree T_5

Edge (2,6) is removed, $E_6' = \{(2,6)\}$

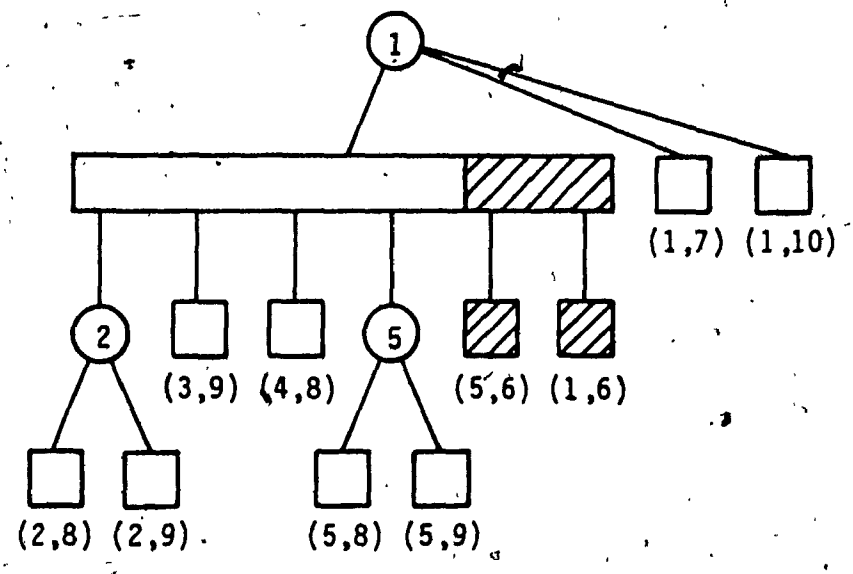


Figure 9.15(b)

PQ-tree T_5^*

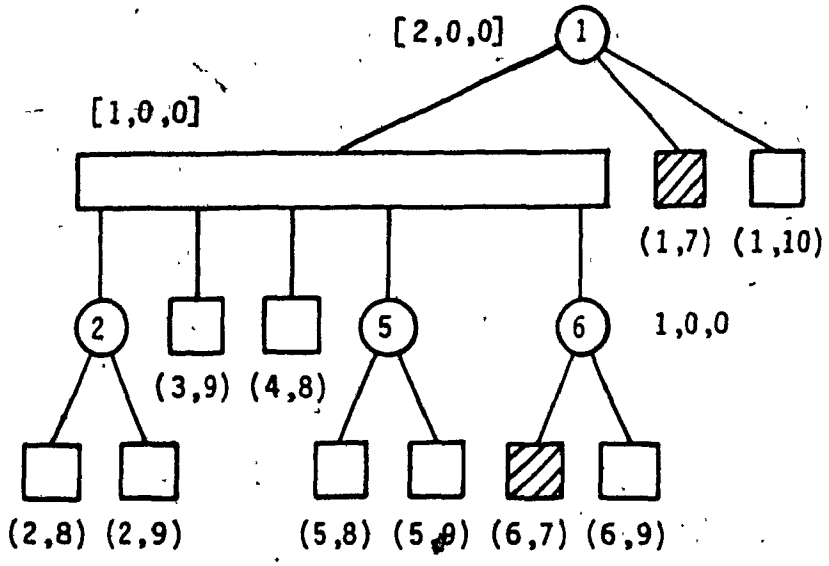


Figure 9.16(a)

PQ-tree T_6

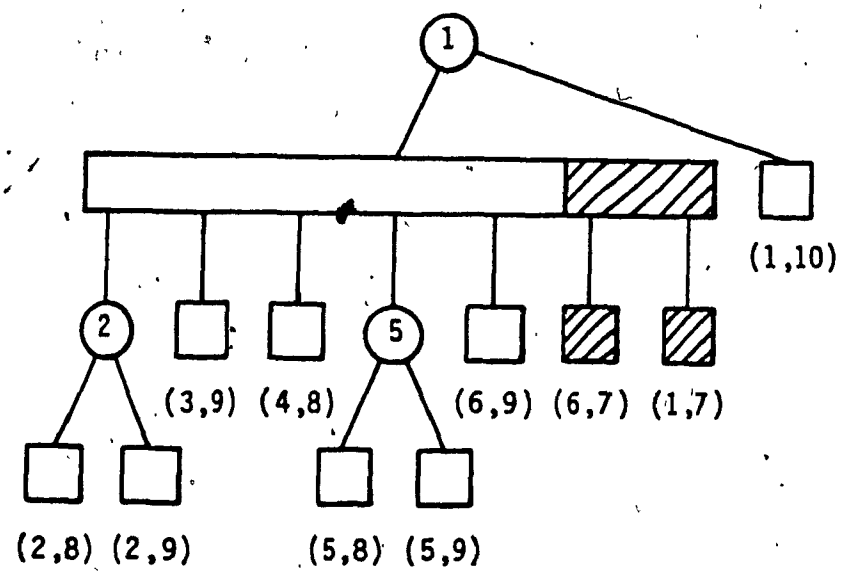


Figure 9.16(b)

PQ-tree T_6^*

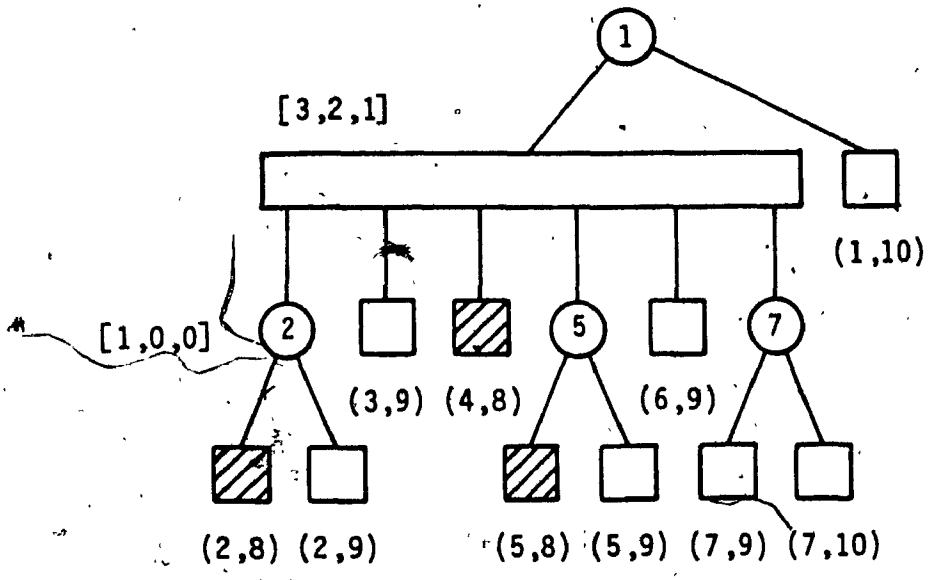


Figure 9.17(a)

PQ-tree T_7

Edge (2,8) is removed, $E_8' = \{(2,8)\}$

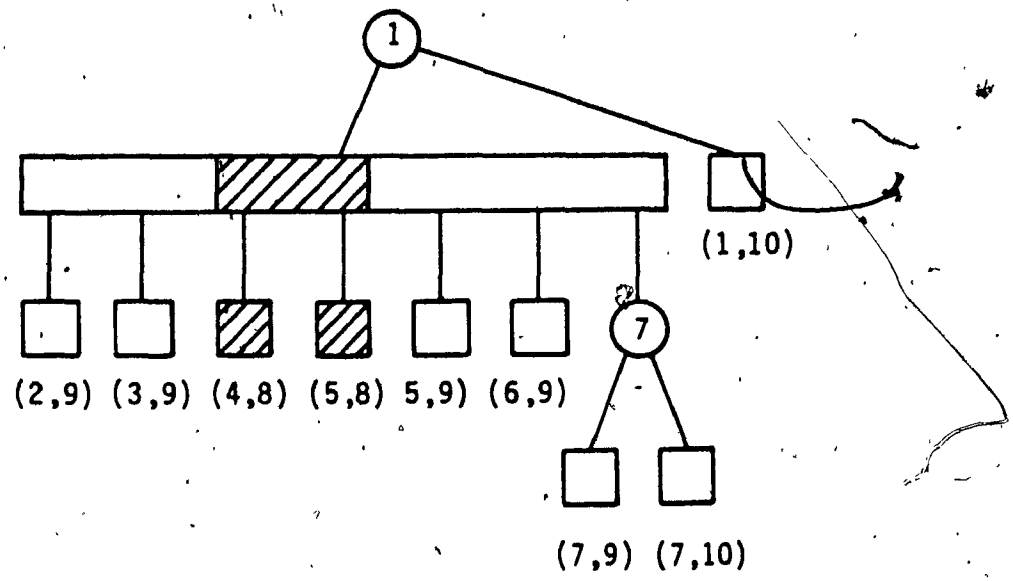


Figure 9.17(b)

PQ-tree T_7^*

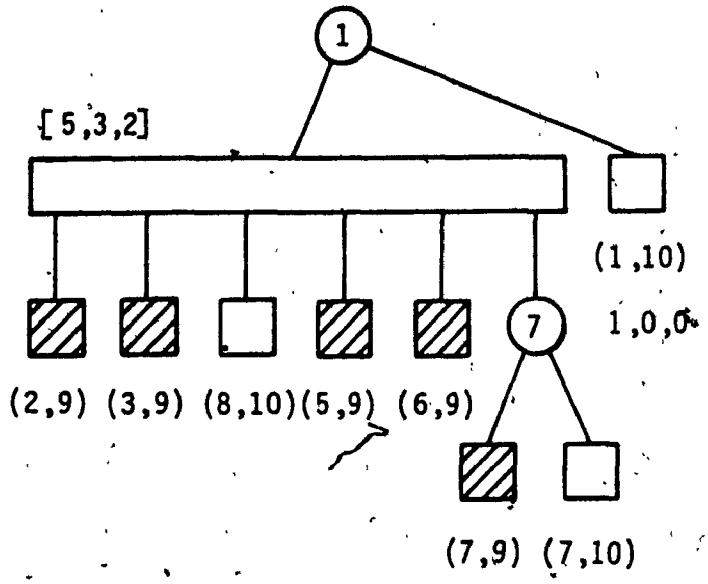


Figure 9.18(a)

PQ-tree T_8

Edges (2,9) and (3,9) are removed, $E'_9 = \{(2,9), (3,9)\}$

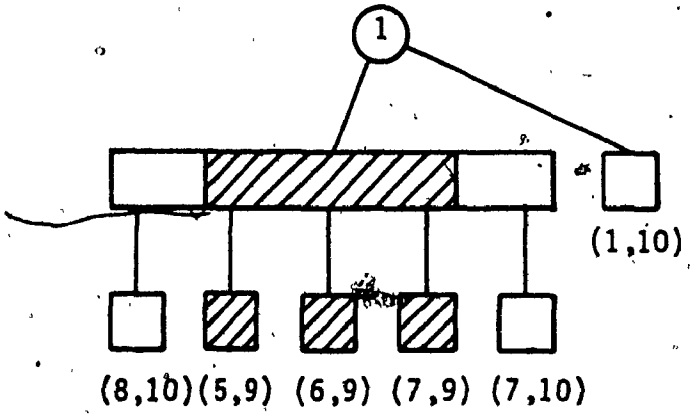


Figure 9.18(b).

PQ-tree T_8^*

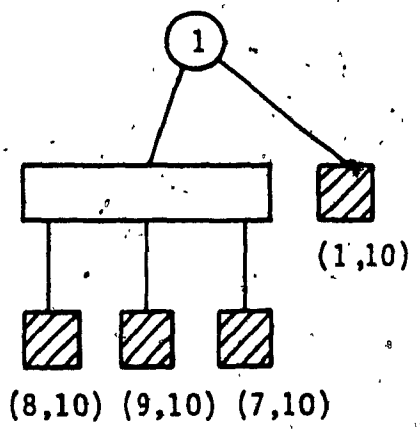


Figure 9.19

PQ-tree T_9

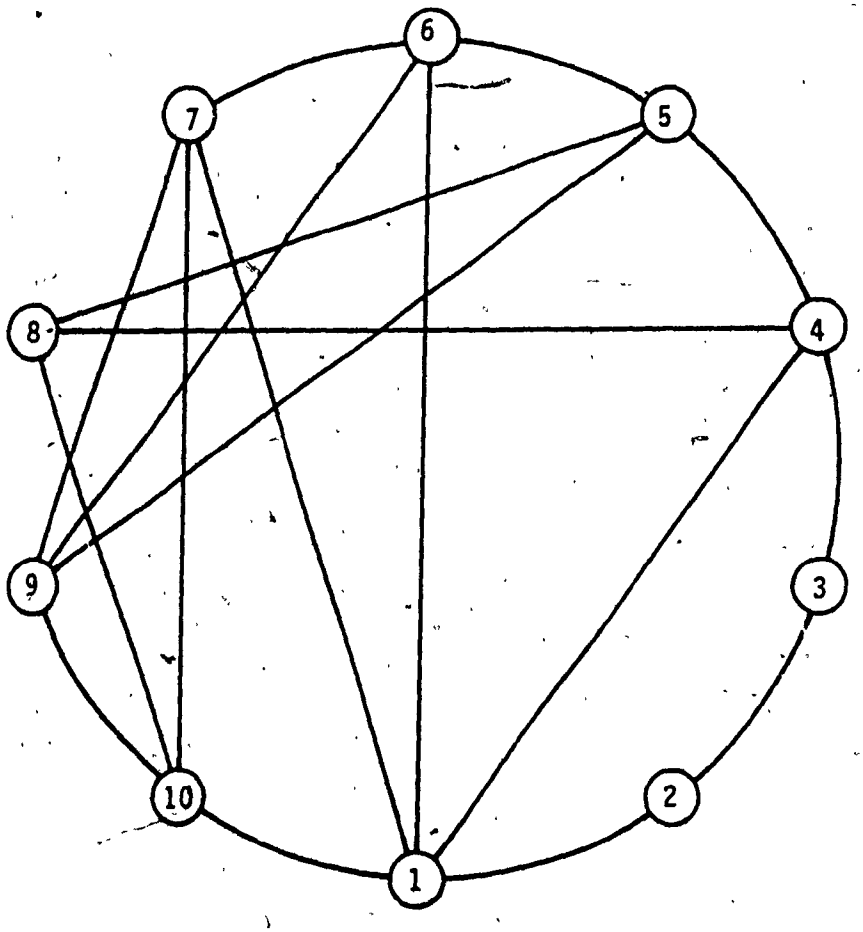


Figure 9.20

Spanning Planar Subgraph G_p

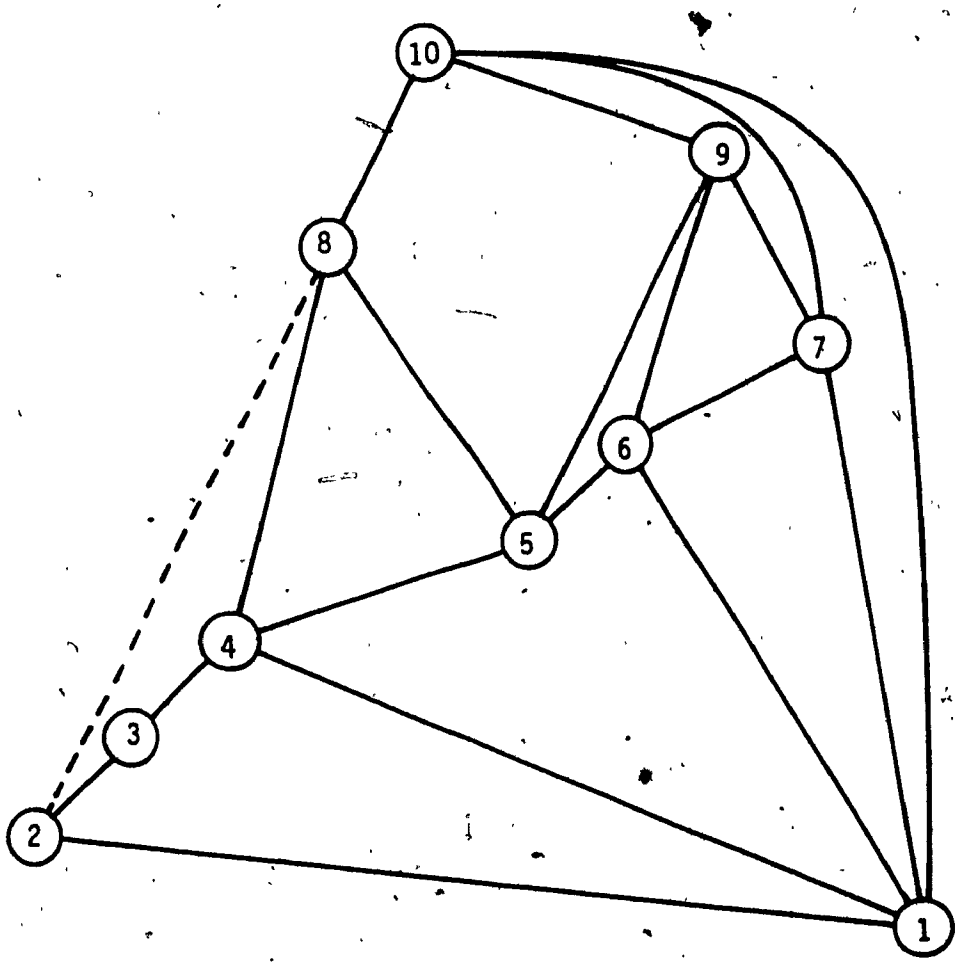


Figure 9.21

Planar Embedding of the Planar Subgraph G_p

Edge (2,8) can be added

maximally planar. In the next section we develop an efficient algorithm which determines a maximal planar subgraph starting with the planar subgraph determined by procedure PLANARIZE.

9.4 A Maximal Planarization Algorithm

In this section we develop an efficient algorithm to maximally planarize the spanning planar subgraph constructed by procedure PLANARIZE described in the previous section. Let G be the given nonplanar graph and G_p be the spanning planar subgraph constructed by procedure PLANARIZE. Let $E'_3 = \phi$, E'_4, ..., E'_{n-1} be the sets of edges removed by procedure PLANARIZE to obtain G_p. Our interest is to add to G_p as many edges from these sets as possible, without affecting the planarity of the resultant graph. We can achieve this in one of two ways.

One approach is to start with G_p and grow its bush forms and the corresponding PQ-trees. After constructing a PQ-tree, say $T_i(p)$, we may add to it as many leaves as possible representing the edges in the corresponding set E'_{i+1}. While doing so we should ensure that the reducibility of $T_i(p)$ is not affected. To add to $T_i(p)$ the leaves corresponding to the edges in E'_{i+1}, we have to first identify the P-nodes representing the lower numbered

vertices of these edges. It may so happen that for some of these edges such P-nodes may not be present in $T_i(p)$. We can overcome this problem by augmenting G by a new vertex n+1 and connecting this vertex to all the other vertices. However, this method is not elegant, though it will be very useful in constructing a nice planar embedding of G_p. So we shall not pursue this line of approach for maximally planarizing G_p.

The alternate approach to maximally planarize G_p is to start with G and construct its PQ-trees. After constructing a PQ-tree, say T_i, we make it reducible by deleting a minimum number of leaves representing the edges in E'_{i+1}. (Note that T_i will become reducible if all these leaves are deleted from T_i.) This can be easily done by computing the [w,h,a] number of the pertinent nodes in T_i. In the following, the leaves in T_i corresponding to the edges in E'_{i+1} will be called the <u>new pertinent leaves</u> of T_i and the other pertinent leaves of T_i (corresponding to the edges entering vertex i+1 in G_p) will be called <u>preferred leaves</u>. To compute the minimum number of new pertinent leaves to be removed from T_i, we may proceed as follows.

To start with we say the new pertinent leaves in T_i are "not processed" and compute the [w,h,a] numbers of all the pertinent nodes in T_i. Note that in the following "full" and "partial" are with respect to the graph G_p. Let X be a

pertinent node in T_i. We call X a preferred node if it has some of the preferred leaves among its descendants. Clearly, if X is full, then it is preferred and it should be retained in T_i. If X is not preferred, then it may either be retained in the reducible T_i or it may be deleted along with all its descendants to make T_i reducible.

Suppose X is a partial node. Then it can have at most two partial preferred children. First we consider the case when X is a P-node. If X has no partial preferred children, then it can be included in the reducible T_i only by making it Type H. So in this case we determine h_child1(X) and the h number of X and also set h_child2(X) and a_child(X) empty. If X has exactly one partial preferred child, then that preferred child has to be retained in T_i. Moreover, in this case X can be made Type H or A in a reducible T_i. So the partial preferred child becomes h_child1(X) and we determine h_child2(X) and the h and a numbers of X. We also set a_child(X) empty. On the other hand, if X has two partial preferred children, then it should be the pertinent root of the reducible T_i. So one of the partial preferred children of X becomes h_child1(X) and the other partial preferred child becomes h_child2(X). It is now easy to determine the a number for X. We also set a_child(X) empty and remember that the pertinent root is processed by setting the Boolean variable ROOT_PROCESSED to true.

If X is a Q-node, then all its preferred pertinent children should appear in one maximal consecutive sequence of pertinent children. In this case, we traverse the children of X from the leftmost child towards the rightmost child and determine the maximal consecutive sequence P'(X) of pertinent children of X such that

(i) P'(X) contains all the preferred children of X;

(ii) only the leftmost node and/or the rightmost node in P'(X) is partial; and

(iii) all the other nodes in P'(X) are full.

In this case X can be made Type H only when

(i) P'(X) appears at the left end of X and the leftmost node in P'(X) is not partial. In this case $P_L(X)$ = P'(X).

(or)

(ii) P'(X) appears at the right end of X and the rightmost node in P'(X) is not partial. In this case $P_R(X)$ = P'(X).

In both the above cases, we set h_child(X) to the leftmost node in P'(X) and compute the h number for X. If P'(X) does not satisfy either of the above two conditions, then $P_A(X)$ = P'(X). In this case X becomes the pertinent root of the reducible T_i. If P'(X) contains only one node, then the node in P'(X) should be made Type H or A corresponding to

the minimum of h and a. If P'(X) is made Type A, then the only node in P'(X) becomes a_child(X). If P'(X) has more than one node, then we set h_child2(X) to the leftmost node in P'(X) and compute the a number for X. We also remember in this case that the pertinent root is processed.

Note that some of the internal nodes in P'(X) and/or their descendants may be non-preferred leaves and all such non-preferred leaves should be deleted from T_i.

Processing the pertinent nodes of T_i upto the pertinent root using the above ideas, we can determine the [w,h,a] number of the pertinent nodes in T_i. This procedure is presented below in ALGOL-like notation.

```
procedure COMPUTE2(T_i);
comment procedure COMPUTE2 computes the [w,h,a] number of
        the pertinent nodes in T_i. For each pertinent node
        X which is not a leaf, a_child(X), h_child1(X) and
        h_child2(X) are also computed.
begin
  mark all old pertinent leaves preferred;
  for each pertinent leaf X in T_i do
    begin
      put X into the queue;
      initialize w := 1, h := 0, and a := 0 for X
    end;
```

```
ROOT_PROCESSED := false;
while the queue is not empty and not ROOT_PROCESSED do
  begin
    remove a node X from the queue;
    w := Σ_{i∈P(X)} w_i;
    if X is full
      then begin
        h := 0;
        a := 0
      end
      else
        if X is a P-node
          then begin
            case number of partial preferred children of X
            of
            0: begin
                 determine h_child1(X) which is the
                 partial child of X having
                   max_{i∈Par(X)} {(w_i-h_i)};
                 h_child2(X) := nil
               end;
            1: begin
                 h_child1(X) := the partial preferred
                 child of X;
                 h_child2(X) := the partial child of X
```

```
        having
              max      {(w_i-hi)};
            i ∈ Par(X)
            i≠h_childl(X)
      end;
2:    begin
        h_childl(X) :=  first  partial  preferred
        child of X;
        h_child2(X)  :=  second partial preferred
        child of X;
        ROOT_PROCESSED := true
      end
  end case
  a_child(X) := nil;
  h := Σ_{i ∈ Par(X)}  w_i - (w_i-h_i)|_{i=h_childl(X)};
  a := h - (w_i-h_i)|_{i=h_child2(X)}
end
else begin
  {X is a Q-node}
  traverse the children of X from left to  right
  and determine the maximal consecutive sequence
  of pertinent children P'(X);
  if any internal node of P'(X) has a descendant
  which is a non-preferred leaf
    then
      delete that non-preferred leaf from T_i;
```

```
if P'(X) = P_A(X)
  then begin
    ROOT_PROCESSED := true;
    if P'(X) has only one node
      then
        if a < h for the node in P'(X)
          then begin
            a_child(X) := the only node in
            P'(X);
            a := w - (w_i - a_i) | i=a_child(X)
          end
          else begin
            h_child2(X) := the only node in
            P'(X);
            a := w - (w_i - h_i) | i=h_child2(X)
          end
      else begin
        h_child2(X) := leftmost node in P'(X);
        a := w - Σ_{i ∈ P'(X)} (w_i - h_i)
      end
  end
  else begin
    h_child1(X) := leftmost node in P'(X);
    h := w - Σ_{i ∈ P'(X)} (w_i - h_i)
```

```
            end
        end
    {PARENT(X) denotes the parent of node X in T_i}
    if all the pertinent children of PARENT(X) are
    processed
      then put PARENT(X) into the queue;
    if X is a preferred node
      then mark PARENT(X) a preferred node;
    if X is the pertinent root
     then
       ROOT_PROCESSED := true
  end
end COMPUTE2;
```

The following lemma gives the complexity of procedure COMPUTE2.

LEMMA 9.5.

Procedure COMPUTE2 computes the [w,h,a] numbers of the pertinent nodes in all the PQ-trees in $O(n^2)$ time.

Proof:

It is easy to see that the computational work done by procedure COMPUTE2 is equal to or less than that of procedure COMPUTE1. Hence the proof follows from Lemma 9.1. □

Having computed the [w,h,a] numbers for the pertinent

nodes in T_i, we can obtain a reducible T_i by traversing the pertinent subtree top-down from the pertinent root using procedure DELETE_NODES. During this processing some of the new pertinent leaves in T_i may not be processed at all. It is easy to see that such pertinent leaves should be deleted from T_i to make it reducible and the edges corresponding to these leaves should also be removed from the nonplanar graph G to obtain a maximal planar subgraph.

Processing the PQ-trees T_2, T_3, ... T_{n-2} this way we obtain a maximal planar subgraph of the nonplanar graph G using the following procedure.

```
procedure MAXIMAL_PLANARIZE(G);
comment procedure MAXIMAL_PLANARIZE determines a maximal
        planar subgraph of the nonplanar graph G. This
        procedure uses the spanning planar subgraph obtained
        by procedure PLANARIZE.
begin
  {Determine the spanning planar subgraph}
  PLANARIZE(G);
  {Maximally planarize the spanning planar subgraph}
  construct the initial PQ-tree T1 = T1*;
  DESCENDANT_LEAVES(1) := out-deg(1);
  for each leaf X corresponding to an edge in E2 do
    DESCENDANT_LEAVES(X) := 1;
  for i := 2 to n-2 do
```

```
begin
  construct the PQ-tree T_i from T'_{i-1};
  UPDATE_DESCENDANTS(T_i);
  for the P-node X corresponding to vertex i do
    DESCENDANT_LEAVES(X) := out-deg(i);
  for each leaf X corresponding to an edge in E_{i+1} do
    DESCENDANT_LEAVES(X) := 1;
  BUBBLE_UP(T_i);
  COMPUTE2(T_i);
  if min{h,a} for the pertinent root is not zero
    then begin
      make the pertinent root Type H or A corresponding
      to the minimum of h and a;
      DELETE_NODES(T_i);
      delete the new pertinent leaves which are not
      processed from T_i
    end;
  reduce T_i and obtain T'_i
end
end MAXIMAL_PLANARIZE;
```

The complexity of procedure MAXIMAL_PLANARIZE is given in the following.

THEOREM 9.5.

Procedure MAXIMAL_PLANARIZE determines a maximal planar subgraph of a nonplanar graph in $O(n^2)$ time and $O(m+n)$

space.

Proof:

The fact that procedure MAXIMAL_PLANARIZE determines a maximal planar subgraph follows when we note that no edge can be added to the resultant planar subgraph without affecting its planarity.

All the procedures used in procedure MAXIMAL_PLANARIZE are of time complexity $O(n^2)$. The PQ-tree reductions can be performed in $O(m+n)$ time. Hence procedure MAXIMAL_PLANARIZE has an $O(n^2)$ time complexity.

Regarding the space complexity, note that the space required by the algorithm is bounded by the space required to store the different PQ-trees, which is $O(m+n)$. □

We now illustrate procedure MAXIMAL_PLANARIZE on the nonplanar graph shown in Fig. 9.10. We start with the spanning planar subgraph G_p determined by procedure PLANARIZE, which is shown in Fig. 9.20. In Figs. 9.22 to 9.30 we show the different PQ-trees obtained during procedure MAXIMAL_PLANARIZE. In these figures, adjacent to each pertinent node we show its [w,h,a] number, and the new pertinent leaves as shown as triangles. From Fig. 9.26(a) we can see that the edge (2,6) from E_6' can be added to G_p without affecting the planarity. The maximal planar

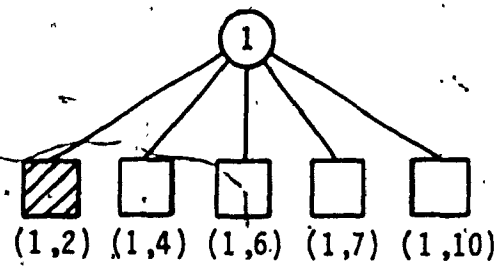


Figure 9.22

PQ-tree $T_1 = T_1^*$

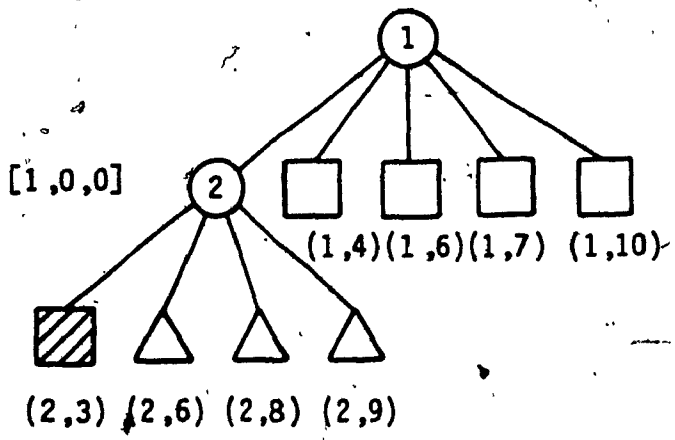


Figure 9.23

PQ-tree $T_2 = T_2^*$

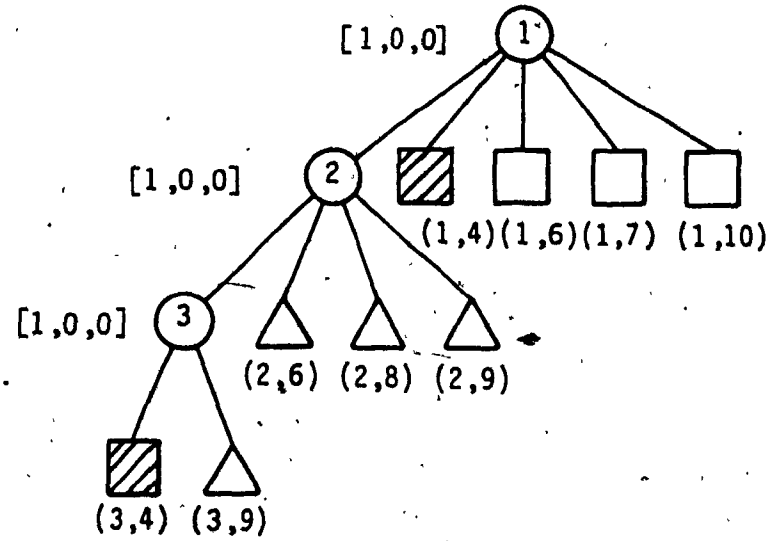


Figure 9.24(a)

PQ-tree T_3

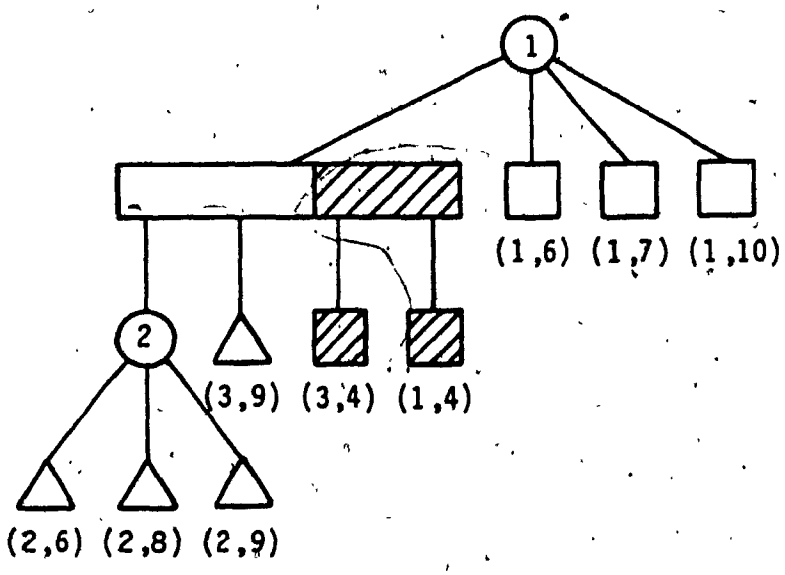


Figure 9.24(b)

PQ-tree T_3^*

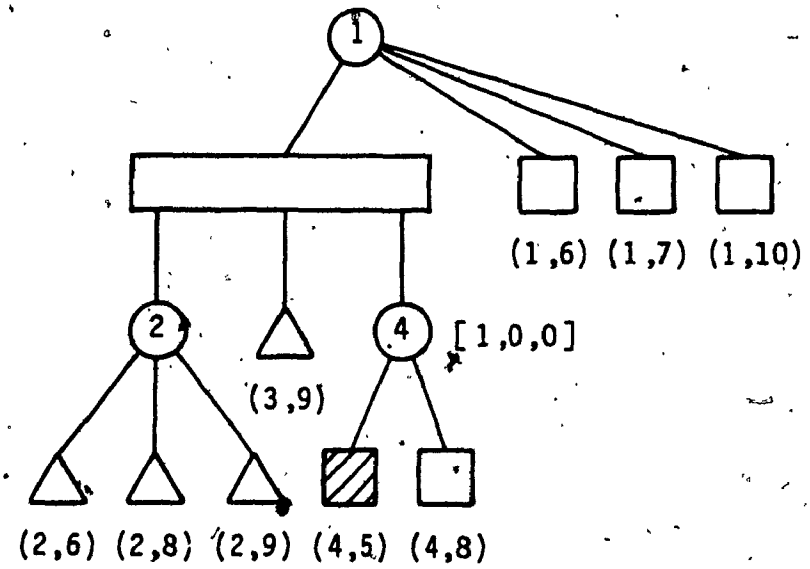


Figure 9.25

PQ-tree $T_4 = T_4^*$

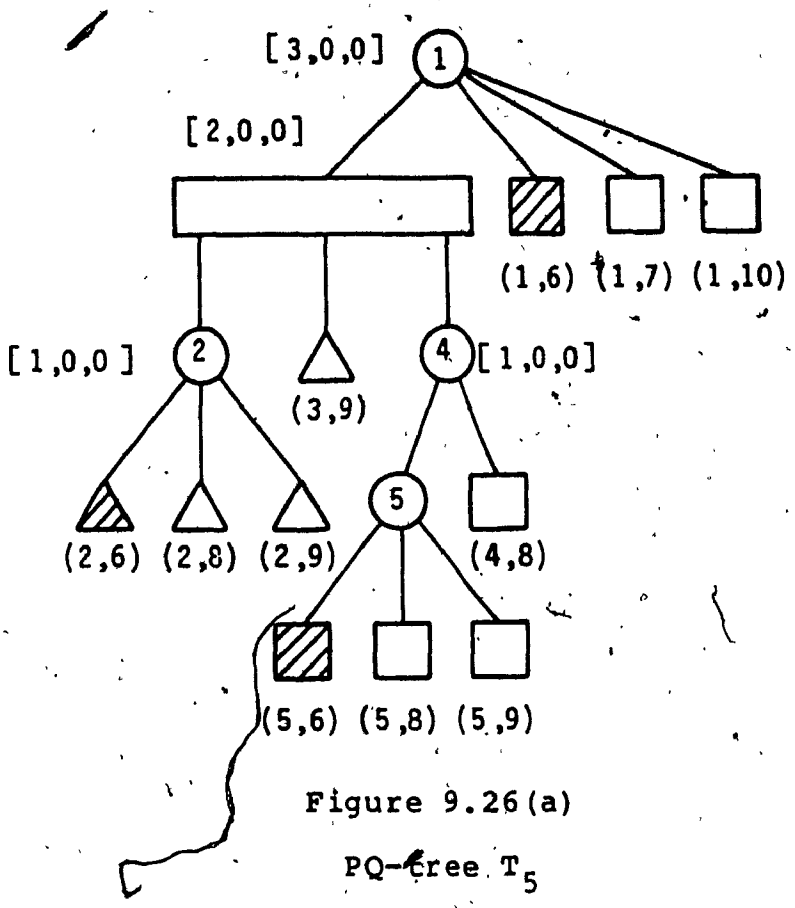


Figure 9.26(a)

PQ-tree T_5

Edge (2,6) can be added

Edges (2,8), (2,9) and (3,9) must be removed

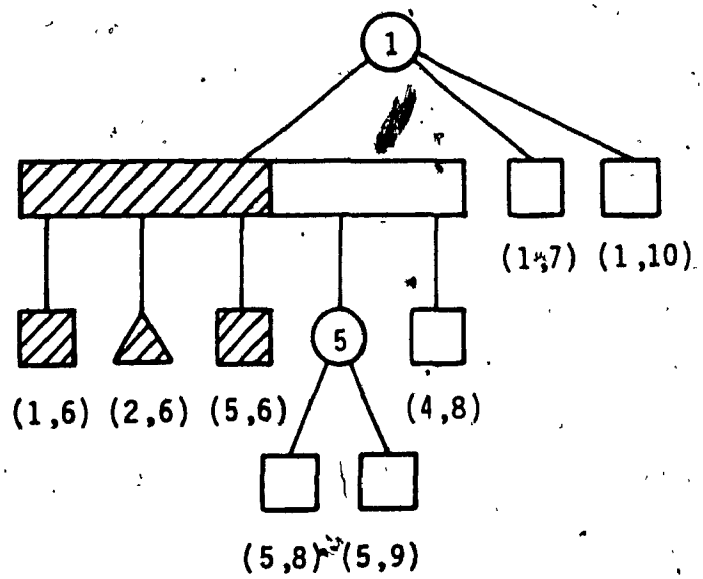


Figure 9.26(b)

PQ-tree T_5^*

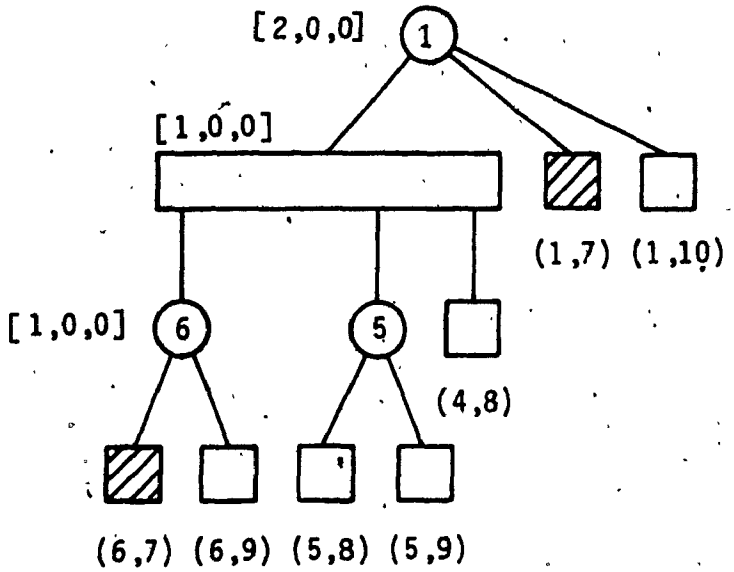


Figure 9.27(a)

PQ-tree T_6

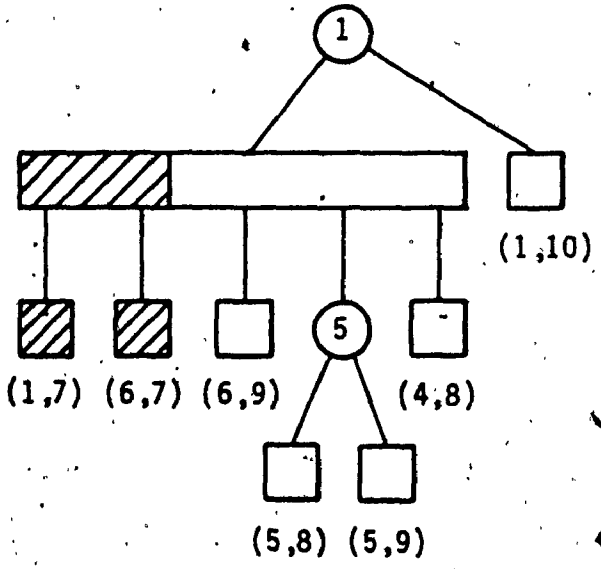


Figure 9.27(b)

PQ-tree T_6^*

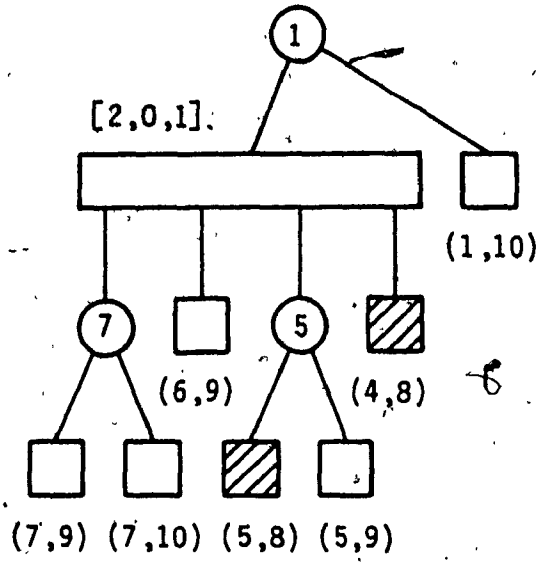


Figure 9.28(a)

PQ-tree T_7

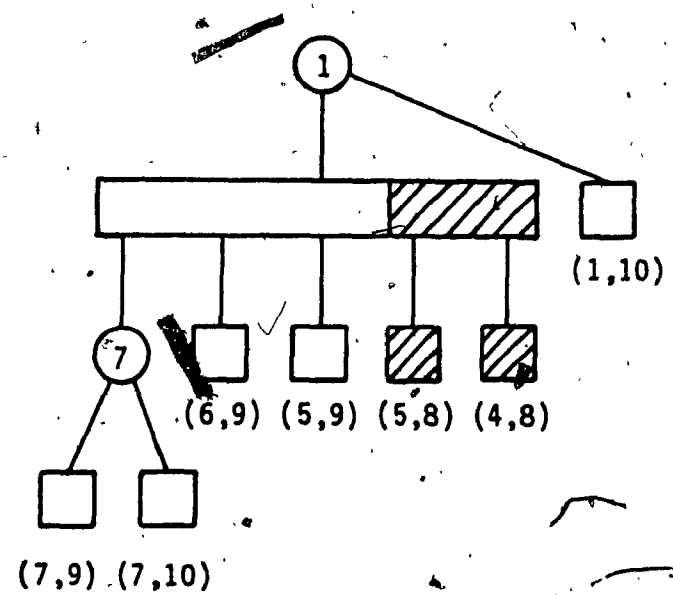


Figure 9.28(b)

PQ-tree T_7^*

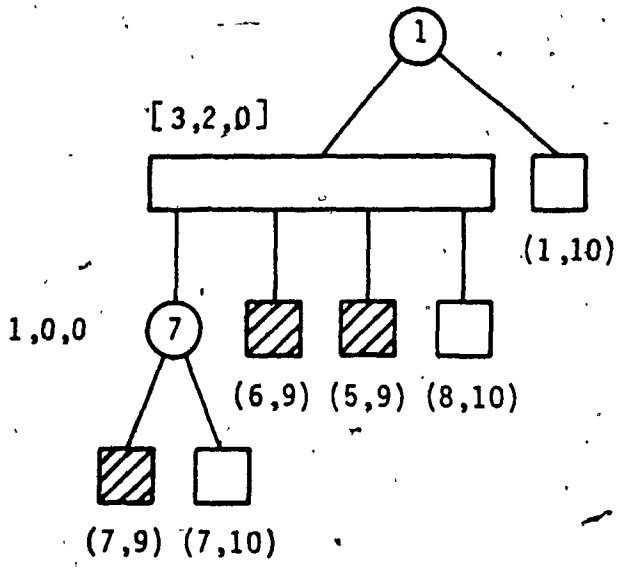


Figure 9.29(a)

PQ-tree T_8

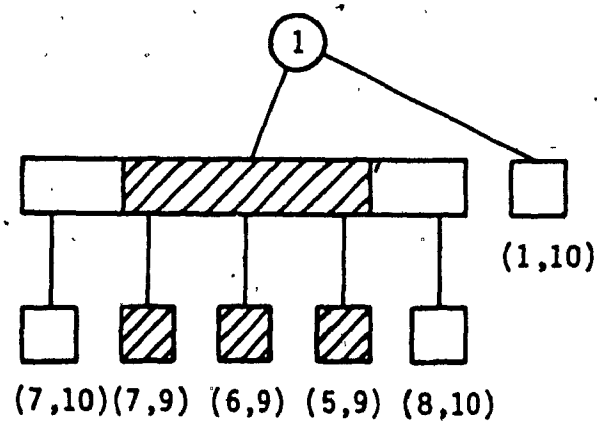


Figure 9.29(b)

PQ-tree T_8^*

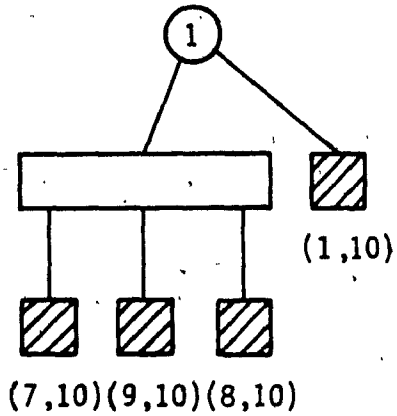


Figure 9.30

PQ-tree T_9

subgraph determined by procedure MAXIMAL_PLANARIZE is shown in Fig. 9.31 and Fig. 9.32 shows a planar embedding of this graph. From Fig. 9.32 we can easily verify that the subgraph determined by procedure MAXIMAL_PLANARIZE is a maximal planar subgraph of the nonplanar graph G shown in Fig. 9.10.

It is easy to see that any biconnected spanning planar subgraph of the nonplanar graph can be used as the starting graph for procedure MAXIMAL_PLANARIZE. However, we use the spanning planar subgraph G_p determined by procedure PLANARIZE as the starting graph because while obtaining G_p we have already attempted to include as many edges as possible and so procedure MAXIMAL_PLANARIZE will be required to add only a small number of edges to G_p to determine the maximal planar subgraph.

From Theorem 9.5 it is clear that the $O(n^2)$ time procedure MAXIMAL_PLANARIZE is computationally superior to both Chiba, Nishioka and Shirakawa's algorithm [64] and Ozawa and Takahashi's algorithm [49]. Moreover, our algorithm can easily be modified to determine a maximal planar subgraph of a nonplanar graph G such that the maximal planar subgraph contains a desired set of edges of G.

We have implemented procedure MAXIMAL_PLANARIZE in PASCAL and tested it on several nonplanar graphs using a CDC

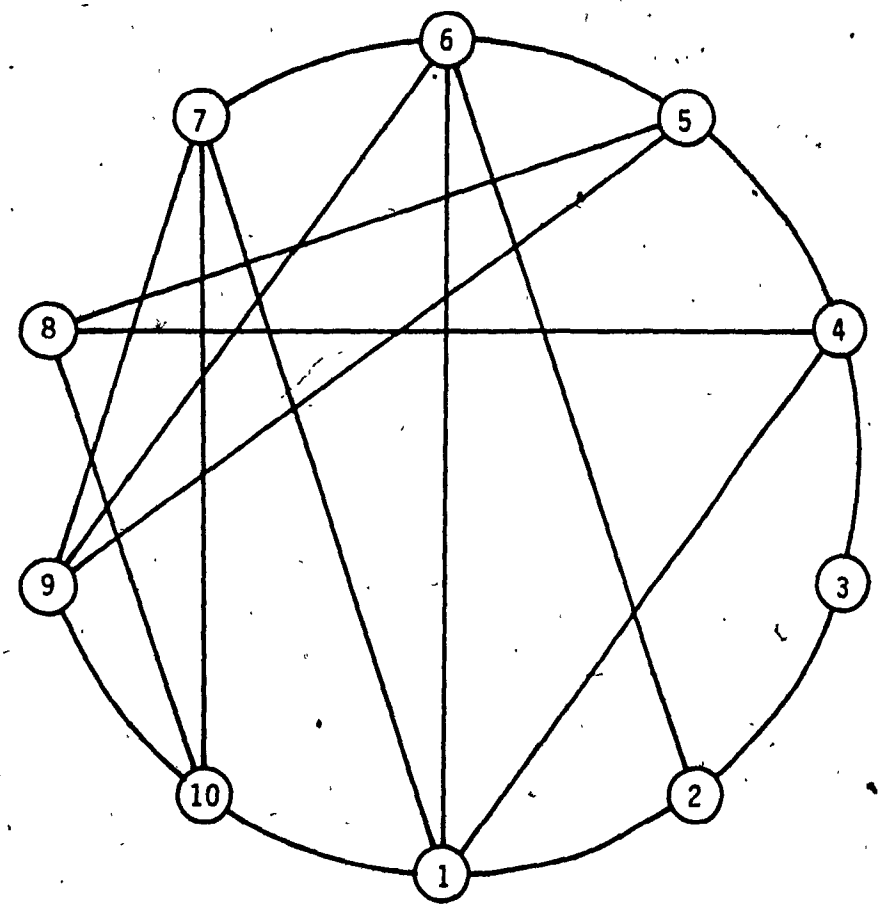


Figure 9.31

Maximal Planar Subgraph

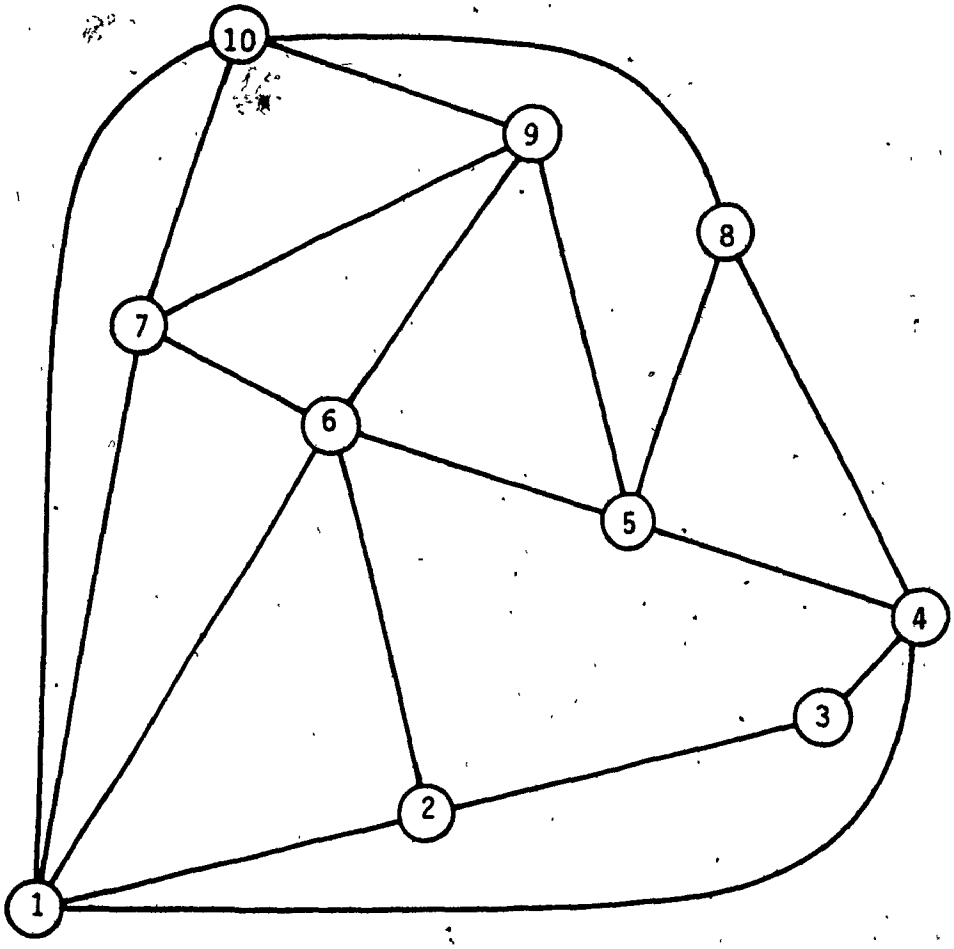


Figure 9.32

Planar Embedding of the Maximal Planar Subgraph

Cyber 170. In Table 9.1 we show the number of edges removed by procedure PLANARIZE and the number of edges added by procedure MAXIMAL_PLANARIZE for some of the test graphs. It can be seen from Table 9.1 that procedure MAXIMAL_PLANARIZE adds only a very small number of edges to the spanning planar subgraph. Finally, in Table 9.2 we show the execution time required to find a maximal planar subgraph for these graphs.

Table 9.1

Number of Edges Removed and Number of Edges Added

Graph	Number of vertices	Number of edges	Number of edges removed by procedure PLANARIZE	Number of edges added by procedure MAXIMAL_PLANARIZE
G_1	10	35	21	3
G_2	20	60	24	0
G_3	30	95	42	5
G_4	40	125	39	2
G_5	50	150	47	4
G_6	60	180	53	3
G_7	70	225	57	0
G_8	80	250	78	7
G_9	90	300	103	5
G_{10}	100	350	124	8

Table 9.2

Execution Time

Graph	Number of vertices	Number of edges	Execution time in seconds
G_1	10	35	0.263
G_2	20	60	0.672
G_3	30	95	0.976
G_4	40	125	1.321
G_5	50	150	1.985
G_6	60	180	3.126
G_7	70	225	4.795
G_8	80	250	5.013
G_9	90	300	6.792
G_{10}	100	350	7.863

CHAPTER 10

SUMMARY AND PROBLEMS FOR FURTHER INVESTIGATION

In this chapter we summarize the main results of the thesis and point out a few problems for further study.

10.1 Summary

In Part I (Chapters 2 to 6) of the thesis a detailed study of the computational complexity of Char's spanning tree enumeration algorithm has been carried out. A brief review of some of the well-known spanning tree enumeration algorithms has been given in Chapter 2. We have given in Chapter 3 a description of Char's algorithm and a detailed analysis of this algorithm for general graphs. Specifically, an expression for the number of sequences - tree sequences and non-tree sequences - generated by Char's algorithm has been derived, and based on this expression, certain properties of the algorithm have been established. The two types of computations performed by the algorithm are identified and the costs of these computations have been obtained. Using a crude bound for the total number of sequences generated, we have shown that Char's algorithm is of complexity $O(n^3t)$, where t is the number of spanning trees. Two heuristics have been proposed for selecting the

initial spanning tree to be used in the algorithm. These heuristics aim at reducing the number of non-tree sequences generated. We have given an implementation of the algorithm using path compression which helps reduce the number of comparisons made by the algorithm. We have also shown that use of path compression does not affect the complexity of the algorithm as obtained before. We have concluded Chapter 3 with our computational experiences with Char's algorithm when implemented using the heuristics and path compression.

Analysis of Char's algorithm for certain special graphs has been carried out in Chapter 4. A class of graphs for which the algorithm is of complexity O(nt) has been identified. The complete graph, the ladder and the wheel belong to this class. For these graphs, we have obtained expressions (as functions of n) for the total number of sequences generated by Char's algorithm. We have also shown that in the cases of the ladder and the wheel, the algorithm requires, on the average, at most 4 computational steps to generate a spanning tree.

An efficient implementation of Char's algorithm has been given in Chapter 5. We have shown that this modified algorithm, called algorithm MOD-CHAR, is of complexity $O(nH_n t)$ which is $O(n^2 t)$ in the worst case. Classes of graphs for which algorithm MOD-CHAR is of complexity O(nt)

have been identified. These classes are more general than the one considered in Chapter 4. We have shown that in the case of a large complete graph ($n \geq 8$), algorithm MOD-CHAR requires, on the average, at most 10 computational steps per spanning tree generated. We have also given our computational experiences with algorithm MOD-CHAR and observed that Char's algorithm is superior to algorithm MOD-CHAR though the latter has a better asymptotic complexity.

In Chapter 6, the final chapter of Part I, a computational evaluation of Char's algorithm in comparison to the algorithm by Gabow and Myers has been given. To make the evaluation independent of implementation details, the number of basic operations performed by these algorithms has been used as a measure of efficiency of the algorithms. Again we have observed that Char's algorithm is superior to both algorithm MOD-CHAR and Gabow and Myers' algorithm. In most of the cases, Char's algorithm is five times as fast as Gabow and Myers' algorithm.

In Part II (Chapters 7 to 9) of the thesis we have developed efficient algorithms for obtaining a planar embedding of a planar graph and for obtaining a maximal planar subgraph of a nonplanar graph. These algorithms are based on Lempel, Even, and Cederbaum's planarity testing algorithm (the LEC algorithm) and its implementation using PQ-trees. To make the discussions in Part II self-

contained, a description of the LEC algorithm and its PQ-tree implementation have been given in Chapter 7.

The planar embedding procedure developed in Chapter 8 starts with an st-numbering of the given planar graph and involves placing the vertices at different vertical and horizontal levels, so that in the final embedding no two vertices appear in the same vertical or horizontal levels. The vertical levels of the vertices are dictated by their st-numbers. The order of the vertices as we scan the final embedding from left to right is called vertex order. The anticlockwise order in which edges from lower numbered vertices enter a vertex in the final planar embedding is called the τ'-order of that vertex. In Chapter 8 first an O(n) algorithm to obtain the τ'-orders of all the vertices has been developed. Then we have designed an O(n) algorithm to obtain the vertex order. This latter algorithm uses the τ'-orders of the vertices. An interesting property of the vertex order so obtained has been established. The vertex order and the st-numbers fix the positions of the vertices in the planar embedding. Finally, we have described a simple procedure to draw by hand the edges without crossovers.

In Chapter 9, the problem of determining a maximal planar subgraph of a nonplanar graph has been considered. First we have shown that Ozawa and Takahashi's planarization

algorithm does not, in general, obtain a maximal planar subgraph. However, we have established that this algorithm determines a maximal planar subgraph in the case of a complete graph. The new maximal planarization algorithm described in this chapter is in two phases. In the first phase a spanning planar subgraph of the given nonplanar graph is determined. We have developed formulas to determine the minimum number of edges that need to be removed at each step in the first phase. In the second phase edges are added to the spanning planar subgraph so that at each step in this phase, a maximum number of edges are added to determine the maximal planar subgraph. We have shown that the complexity of this maximal planarization algorithm is $O(n^2)$. Finally, results relating to the maximal planarization algorithm have been tabulated.

10.2 Problems for Further Investigation

Our analysis in Part I has shown that Char's algorithm can be implemented with complexity $O(nH_n t)$, which is $O(n^2 t)$ in the worst case. We may recall that Gabow and Myers' algorithm has $O(nt)$ complexity. We believe that the poor complexity of Char's algorithm in relation to Gabow and Myers' algorithm is more a result of our inability to obtain a bound for H_n which is tighter than the one, namely $H_n \leq n$, we have used. To conclusively establish the superiority of

Char's algorithm, we have to investigate H_n further.

One line of approach is to show that all biconnected graphs with minimum degree greater than or equal to three admit the M-numbering defined in Section 5.2. Such a result will prove that Char's algorithm is of complexity O(nt), since this class of graphs is general enough as far as the spanning tree enumeration algorithms are concerned.

We can see that H_n is in fact the ratio of t and the sum of t(k)'s. Thus H_n can be expressed in terms of the determinant and the principal minors of the matrix AA^t, where A is a reduced incidence matrix of the graph. Thus another line of approach is to study H_n using this determinant approach.

Consider an n-vertex biconnected resistance network N consisting of only one ohm resistances. If the vertices of N are numbered as in Char's algorithm, then d_{n-1} is the driving point admittance of N across the terminals (n,n-1). A third line of approach to prove the superiority of Char's algorithm is to show that n/d_{n-1} converges to a constant for large values of n.

We believe that studies along the above lines might be fruitful.

As regards the planar embedding problem, when we set out to study this problem, our aim was to obtain an embedding in which all the edges are straight-line segments. Such an embedding is possible if the graph has no parallel edges or self-loops. Our choice of Lempel, Even, and Cederbaum's planarity testing algorithm to study this problem was motivated by two considerations. One was that no published work to obtain a planar embedding which uses this algorithm was available. The other was that this algorithm tests for planarity by building a planar embedding, and the way it is achieved appears more appropriate for constructing an embedding with straight-line segments. However, we have not been able to achieve our goal of obtaining a straight-line embedding. It seems that augmenting the graph by an additional vertex, as mentioned in Section 9.4, might help in getting more information about the relative locations of the vertices on the outside window of a block. An examination of the embedding procedure described in Chapter 8 will show that in our embedding almost all the edges, except those entering a vertex, say i, from lower numbered cut vertices in the corresponding bush form B_{i-1}, can be drawn as straight-line segments. Thus it seems that for this study, using the idea of augmentation may be helpful, since it might provide more information about the vertex order.

REFERENCES

[1] R.G. Busacker and T.L. Saaty, "Finite Graphs and Networks: An Introduction with Applications", (Book) McGraw-Hill, New York, 1965.

[2] N. Deo, "Graph Theory with Applications to Engineering and Computer Science", (Book) Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[3] M.N.S. Swamy and K. Thulasiraman, "Graphs, Networks, and Algorithms", (Book) Wiley-Interscience, New York, 1981.

[4] W.K. Chen, "Applied Graph Theory: Graphs and Electrical Networks", (Book) North-Holland Publishing Company, New York, 1971.

[5] S. Bedrosian, "Application of Linear Graphs to Multi-level Maser Analysis", Journal of the Franklin Institute, Vol. 274, No. 4, 278-283 (October 1962).

[6] R.A. Bari, "Chromatic Polynomials and the Internal and External Activities of Tutte", in Graph Theory and Related Topics, (Ed.) J.A. Bondy and U.S.R. Murty, Academic Press, New York, 1979, pp. 41-52.

[7] A. Satyanarayana and J.N. Hagstrom, "A New Algorithm for the Reliability Analysis of Multi-Terminal Networks", IEEE Trans. Reliability, Vol. R-30, No. 4, 325-334 (October 1981).

[8] S.M. Chase, "Analysis of Algorithms for Finding All Spanning Trees of a Graph", Report No. 401, Department

of Computer Science, University of Illinois, Urbana, October 1970.

[9] Y. Kajitani, "A Tree Listing Algorithm whose Computational Time is Asymptotically 0", IEEE Conference Record of the Fourteenth Asilomar Conference on Circuits, Systems and Computers, 51-54 (November 1980).

[10] G.J. Minty, "A Simple Algorithm for Listing All the Trees of a Graph", IEEE Trans. Circuit Theory, Vol. CT-12, No. 1, 120 (March 1965).

[11] R.C. Read and R.E. Tarjan, "Bounds on Backtrack Algorithms for Listing Cycles, Paths and Spanning Trees", Networks, Vol. 5, No. 3, 237-252 (July 1975).

[12] H.N. Gabow and E.W. Myers, "Finding All Spanning Trees of Directed and Undirected Graphs", SIAM Journal on Computing, Vol. 7, No. 3, 280-287 (August 1978).

[13] J.P. Char, "Generation of Trees, Two-Trees and Storage of Master Forests", IEEE Trans. Circuit Theory, Vol. CT-15, No. 3, 228-238 (September 1968).

[14] R. Jayakumar, "Analysis and Study of a Spanning Tree Enumeration Algorithm", M.S. Thesis, Department of Computer Science, Indian Institute of Technology, Madras, India, 1980.

[15] R. Jayakumar and K. Thulasiraman, "Analysis of a Spanning Tree Enumeration Algorithm", in Combinatorics and Graph Theory, Springer-Verlag Lecture Notes in Mathematics, (Ed.) S.B. Rao, No. 885, 1981, pp. 284-289.

[16] R. Jayakumar, K. Thulasiraman and M.N.S. Swamy, "Complexity of Computation of a Spanning Tree Enumeration Algorithm", IEEE Trans. Circuits and Systems, Vol. CAS-31, No. 10, 853-860 (October 1984).

[17] M.N.S. Swamy and K. Thulasiraman, "A Theorem in the Theory of Determinants and the Number of Spanning Trees of a Graph", Canadian Electrical Engineering Journal, Vol. 8, No. 4, 147-152 (October 1983).

[18] G. Tinhofer, "On the Generation of Random Graphs with Given Properties and Known Distribution", in "Graphs, Data Structures, Algorithms", Proceedings of the Workshop on Graph-theoretic Concepts in Computer Science, (Ed.) Manfred Nagl and H.J. Schneider, Carl Hanser Verlag, 1979, pp. 265-297.

[19] R.E. Tarjan, "Applications of Path Compression on Balanced Trees", J. ACM, Vol. 26, No. 4, 690-715 (October 1979).

[20] A.J.W. Hilton, "The Number of Spanning Trees of Labelled Wheels, Fans, and Baskets", in Combinatorics, published by Inst. Math. Appl., 1972, pp. 203-206.

[21] B.R. Myers, "Number of Trees in a Cascade of 2-Port Networks", IEEE Trans. Circuit Theory, Vol. CT-14, No. 3, 284-290 (September 1967).

[22] N.K. Bose, R. Feick and F.K. Sun, "General Solution to the Spanning Tree Enumeration Problem in Multigraph Wheels", IEEE Trans. Cicuit Theory, Vol. CT-20, No. 1, 69-71 (January 1973).

[23] J. Hopcroft and R. Tarjan, "A VlogV Algorithm for Isomorphism of Triconnected Planar Graphs", J. Comput. Syst. Sci., Vol. 7, No. 3, 323-331 (June 1973).

[24] J. Hopcroft and J.K. Wong, "Linear Time Algorithms for Isomorphism of Planar Graphs", Proc. Sixth Annual ACM Symposium on Theory of Computing, 172-184 (1974).

[25] F. Hadlock, "Finding a Maximum Cut in a Planar Graph in Polynomial Time", SIAM J. Comput., Vol. 4, No. 3, 221-225 (September 1975).

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

[27] C. Kuratowski, "Sur le probleme des corbes gauches en topologie", Fundamenta Mathematicae, Vol. 15, 271-283 (1930).

[28] P. Mei and N. Gibbs, "A Planarity Algorithm based on the Kuratowski Theorem", Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, New Jersey, pp. 91-95.

[29] H. Whitney, "Non-separable and Planar Graphs", Trans. Am. Math. Society, Vol. 33, 339-362 (1932).

[30] S. MacLane, "A Structural Characterization of Planar Combinatorial Graphs", Duke. Math J., Vol. 3, 460-472 (September 1937).

[31] L. Auslander and S.V. Parter, "On imbedding Graphs in the Plane", J. Math. and Mech., Vol. 10, 517-523 (May 1961).

[32] A.J. Goldstein, "An Efficient and Constructive

Algorithm for Testing whether a Graph can be Embedded in a Plane", Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Math., Princeton University, May 16-18, 1963.

[33] R.W. Shirey, "Implementation and Analysis of Efficient Graph Planarity Testing Algorithms", Ph.D. Thesis, University of Wisconsin, June 1969.

[34] J. Hopcroft and R. Tarjan, "Planarity Testing in VlogV Steps", Extended Abstract, Proc. IFIP Cong. 1971: Foundations of Information Processing, Ljubljana, Yugoslavia, August 1971, North-Holland Publishing Co., Amsterdam, pp. 18-22.

[35] J. Hopcroft and R. Tarjan, "Efficient Planarity Testing", J. Ass. Comput. Mach., Vol. 21, No. 4, 549-568 (October 1974).

[36] E.M. Reingold, J. Nievergelt and N. Deo, "Combinatorial Algorithms: Theory and Practice" (Book), Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[37] G. Demoucron, Y. Malgrange and R. Pertuiset, "Graphes planaires: Reconnaissance et construction de representations planaires topologiques", Rev. Francaise de Rech. Operationelle, Vol. 8, 33-47 (1964).

[38] F. Rubin, "An Improved Algorithm for Testing the Planarity of a Graph", IEEE Trans. on Computer, Vol. C-24, No. 2, 113-121 (February 1975).

[39] G.J. Fisher and O. Wing, "Computer Recognition and

Extraction of Planar Graphs from the Incidence Matrix", IEEE Trans. on Circuit Theory, Vol. CT-13, No. 2, 154-163 (June 1966).

[40] L. Mondshein, "Combinatorial Orderings and Embedding of Graphs", Tech. Note 1971-35, Lincoln Lab., M.I.T., August 1971.

[41] A. Lempel, S. Even and I. Cederbaum, "An Algorithm for Planarity Testing of Graphs", Theory of Graphs: International Symposium: Rome, July, 1966, P. Rosenstiehl (Ed.), Gordon and Breach, New York, 1967, pp. 215-232.

[42] K.S. Booth and G.S. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs and Graph Planarity Using PQ-tree Algorithms", J. of Comp. and Syst. Sciences, Vol. 13, No. 3, 335-379 (December 1976).

[43] J. Bruno, K. Steiglitz and L. Weinberg, "A New Planarity Test Based on 3-Connectivity", IEEE Trans. on Circuit Theory, Vol. CT-17, No. 2, 197-206 (May 1970).

[44] S. Even, "Graph Algorithms" (Book), Computer Science Press, Potomac, Maryland, 1979.

[45] S. Even and R.E. Tarjan, "Computing an st-numbering", Th. Comp. Sci., Vol. 2, 339-344 (1976).

[46] J. Ebert, "st-Ordering the Vertices of Biconnected Graphs", Computing, Vol. 30, 19-33 (1983).

[47] S. Fujishige, "An Efficient Algorithm for Solving the Graph-realization Problem by means of PQ-trees",

Proc. of 1979 Int. Symp. on Circuits and Systems, pp. 1012-1015.

[48] T. Ohtsuki and H. Mori, "On Minimal Augmentation of a Graph to Obtain an Interval Graph", J. Comput. and Sys. Sciences, Vol. 22, No. 1, 60-97 (1981).

[49] T. Ozawa and H. Takahashi, "A Graph-Planarization Algorithm and its Application to Random Graphs", in Graph Theory and Algorithms, Springer-Verlag Lecture Notes in Computer Science, Vol. 108, 1981, pp. 95-107.

[50] S. Masuda, T. Kashiwabara and T. Fujisawa, "A Layout Problem on Single Layer Printed Circuit Board", IECE of Japan, Tech. Rep. CAS 81-19, 1981, pp. 93-100.

[51] K. Nakajima and M. Sun, "On an Efficient Implementation of a Planarity Testing Algorithm for a Graph with Local Constraints", Proc. Twentieth Annual Allerton Conference on Communication, Control, and Computing, 1982, pp. 656-661.

[52] W.T. Tutte, "How to Draw a Graph", Proc. London Math. Soc., Vol. 13, No. 3, 743-768 (April 1963).

[53] L. Woo, "An Algorithm for Straight-line Representation of Simple Planar Graphs", Journal of the Franklin Institute, Vol. 287, No. 3, 197-208 (March 1969).

[54] R. Koppe, "Automatische Abbildung eines Planaren Graphen in einen Streckengraphen", Computing, Vol. 10, 317-333 (1972).

[55] O. Wing, "On Drawing a Planar Graph", IEEE Trans. Circuit Theory, Vol. CT-13, No. 1, 112-114 (March

1966).

[56] W. Mály, "An Algorithm for Obtaining the Planar Drawing of a Planar Graph", Proc. IEEE Int. Symp. Circuits and Systems, 1978, pp. 83-87.

[57] A.K. Hope, "A Planar Graph Drawing Program", Software - Practice and Expefience, Vol. 1, 83-91 (1971)..

[58] R. Tarjan, "An Efficient Planarity Algorithm", STAN-CS 244-71, Computer Science Department, Stanford University, November 1971.

[59] S.G. Williamson, "Embedding Graphs in the Plane - Algorithmic Aspects", in Combinatorial Mathematics, Optimal Designs and their Applications, Annals of Discrete Mathematics, (Ed.) J. Srivastava, No. 6, North-Holland Publishing Company, New York, 1980, pp. 349-384.

[60] W.M. Brehaut, "On Planar Graphs and the Plainer Nonplanar Graphs", Doctoral dissertation, University of Waterloo, September 1974.

[61] W.M. Brehaut, "Efficient Planar Embedding", Proc. 7-th South-Eastern Conference on Combinatorics, Graph Theory, and Computing, 1976, pp. 177-190.

[62] K. Pasedach, "Criterion and Algorithms for Determination of Bipartite Subgraphs and their Application to Planarization of Graphs", in Graphen-Sprachen und Algorithmen auf Graphen, Carl Hanser Verlag, 1976, pp. 175-183.

[63] M. Marek-Sadowska, "Planarization Algorithm for Integ-

rated Circuits Engineering", Proc. 1978 IEEE International Symposium on Circuits and Systems, pp. 919-923.

[64] T. Chiba, I. Nishioka, and I. Shirakawa, "An Algorithm of Maximal Planarization of Graphs", Proc. 1979 IEEE International Symposium on Circuits and Systems, pp. 649-652.