AN ALGORITHMIC APPROACH TO EXTREMAL GRAPH PROBLEMS

Xiaofeng Han
(Thesis)

CS-TR-322-91

June 1991

AN ALGORITHMIC APPROACH TO EXTREMAL GRAPH PROBLEMS

XIAOFENG HAN

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JUNE, 1991

DEDICATED TO MY GRANDFATHER DR. CHIYÜ WU

Abstract

The main purpose of this thesis is to study three problems concerning extremal graphs. The first is the problem of finding a *minimal 2-edge connected subgraph* of a *2-edge connected* graph, the second is the problem of finding a *minimal 2-connected subgraph* of a *2-connected graph*, and the third is the problem of finding a *maximal planar subgraph* of a *nonplanar* graph. These problems are extensions of the *2-edge connectivity* problem, the *2-connectivity* problem, and the *planarity testing* problem in graph theory. All three problems are important in the theory of extremal graphs. They also have useful applications in computer network and electronic circuit design.

This thesis is divided into five chapters and an appendix.

In the first chapter, we discuss the theory of extremal graphs and graph algorithm design in general. We explain why the problems solved in the thesis are important. We also include a brief discussion of some recent algorithms for problems in the theory of extremal graphs.

In the second chapter, we present an efficient algorithm for the problem of finding a minimal 2-edge connected subgraph of a 2-edge connected graph. Let $G = (V, E)$ be a 2-edge connected graph. The algorithm finds a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is still 2-edge connected. Graph $(V, E - E_0)$ is a minimal 2-edge connected subgraph of G. Let $\mid V \mid$ be n and $\mid E \mid$ be m. The algorithm runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-edge connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) We want

to find a maximal subset $E_0' \subseteq E'$ such that $(V', E' - E_0')$ is still 2-edge connected. Each recursion is divided into two parts. The first part is divided into twenty-five phases. At the beginning of the first phase, we let $E_0' = \phi$. In each phase, a subset of E' is processed and some edges in this subset are added to E_0'. In the second part, a new graph (V'', E'') is obtained from $(V', E' - E_0')$ such that (1) $\mid V'' \mid \leq 3/4 \mid V' \mid$, (2) (V'', E'') is 2-edge connected, (3) $\mid E'' \mid \leq \mid E' - E_0' \mid$, and (4) we can finish the computation of E_0' by computing a minimal 2-edge connected subgraph of (V'', E''). We then compute a minimal 2-edge connected subgraph of (V'', E'') by using the algorithm recursively. The algorithm stops when V'' contains just one vertex. Since the number of vertices in the graph is decreased by at least one fourth in each recursion, the level of recursions is $O(log(n))$. The whole algorithm takes $O(m+n)$ time and $O(m+n)$ space.

In the third chapter, we present an efficient algorithm for the problem of finding a minimal 2-connected subgraph of a 2-connected graph. Let $G = (V, E)$ be a 2-connected graph. The algorithm finds a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is still 2-connected. Graph $(V, E - E_0)$ is a minimal 2-connected subgraph of G. Let $\mid V \mid$ be n and $\mid E \mid$ be m. Like the algorithm in chapter two, the algorithm in chapter three runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) We want to find a maximal subset $E_0' \subseteq E'$ such that $(V', E' - E_0')$ is still 2-connected. Each recursion is again divided into two parts. The first part is further divided into twenty-five phases. At the beginning of the first phase, we let $E_0' = \phi$. In each phase, a subset of E' is processed and some edges in this subset are added to E_0'. In the second part, a new graph (V'', E'') is obtained from $(V', E' - E_0')$ such that (1) $\mid V'' \mid \leq 5/6 \mid V' \mid$, (2) (V'', E'') is 2-connected, (3) $\mid E'' \mid \leq \mid E' - E_0' \mid$, and (4) we can finish the computation of E_0' by computing a minimal 2-connected subgraph of (V'', E''). We then compute a minimal 2-connected subgraph of (V'', E'') by using the algorithm recursively. The algorithm stops when V'' contains just one vertex. The structure of the

algorithm in chapter three is exactly the same as that of the algorithm in chapter two. The difference between the two algorithms lies in (1) the procedures to process the edges in E' in each phase in the first part of each recursion, and (2) the procedures to obtain (V'', E'') from $(V', E' - E'_0)$ in the second part of each recursion. The algorithm in chapter three also takes $O(m+n)$ time and space.

In the fourth chapter, we discuss an algorithm for the problem of finding a maximal planar subgraph of a nonplanar graph. Let $G = (V, E)$ be a nonplanar graph with $\mid V \mid = n$ and $\mid E \mid = m$. The algorithm finds a minimal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is *planar*. Graph $(V, E - E_0)$ is a maximal planar subgraph of G. In designing the algorithm, we extend the classic planarity testing algorithm of Hopcroft and Tarjan. The algorithm runs in $O(m \cdot log(n))$ time and $O(m+n)$ space. The result in chapter four is the outcome of joint work with Jiazhen Cai and Robert Tarjan.

The fifth chapter discusses a parallel algorithm for transitively orienting comparability graphs. The algorithm is not directly related to problems in extremal graphs. It was discovered during the process of studying extremal graphs, however. The algorithm has many applications in designing parallel algorithms for such special perfect graphs as chordal graphs, permutation graphs and comparability graphs. It is well known that computing a maximum clique or a minimum coloring for an arbitrary graph is NP-complete. The problems are solvable in polynomial time if a graph is known to be perfect. It is an interesting question to ask whether there exist efficient NC algorithms to solve these problems for perfect graphs. Unfortunately, no such algorithms have been found. The algorithm in chapter five can be used to compute a maximum clique and a minimum coloring for a comparability graph.

The appendix contains the definitions used in this thesis and the references. The definitions are standard and can be found in books about graph theory. (See [AHU].)

When a term is first used, it appears in *italics*.

Acknowledgement

It would be impossible to mention everybody who helped me to complete this thesis. My thesis advisor Professor Robert Tarjan introduced me to the field of graph algorithm design. This thesis is influenced greatly by his previous work in this area.

I would like to thank my readers: Professor Andrew Yao and Professor Ken Steiglitz. They read my thesis very carefully and made many comments and suggestions.

I would also like to thank my fellow graduate students at Princeton. Mordecai Golin helped me with the format of this thesis. I had very helpful discussions with Robert Abbot and Richard Squier during the research that led to this thesis.

I also benefited from discussions with people from other universities. I discussed the algorithm in chapter three with Vijaya Ramachandran from University of Texas, Austin. The data structure in chapter four was first proposed by Jiazheng Cai from New York University.

I would like to thank the staff members in the Department of Computer Science of Princeton University for their support. Special thanks must be given to Sharon Rogers, Rebecca Davies, and Steven Beck.

Table of Contents

Abstract 1

Acknowledgments 5

Table of Contents 6

Chapter One. An Algorithmic Approach to Extremal Graph Problems 8

§1.1 Thesis Outline 9

§1.2 Introduction 13

Chapter Two. An Algorithm to Find Minimal 2-Edge Connected Subgraphs 17

§2.1 Introduction 18

§2.2 Outline of the Algorithm 19

§2.3 Finding E'_{sc} in Each Phase 22

§2.4 Processing the Edges in E'_{sc} 27

§2.5 Computing (V'', E'') from $(V', E'_e \cup E'_c)$ 35

§2.6 The Complexity of the Algorithm 41

§2.7 Problems for Future Research 44

Chapter Three. An Algorithm to Find Minimal 2-Connected Subgraphs 45

§3.1 Introduction 46

§3.2 Outline of the Algorithm 48

§3.3 Finding E'_{sc} in Each Phase 50

§3.4 Processing the Edges in E'_{sc} 53

§3.5 Computing (V'', E'') from $(V', E'_e \cup E'_c)$ 67

§3.6 The Complexity of the Algorithm 74

§3.7 Problems for Future Research 77

Chapter Four. An Algorithm to Find Maximal Planar Subgraphs 78

§4.1 Introduction 79
§4.2 Preliminary Results 81
§4.3 An Algorithm for Finding a Maximal Planar Subgraph 89
§4.4 Data Structures and the Time Complexity 97
§4.5 Problems for Future Research 102

Chapter Five. A Parallel Algorithm for Comparability Graphs 103

§5.1 Introduction 104
§5.2 A Transitive Orientation Algorithm 107
§5.3 Applications 115
§5.4 Problems for Future Research 118

Appendix. Graph Definitions and References 119

Graph Definitions 120
References 125

Chapter One

An Algorithmic Approach to Extremal Graph Problems

§1.1 Thesis Outline

The main purpose of this thesis is to study three problems concerning extremal graphs. The first is the problem of finding a minimal 2-edge connected subgraph of a 2-edge connected graph, the second is the problem of finding a minimal 2-connected subgraph of a 2-connected graph, and the third is the problem of finding a maximal planar subgraph of a nonplanar graph. These problems are extensions of the 2-edge connectivity problem, the 2-connectivity problem, and the planarity testing problem in graph theory. All three problems are important in the theory of extremal graphs. They also have useful applications in computer network and electronic circuit design.

This thesis is divided into five chapters and an appendix.

In the first chapter, we discuss the theory of extremal graphs and graph algorithm design in general. We explain why the problems solved in the thesis are important. We also include a brief discussion of some recent algorithms for problems in the theory of extremal graphs.

In the second chapter, we present an efficient algorithm for the problem of finding a minimal 2-edge connected subgraph of a 2-edge connected graph. Let $G = (V, E)$ be a 2-edge connected graph. The algorithm finds a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is still 2-edge connected. Graph $(V, E - E_0)$ is a minimal 2-edge connected subgraph of G. Let $\mid V \mid$ be n and $\mid E \mid$ be m. The algorithm runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-edge connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) We want to find a maximal subset $E'_0 \subseteq E'$ such that $(V', E' - E'_0)$ is still 2-edge connected. Each recursion is divided into two parts. The first part is divided into twenty-five phases. At the beginning of the first phase, we let $E'_0 = \phi$. In each phase, a subset of E' is

processed and some edges in this subset are added to E_0'. In the second part, a new graph (V'', E'') is obtained from $(V', E' - E_0')$ such that (1) $\mid V'' \mid \leq 3/4 \mid V' \mid$, (2) (V'', E'') is 2-edge connected, (3) $\mid E'' \mid \leq \mid E' - E_0' \mid$, and (4) we can finish the computation of E_0' by computing a minimal 2-edge connected subgraph of (V'', E''). We then compute a minimal 2-edge connected subgraph of (V'', E'') by using the algorithm recursively. The algorithm stops when V'' has just one vertex. Since the number of vertices in the graph is decreased by at least one fourth in each recursion, the level of recursions is $O(log(n))$. The whole algorithm takes $O(m+n)$ time and $O(m+n)$ space.

In the third chapter, we present an efficient algorithm for the problem of finding a minimal 2-connected subgraph of a 2-connected graph. Let $G = (V, E)$ be a 2-connected graph. The algorithm finds a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is still 2-connected. Graph $(V, E - E_0)$ is a minimal 2-connected subgraph of G. Let $\mid V \mid$ be n and $\mid E \mid$ be m. Like the algorithm in chapter two, the algorithm in chapter three runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) We want to find a maximal subset $E_0' \subseteq E'$ such that $(V', E' - E_0')$ is still 2-connected. Each recursion is again divided into two parts. The first part is further divided into twenty-five phases. At the beginning of the first phase, we let $E_0' = \phi$. In each phase, a subset of E' is processed and some edges in this subset are added to E_0'. In the second part, a new graph (V'', E'') is obtained from $(V', E' - E_0')$ such that (1) $\mid V'' \mid \leq 5/6 \mid V' \mid$, (2) (V'', E'') is 2-connected, (3) $\mid E'' \mid \leq \mid E' - E_0' \mid$, and (4) we can finish the computation of E_0' by computing a minimal 2-connected subgraph of (V'', E''). We then compute a minimal 2-connected subgraph of (V'', E'') by using the algorithm recursively. The algorithm stops when V'' contains just one vertex. The structure of the algorithm in chapter three is exactly the same as that of the algorithm in chapter two. The difference between the two algorithms lies in (1) The procedures to process the edges in E' in each phase in the first part of each recursion, and (2) The procedures to obtain

(V'', E'') from $(V', E' - E'_0)$ in the second part of each recursion. The algorithm in chapter three also takes $O(m + n)$ time and space.

Similar problems were considered previously. In [ET2], Eswaran and Tarjan gave linear-time algorithms for the problems of bridge connectivity augmentation and 2-connectivity augmentation. P. Kelsen and V. Ramachandran considered devising efficient parallel algorithms for finding a minimal 2-edge connected (or 2-connected) subgraph of a 2-edge connected (or 2-connected) graph. They gave parallel algorithms that run in $O(log^3(n))$ time using $O(m + n)$ processors. (See [KR1].) The algoritms can also sequentialized to run in $O(m + n \cdot log(n))$ time. More recently they have obtained linear time sequential algorithms independently of the work presented here. (See [KR3].) A problem known as the transitive compaction problem was considered by [GKRST] for directed graphs. In this problem, we are given a strongly connected digraph, we wish to find a minimal strongly connected spanning subgraph of it. [GKRST] proposed an NC algorithm for it. The algorithm also has an $O(m + n \cdot log(n))$ sequential implementation.

In the fourth chapter, we discuss an algorithm for the problem of finding a maximal planar subgraph of a nonplanar graph. Let $G = (V, E)$ be a nonplanar graph. The algorithm finds a minimal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is planar. $(V, E - E_0)$ is a maximal planar subgraph of G. In designing the algorithm, we extend the classic planarity testing algorithm of Hopcroft and Tarjan. (See [HT1].) The algorithm runs in $O(m \cdot log(n))$ time and $O(m + n)$ space. The result in chapter four is the outcome of joint work with Jiazhen Cai and Robert Tarjan.

The problem of finding maximal planar subgraphs was considered by many authors. In [Wu], Wu gave algorithms for both planarity testing and the problem of finding maximal planar subgraphs. He used an algebraic equation system to solve both problems. His solution runs in $O(m^2)$ time.

Jayakumar et al (See [JTS]) have also considered the problem. For the special case when a 2-connected spanning planar subgraph is given, their algorithm runs in $O(n^2)$ time and $O(m+n)$ space. For more general situations, their algorithm runs in $O(m*n)$ time.

In [BT], Battista and Tamassia proposed an algorithm for incremental planarity testing. Using their techniques, there is also a maximal planar subgraph algorithm taking $O(m+n \cdot log(n))$ time and $O(m+n)$ space. The approach in [BT] is based on the $P-Q$ tree planarity testing algorithm (see [LEC]) and the algorithm in chapter four is based on the path addition planarity testing algorithm. (See [HT1].)

The fifth chapter discusses a parallel algorithm for transitively orienting comparability graphs. The algorithm is not directly related to problems in extremal graphs. It was discovered during the process of studying extremal graphs, however. The algorithm has many applications in designing parallel algorithms for such special perfect graphs as chordal graphs, permutation graphs and comparability graphs. It is well known that computing a maximum clique or a minimum coloring for an arbitrary graph is NP-complete. The problems are solvable in polynomial time if a graph is known to be perfect. It is an interesting question to ask whether there exist efficient NC algorithms to solve these problems for perfect graphs. Unfortunately, no such algorithms have been found. The algorithm in chapter five can be used to compute a maximum clique and a minimum coloring for a comparability graph.

The appendix contains the definitions used in this thesis and the references. The definitions are standard and can be found in books about graph theory. (See [AHU].) When a term is first used, it appears in *italics*.

§1.2 Introduction

The study of graph theory can be traced back to Euler when he solved the famous problem of the Kőnigsberg bridges. Since then, graph theory has evolved into a branch of combinatorial mathematics with wide applications in engineering, social sciences and natural sciences. (See [CGM], [H], [Te], and [WB].)

After the invention of digital computers, people started to use computers to solve graph problems. This led to the design of graph algorithms. In the early days of graph algorithm design, many important problems were identified and studied. Algorithms were proposed for such problems as finding minimal spanning trees, finding shortest paths, finding network flows and testing planarity. (See [Kr], [Pr], [F], [Wa], [FF], [AP], and [LEC].) In the early days, complexity theory and algorithm analysis techniques were not widely used. Many algorithms were devised, but their efficiency was poorly understood. Many important ideas were formed but not fully explored.

At the beginning of the 70's, important progress was made in the area of algorithm design. Knuth's books had a major influence in persuading people to treat algorithms as mathematical objects and to do rigorous analysis on them. (See [Kn].) Cook discovered the first NP-complete problem. (See [Co].) Karp popularized the concept of NP-completeness by showing that many important combinatorial problems, including many graph problems, are NP-complete. (See [Ka].) Since then, the study of graph algorithms has diverged. For those graph problems that are NP-complete, people use approximation and randomization to study them. For polynomial graph problems, efforts were directed to finding efficient algorithms. In the early 70's, Hopcroft and Tarjan showed in a series of papers that very efficient graph algorithms can be devised by designing good data structures and using depth-first search on graphs. (See [HT1], [HT2], [HT3], [HT4], and [T1].) A notable example of their algorithms is the linear-time planarity testing algorithm. (See [HT1].)

The algorithms in this thesis extend the results of earlier research on polynomial time graph algorithms. They give efficient algorithmic solutions to some extremal graph problems.

The theory of extremal graphs is a branch of graph theory that deals with extremal properties of graphs. The study of the field was initiated by Turan in 1940 when he published the first paper on extremal graph problems. People are interested in the extremal (minimal or maximal) values of graph invariants that ensure that a graph has a certain property.

Over a period of 40 years, many problems of extremal graphs have been considered and many results have been obtained. Most of these results are purely graph-theoretical. Among the problems studied, problems associated with *minimal k-connected* and *minimal k-edge connected* graphs drew special attention from researchers. When $k = 2$, such problems are understood well. (See [D] and [Pl].)

These problems are not only interesting in theory, they also have practical applications. An example is the use of minimal k-edge connected graphs in the design of reliable computer networks.

Let us consider a simple model of a computer network. The network is modeled as a graph G. A computer is represented by a vertex and a communication channel between two computers is represented by an edge. For simplicity, we assume that computers can never fail; only communication channels can break down.

If G is *k-edge connected*, then for any two computers x and y in the network, there are at least k edge-disjoint paths connecting them. This means that if we have at most $k - 1$

breakdowns of communication links, we can still route information from any computer to any other computer in the network. Thus, the edge connectivity of the graph reflects the reliability of the network. If the graph is not *minimal k-edge connected*, we can delete some edges in the graph without sacrificing the edge connectivity (reliability) of the network. The property of minimal edge connectivity represents the most economical way of building a network that satisfies certain edge connectivity (reliability) requirements.

In this situation, we not only need to know the properties of minimal k-edge connected graphs, we also need an algorithm to decide whether a given network is minimal k-edge connected; if it is not, we would like to know how to modify it to make it so.

Another problem of extremal graphs that has interesting applications in electronic circuit design is the problem of finding maximal planar subgraphs of a nonplanar graph.

In designing a electronic circuit, we frequently require that the underlying graph of the circuit be planar. If the underlying graph is nonplanar, we are not able to make a layout of the circuit. In this case, we want to delete a minimal set of edges from the graph to make it planar. This requires finding a maximal planar subgraph of the underlying graph.

The main results of this thesis are expressed in the following theorems.

Main results:

Theorem 1.2.1 Let $G = (V, E)$ be a 2-edge connected graph. Assume that $\mid V \mid = n$ and $\mid E \mid = m$. There is an algorithm that computes a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is 2-edge connected. The algorithm runs in $O(m + n)$ time and space. ∎

Theorem 1.2.2 Let $G = (V, E)$ be a 2-connected graph. Assume that $\mid V \mid = n$ and $\mid E \mid = m$. There is an algorithm that computes a maximal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is 2-connected. The algorithm runs in $O(m + n)$ time and space. ∎

Theorem 1.2.3 Let $G = (V, E)$ be a nonplanar graph. Assume that $\mid V \mid = n$ and $\mid E \mid = m$. There is an algorithm that computes a minimal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is planar. The algorithm runs in $O(m \cdot log(n))$ time and $O(m + n)$ space. and linear space.∎

In what follows, we shall refer to the algorithms in Theorem 1.2.1, 1.2.2, and 1.2.3 as *algorithm 1.2.1, 1.2.2*, and *1.2.3* respectively.

Chapter Two

An Algorithm to Find Minimal 2-Edge Connected Subgraphs

§2.1 Introduction

Let $G = (V, E)$ be a 2-edge connected graph. In this chapter, we develop an $O(m+n)$ time algorithm for finding a minimal 2-edge connected subgraph of G.

In finding a minimal 2-edge connected subgraph of G, we prefer a subgraph with as few edges as possible. The problem of finding a minimal 2-edge connected subgraph of G that has the fewest edges is NP-complete, however, since the problem is polynomially equivalent to the following NP-complete problem.

Lemma 2.1.1 Let $G = (V, E)$ be a 2-edge connected graph and let k be an integer. The problem of deciding whether there exists an edge set $E' \subseteq E$ with $\mid E' \mid = k$ such that $H = (V, E - E')$ is 2-edge connected is NP-complete.

Proof: The problem is obviously in NP. To prove that it is NP-complete, we reduce the undirected Hamilton circuit problem to it. Let $G = (V, E)$ be a 2-edge connected graph. Let $k = m - n$. G contains a Hamilton circuit iff there exists an edge set $E' \subseteq E$ with $\mid E' \mid = k$ such that $(V, E - E')$ is 2-edge connected. ∎

Because of Lemma 2.1.1, we can hope to find only minimal 2-edge connected subgraphs of G. Let m_1 be the number of edges in a minimal 2-edge connected subgraph of G that has the fewest edges. It is true that $m_1 \geq n$. Let m_2 be the number of edges in a minimal 2-edge connected subgraph of G. We know that $m_2 \leq 2n$. We have $m_2/m_1 \leq 2$. It would be nice to have an algorithm such that for every $k > 0$, the algorithm finds a minimal 2-edge connected subgraph with m_2 edges and $m_2/m_1 \leq 1 + k$. Unfortunately, no such algorithm is known right now.

§2.2 Outline of the Algorithm

According to the introduction in chapter one, we wish to find a maximal subset $E_0 \subseteq E$ such that graph $(V, E - E_0)$ is still 2-edge connected. In the presentation of the algorithm, we talk about deleting edges from E. When an edge is deleted, it is added to E_0 and remains there until the end of the algorithm.

In order to find a minimal 2-edge connected subgraph of G, we can use a simple iterative algorithm. For every edge (x, y) in E, we test whether graph $(V, E - \{(x, y)\})$ is 2-edge connected. We keep (x, y) in E if graph $(V, E - \{(x, y)\})$ is not 2-edge connected and delete it from E otherwise. The edges can be processed in any order. At the end of the algorithm, let $E_0 = \{$ the deleted edges $\}$; graph $(V, E - E_0)$ is a minimal 2-edge connected subgraph of G. Since we have m edges to process and processing each edge takes $O(m+n)$ time, this algorithm takes $O(m^2)$ time. Although this is not a very efficient algorithm, it is the starting point for the algorithm in this chapter.

In the simple algorithm, we consider only one edge at an iteration step and we test the 2-edge connectivity of a graph that is only slightly different from the one previously tested. The time complexity of the algorithm can be greatly improved if we can process a group of edges at a time.

The algorithm in this chapter runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-edge connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) In each recursion, we want to find a maximal subset $E'_0 \subseteq E'$ such that $(V', E' - E'_0)$ is still 2-edge connected. Each recursion is divided into two parts. The first part is divided into twenty-five phases. At the beginning of each phase, we have $E' = E'_e \cup E'_c \cup E'_0$ and $E'_e \cap E'_c = E'_c \cap E'_0 =$

$E'_e \cap E'_0 = \phi$. The edges in E'_0 are deleted from E' and do not affect any future processing of (V', E').

The inductive hypothesis at the beginning of each phase is that each edge in E'_e is 2-*edge essential* (see Definition 24 in the appendix.) for $(V', E'_e \cup E'_c)$.

The edges in E'_c are unprocessed edges and are called *candidate* edges. In each phase, a subset $E'_{sc} \subseteq E'_c$ is computed such that $\mid E'_{sc} \mid \geq 1/6 \mid E'_c \mid$. E'_{sc} is then split into two sets E'_{0sc} and E'_{esc}. The edges in E'_{0sc} are added to E'_0 and the edges in E'_{esc} are added to E'_e. The inductive hypothesis is maintained and we go to the next phase. Let $\mid V' \mid = n'$ and $\mid E' \mid = m'$. The first phase takes $O(m' + n')$ time. After the first phase, we have $\mid E'_e \cup E'_c \mid \leq 2n'$. Each of the remaining twenty-four phases takes $O(n')$ time. After the first part of each recursion, we have $\mid E'_c \mid \leq (5/6)^{24} 2n' \leq n'/18$.

In the second part of each recursion, we use the fact that $E'_c \leq n'/18$ to compute a new graph (V'', E'') such that (1) $\mid V'' \mid \leq 3/4 \mid V' \mid$, (2) (V'', E'') is 2-edge connected, (3) $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$, and (4) if we know a minimal 2-edge connected subgraph of (V'', E''), we can easily compute a subset $E'_{ec} \subseteq E'_c$ such $(V', E'_e \cup E'_{ec})$ is a minimal 2-edge connected subgraph of (V', E'). The algorithm then computes a minimal 2-edge connected subgraph of (V'', E'') by using itself recursively.

The following pseudo-code outlines a single recursive call of algorithm 1.2.1.

Procedure 2.2.1

A single recursive call of algorithm 1.2.1

{

Assume we want to compute a minimal 2-edge
connected subgraph of (V', E')
Let $E'_e = \phi, E'_c = E', E'_0 = \phi$;
* first part of the recursive call *\
For $(i = 0; i < 25; i++)$
{
Compute $E'_{sc} \subseteq E'_c$;
Split E'_{sc} into $E'_{0sc} \cup E'_{esc}$;
$E'_e = E'_e \cup E'_{esc}$, $E'_0 = E'_0 \cup E'_{0sc}$;
$E'_c = E'_c - E'_{sc}$, $E'_{sc} = \phi$;
}
* second part of the recursive call *\
Compute (V'', E'') from $(V', E'_e \cup E'_c)$;
Compute a minimal 2-edge connected subgraph of (V'', E'');
Use the result of previous step to compute a minimal 2-edge connected
subgraph of (V', E');
}

The following discussion of the algorithm is divided into five sections. In §2.3, we discuss how to compute E'_{sc} in each phase. In §2.4, we discuss how to split E'_{sc} into E'_{0sc} and E'_{esc}. §2.5 is concerned with the computation of (V'', E'') from $(V', E'_e \cup E'_c)$ and the computation of a minimal 2-edge connected subgraph of (V', E') using the result obtained from the recursive application of the algorithm to (V'', E''). In §2.6, we analyze the complexity of the algorithm. §2.7 discusses some future research problems.

§2.3 Finding E'_{sc} in Each Phase

This section is concerned with finding E'_{sc} at the beginning of each phase. We start with the graph $(V', E'_e \cup E'_c)$ at the beginning of each phase. A *spanning tree* (V', E'_t) of $(V', E'_e \cup E'_c)$ is computed such that E'_t contains as many edges from E'_e as possible. Let $\mid E'_c \mid = K$ and $\mid E'_c \cap E'_t \mid = k$. The edges in $E'_c \cap E'_t$ $(E'_c - (E'_c \cap E'_t))$ are called *tree (nontree) candidate* edges.

Let the connected components of (V', E'_e) be $(V'_1, E'_1), (V'_2, E'_2), ..., (V'_p, E'_p)$. We have $\cup_i V'_i = V'$, $\cup_i E'_i = E_e$, and $V'_i \cap V'_j = \phi$ for $1 \leq i \neq j \leq p$.

We contract the connected components of (V', E'_e) to build an auxiliary graph $G'_a = (V'_a, E'_a)$. We have $V'_a = \{V'_i \mid 1 \leq i \leq p\}$. For every edge (x, y) in E'_c with $x \in V'_i$, $y \in V'_j$, we have an edge $(V'_i, V'_j)_{(x,y)} \in E'_a$. We shall use $E'_a(x, y)$ to denote this edge, i.e., $E'_a(x, y) = (V'_i, V'_j)_{(x,y)}$ and $E'^{-1}_a(V'_i, V'_j)_{(x,y)} = (x, y)$. G'_a may contain multiple edges and loops.

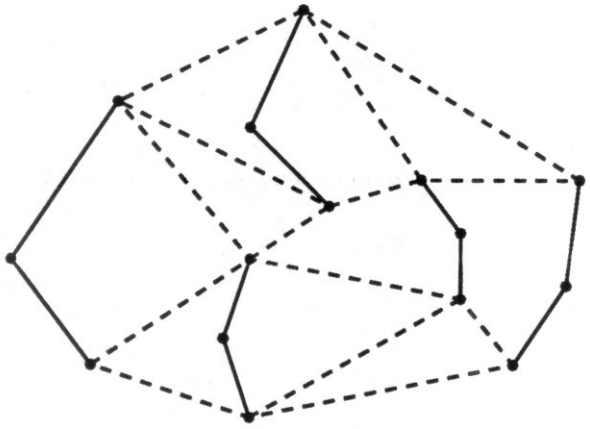

Figure 2.3.1

In Figure 2.3.1, the edges in E'_e are solid and the edges in E'_c are dashed. A spanning

tree (V', E'_t) and the auxiliary graph are drawn in Figure 2.3.2 and 2.3.3.

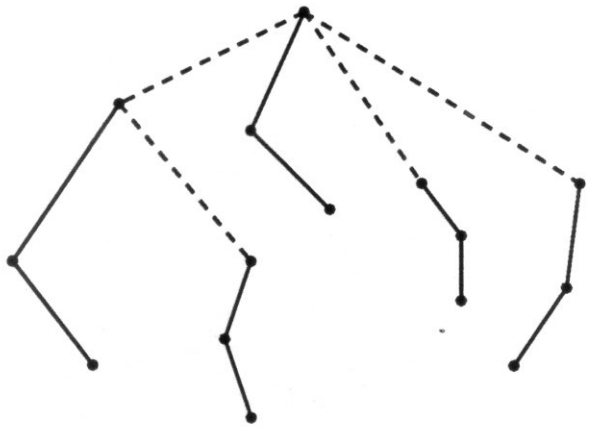

Figure 2.3.2

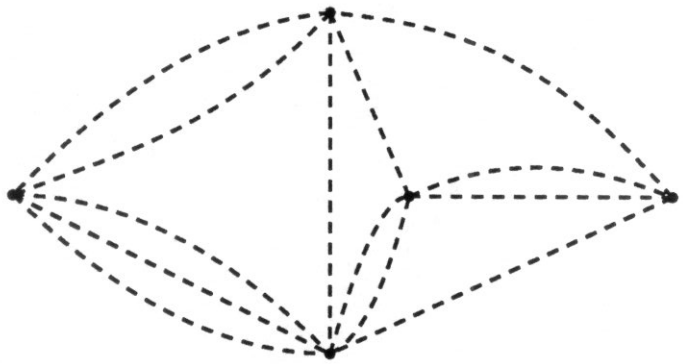

Figure 2.3.3

It is not hard to see that $(V'_a, \{E'_a(x,y) \mid (x,y) \in E'_c \cap E'_t\})$ is a spanning tree for G'_a, The number of vertices in G'_a is $k+1$.

G'_a also satisfies following properties:

Lemma 2.3.1 G'_a is 2-edge connected.

Proof: This is obvious since $(V', E'_e \cup E'_c)$ is 2-edge connected. ∎

Lemma 2.3.2 Let edge $(x, y) \in E'_c$. If $E'_a(x, y)$ is *2-edge essential* in G'_a, edge (x, y) is 2-edge essential in $(V', E'_e \cup E'_c)$.

Proof: Let (x', y') be a *bridge* in $(V'_a, E'_a - E'_a(x, y))$. Then $E_a^{-1}(x', y')$ is a bridge in $(V', (E'_e \cup E'_c) - (x, y))$. Thus edge (x, y) is 2-edge essential for $(V', E'_e \cup E'_c)$. ∎

E'_{sc} is determined according to the following two cases:

Case 2.3.1 $k \leq 5K/6$.

In this case, we let $E'_{sc} = E'_c - E'_t$. Obviously, $| E'_{sc} | = | E'_c - (E'_c \cap E'_t) | = K - k \geq K/6$. A procedure to process the edges in E''_{sc} is given in §2.4.

Case 2.3.2. $k > 5K/6$

In this case, most of the candidate edges are tree candidate edges and we shall prove that there are at least $K/6$ edges in G'_a that are 2-edge essential for G'_a.

Lemma 2.3.3 If case 2.3.2 applies, then there are at least $K/6$ vertices in G'_a with degree equal to 2.

Proof: Since G'_a is 2-edge connected, every vertex in G'_a has degree at least 2. Let the number of degree 2 vertices in G'_a be q. Assume that $q < K/6$. We have $\sum_{v_i \in V'} deg(v_i) \geq 3(k + 1 - q) + 2q = 3(k + 1) - q > 3(k + 1) - K/6 > 2K$. But the sum of degrees equals $2K$ since the number of edges in G_a is K. This is a contradiction. ∎

Let (x', y') be an edge in G'_a. If the degree of x' or y' equals 2, edge (x', y') is obviously a 2-edge essential edge in G'_a. We call (x', y') a *critical edge* of G'_a.

Lemma 2.3.4 If case 2.3.2 applies, then there are at least $K/6$ critical edges in G'_a.

Proof: If case 2.3.2 applies, we know from Lemma 2.3.3 that there are at least $K/6$ vertices in G'_a with degree equal to 2. Each such vertex provides two critical edges and each critical edge is counted at most twice, so the lemma is true. ∎

Let $E'_{sc} = \{(x, y) \mid E'_a(x, y)$ is a critical edge in $G'_a\}$. From Lemma 2.3.2 and 2.3.4, we know that every edge in E'_{sc} is 2-essential for $(V', E'_e \cup E'_c)$ and $\mid E'_{sc} \mid \geq K/6$.

The following pseudo code outlines the procedure to compute E'_{sc} at the beginning of each phase.

Procedure 2.3.1

Input: $(V', E'_e \cup E'_c)$

Output: E'_{sc}

{

Compute a spanning tree (V', E'_t) for $(V', E'_e \cup E'_c)$

such that E'_t contains as many edges of E'_e as possible;

Let $\mid E'_c \mid = K$ and $\mid E'_c \cap E'_t \mid = k$;

Construct $G'_a = (V'_a, E'_a)$;

If $(k \leq 5K/6)$;

Let $E'_{sc} = E'_c - (E'_c \cup E'_t)$;

Else $E'_{sc} = \{(x, y) \mid E'_a(x, y)$ is a critical edge of $G'_a\}$;

```
}
```

§2.4 Processing the Edges in E'_{sc}

From the discussion of §2.3, we know that when $k > 5K/6$, every edge in E'_{sc} is 2-edge essential for $(V', E'_e \cup E'_c)$. In this case, no further processing is needed for the edges in E'_{sc}. All the edges in E'_{sc} are added to E'_e. We go to the next phase in the recursive call.

If $k \leq 5K/6$, then we have $E'_{sc} = E'_c - E'_t$. Using the procedure discussed in this section, some edges in E'_{sc} are added to E'_e and others are added to E'_0. We shall prove that the inductive hypothesis is maintained at the end of the phase.

We again consider graph $(V', E'_e \cup E'_c)$. Let (V', E'_t) be the spanning tree described in §2.3. For the sake of simplicity, the procedure actually processes the edges in $(E'_c \cup E'_e) - E'_t$ (the nontree edges in $(V', E'_e \cup E'_c)$). If we pick a vertex in V, we can convert (V', E'_t) into a *rooted spanning tree*. The procedure works bottom up on the tree. It splits the set $(E'_e \cup E'_c) - E'_t$ into two sets E'_{esc} and E'_{0sc} such that the edges in E'_{esc} are 2-edge essential for $(V', (E'_e \cup E'_{esc}) \cup (E'_c - E'_{sc}))$. The edges in $E'_e - E'_t$ will be added to E'_{esc} automatically in the process.

After (V', E'_t) is turned into a rooted spanning tree, we number the vertices in V' by a *postorder numbering* of (V', E'_t). In the following discussion, the vertices will be referred to by their numbering. We also mark the leaves of (V', E'_t). For each nontree edge (x, y), let $nca(x, y)$ be the *nearest common ancestor* of x and y in (V', E'_t). That is, nca =min $\{v \mid v$ is an ancestor of x and $y\}$. For each vertex $v \in V$, we define:

Definition 2.4.1 $bc(v) = \{(x, y) \mid (x, y)$ is a nontree edge, x or y is a descendant of $v\}$

Definition 2.4.2 $high(v) = \max \{z \mid z = nca(x, y), (x, y) \in bc(v)\}$

Definition 2.4.3 $led(v) = \{(v,x) \mid (v,x) \in bc(v) \text{ and } nca(v,x) = high(v)\}$

In Figure 2.4.1, we have $high(7) = 11$, $high(3) = 7$, $high(2) = 3$. We also have $bc(7) = \{(1,6),(1,2),(4,9),(5,8)\}$, and $bc(2) = \{(1,2)\}$. $led(1) = \{(1,6)\}$, $led(4) = \{(4,9)\}$.

At the beginning of the procedure, we compute *high* for every vertex in V' and *nca* for every nontree edge in $E'_e \cup E'_c$. For each vertex v, we also compute $led(v)$. All these can be computed in linear time. (See [HT5] for the computation of nearest common ancestors of the nontree edges.) The *high* and *led* values remain the same through out the procedure.

For each edge (x,y) in $(E'_e \cup E'_c) - E'_t$, if $nca(x,y) <\min(high(x), high(y))$, we remove it from $E'_e \cup E'_c$ and put it in E'_0. After this operation, we have $nca(x,y) = \min(high(x), high(y))$ for every $(x,y) \in (E'_e \cup E'_c) - E'_t$.

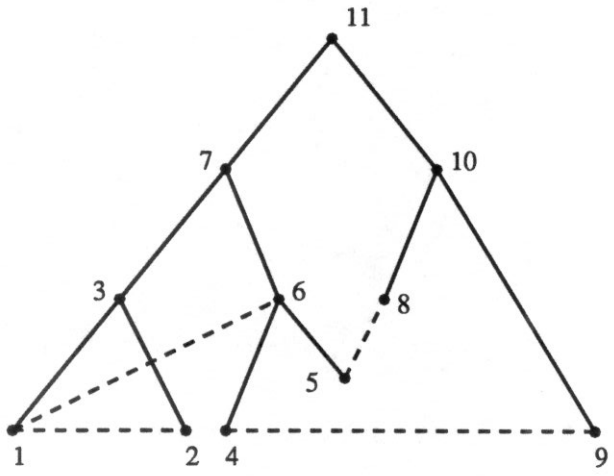

Figure 2.4.1

After computing the *high* values, we partition V' into $V_{h_1}, ..., V_{h_l}$ such that (1) $high(v) = high(w)$ for $v \in V_{h_i}$ and $w \in V_{h_i}$ with $1 \leq i \leq l$, and (2) $high(v) < high(w)$ for $v \in V_{h_i}$ and $w \in V_{h_j}$ with $1 \leq i < j \leq l$. We call $V_{h_1}, ..., V_{h_l}$ the *partition classes* of V'. Let $high(V_{h_i}) = high(v)$ where $v \in V_{h_i}$. This partition can be computed by doing a

bucket sort on V' using the *high* values as the sort key. At the beginning of the procedure, we mark every partition class of V' as unprocessed. In the procedure, the partition classes are processed according to the order of ascending *high* values. After a partition class is processed, it is marked so and the vertices in the partition class are removed from $(V', E'_e \cup E'_c)$.

We also partition $(E'_e \cup E'_c) - E'_t$ into $E_{h_1}, ..., E_{h_l}$ such that (1) $nca(v,w) = nca(x,y)$ for $(v,w) \in E_{h_i}$ and $(x,y) \in E_{h_i}$ with $1 \leq i \leq l$, and (2) $nca(v,w) < nca(x,y)$ for $(v,w) \in E_{h_i}$ and $(x,y) \in E_{h_j}$ with $1 \leq i < j \leq l$. We call $E_{h_1}, ..., E_{h_l}$ the *partition classes* of the nontree edges. Let $nca(E_{h_i}) = nca(x,y)$ where $(x,y) \in E_{h_i}$. This partition can be computed by doing a bucket sort on $(E'_e \cup E'_c) - E'_t$ using the *nca* values as the sort key. It is important to notice that there is a $1-1$ correspondence ($P : V_{h_i} \rightarrow E_{h_i}$ where $high(V_{h_i}) = nca(E_{h_i})$) between the partition classes of V' and the partition classes of $(E'_e \cup E'_c) - E'_t$.

For each vertex $v \in V'$, we associate a field *isleaf* with v. We have $v.isleaf = 1$ if v is a leaf in (V', E'_t) and $v.isleaf = 0$ otherwise. In the procedure, a nonleaf vertex can turn into a leaf vertex, but not vice versa.

The procedure is divided into steps and works bottom up on (V', E'_t). At the beginning of the procedure, we let $E'_{0sc} = \phi$ and $E'_{esc} = \phi$. In order to make the procedure efficient, we wish to design it in such a way that we can consider a group of nontree edges at a time, and decide which edges can be added to E'_{0sc} while others have to be added to E'_{esc}, by studying the local structure of these edges.

At the beginning of each step, let $slea = \{v \mid v$ is a leaf, $v \in V_{h_i}$, and V_{h_i} is the unprocessed partition class of V' with the smallest *high* value $\}$. We call V_{h_i} the *base partition class* of *slea*. We have $\mid slea \mid \leq \mid V_{h_i} \mid$.

Let $v_0 = high(V_{h_i})$. We assume that $v_1, ..., v_p$ are the *children* of v_0 with $high(v_i) = v_0$ and $u_1, ..., u_q$ are the children of v_0 with $high(u_j) > v_0$. Let $T_{v_i} = (V_{v_i}, E_{v_i})$ and $T_{u_j} = (V_{u_j}, E_{u_j})$ be the subtrees of (V', E'_t) that are rooted at v_i and u_j respectively. Let $S = \{(x,y) \mid nca(x,y) = v_0 \text{ and } (x,y) \in (E'_e \cup E'_c) - E'_t\}$. Notice that S is actually a partition class of $(E'_e \cup E'_c) - E'_t$ with $nca(S) = v_0$. (See Figure 2.4.2.)

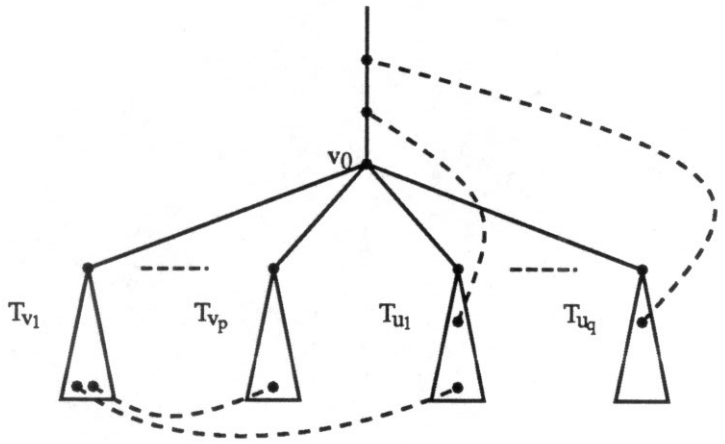

Figure 2.4.2

In each step, a subset CS of S is chosen. If we add the edges in $S - CS$ to E'_{0sc}, $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-edge connected and every edge in CS is 2-edge essential for the graph. Vertex v_0 is chosen in such a way that CS can be computed locally by using the subgraphs of $(V', E'_e \cup E'_c)$ spanned by T_{v_i} and T_{u_j}. The processing of nontree edges in $(V', E'_e \cup E'_c)$ with nearest common ancestor greater than v_0 will not affect the choice of the edges in CS. (See Lemma 2.4.1 for details.)

After CS is chosen, the vertices in V_{h_i} are removed from $(V', E'_e \cup E'_c)$. We also remove any tree edge or nontree edge that is incident on a removed vertex. Let the set of removed nontree edges be S'. We prove that $S' = S$. We use induction on the steps of the procedure. Assume that every vertex v with $high(v) < v_0$ and every nontree edge (x,y) with $nca(x,y) < v_0$ have been removed from $(V', E'_e \cup E'_c)$ at the beginning of the step. Let

$(x,y) \in S'$ and $x \in V_{h_i}$. We have $high(x) = v_0$ and $nca(x,y) = min(high(x), high(y)) = v_0$ since we know that $high(y) \geq v_0$. This means $(x,y) \in S$. Let $(x,y) \in S$. If both $high(x) > v_0$ and $high(y) > v_0$ then we have $nca(x,y) > v_0$. This is a contradiction. Assume that $high(x) = v_0$. We have $v \in V_{h_i}$ and $(x,y) \in S'$. (See also the discussion after substep 2.) After the removal of vertices and edges, we go to the next step of the procedure. When $(V', E'_e \cup E'_c)$ is empty, the procedure terminates. Since the removal of vertices and edges only affect the procedure in this section, the procedure actually works on a copy of $(V', E'_e \cup E'_c \cup E'_0)$ and saves the original $(V', E'_e \cup E'_c \cup E'_0)$ for future processing.

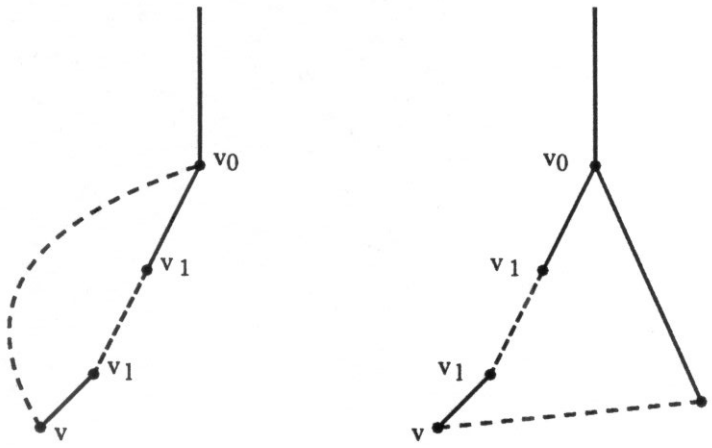

Figure 2.4.3

The process of computing CS involves two substeps. The goal of the substeps is to choose a minimal subset CS of S such that $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-edge connected. Let us consider a vertex v in *slea*; there is a unique path in (V', E'_t) from v_0 to $v : v_0, v_1, ..., v_l, v$. It is obviously true that we must pick at least one edge from $bc(v)$ and put it in CS, since (v_l, v) would be a bridge in $(V', (E'_e \cup E'_c) - (S - CS))$ otherwise. Since we have $high(v) = v_0$, if we put an edge from $led(v)$ in CS, we are sure that none of $(v_0, v_1), ..., (v_l, v)$ will be a bridge in $(V', (E'_e \cup E'_c) - (S - CS))$, see Figure 2.4.3.

Substep 1 For each vertex v in *slea*, we choose an edge from $led(v)$ and put it in CS.

Although every vertex v in *slea* is now covered by at least one edge in CS, CS may not be a minimal subset of S that covers every vertex of *slea*.

Substep 2 We build an auxiliary graph (V_a, CS). $V_a = \{v \mid v \in V', \exists w, (v,w) \in CS\}$. The vertices of V_a that are in *slea* are marked red. By using a minimal edge covering algorithm, (see below) we find a minimal subset of CS that covers all the red vertices of V_a.

We process the edges in CS one by one. When we are processing an edge $(x,y) \in CS$, we delete (x,y) from CS unless x is red and (x,y) is the only edge in CS incident on x or y is red and (x,y) is the only edge in CS incident on y. After the processing, the remaining edges in CS is the required minimal set.

Both substep 1 and 2 can be performed in $O(\mid slea \mid)$ time.

After the computation of substep 2, the edges in CS form a minimal subset of S that covers every vertex in *slea*. We then remove the vertices of V_{h_i} from $(V', E'_e \cup E'_c)$. We also remove every tree edge and nontree edge incident on a removed vertex. If a nonleaf vertex v becomes a leaf after the removal of the vertices in V_{h_i}, we let $v.isleaf = 1$. We already know that the set of removed nontree edges is S. Let the number of removed tree edges be R_t. The removal of vertices and edges can be finished in $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$ time. After the removal of vertices and edges, we go to the next step. The time complexity of each step is $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$. After each step, the size of V' decreases by $\mid V_{h_i} \mid$ and the size of $E'_e \cup E'_c$ decreases by $\mid S \mid + R_t$. The total complexity of the procedure is $O(\mid V \mid + \mid E'_e \mid + \mid E'_c \mid)$.

Let $E'_{esc} = E'_{esc} \cup CS$ and $E'_{0sc} = E'_{0sc} \cup (S - CS)$.

We have the following lemma about CS.

Lemma 2.4.1 Graph $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-edge connected and the edges in CS are all 2-edge essential for the graph.

Proof: Graph $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-edge connected because of the way CS is chosen. If we delete one edge from CS, one of the vertices in *slea* becomes uncovered by the edges in CS. The unique tree edge that enters the vertex is a bridge in $(V', (E'_e \cup E'_c) - (S - CS))$. The edges processed at later steps of the procedure all have *high* values greater than v_0. After the procedure, $(V', E'_e \cup E'_c)$ is not 2-edge connected. ∎

Lemma 2.4.1 shows that after the procedure, the inductive hypothesis is maintained.

The following pseudo code outlines the procedure in this section.

Procedure 2.4.1

{

Let (V', E'_t) be the spanning tree in §2.3;

For each vertex $v \in V'$, compute $high(v)$;

For each non tree edge (v, w), compute $nca(v, w)$;

For each vertex $v \in V'$, compute $led(v)$;

For each $(x, y) \in (E'_e \cup E'_c) - E'_t$ with $nca(x, y) <$min$(high(x), high(y))$

  Remove (x, y) from $E'_e \cup E'_c$ and put it in E'_0;

Partition V' into partition classes: $V_{h_1}, ..., V_{h_l}$;

Partition $(E'_e \cup E'_c) - E'_t$ into partition classes: $E_{h_1}, ..., E_{h_l}$;

Mark each $V_{h_i} (1 \leq i \leq l)$ as unprocessed;

While $(V', E'_e \cup E'_c)$ is not empty

{

Let $slea = \{v \mid v$ is a leaf, $v \in V_{h_i}$, and V_{h_i} is

the unprocessed partition class of V' with the smallest $high$ value $\}$;

Let $v_0 = high(V_{h_i})$;

Let $v_1, ..., v_p$ be the children of v_0 with $high(v_i) = v_0$;

Let $u_1, ..., u_q$ be the children of v_0 with $high(u_j) < v_0$;

Let T_{v_i} and T_{u_j} be the subtree of (V', E'_t)

rooted at v_i and u_j;

Compute CS using the two substeps described above;

Remove the vertices of V_{h_i} and the edges incident

on a vertex in V_{h_i} from $(V', E'_e \cup E'_c)$;

Mark V_{h_i} as processed;

}

}

§2.5 Computing (V'', E'') from $(V', E'_e \cup E'_c)$

After the first part of the algorithm, we obtain a graph $(V', E'_e \cup E'_c)$ such that (1) $(V', E'_e \cup E'_c)$ is 2-edge connected, (2) Every edge in E'_e is 2-edge essential for the graph, and (3) $\mid E'_c \mid \leq n'/18$.

In this section, we discuss a procedure to compute a new graph (V'', E'') such that (1) (V'', E'') is 2-edge connected, (2) $\mid V'' \mid \leq 3n'/4$, (3) $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$, and (4) if we know a minimal 2-edge connected subgraph of (V'', E''), we can process the edge in E'_c in $O(n')$ time and compute a minimal 2-edge connected subgraph of (V', E').

Graph (V'', E'') is built in such a way that it preserves the 2-edge connectivity structure of $(V', E'_e \cup E'_c)$.

In the following discussion, we assume that if $(x, y) \in E'_c$ then $deg(x) > 2$ and $deg(y) > 2$. If $deg(x) = 2$ or $deg(y) = 2$, (x, y) is always 2-edge essential for $(V', E'_e \cup E'_c)$.

There are two cases to consider:

Case 2.5.1 $\mid E'_e \cup E'_c \mid \leq 4n'/3$.

Lemma 2.5.1 If case 2.5.1 applies, there are at least $n'/4$ vertices in $(V', E'_e \cup E'_c)$ with degree equal to 2.

Proof: Since $(V', E'_e \cup E'_c)$ is 2-edge connected, every vertex in the graph has degree at least 2. Let q be the number of degree 2 vertices in the graph. Assume that $q < n'/4$. We

have $\sum_{v \in V'} deg(v) \geq 3(n'-q) + 2q = 3n' - q \geq 11n'/4 > 8n'/3 = 2 \mid E'_e \cup E'_c \mid$. This is a contradiction. ∎

A path $v_1, ..., v_p$ in $(V', E'_e \cup E'_c)$ is called a *chain* if we have $deg(v_1) > 2$, $deg(v_p) > 2$ and $deg(v_i) = 2$ for $1 < i < p$. Every degree two vertex in $(V', E'_e \cup E'_c)$ is on a unique chain in the graph. v_1 and v_p are called the end vertices of the chain. Notice that a chain may be a cycle with $v_1 = v_p$.

If case 2.5.1 applies, we know from Lemma 2.5.1 that there are at least $n'/4$ vertices in $(V', E'_e \cup E'_c)$ with degree 2. We contract these vertices to build (V'', E''). We construct (V'', E'') in the following way:

(1) Let $V'' = V' - \{v \mid deg(v) = 2\}$.

(2) For each edge $(x, y) \in E'_e \cup E'_c$ with $deg(x) > 2$ and $deg(y) > 2$, add an edge (x, y) to E''. (x, y) is called a *regular* edge in E''.

(3) For each chain $v_1, ..., v_p$, we added a new edge (v_1, v_p) to E''. (v_1, v_p) is called a *special* edge in E''.

(V'', E'') may contain multiple edges. Notice that for each regular edge (x', y') in E'', there is a corresponding edge in $E'_e \cup E'_c$. We refer to this corresponding edge as $E''^{-1}(x', y')$. For each special edge (x', y') in E'', there is a corresponding chain $v_1, ..., v_p$ in $(V', E'_e \cup E'_c)$. We also refer to this chain as $E''^{-1}(x', y')$.

It is not hard to see that (V'', E'') is 2-edge connected and preserves the 2-edge connectivity structure of $(V', E'_e \cup E'_c)$. (See also Lemma 2.5.2 and 2.5.3.)

If $(V', E'_e \cup E'_c)$ is the graph in Figure 2.3.1, (V'', E'') is shown in Figure 2.5.1. Notice that the graph in Figure 2.3.1 does not satisfy case 2.5.1, Figure 2.5.1 is included to illustrate how to obtain (V'', E'').

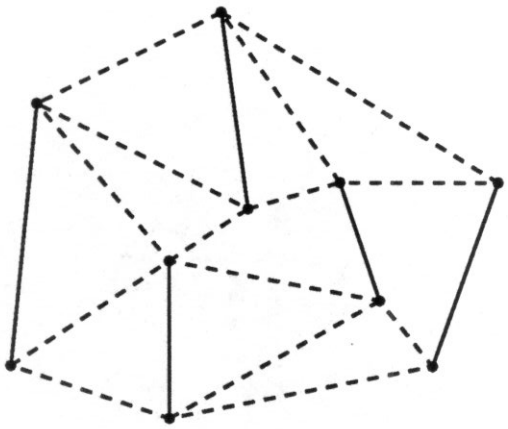

Figure 2.5.1

The construction of (V'', E'') can be done in $O(n')$ time.

Since there are at least $n'/4$ degree 2 vertices in $(V', E'_e \cup E'_c)$, we know that $\mid V'' \mid \leq 3n'/4$. It is also true that $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$.

Lemma 2.5.2 Assume case 2.5.1 applies. Let (V'', E''_e) be a subgraph of (V'', E'') that is 2-edge connected. Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge $\}))$ is also 2-edge connected.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge $\}$. Assume that $(V', E'_e \cup E''_c)$ is not 2-edge connected. Let (v,w) be a bridge in $(V', E'_e \cup E''_c)$. If (v,w) is in a chain $v_1, ..., v_p$, then either (v_1, v_p) is in E''_e and (v_1, v_p) is a bridge in (V'', E''_e) or (V'', E''_e) is not connected. In either case, (V'', E''_e) is not 2-edge connected. If $deg(v) > 2$ and $deg(w) > 2$, (v,w) is a regular edge in E''. If $(v,w) \in E''_e$ then (v,w) is a bridge

in (V'', E''_e); otherwise (V'', E''_e) is not connected. In either case, (V'', E''_e) is not 2-edge connected. This means that $(V', E'_e \cup E''_c)$ is 2-edge connected. ∎

Lemma 2.5.3 Assume case 2.5.1 applies. Let (V'', E''_e) be a minimal 2-edge connected subgraph of (V'', E''). Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in E'' $\}))$ is a minimal 2-edge connected subgraph of $(V', E'_e \cup E'_c)$.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in E'' $\}$. From Lemma 2.5.2 we know that $(V', E'_e \cup E''_c)$ is 2-edge connected. All we have to prove is that every edge in E''_c is 2-edge essential for $(V', E'_e \cup E''_c)$. Let (v,w) be an edge in E''_c. Edge (v,w) is also a regular edge in E'' and $(v,w) \in E''_e$. If $(V', (E'_e \cup E''_c) - \{(v,w)\})$ is still 2-edge connected, then $(V'', E''_e - \{(v,w)\})$ is also 2-edge connected. Graph (V'', E''_e) is not a minimal 2-edge connected subgraph of (V'', E''). This is a contradiction. ∎

After we construct (V'', E''), we do not wish to keep (V', E'). We do not wish to compute E''_c after we compute (V'', E''_e). As a matter of fact, E''_c can be decided while we are computing (V'', E''_e). For each edge (x,y) in E', we associate a field *source* with it. Before the first recursive call of the algorithm, we let $(x,y).source = (x,y)$ for every edge in E. When we construct (V'', E'') from (V', E'), we let $(x,y).source = E''^{-1}(x,y).source$ for every edge regular edge (x,y) in E'' such that $E''^{-1}(x,y)$ is in E'_c and $(x,y).source = null$ for all other edges in E''. Before the first recursive call, we let $E_{fe} = \phi$. When we compute a minimal 2-edge connected subgraph of (V', E'), if we add an edge $(x,y) \in E'$ to E'_e, we add $(x,y).source$ to E_{fe} if $(x,y).source \neq null$. After the algorithm terminates, (V, E_{fe}) is a minimal 2-edge connected subgraph of (V, E).

Case 2.5.2 $\mid E'_e \cup E'_c \mid > 4n'/3$.

If case 2.5.2 applies, we know that $\mid E'_e \mid > 23n'/18$. Let $(V_1, E_1), ..., (V_p, E_p)$ be the *2-edge connected components* of (V', E'_e). We have $\cup_{i=1}^{p} V_i = V'$ and $V_i \cap V_j = \phi$ for $1 \leq i \neq j \leq p$.

Lemma 2.5.4 The number of 2-edge connected components in (V', E'_e) is no more than $13n'/18$.

Proof: Let (V', E'_T) be a spanning forest of (V', E'_e). We have $\mid E'_T \mid \leq n' - 1$ and $\mid E'_e - E'_T \mid \geq 5n'/18$. Every vertex in V' is in a 2-edge connected component by itself in (V', E'_T). Now we add the edges in $E'_e - E'_T$ to E'_T one by one. When we add edge $(x, y) \in E'_e - E'_T$ to E'_T, (x, y) will cause two 2-edge connected components in (V', E'_T) to merge and decrease the number of 2-edge connected components in (V', E'_T) by at least one. This is true because every edge in E'_e is 2-edge essential for $(V', E'_e \cup E'_c)$. The lemma is true since we have $\mid E'_e - E'_T \mid \geq 5n'/18$. ∎

If case 2.5.2 applies, we construct (V'', E'') from (V', E') in the following way:

(1) We contract V_i $(1 \leq i \leq q)$ into a vertex v''_i. Let $V'' = \{V_1, ..., V_p)\}$.

(2) For every edge (x, y) in E' with $x \in V_i$ and $y \in V_j$, we put an edge $(V_i, V_j)_{(x,y)}$ in E''. We let $E''(x, y) = (V_i, V_j)_{(x,y)}$ and $E''^{-1}(V_i, V_j)_{(x,y)} = (x, y)$. For each edge (x', y') in E'', we let $(x', y').source$ equal to $E''^{-1}(x', y').source$ if $E''^{-1}(x', y')$ is in E'_c and $(x', y').source = null$ otherwise.

Lemma 2.5.5 $\mid V'' \mid \leq 13n'/18 \leq 3n'/4$ and $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$.

Proof: This is obvious from Lemma 2.5.4 and the way (V'', E'') is built. ∎

It is not hard to see that (V'', E'') is 2-edge connected and preserves the 2-edge connectivity structure of $(V', E'_e \cup E'_c)$. (See also Lemma 2.5.6 and 2.5.7.)

Lemma 2.5.6 Assume case 2.5.2 applies. Let (V'', E''_e) be a subgraph of (V'', E'') that is 2-edge connected. Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e\}))$ is also 2-edge connected.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e\}$. Assume that $(V', E'_e \cup E''_c)$ is not 2-edge connected. Let (v,w) be a bridge in $(V', E'_e \cup E''_c)$. If $E''(v,w) \in E''_e$, then $(v.w)$ is a bridge in (V'', E''_e). If $E''(v,w) \notin E''_e$, then (V'', E''_e) is not connected. In either case, (V'', E''_e) is not 2-edge connected. This is a contradiction. ∎

Lemma 2.5.7 Assume case 2.5.2 applies. Let (V'', E''_e) be a minimal 2-edge connected subgraph of (V'', E''). Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e\}))$ is a minimal 2-edge connected subgraph of $(V', E'_e \cup E'_c)$.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e\}$. From Lemma 2.5.6 we know that $(V', E'_e \cup E''_c)$ is 2-edge connected. All we have to prove is that every edge in E''_c is 2-edge essential for $(V', E'_e \cup E''_c)$. Let (v,w) be an edge in E''_c. If $(V', (E'_e \cup E''_c) - \{(v,w)\})$ is still 2-edge connected, then $(V'', E''_e - \{E''(v,w)\})$ is also 2-edge connected. Graph (V'', E''_e) is not a minimal 2-edge connected subgraph of (V'', E''). This is a contradiction. ∎

In this section, we discuss the second part of each recursive call of algorithm 1.2.1. After we construct (V'', E'') from (V', E'), we compute a minimal 2-edge connected subgraph of (V'', E'') by using the algorithm recursively. A minimal 2-edge connected subgraph of (V', E') can be decided while we are computing a minimal 2-edge connected subgraph of (V'', E'').

§2.6 The Complexity of the Algorithm

The algorithm discussed in this chapter can be implemented to run in $O(m+n)$ time and space.

There are $O(log(n))$ recursive calls to the algorithm. In each recursive call, we want to compute a minimal 2-edge connected subgraph of (V', E') with $\mid V' \mid = n'$ and $\mid E' \mid = m'$.

Each recursive call is divided into two parts.

There are twenty-five phases in the first part. In each phase, two procedures are involved. Edge set E'_{sc} is first computed and then processed. (See §2.3 and 2.4.) In computing E'_{sc}, (See also procedure 2.3.1.) spanning tree E'_t can be obtained by computing a minimum spanning tree on a $0-1$ weighted graph. In graph $(V', E_e \cup E'_c)$, we assign a weight to each edge. If $(x,y) \in E'_e$, we let $(x,y).weight = 0$. If $(x,y) \in E'_c$, we let $(x,y).weight = 1$. A minimum spanning tree of $(V', E'_e \cup E'_c)$ can be computed in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. This spanning tree contains as many edges of E'_e as possible. After E'_t is computed, G'_a can be obtained in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. The critical edges in G'_a are easily identified. Edge set E'_{sc} can be computed in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time.

In §2.4, we discuss a procedure to process E'_{sc}. Before the procedure, we compute *high* and *led* for each vertex $v \in V'$. This computation can be done in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time by a preorder traversal of the rooted spanning tree E'_t. (See also [T1].) We also compute the nearest common ancestor of each nontree edge $(x,y) \in E'_e \cup E'_c$. This can be done in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. (See [HT5].) The partition of V' and $(E'_e \cup E'_c) - E'_t$ into partition classes can be implemented in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time by doing

bucket sort on V' and $(E'_e \cup E'_c) - E'_t$ respectively.

The procedure in §2.4 is divided into steps. At the beginning of each step, *slea* can be computed in $O(\mid V_{h_i} \mid)$ time where V_{h_i} is the base partition class of *slea*. Each step is further divided into two substeps. Both substeps can be implemented in $O(\mid slea \mid)$ time. We also have $\mid slea \mid \leq \mid V_{h_i} \mid$.

After the two substeps, we remove the vertices from V_{h_i} and the edges incident on a removed vertex from $(V', E'_e \cup E'_c)$. Let the number of removed tree edges be R_t. The removal of vertices and edges can be finished in $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$ time. The time complexity of each step is $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$. After each step, the size of V' decreases by $\mid V_{h_i} \mid$ and the size of $E'_e \cup E'_c$ decreases by $\mid S \mid + R_t$. The total complexity of the procedure is $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$.

In the first part of the algorithm, assume that we have $m' > 2n'$ at the beginning of the first phase. We have $k = n' - 1 \leq 5K/6 = 5m'/6$ when we compute E'_{sc} for this phase. The procedure in §2.4 is used to process the edges in E'_{sc}. After this procedure, we have $\mid E'_e \cup E'_c \mid \leq 2n'$. Thus the first phase takes $O(m' + n')$ time and each of the remaining twenty-four phases takes $O(n')$ time.

In the second part of each recursive call, (V'', E'') can be computed from $(V', E'_e \cup E'_c)$ in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. A minimal 2-edge connected subgraph of $(V', E'_e \cup E'_c)$ can be obtained while we are computing a minimal 2-edge connected subgraph of (V'', E''). (See Lemma 2.5.3, 2.5.7 and the discussion after Lemma 2.5.3.)

The number of vertices is decreased by at least one fourth in each recursive call, thus the total time complexity of the algorithm is $O(m + n) + O((3/4)n) + O((3/4)^2 n) + ... +$

$O(1) = O(m + n)$.

In each recursive call, after we construct (V'', E''), we can release the space used by (V', E'). The space used by the algorithm never exceeds $O(m + n)$.

§2.7 Problems for Future Research

The algorithm in this chapter is optimal to within a constant factor for solving the minimal 2-edge connected subgraph problem. As we mentioned in §1.2 this problem is actually a special case of the minimal k-edge connected subgraph problem. There is a polynomial algorithm to test k-edge connected graphs. (See [ET3].) Combining this algorithm with the simple iterative algorithm presented at the beginning of §2.2, we have a polynomial algorithm solving the minimal k-edge connected subgraph problem. It would be interesting to know if there is a more efficient (near linear time) algorithm to solve the minimal k-edge connected subgraph problem. Another interesting problem is to improve the parallel algorithm in [KV] for finding a minimal 2-edge connected subgraph of a 2-edge connected graph.

Chapter Three

An Algorithm to Find Minimal 2-Connected Subgraphs

§3.1 Introduction

Let $G = (V, E)$ be a 2-connected graph. In this chapter, we develop an $O(m+n)$ time algorithm (algorithm 1.2.2) for finding a minimal 2-connected subgraph of G. Algorithm 1.2.2 is in many ways similar to algorithm 1.2.1. The presentation in this chapter is parallel to that of the last chapter. A reader will find that some discussions in this chapter have already appeared in chapter two. They are included in chapter three to make the presentation of algorithm 1.2.2 self-contained. Certain proofs and procedures are omitted in chapter three. A reader is referred to the appropriate sections in chapter two.

In finding a minimal 2-connected subgraph of G, we prefer a subgraph with as few edges as possible. The problem of finding a minimal 2-connected subgraph of G that has the fewest edges is NP-complete, however, since the problem is polynomially equivalent to the following NP-complete problem.

Lemma 3.1.1 Let $G = (V, E)$ be a 2-connected graph and k be an integer. The problem of deciding whether there exists an edge set $E' \subseteq E$ with $\mid E' \mid = k$ such that $H = (V, E - E')$ is 2-connected is NP-complete.

Proof : The problem is obviously in NP. To prove that it is NP-complete, we can reduce the undirected Hamilton circuit problem to it. Let $G = (V, E)$ be a 2-connected graph. Let $k = m - n$. G contains a Hamilton circuit iff there exists an edge set $E' \subseteq E$ with $\mid E' \mid = k$ such that $(V, E - E')$ is 2-connected. ∎

Because of Lemma 3.1.1, we can hope to find only minimal 2-connected subgraphs for G. Let m_2 be the number of edges in the minimal 2-connected subgraph found by the algorithm. Let m_1 be the number of edges in the minimal 2-connected subgraph that has

the fewest edges. As in chapter two, we know that $m_2/m_1 \leq 2$. It would be nice to have an algorithm such that for every $k > 0$, the algorithm finds a minimal 2-connected subgraph with m_2 edges and $m_2/m_1 \leq 1 + k$. Unfortunately, no such algorithm is known right now.

§3.2 Outline of the Algorithm

We wish to find a maximal subset $E_0 \subseteq E$ such that graph $(V, E - E_0)$ is still 2-connected. In the presentation of the algorithm, we talk about deleting edges from E. When an edge is deleted, it is added to E_0 and remains there until the end of the algorithm.

In order to find a minimal 2-connected subgraph of G, we can use a simple iterative algorithm. For every edge (x, y) in E, we test whether graph $(V, E - \{(x, y)\})$ is 2-connected. We keep (x, y) in E if graph $(V, E - \{(x, y)\})$ is not 2-connected and delete it from E otherwise. The edges can be processed in any order. At the end of the algorithm, let $E_0 = \{$ the deleted edges $\}$; graph $(V, E - E_0)$ is a minimal 2-connected subgraph of G. Since we have m edges to process, and processing each edge takes $O(m + n)$ time, this algorithm takes $O(m^2)$ time. Although this is not a very efficient algorithm, it is the starting point for the algorithm in this chapter.

In the simple algorithm, we consider only one edge at an iteration step and we test the 2-connectivity of a graph that is only slightly different from the previously tested graph. The time complexity of the algorithm can be greatly improved if we can process a group of edges at a time.

As does algorithm 1.2.1, algorithm 1.2.2 runs recursively. Let us assume that at the beginning of each recursion, we want to compute a minimal 2-connected subgraph of (V', E'). (At the beginning of the first recursion, we have $V' = V$ and $E' = E$.) In each recursion, we want to find a maximal subset $E'_0 \subseteq E'$ such that $(V', E' - E'_0)$ is still 2-connected. Each recursion is divided into two parts. The first part is divided into twenty-five phases. At the beginning of each phase, we have $E' = E'_e \cup E'_c \cup E'_0$ and $E'_e \cap E'_c = E'_c \cap E'_0 = E'_e \cap E'_0 = \phi$. The edges in E'_0 are deleted from E' and do not affect any future processing of (V', E').

The inductive hypothesis at the beginning of each phase is that each edge in E'_e is 2-*essential* (see Definition 24 in the appendix.) for $(V', E'_e \cup E'_c)$.

The edges in E'_c are unprocessed edges and are called *candidate* edges. In each phase, a subset $E'_{sc} \subseteq E'_c$ is computed such that $\mid E'_{sc} \mid \geq 1/6 \mid E'_c \mid$. E'_{sc} is then split into two sets E'_{0sc} and E'_{esc}. The edges in E'_{0sc} are added to E'_0 and the edges in E'_{esc} are added to E'_e. The inductive hypothesis is maintained and we go to the next phase. Let $\mid V' \mid = n'$ and $\mid E' \mid = m'$. The first phase takes $O(m' + n')$ time. After the first phase, we have $\mid E'_e \cup E'_c \mid \leq 2n'$. Each of the remaining twenty-four phases takes $O(n')$ time. After the first part of each recursion, we have $\mid E'_c \mid \leq (5/6)^{24} 2n' \leq n'/18$.

In the second part of each recursion, we use the fact that $E'_c \leq n'/18$ to compute a new graph (V'', E'') such that (1) $\mid V'' \mid \leq 5/6 \mid V' \mid$, (2) (V'', E'') is 2-connected, (3) $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$, and (4) if we know a minimal 2-connected subgraph of (V'', E''), we can easily compute a subset $E'_{ec} \subseteq E'_c$ such that $(V', E'_e \cup E'_{ec})$ is a minimal 2-connected subgraph of (V', E'). The algorithm then computes a minimal 2-connected subgraph of (V'', E'') by using itself recursively.

Procedure 2.2.1 is also an outline of algorithm 1.2.2. The difference between the algorithms lies in (1) the procedures to split E'_{sc} in each phase in the first part of each recursion, and (2) the procedures to compute (V'', E'') from $(V', E'_e \cup E'_c)$ in the second part of each recursion.

§3.3 Finding E'_{sc} in Each Phase

This section is concerned with finding E'_{sc} at the beginning of each phase. We start with graph $(V', E'_e \cup E'_c)$. A spanning tree (V', E'_t) of $(V', E'_e \cup E'_c)$ is computed such that E'_t contains as many edges from E'_e as possible. Let $\mid E'_c \mid = K, \mid E'_c \cap E'_t \mid = k$. The edges in $E'_c \cap E'_t$ $((E'_c - (E'_c \cap E'_t)))$ are called tree (nontree) candidate edges. The procedure for finding E'_{sc} is exactly the same as the procedure for finding E'_{sc} in §2.3. The procedure is included here to make the presentation of algorithm 1.2.2 self contained. A reader can skip this section if he (or she) has read §2.3.

Let the connected components of (V', E'_e) be $(V'_1, E_1), (V'_2, E'_2), ..., (V'_p, E'_p)$. We have $\cup_i V'_i = V'$, $\cup_i E'_i = E'_e$, and $V'_i \cap V'_j = \phi$ for $1 \leq i \neq j \leq p$.

We contract the vertices in each V'_i into a single vertex to build an auxiliary graph $G'_a = (V'_a, E'_a)$. We have $V'_a = \{V'_i \mid 1 \leq i \leq p\}$. For every edge (x, y) in E'_c with $x \in V'_i$, $y \in V'_j$, we have an edge $(V'_i, V'_j)_{(x,y)} \in E'_a$. We shall use $E'_a(x, y)$ to denote this edge, i.e. $E'_a(x, y) = (V'_i, V'_j)_{(x,y)}$ and $E'^{-1}_a(V'_i, V'_j)_{(x,y)} = (x, y)$. G'_a may contain multiple edges and loops. See Figures 2.3.1, 2.3.2, and 2.3.3.

It is not hard to see that $(V'_a, \{E'_a(x, y) \mid (x, y) \in E'_c \cap E'_t\})$ is a spanning tree for G'_a, The number of vertices in G'_a is $k + 1$.

G_a also satisfies following properties:

Lemma 3.3.1 G'_a is 2-edge connected.

Proof: This is obvious since $(V', E'_e \cup E'_c)$ is 2-connected. ∎

Lemma 3.3.2 Let edge $(x,y) \in E'_c$. If $E'_a(x,y)$ is *2-edge essential* in G'_a, edge (x,y) is *2-essential* in $(V', E'_e \cup E'_c)$.

Proof: Let (x',y') be a *bridge* in $(V'_a, E'_a - E'_a(x,y))$. Then ${E'_a}^{-1}(x',y')$ is a bridge in $(V', (E'_e \cup E'_c) - (x,y))$. Thus edge (x,y) is a 2-essential edge of $(V', E'_e \cup E'_c)$. ∎

E'_{sc} is determined according to the following two cases:

Case 3.3.1 $k \leq 5K/6$.

In this case, we let $E'_{sc} = E'_c - E'_t$. Obviously, $\mid E'_{sc} \mid = \mid E'_c - (E'_c \cap E'_t) \mid = K - k \geq K/6$. A procedure to process the edges in E'_{sc} is given in §3.4.

Case 3.3.2 $k > 5K/6$.

In this case, most of the candidate edges are tree candidate edges and we shall prove that there are at least $K/6$ edges in G'_a that are 2-edge essential for G'_a.

Lemma 3.3.3 If case 3.3.2 applies, there are at least $K/6$ vertices in G'_a with degree equal to 2.

Proof: Since G'_a is 2-edge connected, every vertex in G'_a has degree at least 2. Let q be the number of degree 2 vertices. Assume that $q < K/6$. We have $\sum_{v_i \in V'} deg(v_i) \geq 3(k+1-q) + 2q = 3(k+1) - q > 3(k+1) - K/6 > 2K$. But the sum of degrees equals $2K$ since the number of edges in G_a is K. This is a contradiction. ∎

Let (x',y') be an edge in G'_a. If the degree of x' or y' equals 2, edge (x',y') is obviously

a 2-edge essential edge in G'_a. We call (x', y') a *critical* edge of G'_a.

Lemma 3.3.4 If case 3.3.2 applies, there are at least $K/6$ critical edges in G'_a.

Proof: We know from Lemma 3.3.3 that there are at least $K/6$ vertices in G'_a with degree equal to 2. Each such vertex provides two critical edges and each critical edge is counted at most twice, so the lemma is true. ∎

Let $E'_{sc} = \{(x, y) \mid E'_a(x, y)$ is a critical edge in $G'_a\}$. From Lemma 3.3.2 and 3.3.4, we know that every edge in E'_{sc} is 2-essential for $(V', E'_e \cup E'_c)$ and $\mid E'_{sc} \mid > K/6$.

The following pseudo code outlines the procedure to compute E_{sc} at the beginning of each phase.

Procedure 3.3.1

Input: $(V', E'_e \cup E'_c)$
Output: E'_{sc}
{
Compute a spanning tree (V', E'_t) for $(V', E'_e \cup E'_c)$;
such that E'_t contains as many edges of E'_e as possible;
Let $\mid E'_c \mid = K$ and $\mid E'_c \cap E'_t \mid = k$;
Construct $G'_a = (V'_a, E'_a)$;
If $(k \leq 5K/6)$;
Let $E'_{sc} = E'_c - E'_c \cup E'_t$;
Else $E'_{sc} = \{(x, y) \mid E'_a(x, y)$ is a critical edge of $G'_a\}$;
}

§3.4 Processing the Edges in E'_{sc}

From the discussion of §3.3, we know that when $k > 5K/6$, every edge in E'_{sc} is 2-essential for $(V', E'_e \cup E'_c)$. In this case, no further processing is needed for the edges in E'_{sc}. All the edges in E'_{sc} are added to E'_e. We go to the next phase of the algorithm.

If $k \leq 5K/6$, then we have $E'_{sc} = E'_c - E'_t$. Using the procedure discussed in this section, some edges in E'_{sc} are added to E'_e and others are added to E'_0. We shall prove that the inductive hypothesis is maintained at the end of the procedure.

We again consider graph $(V', E'_e \cup E'_c)$. Let (V', E'_t) be the spanning tree described in §3.3. For the sake of simplicity, the procedure actually processes the edges in $(E'_c \cup E'_e) - E'_t$ (the nontree edges in $(V', E'_e \cup E'_c)$). If we pick a vertex in V', we can convert (V', E'_t) into a rooted spanning tree. The procedure works bottom-up on the tree. E'_{sc} is split into E'_{0sc} and E'_{esc} such that every edge in E'_{esc} is 2-essential for$(V', (E'_e \cup E'_{esc}) \cup (E'_c - E'_{sc})$. The edges in $E'_e - E'_t$ will be added to E'_{esc} automatically in the process.

After (V', E'_t) is turned into a rooted spanning tree, we number the vertices in V' by a postorder numbering of (V', E'_t). In the following discussion, the vertices will be referred to by their number. We also mark the leaves of (V', E'_t). For each nontree edge (x, y), let $nca(x, y)$ be the nearest common ancestor of x and y in (V', E'_t). That is , $nca(x, y) = \min \{v \mid v \text{ is an ancestor of } x \text{ and } y\}$. For each vertex v in V, let $bc(v)$, $high(v)$, $led(v)$ be as defined in §2.3.

At the beginning of the procedure, we compute *high* for every vertex in V' and *nca* for every nontree edge in $E'_e \cup E'_c$. For each vertex v, we also compute $led(v)$. All these can be computed in linear time (See [HT5] for the computation of nearest common ancestors

of nontree edges).

For each edge (x,y) in $(E'_e \cup E'_c) - E'_t$, if $nca(x,y)$ <min$(high(x), high(y))$, we remove it from $E'_e \cup E'_c$ and put it in E'_0. After this operation , we have $nca(x,y) = \min(high(x), high(y))$ for every $(x,y) \in (E'_e \cup E'_c) - E'_t$.

After computing the *high* values, we partition V' into $V_{h_1}, ..., V_{h_l}$ such that (1) $high(v) = high(w)$ for $v \in V_{h_i}$ and $w \in V_{h_i}$ with $1 \leq i \leq l$, and (2) $high(v) < high(w)$ for $v \in V_{h_i}$ and $w \in V_{h_j}$ with $1 \leq i < j \leq l$. We call $V_{h_1}, ..., V_{h_l}$ the *partition classes* of V'. Let $high(V_{h_i}) = high(v)$ where $v \in V_{h_i}$. This partition can be computed by doing a bucket sort on V' using the *high* values as the sort key. At the beginning of the procedure, we mark every partition class of V' as unprocessed. In the procedure, the partition classes are processed according to the order of ascending *high* values. After a partition class is processed, it is marked so and the vertices in the partition class are removed from $(V', E'_e \cup E'_c)$.

We also partition $(E'_e \cup E'_c) - E'_t$ into $E_{h_1}, ..., E_{h_l}$ such that (1) $nca(v,w) = nca(x,y)$ for $(v,w) \in E_{h_i}$ and $(x,y) \in E_{h_i}$ with $1 \leq i \leq l$, and (2) $nca(v,w) < nca(x,y)$ for $(v,w) \in E_{h_i}$ and $(x,y) \in E_{h_j}$ with $1 \leq i < j \leq l$. We call $E_{h_1}, ..., E_{h_l}$ the *partition classes* of the nontree edges. Let $nca(E_{h_i}) = nca(x,y)$ where $(x,y) \in E_{h_i}$. This partition can be computed by doing a bucket sort on $(E'_e \cup E'_c) - E'_t$ using the *nca* values as the sort key. It is important to notice that there is a $1-1$ correspondence ($P : V_{h_i} \rightarrow E_{h_i}$ where $high(V_{h_i}) = nca(E_{h_i})$) between the partition classes of V' and the partition classes of $(E'_e \cup E'_c) - E'_t$.

For each vertex $v \in V'$, we associate a field $isleaf$ with v. We have $v.isleaf = 1$ if v is a leaf in (V', E'_t) and $v.isleaf = 0$ otherwise. In the procedure, a nonleaf vertex can turn into a leaf vertex, but not vice versa.

The procedure is divided into steps and works bottom-up on (V', E'_t). At the beginning of the procedure, we let $E'_{0sc} = \phi$ and $E'_{esc} = \phi$. In order to make the procedure efficient, we wish to design it in such a way that we can consider a group of nontree edges at a time, and decide which edges should be added to E'_{0sc} and which should be added to E'_{esc}, by studying the local structure of these edges.

At the beginning of each step, let $slea = \{v \mid v$ is a leaf, $v \in V_{h_i}$, and V_{h_i} is the unprocessed partition class of V' with the smallest $high$ value $\}$. We call V_{h_i} the *base partition class* of $slea$. We have $| slea | \leq | V_{h_i} |$.

Let $v_0 = high(V_{h_i})$. We assume that $v_1, ..., v_p$ are the *children* of v_0 with $high(v_i) = v_0$ and $u_1, ..., u_q$ are the children of v_0 with $high(u_j) > v_0$. Let $T_{v_i} = (V_{v_i}, E_{v_i})$ and $T_{u_j} = (V_{u_j}, E_{u_j})$ be the subtrees of (V', E'_t) that are rooted at v_i and u_j respectively. Let $S = \{(x, y) \mid nca(x, y) = v_0$ and $(x, y) \in (E'_e \cup E'_c) - E'_t\}$. Notice that S is a partition class of $(E'_e \cup E'_c) - E'_t$ where $nca(S) = v_0$. (See Figure 3.4.1).

In each step, a subset CS of S is chosen. If we add the edges in $S - CS$ to E'_{0sc}, $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-connected and every edge in CS is 2-essential for the graph. Vertex v_0 is chosen in such a way that CS can be computed locally by using the subgraphs of $(V', E'_e \cup E'_c)$ spanned by T_{v_i} and T_{u_j}. The processing of nontree edges in $(V', E'_e \cup E'_c)$ with nearest common ancestor greater than v_0 will not affect the choice of the edges in CS. (See Lemma 3.4.7 for details.)

After CS is chosen, the vertices in V_{h_i} are removed from $(V', E'_e \cup E'_c)$. We also remove any tree edge or nontree edge that is incident on a removed vertex. Let the set of removed nontree edges be S'. We prove that $S' = S$. We use induction on the steps of the procedure. Assume that every vertex v with $high(v) < v_0$ and every nontree edge (x, y) with $nca(x, y) < v_0$ have been removed from $(V', E'_e \cup E'_c)$ at the beginning of the step. Let

$(x,y) \in S'$ and $x \in V_{h_i}$. We have $high(x) = v_0$ and $nca(x,y) = min(high(x), high(y)) = v_0$ since we know that $high(y) \geq v_0$. This means $(x,y) \in S$. Let $(x,y) \in S$. If both $high(x) > v_0$ and $high(y) > v_0$ then we have $nca(x,y) > v_0$. This is a contradiction. Assume that $high(x) = v_0$. We have $v \in V_{h_i}$ and $(x,y) \in S'$. (See also the discussion after substep 4.) After the removal of vertices and edges, we go to the next step of the procedure. When $(V', E'_e \cup E'_c)$ is empty, the procedure terminates. Since the removal of vertices and edges only affect the procedure in this section, the procedure actually works on a copy of $(V', E'_e \cup E'_c \cup E'_0)$ and saves the original $(V', E'_e \cup E'_c \cup E'_0)$ for future processing.

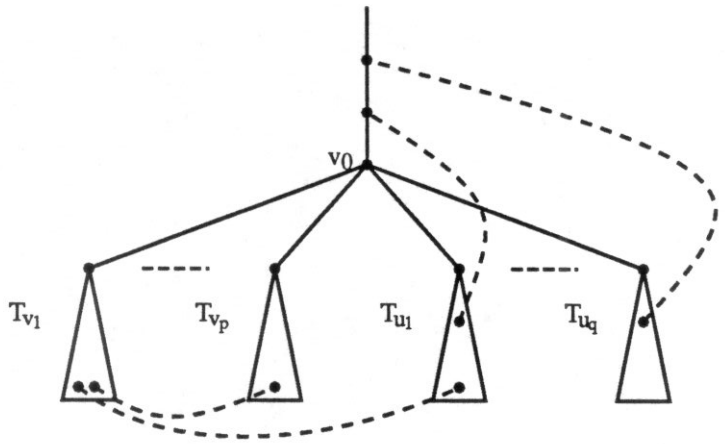

Figure 3.4.1

The process of computing CS involves four substeps (remember that computing CS involves only two substeps in algorithm 1.2.1). The goal of the substeps is to choose a minimal subset CS of S such that $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-connected. Let us consider a vertex v in *sleά*. There is a unique path in (V', E'_t) from v_0 to v, say $v_0, v_1, ..., v_l, v$. We must pick at least one edge from $bc(v)$ and put it in CS, since v_l would be an articulation point in $(V', (E'_e \cup E'_c) - (S - CS))$ otherwise. Since we have $high(v) = v_0$, if we put an edge from $led(v)$ in CS, we are sure that $(v_0, v_1), ..., (v_l, v)$ will be in the same 2-connected component in $(V', (E'_e \cup E'_c) - (S - CS))$. See Figure 3.4.2.

Substep 1 For each vertex v in *slea*, we choose an edge from $led(v)$ and put it in CS.

Although every vertex v in *slea* is now covered by at least one edge in CS, CS may not be a minimal subset of S that covers every vertex of *slea*.

Substep 2 We build an auxiliary graph (V_a, CS). $V_a = \{v \mid \exists w, (v,w) \in CS\}$. The vertices of V_a that are in *slea* are marked red. By using a minimal edge covering algorithm, we find a minimal subset of CS that covers all the red vertices of V_a.

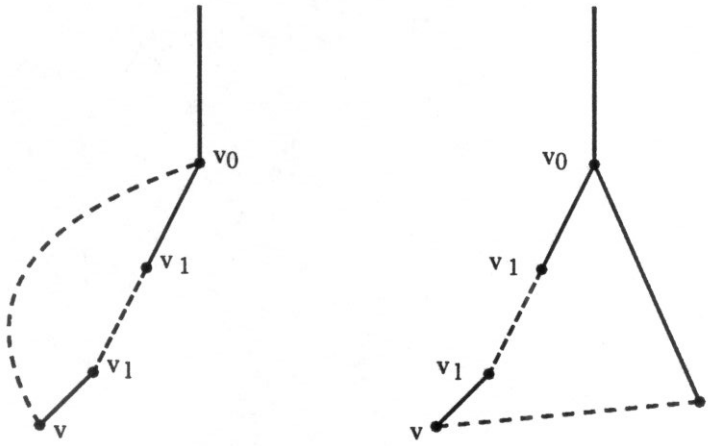

Figure 3.4.2

Both substep 1 and substep 2 can be performed in $O(|\ slea\ |)$ time.

Let CS be the minimal subset found in substep 2. Now the edges in CS satisfy the following lemma:

Lemma 3.4.1 (1) For any set of four vertices u, v, x, y in *slea*, if both (u,v) and (x,y) are in CS, then (v,x) cannot be in CS. (2) If (x,y) is in CS and y is not in *slea*, then (x,y) must be the unique edge in CS that is incident on x.

Proof: The properties are direct results from substep 2. ∎

From Lemma 3.4.1, we know that (V_a, CS) has no path with length longer than 2 after substep 2. Let (x, y) be in CS and $x \in slea$. (x, y) can be in one of the following cases (1) $y \notin slea$ and (x, y) is the only edge in CS that is incident on x. (2) $y \in slea$ and (x, y) is the only edge in CS that is incident on both x and y. (3) $y \in slea$, (x, y) is the only edge in CS incident on y and (x, y) is not the only edge in CS incident on x. The cases are shown in Figure 3.4.3. We call the third case the *star* case. The partition of edges in CS into three cases will be useful in substep 4.

After we pick edges to cover the vertices in *slea*, some subtrees T_{v_i} are connected by edges in CS to form biconnected components in $(V', (E'_e \cup E'_c) - (S - CS))$. In order to make $(V', (E'_e \cup E'_c) - (S - CS))$ 2-connected, more edges may need to be added to CS to connect these components. These edges are chosen by studying the connectivity structure of $(V', (E'_e \cup E'_c) - (S - CS))$ at v_0. The structure is represented in the following auxiliary graph.

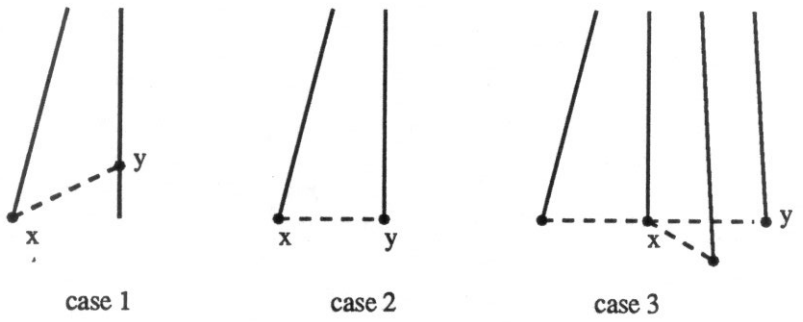

Figure 3.4.3

We build an auxiliary graph (V_{ag}, E_{ag}) where $V_{ag} = \{T_{v_i} \mid 1 \leq i \leq p\} \cup \{y \mid \exists (x, y) \in S, high(y) > v_0\}$ and $E_{ag} = \{(T_{v_i}, T_{v_j}) \mid \exists (x, y) \in S, x \in V_{v_i}, y \in V_{v_j}\} \cup \{(T_{v_i}, y) \mid \exists (x, y) \in$

$S, x \in V_{v_i}, high(y > v_0\}$. When we construct (V_{ag}, E_{ag}), we first propagate a symbol "T''_{v_i} throughout V_{v_i} so that every vertex in V_{v_i} knows that it is in T_{v_i}. This can be done in $O(\sum_{i=1}^{p} \mid V_{v_i} \mid)$ time. Graph (V_{ag}, E_{ag}) can then be constructed in $O(\sum_{i=1}^{p} \mid V_{v_i} \mid + \mid S \mid)$ time. Since we have $\sum_{i=1}^{p} \mid V_{v_i} \mid \leq \mid V_{h_i} \mid$. Graph (V_{ag}, E_{ag}) can be constructed in $O(\mid V_{h_i} \mid + \mid S \mid)$ time. We also have $\mid V_{ag} \mid \leq \mid V_{h_i} \mid + \mid S \mid$ and $\mid E_{ag} \mid \leq \mid S \mid$.

For each edge (x, y) in S, there is a corresponding edge in E_{ag}. We call this edge $E_{ag}(x, y)$. Each vertex y is marked red in (V_{ag}, E_{ag}). (See Figure 3.4.4.)

In Figure 3.4.4, $\{T_{v_1}, T_{v_2}\}$ forms a connected component of (V_{ag}, E_{ag}) that contains no red vertex. It is obviously true that more edges need to be added to CS. Otherwise, v_0 will be an articulation point in $(V', (E'_c \cup E'_e) - (S - CS))$ with T_{v_1} and T_{v_2} forming a 2-connected component. In algorithm 1.2.1, it is enough to have $(V', (E'_e \cup E'_c) - (S - CS))$ 2-edge connected. Substeps 3 and 4 are not needed there.

Let us consider graph $(V_{ag}, \{E_{ag}(x, y) \mid (x, y) \in CS\})$. Let $CT_1, ..., CT_k$ be the connected components of this graph. If there exists a CT_i that contains no red vertex, then $(V', (E'_e \cup E'_c) - (S - CS))$ is not 2-connected. More edges need to be put in CS.

Substep 3 We find a minimal subset CS_1 of $S - CS$ such that every connected component in $(V_{ag}, \{E_{ag}(x, y) \mid (x, y) \in CS \cup CS1\})$ contains at least one red vertex. CS_1 exists since $(V', E'_e \cup E'_c)$ is 2-connected. CS_1 can be found in $O(\mid V_{ag} \mid + \mid E_{ag} \mid)$ time because of Lemma 3.4.2.

Lemma 3.4.2 Let $G_l = (V_l, E_l)$ be a connected graph. At least one of the vertices in V_l is marked red. In $O(\mid V_l \mid + \mid E_l \mid)$ time, we can find a minimal subset $E'_l \subseteq E_l$ such that every connected component in $O(V_l, E'_l)$ contains at least one red vertex.

Proof: We choose a red vertex v in V_l and construct a rooted spanning tree (V_l, E_{lt}) for G_l. For each red vertex v in V_0, let $f(v)$ be the parent of v in (V_l, E_{lt}). Let $E'_l = E_{lt} - \{(v, f(v)) \mid v \text{ is red }\}$. E'_l is the required set. ∎

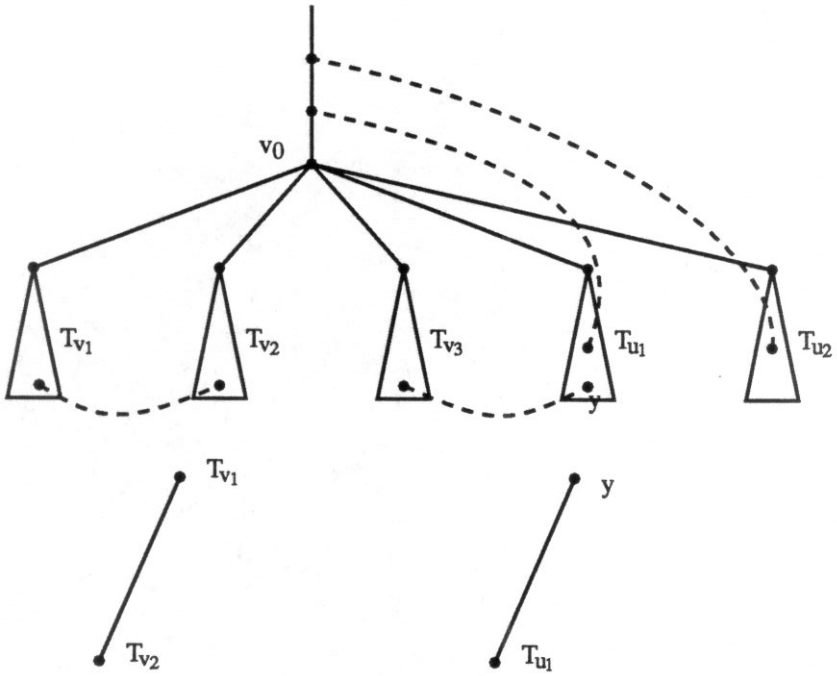

Figure 3.4.4

The reader has probably noticed that we are not adding the edges of CS_1 to CS immediately. This caution has a good reason. The problem is that the edges in CS_1 may also cover vertices in *slea*, thus making some edges in CS unnecessary. Substep 4 removes these edges.

Substep 4 In this substep, we will manipulate the subsets of CS. A subset CS' of CS is called a *valid* set, if

Condition 3.4.1 $CS' \cup CS_1$ covers every vertex in *slea*.

Condition 3.4.2 If a connected component CT_i in $(V_{ag}, \{E_{ag}(x,y) \mid (x,y) \in CS\})$ does not contain a red vertex, it is still a connected component in $(V_{ag}, \{E_{ag}(x,y) \mid (x,y) \in CS'\})$.

Condition 3.4.3 If a connected component CT_i in $(V_{ag}, \{E_{ag}(x,y) \mid (x,y) \in CS\})$ contains a red vertex, it can split into several components in $(V_{ag}, \{E_{ag}(x,y) \mid (x,y) \in CS'\})$, each containing a red vertex.

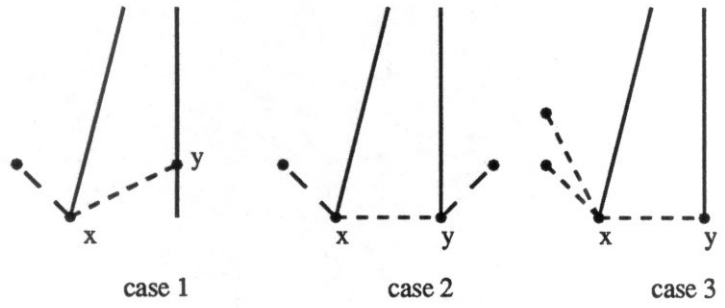

Figure 3.4.5

The purpose of substep 4 is to find a minimal subset of CS that is valid. Since CS_1 is chosen in substep 3 to connect the components of $(V_{ag}, \{E_{ag}(x,y) \mid (x,y) \in CS\})$, a valid set is required to satisfy condition 3.4.2 and 3.4.3 to maintain $(V', (E'_e \cup E'_c) - (S - CS))$ 2-connected after the substep.

For every edge $(x,y) \in CS$, let $x \in slea$. Because of Lemma 3.4.1, (x,y) can be taken out of CS iff one of the following cases happens.

Case 3.4.1 Only one edge in CS is incident on x; $y \notin slea$ and x is incident on an edge in CS_1.

Case 3.4.2 $y \in slea$, x, y are both incident on edges in CS_1.

Case 3.4.3 More than one edge in CS is incident on x, x is not on an edge in $CS1$, $y \in slea$ and y is incident on an edge in CS_1.

Notice that these three cases correspond to the three cases in Figure 3.4.3.

In Figure 3.4.5, the edges in CS_1 are long dashed and the edges in CS are short dashed.

If we delete (x, y) from CS in case 3.4.1 and case 3.4.2, the vertices in *slea* are still covered by edges in CS_1. Case 3.4.3 corresponds to the *star* case in Figure 3.4.3. In this case, if all the edges in CS that are incident on x are deleted from CS, x is left uncovered by $CS \cup CS_1$.

Let $CD_1 = \{(x, y) \in CS \mid (x, y)$ satisfies one of the cases 3.4.1, 3.4.2 , and 3.4.3 $\}$. $(CS - CD_1) \cup CS_1$ may leave some vertices x in case 3.4.3 uncovered. If this happens, we pick an arbitrary edge (x, y) in CD_1 that covers x. Let CD_2 be the set of all such edges. We have $CD_2 \subseteq CD_1$.

Consider graph $(V_{ag}, \{E_{ag}(x, y) \mid (x, y) \in (CS - CD_1) \cup CD_2\}$. We choose a minimal subset CS_2 of $CD_1 - CD_2$ such that $(CS - CD_1) \cup CS_2 \cup CD_2$ is a valid set. Such a set CS_2 exists since CS itself is valid. CS_2 can be computed in $O(\mid V_{ag} \mid + \mid E_{ag} \mid)$ time because of Lemma 3.4.3 and 3.4.4.

Lemma 3.4.3 Let $G_l = (V_l, E_l)$ be a connected graph. Let $E'_l \subseteq E_l$. In $O(\mid V_l \mid + \mid E_l \mid)$ time, we can find a minimal subset $E''_l \subseteq E'_l$ such that $O(V_l, (E_l - E'_l) \cup E''_l)$ is connected.

Proof: Let the connected components of $(V_l, E_l - E'_l)$ be $C_1, ..., C_p$. We contract C_i into a vertex v_i to obtain a new graph (V_{l1}, E_l). Every edge in $E_l - E'_l$ is a self loop in (V_{l1}, E_l). Let (V_{l1}, E_t) be a spanning tree of (V_{l1}, E_l). E_t is a minimal subset of E'_l such that $(V_l, (E_l - E'_l) \cup E_t)$ is connected. ∎

Lemma 3.4.4 Let $G_l = (V_l, E_l)$ be a connected graph. At least one of the vertices in V_l is red. Let $E'_l \subseteq E_l$. In $O(\mid V_l \mid + \mid E_l \mid)$ time, we can find a minimal subset $E''_l \subseteq E'_l$ such that every connected component in $(V_l, (E_l - E'_l) \cup E''_l)$ contains at least one red vertex.

Proof: Let the connected components of $(V_l, E_l - E'_l)$ be $C_1, ..., C_p$. We contract C_i into a vertex v_i to obtain a new graph (V_{l1}, E_l). Every edge in $E_l - E'_l$ is a self loop in (V_{l1}, E_l). Lemma 3.4.2 can be used to find a minimal set $E''_l \subseteq E'_l$ such that every connected component in (V_{l1}, E''_l) contains a red vertex. E''_l is the required set. ∎

Remember that the edges in CD_2 are chosen to cover the vertices in case 3.4.3 of Figure 3.4.5. Let $(x, y) \in CS_2$. The nice thing about the computation of CS_2 is that no matter how we process the edges in CD_2 later, $(CS - CD_1) \cup CD_2 \cup (CS_2 - \{(x, y)\})$ is not a valid set.

Lemma 3.4.5 Let $(x, y) \in (CS - CD_1) \cup CS_2$. For every $CD'_2 \subseteq CD_2$, $(CS - CD_1) \cup CS_2 \cup CD'_2 - \{(x, y)\}$ is not a valid set.

Proof: If $(x, y) \in CS - CD_1$, then $(CS - CD_1) \cup CS_2 \cup CD'_2 - \{(x, y)\} \cup CS_1$ will leave at least one vertex in *slea* uncovered. If $(x, y) \in CS_2$, at least one connected component in $(V_{ag}, \{E_{ag}(x, y) \mid (x, y) \in ((CS - CD_1) \cup CS_2 \cup CD'_2 - \{(x, y)\}) \cup CS_1\}$ does not contain a red vertex. ∎

The edges in CD_2 were chosen to cover the vertices in *slea*. Since some edges in CS_2 also cover vertices in *slea*, some edges in CD_2 may not be necessary.

Let $CD_3 = \{(x,y) \mid (x,y) \in CD_2\}$ and x is covered by an edge in CS_2. Using Lemma 3.4.3 and 3.4.4, we can compute a minimal subset $CS_3 \subseteq CD_3$ such that $(CS - CD_1) \cup CS_2 \cup (CD_2 - CD_3) \cup CS_3)$ is a valid set. This time the complicated *star* case (case 3.4.3 in Figure 3.4.3) goes away and we are sure that every vertex in *slea* is covered by an edge in $(CS - CD_1) \cup CS_2 \cup (CD_2 - CD_3)$.

Let $CS = (CS - CD_1) \cup CS_2 \cup (CD_2 - CD_3) \cup CS_3$.

Lemma 3.4.6 CS is a valid set. For any edge $(x,y) \in CS$, $CS - \{(x,y)\}$ is not a valid set.

Proof: The proof is similar to that of Lemma 3.4.5 and is omitted here. ∎

Let $CS = CS \cup CS_1$.

Substep 4 can be implemented in $O(\mid V_{ag} \mid + \mid E_{ag} \mid)$ time.

After the computation of substep 4, we remove the vertices of V_{h_i} from $(V', E'_e \cup E'_c)$. We also remove every tree edge and nontree edge incident on a removed vertex. If a nonleaf vertex v becomes a leaf after the removal of the vertices in V_{h_i}, we let $v.isleaf = 1$. We already know that the set of removed nontree edges is S. Let the number of removed tree edges be R_t. The removal of vertices and edges can be finished in $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$ time. After the removal of vertices and edges, we go to the next step. The time complexity of each step is $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$. After each step, the size of V' decreases by $\mid V_{h_i} \mid$

and the size of $E'_e \cup E'_c$ decreases by $\mid S \mid + R_t$. The total complexity of the procedure is $O(\mid V \mid + \mid E'_e \mid + \mid E'_c \mid)$.

Let $E'_{esc} = E'_{esc} \cup CS$ and $E'_{0sc} = E'_{0sc} \cup (S - CS)$.

Lemma 3.4.7 Graph $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-connected and the edges in CS are all 2-essential.

Proof: $(V', (E'_e \cup E'_c) - (S - CS))$ is 2-connected because of the way CS is chosen. If we delete one edge (x, y) from CS, either one of the vertices in *slea* becomes uncovered or v_0 becomes an articulation point. The edges processed at later steps of the procedure will all have *high* values greater than v_0. After the procedure, $(V', (E'_e \cup E'_c) - (S - CS))$ is not 2-connected. ∎

The following pseudo code outlines the procedure in this section.

Procedure 3.4.1

{

Let (V', E'_t) be the spanning tree in §3.3;

For each vertex $v \in V'$, compute $high(v)$;

For each nontree edge (v, w), compute $nca(v, w)$;

For each vertex $v \in V'$, compute $led(v)$;

For each $(x, y) \in (E'_e \cup E'_c) - E'_t$ with $nca(x, y) <$min$(high(x), high(y))$

  Remove (x, y) from $E'_e \cup E'_c$ and put it in E'_0;

Partition V' into partition classes: $V_{h_1}, ..., V_{h_l}$;

Partition $(E'_e \cup E'_c) - E'_t$ into partition classes: $E_{h_1}, ..., E_{h_l}$;

Mark each $V_{h_i}(1 \leq i \leq l)$ as unprocessed;

While $(V', E'_e \cup E'_c)$ is not empty

{

Let $slea = \{v \mid v$ is a leaf, $v \in V_{h_i}$, and V_{h_i} is

the unprocessed partition class of V' with the smallest $high$ value $\}$;

Let $v_0 = high(V_{h_i})$;

Let $v_1, ..., v_p$ be the children of v_0 with $high(v_i) = v_0$;

Let $u_1, ..., u_q$ be the children of v_0 with $high(u_j) < v_0$;

Let T_{v_i} and T_{u_j} be the subtree of (V', E'_t)

rooted at v_i and u_j;

Compute CS using the four substeps described above;

Remove the vertices of V_{h_i} and the edges incident

on a vertex in V_{h_i} from $(V', E'_e \cup E'_c)$;

Mark V_{h_i} as processed;

}

}

§3.5 Computing (V'', E'') from $(V', E'_e \cup E'_c)$

After the first part of the algorithm, we obtain a graph $(V', E'_e \cup E'_c)$ such that (1) $(V', E'_e \cup E'_c)$ is 2-connected, (2) every edge in E'_e is 2-essential for the graph, and (3) $\mid E'_c \mid \leq n'/18$.

In this section, we discuss a procedure to compute a new graph (V'', E'') such that (1) (V'', E'') is 2-connected, (2) $\mid V'' \mid \leq 5n'/6$, (3) $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$, and (4) if we know a minimal 2-connected subgraph of (V'', E''), we can process the edges in E'_c in $O(n')$ time and compute a minimal 2-edge connected subgraph of (V', E').

Graph (V'', E'') is constructed in such a way that it preserves the 2-connectivity structure of $(V', E'_e \cup E'_c)$.

In the following discussion, we assume that if $(x, y) \in E'_c$ then $deg(x) > 2$ and $deg(y) > 2$. If $deg(x) = 2$ or $deg(y) = 2$, (x, y) is always 2-essential for $(V', E'_e \cup E'_c)$.

There are two cases to consider:

Case 3.5.1 $\mid E'_e \cup E'_c \mid \leq 4n'/3$.

Lemma 3.5.1 If case 3.5.1 applies, there are at least $n'/4$ vertices in $(V', E'_e \cup E'_c)$ with degree equal to 2.

Proof: See proof of Lemma 2.5.1. ∎

A path $v_1, ..., v_p$ in $(V', E'_e \cup E'_c)$ is a chain if we have $deg(v_1) > 2$, $deg(v_p) > 2$ and

$deg(v_i) = 2$ for $1 < i < p$. Every degree two vertex in $(V', E'_e \cup E'_c)$ is on a unique chain in the graph. v_1 and v_p are called the end vertices of the chain. Notice that a chain may be a cycle with $v_1 = v_p$.

If case 3.5.1 applies, we know from Lemma 3.5.1 that there are at least $n'/4$ vertices in $(V', E'_e \cup E'_c)$ with degree 2. We contract these vertices to build (V'', E''). We construct (V'', E'') in the following way:

(1) Let $V'' = V' - \{v \mid deg(v) = 2\}$.

(2) For each edge $(x, y) \in E'_e \cup E'_c$ with $deg(x) > 2$ and $deg(y) > 2$, add an edge (x, y) to E''. (x, y) is called a *regular* edge in E''. For each regular edge $(x, y) \in E''$, let $E''^{-1}(x, y)$ be the corresponding edge in $(V', E'_e \cup E'_c)$.

(3) For each chain $v_1, ..., v_p$, we add a new edge (v_1, v_p) to E''. (v_1, v_p) is called a *special* edge in E''. For each special edge $(x, y) \in E''$, let $E''^{-1}(x, y)$ be the corresponding chain in $(V', E'_e \cup E'_c)$.

It is not hard to see that (V'', E'') is 2-connected and preserves the 2 con- nectivity structure of $(V', E'_e \cup E'_c)$ (See also Lemma 3.5.2 and 3.5.3). It is also true that $| E'' | \leq | E'_e \cup E'_c |$.

Lemma 3.5.2 Assume case 3.5.1 applies. Let (V'', E''_e) be a subgraph of (V'', E'') that is 2-connected. Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x, y) \mid (x, y) \in E''_e$ and (x, y) is a regular edge $\}))$ is also 2-connected.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x, y) \mid (x, y) \in E''_e$ and (x, y) is a regular edge $\}$. Assume that

$(V', E'_e \cup E''_c)$ is not 2-connected. Let v be an articulation point in $(V', E'_e \cup E''_c)$. If v is in a chain $v_1, ..., v_p$, then both v_1 and v_p are in V'' and they are articulation points in (V'', E''_e) . (V'', E''_e) is not 2-connected. If $deg(v) > 2$, then $v \in V''$ and v is an articulation point in (V'', E''_e). (V'', E''_e) is not 2-connected. This means that $(V', E'_e \cup E''_c)$ is 2-connected. ∎

Lemma 3.5.3 Assume case 3.5.1 applies. Let (V'', E''_e) be a minimal 2-connected subgraph of (V'', E''). Then $(V', E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in E'' $\}))$ is a minimal 2-connected subgraph of $(V', E'_e \cup E'_c)$.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in E'' $\}$. From Lemma 3.5.2 we know that $(V', E'_e \cup E''_c)$ is 2-connected. All we have to prove is that every edge in E''_c is 2-essential for $(V', E'_e \cup E''_c)$. Let (v,w) be an edge in E''_c. (v,w) is also a regular edge in E'' and $(v,w) \in E''_e$. If $(V', (E'_e \cup E''_c) - \{(v,w)\})$ is still 2-connected, then $(V'', E''_e - \{(v,w)\})$ is also 2-connected. (V'', E''_e) is not a minimal 2-connected subgraph of (V'', E''). This is a contradiction. ∎

After we construct (V'', E''), we do not wish to keep (V', E'). We do not wish to compute E''_c after we compute (V'', E''_e). As a matter of fact, E''_c can be decided while we are computing (V'', E''_e). For each edge (x,y) in E', we associate a field *source* with it. Before the first recursive call of the algorithm, we let $(x,y).source = (x,y)$ for every edge in E. When we construct (V'', E'') from (V', E'), we let $(x,y).source = E''^{-1}(x,y).source$ for every regular edge $(x,y) \in E''$ such that $E''^{-1}(x,y) \in E'_c$ and $(x,y).source = null$ for all other edges in E''. Before the first recursive call, we let $E_{fe} = \phi$. When we compute a minimal 2-connected subgraph of (V', E'), if we add an edge $(x,y) \in E'$ to E'_e, we add $(x,y).source$ to E_{fe} if $(x,y).source$ is not null. After the algorithm terminates, (V, E_{fe}) is a minimal 2-connected subgraph of (V, E).

Case 3.5.2 $\mid E'_e \cup E'_c \mid > 4n'/3$.

If case 3.5.2 applies, we know that $\mid E'_e \mid > 23n'/18$. Let $(V_1, E_1), ..., (V_p, E_p)$ be the 2-*connected components* of (V', E'_e). We have $\cup_{i=1}^{p} V_i = V'$ and $V_i \cap V_j = \phi$ for $1 \leq i \neq j \leq p$.

Lemma 3.5.4 If condition 3.5.2 applies, there are at least $5n'/18$ vertices in (V', E'_e) that are not *articulation points.*

Proof: If case 3.5.2 applies, we have $\mid E'_e \mid > 4n'/3 - n'/18 = 23n'/18$. Let (V', E'_T) be a spanning forest of (V', E'_e). We have $\mid E'_e - E'_T \mid > 5n'/18$. Every edge in E'_T is in a 2-connected component by itself in (V', E'_T). We add the edges in $E'_e - E'_T$ to E'_T one by one. When we add (x, y) to E'_T, two 2-connected components in (V', E'_T) merge. The number of 2-connected components in (V', E'_T) is decreased by at least one. This is true since every edge in E'_e is 2-essential for $(V', E'_e \cup E'_c)$. After we add all the edges in $E'_e - E'_T$ to E'_T, (V', E'_T) has less than $13n'/18$ 2-connected components. This proves that (V', E'_e) has less than $13n'/18$ 2-connected components. The number of articulation points in (V', E'_e) cannot exceed the number of its 2-connected components.(See Lemma 3.5.4'.) This proves the lemma. ∎

Lemma 3.5.4' Let $G_l = (V_l, E_l)$ be a graph. The number of articulation points in G_l cannot exceed the number of its 2-connected components.

Proof: Let the 2-connected components of G_l be $C_1 = (V_1, E_1), ..., C_p = (V_p, E_p)$. Let the articulation points of G_l be $v_1, ..., v_q$. We build an auxiliary graph (V_{au}, E_{au}) such that (1) $V_{au} = \{C_1, ..., C_p\} \cup \{v_1, ..., v_q\}$, and (2) $E_{au} = \{(C_i, v_j) \mid v_j \in V_i)\}$. It is not hard to see that (1) (V_{au}, E_{au}) is a forest, and (2) vertex $v_j (1 \leq j \leq q) \in V_{au}$ cannot be a leaf or an isolation point in (V_{au}, E_{au}). From these two facts we can conclude that $\mid \{C_1, ..., C_p\} \mid \geq \mid \{v_1, ..., v_q\} \mid$. This proves the lemma. ∎

Lemma 3.5.5 If case 3.5.2 applies, there are at least $n'/6$ vertices in $(V', E'_e \cup E'_c)$ that are not articulation points in (V', E'_e) and are not incident on an edge in E'_c.

Proof: If case 3.5.2 applies, we know that $\mid E'_c \mid \leq n'/18$. From Lemma 3.5.3, we have at least $5n'/18$ vertices that are not articulation points in (V', E'_e). Since we have $\mid E'_c \mid \leq n'/18$, there are at least $5n'/18 - 2n'/18 = n'/6$ such vertices that are not incident on an edge in E'_c. ∎

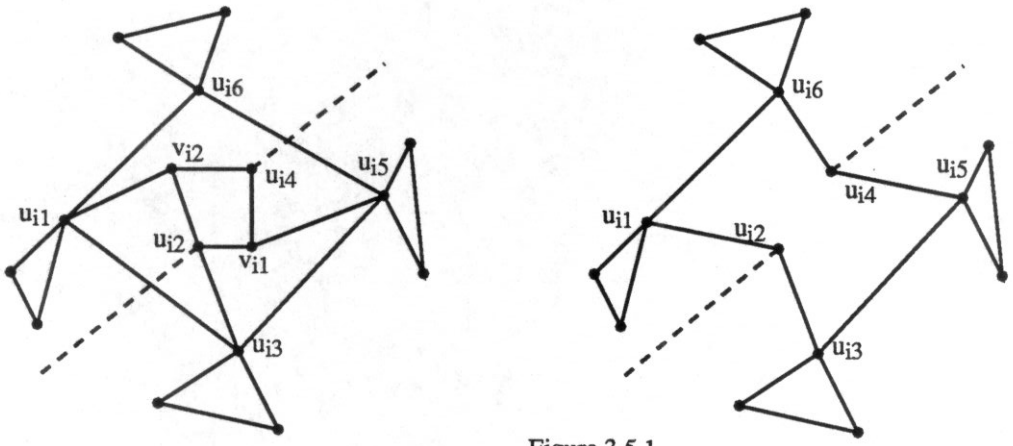

Figure 3.5.1

If case 3.5.2 applies, we construct (V'', E'') in the following way:

(1) Let $V'' = \phi$ and $E'' = \phi$.

(2) For $1 \leq i \leq p$, Let $V_i = \{v_{i1}, ..., v_{ia}\} \cup \{u_{i1}, ..., u_{ib}\}$ (V_i is a 2-connected component of (V', E'_e)) where v_{ij} $(1 \leq j \leq a)$ is a vertex in V_i that is not an articulation point in (V', E'_e) and is not incident on an edge in E'_c. Let $V'' = V'' \cup \{u_{i1}, ..., u_{ib}\}$. Let $E'' = E'' \cup \{(u_{i1}, u_{i2}), (u_{i2}, u_{i3}), ..., (u_{ib}, u_{i1})\}$. Edges $(u_{i1}, u_{i2}), ..., (u_{ib}, u_{i1})\}$ are called special edges in E''. We let the *source* field of the special edges be *null*.

(3) For each edge (x,y) in E'_c, both x and y are in V''. Let $E'' = E'' \cup \{(x,y)\}$. Edge (x,y) is called a regular edge in E''. Let (x',y') be a regular edge in E''. We shall denote the corresponding edge in E'_c by $E''^{-1}(x',y')$ and let $(x',y').source = E''^{-1}(x'.y').source$.

Lemma 3.5.6 $\mid V'' \mid \leq 5n'/6$ and $\mid E'' \mid \leq \mid E'_e \cup E'_c \mid$.

Proof: This is obvious from the way (V'',E'') is built. ∎

It is not hard to see that (V'',E'') is 2-connected and preserves the 2 connectivity structure of $(V',E'_e \cup E'_c)$. (See also Lemma 3.5.7 and 3.5.8.)

In Figure 3.5.1, the edges in E'_e are solid and the edges in E'_c are dashed.

Lemma 3.5.7 Assume case 3.5.2 applies. Let (V'',E''_e) be a subgraph of (V'',E'') that is 2-connected. Then $(V',E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in $E''\}))$ is also 2-connected.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in $E''\}$. Assume that $(V',E'_e \cup E''_c)$ is not 2-connected. Let v be an articulation point in $(V',E'_e \cup E''_c)$, then v is an articulation point in (V',E'_e) and we have $v \in V''$. v is an articulation point in (V'',E''_e). (V'',E''_e) is not 2-connected. This is a contradiction. ∎

Lemma 3.5.8 Assume case 3.5.2 applies. Let (V'',E''_e) be a minimal 2-connected subgraph of (V'',E''). Then $(V',E'_e \cup (E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e$ and (x,y) is a regular edge in $E''_e\ \}))$ is a minimal 2-connected subgraph of $(V',E'_e \cup E'_c)$.

Proof: Let $E''_c = E'_c \cap \{E''^{-1}(x,y) \mid (x,y) \in E''_e\ \}$. From Lemma 3.5.7 we know that

$(V', E'_e \cup E''_c)$ is 2-connected. All we have to prove is that every edge in E''_c is 2-essential for $(V', E'_e \cup E''_c)$. Let (v, w) be an edge in E''_c. If $(V', (E'_e \cup E''_c) - (v, w))$ is still 2-connected, then $(V'', E''_e - \{(v, w)\})$ is also 2-connected. (V'', E''_e) is not a minimal 2-connected subgraph of (V'', E''). This is a contradiction. ∎

We conclude this section with a discussion of the second part of each recursive call of algorithm 1.2.2. After we construct (V'', E'') from (V', E'), we compute a minimal 2-connected subgraph of (V'', E'') by using the algorithm recursively. A minimal 2-connected subgraph of (V', E') can be decided while we are computing a minimal 2-connected subgraph of (V'', E'').

§3.6 The Complexity of the Algorithm

The algorithm discussed in §3.3 and 3.4 can be implemented to run in $O(m+n)$ time and space.

There are $O(log(n))$ recursive calls to the algorithm. In each recursive call, we want to compute a minimal 2-connected subgraph of (V', E') with $\mid V' \mid = n'$ and $\mid E' \mid = m'$.

Each recursive call is divided into two parts.

There are twenty-five phases in the first part. In each phase, two procedures are involved. Edge set E'_{sc} is first computed and then processed. (See §2.3 and 2.4.) In computing E'_{sc}, (see also procedure 2.3.1) spanning tree E'_t can be obtained by computing a minimum spanning tree on a $0-1$ weighted graph. In graph $(V', E_e \cup E'_c)$, we assign a weight to each edge. If $(x,y) \in E'_e$, we let $(x,y).weight = 0$. If $(x,y) \in E'_c$, we let $(x,y).weight = 1$. A minimum spanning tree of $(V', E'_e \cup E'_c)$ can be computed in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. This spanning tree contains as many edges of E'_e as possible. After E'_t is computed, G'_a can be obtained in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. The critical edges in G'_a are easily identified. Edge set E'_{sc} can be computed in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time.

In §2.4, we discuss a procedure to process E'_{sc}. Before the procedure, we compute *high* and *led* for each vertex $v \in V'$. This computation can be done in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time by a preorder traversal of the rooted spanning tree E'_t. (See also [T1].) We also compute the nearest common ancestor of each nontree edge $(x,y) \in E'_e \cup E'_c)$. This can be done in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$. (See [HT5].) The partition of V' and $(E'_e \cup E'_c) - E'_t$ into partition classes can be implemented in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time using bucket

sort on V' and $(E'_e \cup E'_c) - E'_t$ respectively.

The procedure in §3.4 is divided into steps. At the beginning of each step, *slea* can be computed in $O(\mid V_{h_i} \mid)$ time where V_{h_i} is the base partition class of *slea*. Each step is further divided into four substeps. The first two substeps can be implemented in $O(\mid slea \mid)$ time. We also have $\mid slea \mid \leq \mid V_{h_i} \mid$.

Both substep 3 and 4 can be implemented in $O(\mid V_{ag} \mid + \mid E_{ag} \mid)$ time. We also have $\mid V_{ag} \mid \leq \mid V_{h_i} \mid + \mid S \mid$ and $\mid E_{ag} \mid \leq \mid S \mid$.

After the substeps, we remove the vertices from V_{h_i} and the edges incident on a removed vertex from $(V', E'_e \cup E'_c)$. Let the number of removed tree edges be R_t. The removal of vertices and edges can be finished in $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$ time. The time complexity of each step is $O(\mid V_{h_i} \mid + \mid S \mid + R_t)$. After each step, the size of V' decreases by $\mid V_{h_i} \mid$ and the size of $E'_e \cup E'_c$ decreases by $\mid S \mid + R_t$. The total complexity of the procedure is $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$.

In the first part of the algorithm, assume that we have $m' > 2n'$ at the beginning of the first phase. We have $k = n' - 1 \leq 5K/6 = 5m'/6$ when we compute E'_{sc} for this phase. The procedure in §3.4 is used to process the edges in E'_{sc}. After this procedure, we have $\mid E'_e \cup E'_c \mid \leq 2n'$. Thus the first phase takes $O(m' + n')$ time and each of the remaining twenty-four phases takes $O(n')$ time.

In the second part of each recursive call, (V'', E'') can be computed from $(V', E'_e \cup E'_c)$ in $O(\mid V' \mid + \mid E'_e \mid + \mid E'_c \mid)$ time. A minimal 2-edge connected subgraph of $(V', E'_e \cup E'_c)$ can be obtained while we are computing a minimal 2-edge connected subgraph of (V'', E''). (See Lemma 2.5.3, 2.5.7 and the discussion after Lemma 2.5.3.)

The number of vertices decreases by at least one sixth in each recursive call. Thus the total time complexity of the algorithm is $O(m+n)+O((5/6)n)+((5/6)^2n)+...+O(1)=O(m+n)$.

In each recursive call, after we construct (V'', E''), we can release the space used by (V', E'). The space complexity of the algorithm is also $O(m+n)$.

§3.7 Problems for Future Research

The algorithm is optimal to within a constant factor for solving the minimal 2-connected subgraph problem. As we discussed in §1.2, this problem is actually a special case of finding a minimal k-connected subgraph for a k-connected graph. There is a polynomial algorithm to test k-connected graphs (See [ET3]). Combining this algorithm with the simple iterative algorithm presented at the beginning of §3.2, we have a polynomial algorithm solving the minimal k-connected subgraph problem. It would be interesting to know if there exits an more efficient (near linear time) algorithm to solve the minimal k-connected subgraph problem. Another interesting problem is to improve the parallel algorithm in [KV] for finding a minimal 2-connected subgraph of a 2-connected graph.

Chapter Four

An Algorithm to Find Maximal Planar Subgraphs

§4.1 Introduction

Problems of planar graphs are always important in the study of graph theory. A fundamental problem in dealing with planar graphs is the planarity testing problem. The problem was solved satisfactorily by Hopcroft and Tarjan in 1972. They gave the first linear time algorithm for testing the planarity of a graph. (See [HT1].)

Their algorithm starts by finding a simple cycle in a graph. Deleting this cycle breaks the graph into one or more disconnected pieces. The planarity of each piece is tested by using the algorithm recursively. If the pieces are planar, they are put together to see if the whole graph is still planar.

Another solution for planarity testing is due to Lempel, Even and Cederbaum. (See [LEC].) Their algorithm proceeds by adding one vertex to a *planar embedding* at a time. The vertices of the graph are ordered according to *st-numbering*. Because of the properties of this ordering, only local tests need to be done when a vertex is added to the embedding. This algorithm was proven by Even, Tarjan (See [ET1]) and Booth, Leuker (See [BL]) to run in linear-time if implemented appropriately.

In this chapter, we consider a problem that is one step beyond planarity testing. Let $G = (V, E)$ be a nonplanar graph with $\mid V \mid = n$ and $\mid E \mid = m$. We wish to find a minimal subset $E_0 \subseteq E$ such that $(V, E - E_0)$ is planar. Graph $(V, E - E_0)$ is called a maximal planar subgraph of G.

In finding a maximal planar subgraph of G, we prefer a subgraph with as many edges as possible. The problem of finding a maximal planar subgraph of G that has the most edges is NP-complete, however. An interested reader can read [GJ] for details.

The problem of finding a maximal planar subgraphs of a nonplanar graph has been considered by many authors; see §1.2 for a brief discussion of previous results on the problem. In this chapter, we develop an $O(m + n \cdot log(n))$ algorithm to solve this problem.

§4.2 Preliminary Results

The algorithm described in this chapter is based on the linear-time planarity testing algorithm of Hopcroft and Tarjan. (See [HT1].) This section presents some preliminary results for the planarity testing algorithm and the problem of finding a maximal planar subgraph.

Let $G = (V, E)$ be an undirected graph with $\mid V \mid = n$ and $\mid E \mid = m$. We can draw a picture of G in the plane as follows: for each vertex $v \in V$, we draw a distinct point v'; for each edge $(u, v) \in E$, we draw an arc connecting u' and v'. We call this arc an *embedding* of edge (u, v). If the points and the arcs can be drawn in such a way that no arcs cross each other, we say G is a *planar graph*, and the drawing is a *planar embedding* of G.

A depth-first search will convert G into a directed graph $G = (V, T \cup B)$, where T is the set of tree edges, and B is the set of back edges. After the depth-first search, the vertices of G are numbered by a preorder traversal of T. We will refer the vertices by their preorder numbers. Graph G is assumed to be a digraph in the following discussions.

We assume that G is 2-connected. If G is not 2-connected, we can find a maximal planar subgraph for each 2-connected component of G and put these subgraphs together to obtain a maximal planar subgraph of G. Since G is 2-connected, there is a unique tree edge e_0 in T that exits the root of T.

For any edge $e =< a, b > \in E$, we associate a segment $S(e)$ with e as follows: if $e \in B$, then $S(e)$ is e itself; if $e \in T$, then $S(e)$ is the subgraph of G that consists of e, the subtree T_e of T rooted at b, and all the back edges that emanate from vertices of T_e.

For each vertex $v \in V$, let $S_v = \{y \in V \mid \exists x, x$ is a descendent of $v, (x,y) \in B$ and y is an ancestor of $v\}$.

Definition 4.2.1 $low_1(e) = b$ if $e \in B$

$$min\{S_b \cup \{n+1\}\} \text{ if } e \in T$$

Definition 4.2.2 $low_2(e) = n+1$ if $e \in B$

$$min\{S_b - \{low_1(e)\}\} \cup \{n+1\}\} \text{ if } e \in T$$

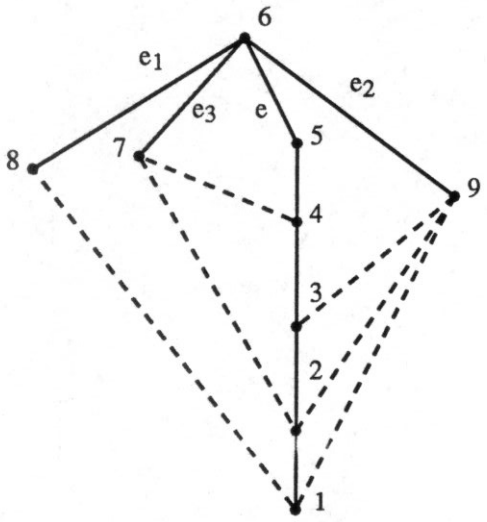

Figure 4.2.1

These two functions low_1 and low_2 can be computed in $O(m)$ time during the depth-first search. (See [HT1].)

We define function ϕ on every edge $e =< a, b >$ of E as follows:

Definition 4.2.3 $\phi(e) = 2b$ if $e \in B$

$$2low_1(e) \text{ if } e \in T \text{ and } low_2(e) \geq a$$

$$2low_1(e) + 1 \text{ if } e \in T \text{ and } low_2(e) \leq a$$

We use $A(e)$ to denote the set of back edges in $S(e)$ that go into proper ancestors of a. Each back edge in $A(e)$ is called an *attachment* of e.

Let $e' =< u, v >$ be an attachment of e. Then $low_1(e) \leq v < a$. If $low_1(e) < v$, then we say that e' is *normal*. Otherwise we say that e' is *special*.

The following lemma follows directly from the definition of ϕ:

Lemma 4.2.1 Let e_i and e_j be any two edges leaving b. We have

1. If $low_1(e_i) < low_1(e_j)$, then $\phi(e_i) < \phi(e_j)$;

2. $low_2(e_i) \geq b$ iff $\mid \{y \mid < x, y > \in A(e_i)\} \mid \leq 1$;

3. If $low_1(e_i) = low_1(e_j)$, then $\phi(e_i) < \phi(e_j)$ iff $low_2(e_i) \geq b$ and $low_2(e_j) < b$.

Proof: See [HT1]. ∎

Let $L(e) = [e_1, ..., e_k]$ be a list of edges leaving b in an order such that for all $i, j \in \{1..k\}, i < j$ implies $\phi(e_i) \leq \phi(e_j)$. We call $e_1, ..., e_k$ the *successors* of e. We can compute $L(e)$ for all $e \in T$ in $O(m)$ time using a bucket sort. (See [HT1].) We define $cycle(e)$ as follows: if e is a back edge, $cycle(e) = e$ + the tree path from b to a, otherwise $cycle(e) = cycle(e_1)$. It is not difficult to see that node $low_1(e)$ is always on $cycle(e)$. We use $sub(e)$ to denote subgraph $S(e) + cycle(e)$. Edge e is said to be *planar* if $sub(e)$ is planar.

In Figure 4.2.1, the tree edges are solid while the back edges are dashed. Let $e =<$

$5,6>$. We have $L(e) = \{<6,8>,<6,9>,<6,10>\}$; $sub(e)$ is the whole graph; $S(e)$ contains all edges in $sub(e)$ except $<1,2>$, $<2,3>$, $<3,4>$, $<4,5>$.

The following facts are important to our discussion:

Fact 4.2.1 Let C be a simple cycle in the plane; let a be a point inside C and b a point outside C. Then any curve that joins a and b will cross C. See Figure 4.2.2.

Fact 4.2.2 Let G_1 be the undirected graph on the right side of Figure 4.2.2. In any embedding of G_1, all the edges of path P_2 must be on the same side of cycle C.

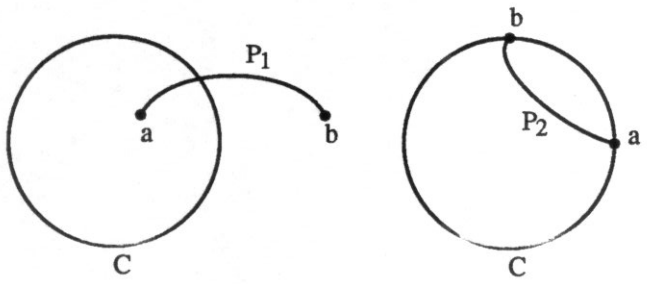

Figure 4.2.2

Fact 4.2.3 Let G_2 be the undirected graph on the left side of Figure 4.2.3. In any embedding of G_2, the two paths P_1 and P_2 must be on different sides of cycle C. See Figure 4.2.3.

Fact 4.2.4 Let G_3 be the undirected graph on the right side of Figure 4.2.3. In any embedding of G_3, the two subgraphs P_1 and P_2 must be on different sides of cycle C. See Figure 4.2.3.

All the facts are intuitively obvious but can be proved from the Jordan curve theorem.

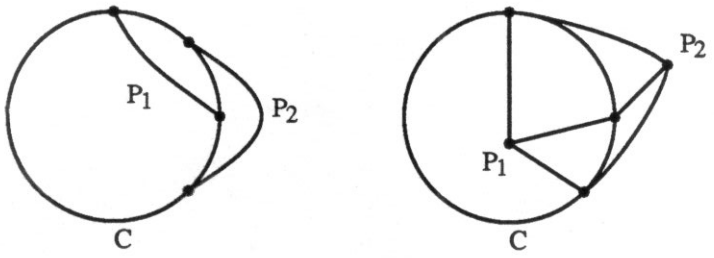

Figure 4.2.3

Let $e =< a, b >$ be a tree edge. The following problem is basic in planarity testing: given that all the edges leaving b are planar, how can we determine the planarity of e?

Suppose e is planar. We divide $A(e)$ into blocks. Two attachments of e are in the same block if and only if they are embedded on the same side of $cycle(e)$ in all embeddings of $sub(e)$.

Let B_1 and B_2 be two blocks of $A(e)$. Let $S_1 \subseteq B_1$ and $S_2 \subseteq B_2$. S_1 and S_2 are said to *interlace* if they cannot be on the same side of $cycle(e)$ in any embedding of $sub(e)$.

If $A(e) \neq \phi$, then the only attachment of e that is on $cycle(e)$ forms a block by itself, and this block does not interlace with any other block of e.

Lemma 4.2.2 If e is planar, each block of $A(e)$ can interlace with at most one block in $A(e)$. (See [HT1].) ∎

We will represent a block of attachments $H = \{< b_1, a_1 >, ..., < b_l, a_l >\}$ by a list $K = [a_1, a_2, ..., a_l]$, where we assume that $a_1 \leq a_2, ..., \leq a_l$. Repeated elements in K can be omitted. Frequently, we will identify blocks with their list representations. Define $first(H) = first(K) = a_1$ and $last(H) = last(K) = a_l$. If K is empty, we define $first(H) = first(K) = n + 1$, and $last(H) = last(K) = 0$. If H contains any normal attachment of e, then we say that H and K are *normal.* Otherwise, we say that they are *special.* We say that e is ***strongly planar*** if e is planar and all the normal blocks of $A(e)$ do not interlace with each other.

We organize the blocks of $A(e)$ as follows: if two blocks K_1 and K_2 interlace, we put them into a pair $[K_1, K_2]$. We assume that $last(K_1) \leq last(K_2)$. If a block K does not interlace with any other block, we form a pair $[[], K]$. Let $[A_1, A_2]$ and $[B_1, B_2]$ be two such pairs. We say $[A_1, A_2] \leq [B_1, B_2]$ if $last(A_2) \leq min\ (first(B_1),\ first(B_2))$.

Assume that e is planar. Let $[A_1, A_2], [B_1, B_2]$ be two pairs of blocks of $A(e)$. Then either $[A_1, A_2] \leq [B_1, B_2]$ or vice versa.

Let $S = [p_1, ..., p_m]$ be the set of pairs of blocks of $A(e)$. There is a linear ordering on S, say, $p_1 \leq p_2 ... \leq p_m$. We represent $A(e)$ by $att(e) = [p_1, ..., p_m]$. We call $att(e)$ the list representation of $A(e)$.

List $att(e)$ can be computed inductively as follows:

If $e =< a, b >$ is a back edge, its only attachment is e itself. Therefore $att(e) = [[], [b]]$

Now assume that $e =< a, b > \in T$, and $att(e_i)$ has been computed for each e_i emanating from b. We compute $att(e)$ in four steps.

Step 4.2.1 For each $i = 2, ..., k$ merge all the blocks of $att(e_i)$ into one intermediate block B_i.

According to fact 4.2.2, every edge in $sub(e_i)$ must be embedded on the same side of $cycle(e)$. Therefore this step can be done only if the normal blocks of $att(e_i)$ do not interlace with each other (i.e. e_i is strongly planar). Note that there is no restriction on the interlacing of special blocks. To merge a series of blocks, simply concatenate their ordered list representations. The ordering of the blocks is maintained. See Figure 4.2.4.

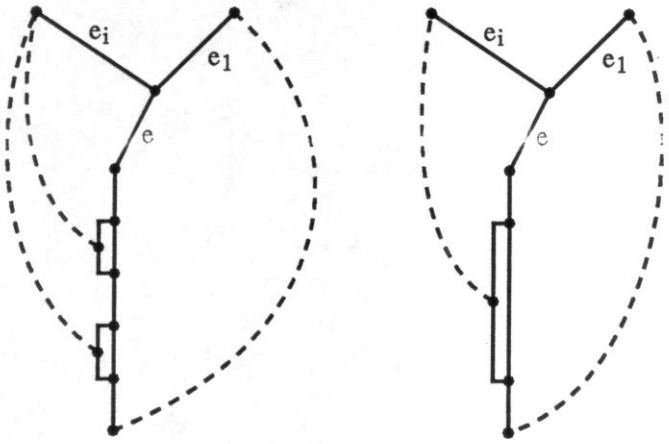

Figure 4.2.4

Step 4.2.2 All blocks D in $att(e_1)$ that have $last(D) > low_1(e_2)$ must be merged into one block B_1. See Figure 4.2.5.

The merge can be done from the high end of the list of pairs in $att(e_1)$. Now $att(e_1)$ is changed to a list of pairs $p_1 \leq ... \leq p_h$ with only p_h possibly interlacing B_2. Take this list to be the initial value of $att(e)$.

Step 4.2.3 Merge blocks $B_2, ..., B_k$ into $att(e)$. This is done as follows:

Let $[X,Y]$ be the last pair in $att(e)$, i.e., $[X,Y] \geq [X',Y']$ for any $[X',Y'] \in att(e)$. We process $B_2, ..., B_k$ one by one. To process B_i, we have three subcases:

(1) If B_i interlaces with both X and Y, e is not planar.

(2) If B_i interlaces with only Y, then merge X and B_i.

(3) If B_i interlaces with neither X nor Y, then $P = [[], B_i]$ and add P to $att(e)$.

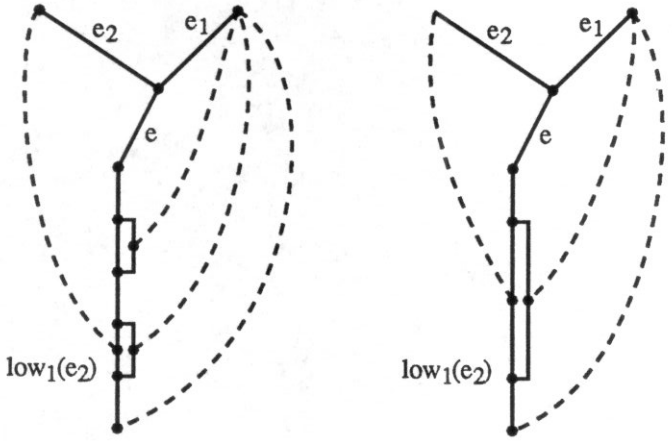

Figure 4.2.5

Step 4.2.4 Delete all instances of a from $att(e)$.

Since G is 2-connected, there is only one edge in T that leaves the root of T. Let this edge be e_0. The planarity of G can be decided by computing the planarity of e_0.

§4.3 An Algorithm for Finding a Maximal Planar Subgraph

This section gives an algorithm that computes a maximal planar subgraph of G when G is not planar. The algorithm proceeds by finding a minimal subset $E_0 \subseteq B$ such that graph $(V, (T \cup B) - E_0)$ is planar. In the algorithm, when we talk about deleting an edge, we mean to put the edge in E_0.

We will assume that G is 2-connected. When G is not 2-connected, we can find a maximal planar subgraph for each 2-connected component of G and put these subgraphs together to form a maximal planar subgraph of G. Since G is 2-connected, there is only one tree edge leaving the root of T. Let this edge be e_0.

Let $e =< a, b >$ be a tree edge in G. Let $e_1, ..., e_k$ be the successors of e. In the maximal planar subgraph algorithm, certain back edges are deleted. This means that the low_1 values of $e_1, ..., e_k$ are subject to change. In the planarity testing algorithm, $e_1, ..., e_k$ are ordered on their ϕ values. It is not convenient to maintain this ordering dynamically. For the purpose of finding a maximal planar subgraph, we modify the algorithm so that $e_1, ..., e_k$ are ordered on their low_1 values. A ϕ ordering implies a low_1 ordering, but not vice versa. We redefine $L(e) = [e_1, ..., e_k]$ to be the successors of e in increasing order of their low_1 values.

Just as the concept of a strongly planar graph is important to the planarity testing algorithm, the concept of a *l-planar* graph is important to the problem of finding maximal planar subgraphs.

An attachment (u, v) of e is *l-normal* if $v \in \{low_1(e) + 1, \ldots, i - 1\}$. A subset D of $A(e)$ is l-normal if D contains a l-normal attachment. A block of $att(e)$ is *l-normal* if it

contains an element $v \in \{low_1(e)+1, ..., l-1\}$.

An edge $e =< a, b >$ is called *l-planar* if e is planar and all the l-normal blocks of $att(e)$ do not interlace with each other. Thus, e is planar iff e is $low_1(e) - planar$, and e is strongly planar iff e is $a - planar$. If we have an algorithm that can compute a maximal l-planar subgraph of $sub(e)$ for any $low_1(e) \leq l \leq a$, then we can compute a maximal planar subgraph of $sub(e)$.

In Figure 4.3.1, the graph on the left is l-planar and the graph on the right is not.

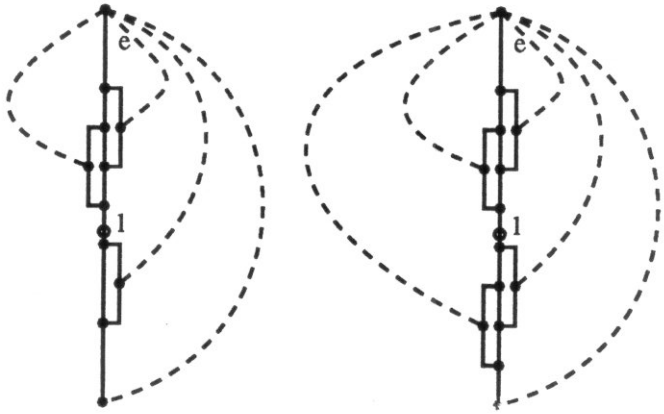

Figure 4.3.1

In the following discussion, we present an procedure to compute a maximal l-planar subgraph $sub_l(e)$ of $sub(e)$. To compute $sub_l(e)$, we try to compute a planar embedding of $sub(e)$. We delete back edges from $sub(e)$ if we cannot proceed with the embedding. The procedure is based on the planarity testing algorithm in §4.2. Again, we describe it inductively.

Assume $e =< a, b >$.

Case 4.3.1 e is a back edge. Assign $[[], [b]]$ to $att(e)$, then return.

Case 4.3.2 e is a tree edge, and e has no successor. Assign [] to $att(e)$, then return.

Case 4.3.3 e is a tree edge and there are k successors $e_1, ..., e_k$ from b. We construct a sequence $G_1, ..., G_k$ of l-planar subgraphs of $sub(e)$ such that G_1 is a maximal l-planar subgraph of $sub(e_1)$ and G_k is a maximal l-planar subgraph of $sub(e)$. Each $G_i (2 < i \leq k)$, is obtained from G_{i-1} by adding to it a subgraph of $sub(e_i)$, where e_i is a successor of e that has the smallest low_1 value and is not contained in G_{i-1}.

In case 4.3.3, the following two steps are involved:

Step 4.3.1 We first mark $e_1, ..., e_k$ unprocessed. Let e_1 be one of the successors of e with the smallest low_1 value. Recursively compute a maximal l-planar subgraph of $sub(e_1)$ and $att(e_1)$. Mark e_1 as processed. Make the resulting planar subgraph of $sub(e_1)$ the initial value of $sub(e)$, and make $att(e_1)$ the initial value of $att(e)$.

Step 4.3.2 While there is a successor of e that is unprocessed and with low_1 value less than or equal to a, (see also Fact 4.3.1.) do the following: Let e_i be an unprocessed successor having the smallest low_1 value. If merging $sub(e_i)$ into $sub(e)$ results in a l-planar subgraph of $sub(e)$, then recursively compute a maximal $b - planar$ (i.e. strongly planar) subgraph of $sub(e_i)$, merge this subgraph into $sub(e)$ as described in the planarity testing algorithm, and mark e_i as processed. If merging $sub(e_i)$ into $sub(e)$ results a non-l-planar subgraph, delete some back edges from $sub(e_i)$ that cause the failure in the merge of $sub(e_i)$ and $sub(e)$ and repeat step 4.3.2.

The following pseudo code outlines the two steps in case 4.3.3.

Procedure 4.3.1

Input: e and $e_1, ..., e_k$

Output: a maximal l-planar subgraph of $sub(e)$

{

Mark $e_1, ..., e_k$ as unprocessed;

Recursively compute a maximal l-planar subgraph of $sub(e_1)$;

Make $sub(e_1)$ the initial value of $sub(e)$;

Make $att(e_1)$ the initial value of $att(e)$;

While there are unprocessed edges

{

l_1 : Let e_i be the unprocessed edge with the smallest low_1 value;

If $(low_1(e_i) \geq l)$ break; * see also Fact 4.3.1 *\

If merging $sub(e_i)$ to $sub(e)$ results in a non-l-planar subgraph of $sub(e)$

then

{

Delete some back edges from $sub(e_i)$;

Update the low_1 value of e_i;

Goto l_1;

}

Merge $sub(e_i)$ to $sub(e)$;

Mark e_i as processed; }

}

Let $e_0 =< a_0, b_0 >$ be the unique tree edge leaving the root of T. Let $e_{10}, ..., e_{k0}$ be the successors of e_0. In order to compute a maximal planar subgraph of G, all we have to do is to call the procedure above to compute a maximal $low(e_0) - planar$ subgraph of $sub(e_0)$.

The correctness of this algorithm depends on (1) how to test whether the merge of $sub(e_i)$ and $sub(e)$ results in a l-planar subgraph of $sub(e)$, and (2) how to select back edges in $sub(e_i)$ to delete if (1) fails. Before we solve these two problems, we point out the following two facts about the maximal l-planar subgraph algorithm.

Fact 4.3.1 By induction, it is easily seen that when we are at $e =< a, b >$ and processing $e_1, ..., e_k$ leaving b; we always test whether the merge of $sub(e_i)$ and $sub(e)$ results a $a-planar$ subgraph of $sub(e)$. To put it in another way, we always have $l = a$.

Fact 4.3.2 We also notice that when we compute a maximal $b-planar$ subgraph of $sub(e_i)$ recursively, the low_1 value of e_i is going to remain a constant through the rest of the algorithm and at least one back edge in $sub(e_i)$ with low_1 value equal to $low_1(e_i)$ is going to be in the final maximal planar subgraph of G. This point is crucial in proving the correctness of the algorithm.

Let e_j be the last processed edge, $l_i = \max\{last(X) : [X,Y] \in att(e)\}$ after e_j is processed, $l_1 = \max\{last(X) : [X,Y] \in att(e_1)\}$. Let e_i be an unprocessed successor with the smallest low_1 value and $low_1(e_i) < l$. According to what we know about planarity testing, the merge of $sub(e_i)$ and $sub(e)$ results in a non l-planar subgraph when one of the following five conditions happens:

Condition 4.3.1 $low_1(e_i) = low_1(e_1)$, $low_1(e_1) < l_1 < l$, and $low_1(e_i) < low_2(e_i) < l$. See Figure 4.3.1.

Condition 4.3.2 $low_1(e_i) = low_1(e_j) = low_1(e_1)$, $low_1(e_1) < l_i < l$, and $low_1(e_i) < low_2(e_i) < l$. See Figure 4.3.1.

Condition 4.3.3 $low_1(e_i) = low_1(e_1)$, $low_1(e_i) < low_2(e_i) \leq l$, and there exist previously processed edges e_{j_1}, e_{j_2} such that $low_1(e_{j_1}) = low_1(e_{j_2}) = low_1(e_1)$, $low_2(e_{j_1}) \leq l$ after $sub(e_{j_1})$ is added to $att(e)$ and $low_2(e_{j_2}) = l$ after $sub(e_{j_2})$ is added to $att(e)$. See Figure 4.3.2.

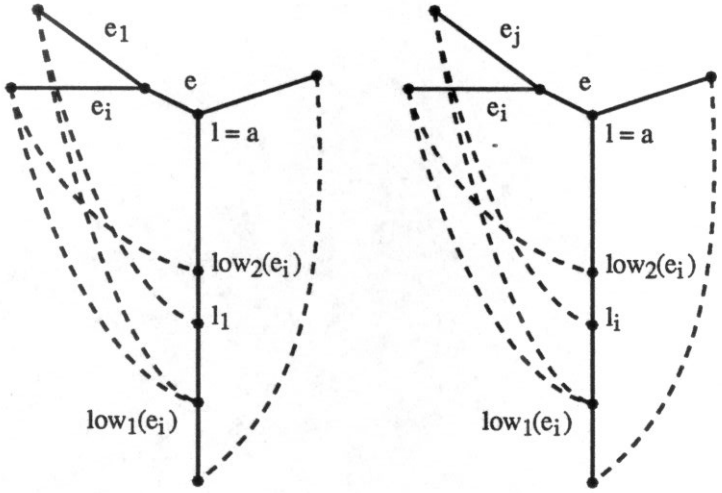

Figure 4.3.2

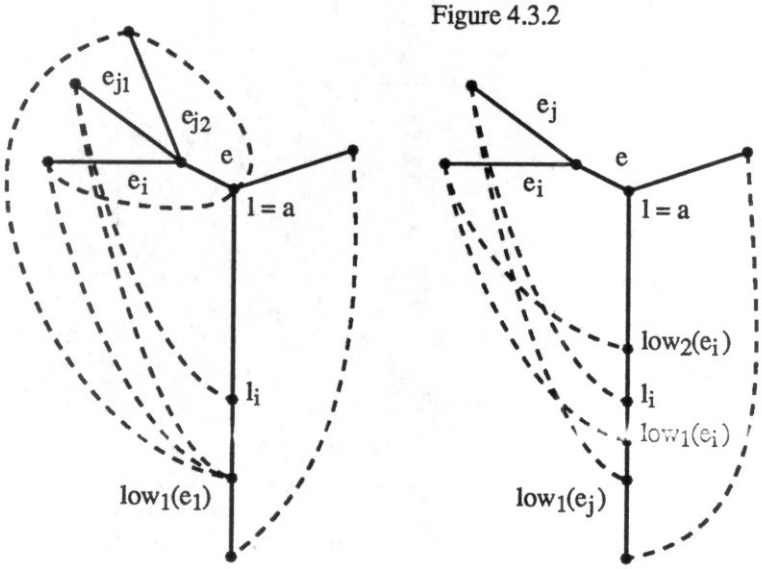

Figure 4.3.3

Condition 4.3.4 $low_1(e_i) > low_1(e_j) \geq low_1(e_1)$, and $low_1(e_i) < l_i$. See Figure 4.3.2.

Condition 4.3.5 $low_1(e_i) = low_1(e_j) > low_1(e_1)$, $l_i \leq l$, and $low_1(e_i) < low_2(e_i) \leq l$. See

Figure 4.3.3.

Conditions 4.3.1, 4.3.2, 4.3.4, and 4.3.5 are very easy to test. To test condition 4.3.3, we keep two flags $e.flag_1$ and $e.flag_2$ with e in the algorithm. We have $e.flag_1 = 0$ and $e.flag_2 = 0$ initially. Whenever $sub(e_j)$ is added to $att(e)$ after we process e_j, if $low_1(e_j) = low_1(e_1)$ and $low_2(e_j) \leq l$ we set $e.flag_1 = 1$. After $e.flag_1$ is set, if there is another $sub(e_j)$ that satisfing the same condition, we set $e.flag_2 =$. We can test condition 4.3.3 in $O(1)$ time by using $e.flag_1$ and $e.flag_2$.

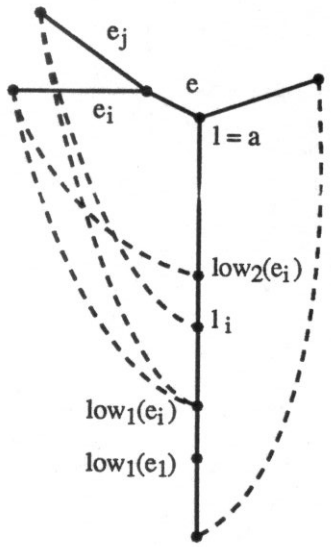

Figure 4.3.4

If condition 4.3.4 is true, we can make it false only by changing the low_1 value of e_i. In this case, the back edge selected for deletion is a back edge (u, v) of $sub(e_i)$ with $v = low_1(e_i)$. If any of the conditions 4.3.1, 4.3.2, 4.3.3, 4.3.5 is true, we can make it false by changing the value of either $low_1(e_i)$ or $low_2(e_i)$. If we choose to change $low_1(e_i)$, it may happen that all the attachments in $A(e_i)$ are deleted when we compute a maximal $b - planar$ subgraph of $sub(e_i)$. This may result a subgraph that is not maximal l-planar since we can add back a back edge that was deleted before the recursive computation for $sub(e_i)$. We select one of the back edges (u, v) of $sub(e_i)$ with $v = low_2(e_i)$ for deletion in this case. As we noted in fact 4.3.2, at least one back edge in $sub(e_i)$ with low_1 value equal

to $low_1(e_i)$ is going to be in $sub(e_i)$ after we compute a maximal $b-planar$ subgraph of $sub(e_i)$. This will ensure that none of the deleted back edges can be added back.

It should be clear that algorithm described above computes an l-planar subgraph of $sub(e_i)$. To see that the resulting subgraph is maximal, we have the following lemma:

Lemma 4.3.1 Procedure 4.3.1 computes a maximal l-planar subgraph of $sub(e)$.

Proof: Lemma 4.3.1 is true because of the following two facts: (1) our algorithm never deletes any edge in $sub(e_i)$ once e_i is processed. (2) when the algorithm is recursively applied to e_i, $low_1(e_i)$ will not change. If any of the edges deleted by the above strategy is added back to $sub(e_j)$, one of the five conditions will be true again, and $sub(e)$ would not be l-planar. ∎

§4.4 Data Structures and the Time Complexity

In the algorithm described in §4.3, we need to repeatedly select an unprocessed successor of e with the smallest low_1 value, and the low_1 values of tree edges are constantly changing. Therefore, we maintain a heap (see [T2]) on low_1 values of the unprocessed successors of the tree edge e currently being processed. Since the algorithm is recursive, we actually maintain simultaneously a heap of unprocessed successors for each tree edge along the path of the currently active tree edge. The total size of these heaps is $O(m)$. The initialization of all the heaps takes a total of $O(m)$ time. When the low_1 value of some element in a heap increases, we modify the heap accordingly. It is important to notice that any two edges in active heaps are unrelated; thus the deletion of a single attachment can modify the low_1 value of only a single such edge. It follows that the total number of modifications and deletions is $O(m)$. The time for the heap operations is $O(log(n))$ time per operation, for a total of $O(m \cdot log(n))$ time.

We also need a data structure for the back edges of $sub(e)$ so that the following operations can be performed efficiently:

(1) Delete one of the attachments $< u, v >$ of e with $v \in \{low_1(e), low_2(e)\}$.

(2) Modify the low_1 and low_2 values of e.

(3) Split the data structure into several pieces, one for each successor of e.

One solution that meets these requirements is the selection tree data structure. (See [HS].) A selection tree is a binary tree. We store values in the leaves of the selection tree. In each internal node v, we store the smaller of the values stored in the children of v. We

will modify the selection tree to fit our needs.

To store a set of back edges B_0 in a selection tree T_0, we store the back edges in the leaves of T_0 from left to right in increasing order (by preorder numbering) of their tails. The back edges with the same tail are ordered arbitrarily. For each internal node v in T_0, let T_v be the subtree of T_0 rooted at v. Let S be the set of back edges stored in the leaves of T_v. Let $l = \min \{x \mid < x, y > \in S\}$ and $r = \max\{x \mid < x, y > \in S\}$. Let $low_1 = \min \{y \mid < x, y > \in S\}$, and $low_2 = \min\{y \mid < x, y > \in S, y \neq low_1\}$. Four values l, r, low_1, low_2 are stored in v in fields $v.l, v.r, v.low_1, v.low_2$. These values can be computed inductively by doing a postorder traversal on T_0. There are also two fields $v.lchild$ and $v.rchild$ in v pointing to the children of v.

In the following discussion, we will refer to a tree by its root. Let r_1 and r_2 be two selection trees representing two disjoint sets of back edges E_1 and E_2. If $u_1 \leq u_2$ for all $< u_1, v_1 > \in E_1$ and $< u_2, v_2 > \in E_2$, we can merge r_1 and r_2 to get a selection tree for $E_1 \cup E_2$ in $O(1)$ time.

Procedure 4.4.1 $merge(r_1, r_2)$

{
If $r_1 = null$ then
Return r_2;
If $r_2 = null$ then
Return r_1;
$r = newnode()$;
$r.lchild = r_1$;
$r.rchild = r_2$;
$r.l = r_1.l$;

$r.r = r_2.r;$

$r.low_1 = min(r_1.low_1, r_2.low_1)$

$r.low_2 = min(\{r_1.low_1, r_1.low_2, r_2.low_1, r_2.low_2\} - \{r.low_1\})$

Return r;

}

Let r be a selection tree representing a set E_0 of back edges. To split E_0 into two sets $E_1 = \{< u, v >\in E_0 \mid u \leq u_x\}$ and $E_1 = \{< u, v >\in E_0 \mid u > u_x\}$ with respect to u_x, we have the following procedure.

Procedure 4.4.2 $split(r, u_x)$

{

If $u_x < r.l$ return $(null, r)$;

Else If $u_x > r.r$ return $(r, null)$;

Else

{

Let $(r_l, r_r) = (r.lchild, r.rchild)$;

If $u_x < r_l.r$

{

$(r_{l1}, r_{l2}) = split(r_l, u_x)$;

Return $(r_{l1}, merge(r_{l2}, r_r))$;

}

Else

{

$(r_{r1}, r_{r2}) = split(r_r, u_x)$;

Return $(merge(r_l, r_{r1}), r_{r2})$;

}

}

}

Let the height of r be R. The height of the trees returned by $split(r, u_x)$ cannot be greater than R. The complexity of the procedure is $O(R)$.

To select and delete an edge $< x, v >$ from a tree r, where $v \in \{r.low_1, r.low_2\}$, we have the following procedure.

Procedure 4.4.3 $delete(r, v)$

{

If r is a leaf and the edge stored in r is $< x, v >$

{

Mark $< x, v >$ as deleted;

Return ϕ;

}

Else

{

$(r_l, r_r) = (r.lchild, r.rchild)$;

If $v = r_l.low_1$ or $v = r_l.low_2$ Return $merge(delete(r_l, v), r_r)$;

Else Return $merge(delete(r_l, v), r_r)$;

}

}

At the beginning of the algorithm, we construct a balanced selection tree $tree(e_0)$ to store all the back edges in B where e_0 is the only tree edge leaving the root of T. The

height of $tree(e_0)$ is $O(log(n))$. The time and space needed to construct $tree(e_0)$ are both $O(m)$.

Each time we begin to process an edge e, we use Procedure 4.4.2 to split $tree(e)$ into several pieces $tree(e_1), ..., tree(e_k)$, where $e_1, ..., e_k$ are the successors of e. For each e_i, $tree(e_i)$ is a selection tree representing the set of back edges in $sub(e_i)$. Tree $tree(e_i)$ is obtained as follows. If e_i is a back edge, $tree(e_i)$ can be constructed from the definition. If e_i is a tree edge, let $e_i =< b, b_i >$. Let n_i be the number of descendants of b_i. It is well known that a back edge $< u, v >$ is in $sub(e_i)$ iff $b_i \leq u < b_i + n_i$. $tree(e_1), ..., tree(e_k)$ can be obtained by using the split procedure successively. Each call of the split procedure takes $O(log(n))$ time. There are $O(m)$ splits for the whole algorithm. After each split, the total size of the trees is still $O(m)$.

We use a procedure $delete(r, v')$ to delete one of the back edges $< u, v >$ stored in the leaves of selection tree r with $v = v'$, where $v' \in \{r.low_1, r.low_2\}$. Every selection tree has height $O(log(n))$. This operation can be done in $O(log(n))$ time. There are $O(m)$ delete operations for the whole algorithm. The total cost for *delete* is thus $O(m \cdot log(n))$.

The other costs of the algorithm are the same as in the planarity testing algorithm. The total time complexity of the algorithm is $O(m \cdot log(n))$.

§4.5 Problems for Future Research

Unlike the algorithm for finding minimal 2-connected subgraphs, there is a gap between the upper bound of the algorithm in this chapter and the trivial linear lower bound for the maximal planar subgraph problem. It is possible that the $O(m \cdot log(n))$ upper bound can be improved.

Another interesting problem is to design an efficient NC algorithm for the maximal planar subgraph problem. There is an NC algorithm that solves the planarity testing problem. (See [KR2].) This algorithm is based on the [LEC] planarity testing algorithm and the $P-Q$ tree data structure. It is not known how to parallelize the [HT] algorithm since the parallel computation of a depth first search tree is not known. In [BT], the dynamic planarity testing algorithm is also based on the [LEC] algorithm. It is conceivable that there is an NC algorithm for the maximal planar subgraph problem by combining the results in [BT] and [KR2].

Chapter Five

A Parallel Algorithm for Comparability Graphs

§5.1 Introduction

In this chapter, we present a fast parallel algorithm for transitively orienting the edges of a comparability graph. Using this algorithm, we give efficient algorithms for many problems on special perfect graphs.

An undirected graph $G = (V, E)$ is a comparability graph if the edges of G can be directed in such a way that the resulting digraph G' is transitive; i.e., if both $a \rightarrow b$ and $b \rightarrow c$ are in G', then $a \rightarrow c$ is in G'.

The main result of this chapter is to give efficient parallel algorithms for:

(1) Testing the comparability of an undirected graph.

(2) If a graph is a comparability graph, find a transitive orientation for the edges.

Comparability graphs are important in the theory of perfect graphs. Many special perfect graphs, such as interval graphs, permutation graphs, and threshold graphs can be totally or partially characterized by comparability graphs. (See also [G].)

Let us recall the definition of a perfect graph.

(1) Let $K(G)$ be the size of a *maximum clique* of G.

(2) Let $C(G)$ be the fewest number of colors needed to properly *color* the vertices of G.

(3) Let $I(G)$ be the size of a *maximum independent set* of G. (See Definition 30 in the

appendix.)

(4) Let $W(G)$ be the fewest number of cliques needed to *cover* the vertices of G.

G is called a perfect graph if (1)$K(G_A) = C(G_A)$ for all A, A a subset of V, and (2) $I(G_A) = W(G_A)$ for all A, A a subset of V, where G_A is the subgraph of G *induced* (see Definition 5 in the appendix) by A.

As a matter of fact, (1) implies (2) and (2) implies (1) as the "perfect graph" theorem states. (See [L].) Thus G is perfect iff (1) or (2) holds.

It is well known that finding $K(G), C(G), I(G)$, and $W(G)$ for an arbitrary graph G is NP-complete. However, there are polynomial algorithms for computing these numbers if G is known to be perfect. It would be interesting to ask whether there exist fast parallel algorithms to find these numbers for perfect graphs. This question is unanswered for general perfect graphs. There are efficient parallel algorithms for special cases such as the fast parallel algorithms for computing $K(G), C(G), I(G)$, and $W(G)$ for chordal graphs . (See [NNS].) For comparability graphs, due to their special structure, $K(G)$ and $C(G)$ can be computed easily once we know a transitive orientation of G. $I(G)$ and $W(G)$ seem harder to compute. They are related to some flow problems for which no efficient parallel algorithm is known. Since interval graphs and permutation graphs can be characterized by comparability graphs, the parallel transitive orientation algorithm can be applied to these graphs to give parallel algorithms to compute all or some of the $K(G), C(G), I(G)$, and $W(G)$ values.

The parallel machine model we use in this paper is the $PRAM$. We use a variation known as the $CRCW\ PRAM$, in which concurrent reads and writes to the same memory

location are allowed. Under this machine model, the algorithm that transitively orients the edges of a comparability graph takes $O(log^4(n))$ time using $O(n^3)$ processors. That is, the algorithm is in NC^4.

There are two previous parallel algorithms to transitively orient the edges of a comparability graph. (See [KVV] and [HM].) The two algorithms use basically the same approach: they both directly parallelize a known sequential algorithm for doing the transitive orientation. The first paper just put the problem in NC but did not give any explicit complexity analysis. The second algorithm is claimed to run in $O(log^3(n))$ time using $O(n^4)$ processors on a $CRCWPRAM$. This seems to be incorrect, however. Using two different parallel maximal independent set algorithms, the algorithm can either run in $O(log^3(n))$ time using $O(n^8)$ processors or in $O(log^5(n))$ time using $O(n^3)$ processors.

The algorithm in this chapter uses results developed for counting the number of transitive orientations of a comparability graph. Proofs of the results are usually omitted. A reader interested in the details can read [G] for reference.

The following discussion is divided into three sections, §5.2 contains the transitive orientation algorithm. §5.3 discusses some applications of the algorithm to interval graphs and permutation graphs. §5.4 describes some future research problems.

§5.2 A Transitive Orientation Algorithm

Let $G = (V, E)$ be an undirected graph. Define $E_d = \{< a, b >, < b, a > | \ (a, b) \in E\}$ where $< a, b >$ and (a, b) are ordered and unordered pairs of a, b respectively.

Define a binary symmetrical relation on E_d as follows:

$< a, b > T < a', b' >$ iff either $a = a'$ and (b, b') is not in E or $b = b'$ and (a, a') is not in E.

We say that (a, b) forces (a', b') whenever $< a, b > T < a', b' >$. The intuition behind this definition is that if $< a, b > T < a', b' >$ and $a = a'$then $(a, b), (a', b')$ can only be directed as $< a, b >< a', b' >$ or $< b, a >< b', a' >$.

Taking the reflexive and transitive closure T^* of T, we obtain an equivalence relation on E_d. This equivalence relation partitions E_d into equivalence classes. Let us call the collection of the classes $b(G)$ and call the classes the color classes.

For every subset A of E_d, we define

(1) $A^{-1} = \{< a, b > | < b, a > \in A\}$

(2) $A^* = A \cup A^{-1}$

We have the following theorems about the color classes:

Theorem 5.2.1 Let A be a color class of G. Then one of the following is true:

(1) $A = A^* = A^{-1}$, or

(2) $A \cap A^{-1} = \phi$, and both A and A^{-1} are transitive. ∎

Theorem 5.2.2 G is transitively orientable iff for every color class $A \in b(G)$, $A \cap A^{-1} = \phi$. ∎

The proofs of theorem 5.2.2 and 5.2.3 can be found in [G].

Theorem 5.2.2 provides a simple way of testing the comparability of a graph.

We build an auxiliary undirected graph $G' = (V', E')$ as follows:

(1) $V' = E_d$.

(2) For each $< a, b >$ and $< c, d >$ in V', $(< a, b >, < c, d >)$ is in E' iff $< a, b > T < c, d >$.

The connected components of G' form the color classes of G. A check is then done on whether there is a component that contains both $< a, b >$ and $< b, a >$ for some (a, b) in E. The algorithm can be implemented in $O(log(n))$ time using $O(n^2)$ processors.

When G is a comparability graph, we need to construct a transitive orientation for G. One may wonder if we can arbitrarily pick A or A^{-1} for each color class, directing the edges accordingly to obtain a transitive orientation. Unfortunately, there is interaction between the color classes; thus the orientation of the color classes cannot be decided independently. Understanding this interaction and making use of it to transitively orient G is the focus of the following discussion.

G is now assumed to be a comparability graph. Let the color classes of G be $A_1, ..., A_k$. Since G is a comparability graph, the color classes of G can be paired up. Without loss of generality, we assume that $A_{k/2+1} = A_1^{-1}, ..., A_k = A_{k/2}^{-1}$. We create a partition $B_1, ..., B_{k/2}$ of E. For each edge $(a.b) \in E$. we put (a, b) in $B_i (1 \leq i \leq k/2)$ if we have $< a, b > \in A_i$ or $< b, a > \in A_i$. Every edge in E is in one and only one B_i. We say that $B_1, ..., B_{k/2}$ are the color classes of graph G and (a, b) is colored by B_i. In the following discussion, the term *color* and *color class* will be used in this new sense on the undirected graph G.

A tricolored triangle in G is a triangle whose edges are colored by three different colors. The tricolored triangles play a central role in describing the interactions of the color classes.

Lemma 5.2.1 If there is a tricolored triangle (a, b, c) in G such that (a, b), (b, c) (c, a) are colored C, A, B respectively, then (1) for every edge (x, y) colored C, (c, x, y) is a tricolor triangle with edges colored by C, A, B, and (2) no edge colored C touches c. ∎

Lemma 5.2.2 If there is a tricolored triangle (a, b, c) in G such that (a, b), (b, c), (c, a) are colored C, A, B respectively. Let v be another vertex such that (v, b) is colored B. Then $(v, c) \in E$ and is colored C. ∎

Lemma 5.2.3 Let (a, b, c) be a tricolored triangle in G such that (a, b), (b, c), (c, a) are colored C, A, B respectively. If (x, y) is colored C and (x, z) is colored B, then $(y, z) \in E$ and is colored A. ∎

A complete subgraph (V_s, S) on $r + 1$ vertices of G is called a simplex of rank r if every edge of S is colored by a different color. A simplex is maximal if it is not properly contained in any larger simplex.

The multiplex generated by a simplex S of rank r is defined to be the undirected subgraph of G that is spanned by the edges in the color classes of S. A multiplex is maximal if it is not properly contained in a larger multiplex.

The importance of the notions of simplex and multiplex lies in following lemmas.

Lemma 5.2.4 Let M be the multiplex generated by a simplex S. M is maximal iff S is maximal. ∎

Lemma 5.2.5 Let M_1 and M_2 be two maximal multiplexes of G. Then either $M_1 = M_2$ or M_1 and M_2 do not share an edge. ∎

Lemma 5.2.6 Let S be a simplex contained in a multiplex M. There exists a maximal simplex S_M generating M that contains S. ∎

Lemma 5.2.7 Each maximal multiplex can be transitively oriented independently to give a transive orientation of G. ∎

Lemma 5.2.8 Let S be a maximal simplex that generates a maximal multiplex M. Every transitive orientation of S uniquely determines a transitive orientation of M. ∎

The proofs of Lemmas 5.2.1-5.2.8 can be found in [G].

According to the above lemmas, an algorithm to find a transitive orientation of G can be divided into four steps:

(1) Find the maximal multiplexes of G.

(2) For each maximal multiplex M, find a maximal simplex S that generates M.

(3) Transitively orient S.

(4) Extend the orientation of S to M.

The steps are discussed more carefully in the following:

Step 5.2.1 Remember that the maximal multiplexes partition the edges of G and they are the union of the color classes. So they also partition the color classes.

Lemma 5.2.9 Color classes A and B are in the same maximal multiplex iff there is a sequence of color classes $A = A_0, A_1, ..., A_k = B$ such that A_i and A_{i+1} color two edges of a tricolored triangle for $0 \leq i < k$.

Proof: (if) Since maximal multiplexes partition the color classes, the relation that two color classes are in the same maximal multiplex is transitive. A tricolored triangle is a simplex of rank 2. Since every simplex is contained in a maximal simplex, the three color classes of a triangle are in the same maximal multiplex.

(only if) There is a maximal simplex in G with two edges (a, b) and (c, d) colored A and B. Either $(a, b), (c, d)$ are two edges of a tricolored triangle or (a, b, d) and (c, b, d) are two tricolored triangles that share an edge and contain (a, b) and (c, d). ∎

In order to find the color classes that are in the same maximal multiplex, we build an auxiliary bipartite graph $G_a = (V_a, E_a)$

(1) $V_a = V_1 \cup V_2$ where $V_1 = \{v \mid v$ is a color class of $G\}$ and $V_2 = \{w \mid w$ is a tricolored triangle of $G\}$.

(2) $E_a = \{(v, w) \mid$ some edge of w is colored by $v\}$.

The color classes of each connected component of G_a are the color classes of a maximal multiplex.

Step 5.2.2 For each maximal multiplex M, let E_M be the set of edges in M, and let $V_M = \{v \mid v \in V$ and $\exists w, (v, w) \in E_M\}$. We arbitrarily pick an edge (a, b) in M. Since (a, b) itself is a simplex of rank 1, it is contained in a maximal simplex S that generates M. (See Lemma 5.2.6.)

Let $M = (V_M, E_M)$.

Define an auxiliary graph associated with $(a, b), G_{(a,b)} = (V_{(a,b)}, E_{(a,b)})$, as follows:

(1) $V_{(a,b)} = \{v \mid v$ is in $V_M, v \neq a, v \neq b, (v, a, b)$ is tricolored triangle$\}$.

(2) (v, w) is not in $E_{(a,b)}$ iff $(v, a), (v, b), (w, a), (w, b), (v, w)$, and (a, b) are all colored differently and $(v.w)$ is in $E_{(a,b)}$ otherwise.

Let I be an arbitrary maximal independent set of $G_{(a,b)}$. We have the following lemma.

Lemma 5.2.10 $I \cup \{a, b\}$ forms the vertex set of a maximal simplex that generates M.

Proof: For every v, w in I and $v \neq w$, we have (v, w) in E since (v, a) and (w, a) are

in different color classes. Thus the vertices in $I \cup \{a, b\}$ form a complete subgraph of G. Assume that two edges (v, w) and (x, y) in this subgraph are colored the same color. Then following cases can occur:

(1) v, w, x, y are in I, $(v, w), (x, y)$ have the same color A.

Subcase 1 The two edges share a vertex, let the vertex be $v = x$. Applying Lemma 5.2.2 to triangles (b, x, y) and (b, v, w)we know that (b, w) and (b, y) have the same color, a contradiction. (See Figure 5.2.1.)

Subcase 2 The two edges do not share a vertex. From Lemma 5.2.1 we know that either (b, x) or (b, y) must have the same color as (b, v), again a contradiction. (See Figure 5.2.1.)

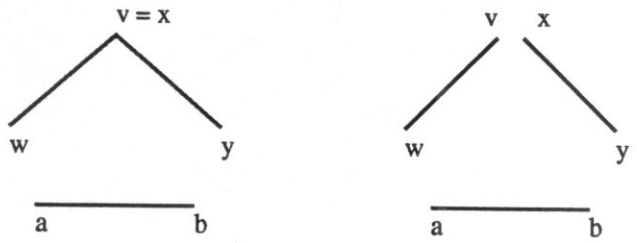

Figure 5.2.1

(2) Vertices v, w, x are in $I, and (v, w)$ (x, a) have the same color A. In this case, (x, b) has the same color as (v, b) or (w, b) and this is impossible. (See Figure 5.2.2.)

If $I \cup \{a, b\}$ is not a maximal simplex, then by Lemma 5.2.6 it is contained in a larger

simplex, which means that there is a larger independent set I' that contains I in $G_{(a,b)}$. ∎

Step 5.2.3 and 5.2.4 After finding a maximal simplex for M, the vertices of the simplex can be numbered so that if we direct each edge in the simplex from the higher numbered vertex to the lower numbered vertex, then we have a transitive orientation for the simplex. This orientation gives direction to the edges in each color class and can be easily extended to the whole multiplex.

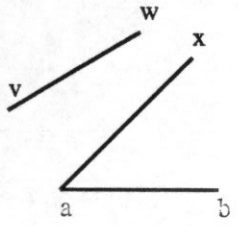

Figure 5.2.2

The complexity of the algorithm is dominated by the maximal independent set computation in step 2. We need to find maximal independent sets on $O(n)$ graphs, each may contain $O(n)$ vertices and $O(n^2)$ edges. The best known parallel maximal independent set algorithm runs in $O(log^4(n))$ time using $O(n+m)$ processors on a graph with n vertices and m edges. (See [GS].) Thus the transitive orientation algorithm runs in $O(log^4(n))$ time with $O(n^3)$ processors.

§5.3 Applications

The parallel algorithm that transitively orients the edges of a comparability graph can be used to give efficient parallel algorithms for following problems:

Algorithm 5.3.1 Find a maximum clique for a comparability graph.

Algorithm 5.3.2 Find a minimum coloring for a comparability graph.

Algorithm 5.3.3 Find a maximum clique, a minimum coloring, a maximum independent set, and a minimum clique covering for a permutation graph.

Algorithm 5.3.4 Find a permutation representation for a permutation graph.

All these algorithms have the same complexity as the transitive orientation algorithm.

Let the transitive orientation of G be G'. G' is a digraph. It is easily seen that G' is acyclic and there is a one-one correspondence between the cliques in G and the paths in G'. The longest path in G' can be computed by a list ranking algorithm. (See [CV].) As the matter of fact, a maximal weighted clique can be computed for G by computing a weighted longest path in G'. The computation of list ranking is much more efficient than the maximal independent set computation in §5.2. The complexity of algorithm 5.3.1 is dominated by the maximal independent set computation. For each vertex $v \in V$, we let $l(v)$ be the length of the longest path in G' starting from v. Let $c(v) = l(v) + 1$. It is not hard to see that $C : v \rightarrow c(v)$ is a proper coloring of G. The number of colors used in C equals the length of the longest path in G' plus one, which is equal to the size of the maximum clique in G. C is a minimum coloring of G. $l(v)$ can also be computed by the

list ranking algorithm. We can find a minimum coloring for comparability graph with the same complexity as the transitive orientation algorithm.

A graph G is a permutation graph if there exists a 1-1 and onto mapping $p : V \rightarrow \{1, 2, ..., n\}$ and a permutation P of $\{1, 2, ..., n\}$ such that (v, w) is in E iff $p(v)$ and $p(w)$ form an inversion pair in P. P is called a permutation representation of G.

Permutation graphs can be characterized completely by comparability graphs. Lemma 5.3.1 is from [EPL].

Lemma 5.3.1 A graph G is a permutation graph iff both G and the complement of G are comparability graphs.

A maximum clique of the complement of G is a maximum independent set of G. Let C_1, ..., C_p be a minimum coloring of the complement of G. Then $\{\{v \mid v \text{ is colored by } C_i\} \mid 1 \leq i \leq p\}$ is a minimum clique cover of G. When both G and the complement of G are comparability graphs, we can compute a maximum clique, a maximum independent set, a minimum coloring , and a minimum clique cover for both G and the complement of G.

If G is a permutation graph, let $G_1 = (V, F_1)$ and $G_2 = (V, F_2)$ be transitive orientations for G and the complement of G. It can be shown that the following two graphs are both acyclic and transitive: $(V, F_1 \cup F_2)$, $(V, F_1 \cup F_2^{-1})$. The undirected graphs underlying them are both complete. Both graphs represent a total linear ordering on V. Let a *topological ordering* in the two graphs be $l(v)$, $l'(v)$. $P : l(v) \rightarrow l'(v)$ is a permutation that represents G.

In this section, we discuss some applications of the algorithm presented in §5.2. One interesting problem that cannot be solved by the algorithm in §5.2 is to find a maximum independent set for a comparability graph. Let G' be the digraph obtained from transitively orienting the edges of G. The problem of finding a maximum independent set of G is related to network flow problems on G'. Unfortunately, no efficient NC algorithms are known for general network flow problems.

§5.4 Problems for Future Research

The transitive orientation algorithm in this chapter improves the previous algorithms. It is still not very efficient. The number of processors in the algorithm is $O(n^3)$, which is not realistic for real computation. It is conceivable that both the processor bound and the time bound can be improved in the future.

It would be interesting to know if there exists a parallel algorithm to test whether a graph is perfect. For perfect graphs, it is interesting to design efficient parallel algorithms to compute their maximum cliques, minimum colorings, maximum independent sets, and minimum clique covers.

Appendix

Graph Definitions and References

Graph Definitions

This appendix contains a collection of definitions used in this thesis. They are standard and can be found in many textbooks. (See [AHU].)

Definition 1 A *graph* G is an ordered pair of disjoint sets (V, E) such that $V \neq \phi$, $E \subseteq \{(v,w) \mid v,w \in V\} \cup \{< v,w > \mid v,w \in V,\}$. V is the vertex set, E is the edge set. (v,w) and $< v,w >$ are the ordered and the unordered pairs of v and w respectively.

We always use G to represent a graph. We assume V, E are finite and denote $\mid V \mid$ by n, $\mid E \mid$ by m.

Definition 2 If an edge of G is an ordered pair, it is a *directed edge*, otherwise it is undirected. If all the edges in G are *directed* (or *undirected*), G is a *digraph* (or *undirected graph*), otherwise G is a mixed graph. When G is a digraph, we also use $v \rightarrow w$ to represent an edge $< v,w >$ of G.

All graphs are undirected unless otherwise stated.

Definition 3 Let (v,w) be an edge of G. Edge (v,w) is *incident* on v and w. Vertices v and w are the *end vertices* of (v,w). The *degree* of v (denoted by $deg(v)$) is the number of edges that are incident on v.

Definition 4 Let G be a digraph and let $< v,w >$ be an edge of G. Vertex v is the *tail* of (v,w) and w is the *head* of (v,w). The *indegree* (or *outdegree*) of v is the number of times v occurs as head (or tail) of an edge.

Definition 5 A *subgraph* $G' = (V', E')$ of G is a graph such that $V' \subseteq V$ and $E' \subseteq E$. If $V' = V, G'$ is a *spanning subgraph* of G. Let $V' \subseteq V$, graph $(V', \{(x, y) \mid (x, y) \in E, x \in V',$ and $y \in V'\}$ is the subgraph of G *induced* by V'.

Definition 6 Let G be an undirected graph or a digraph. A *path* in G is a set of vertices $x_0, x_1, ..., x_k$ such that (x_i, x_{i+1}) (or $< x_i, x_{i+1} >$ if G is a digraph) is an edge of G for every i, $0 \leq i \leq k-1$. The path connects x_0 and x_k. If $x_0 = x_k$, the path is a *cycle*. A path is *simple* if no vertex is repeated, except for the possibility of $x_0 = x_k$.

Definition 7 A graph is *connected* if for every two vertices x and y in G, there is a path connecting them.

Definition 8 A *tree* is a graph that is connected and contains no cycle.

Definition 9 A *spanning tree* of G is a spanning subgraph of G that is a tree.

Definition 10 A *rooted spanning tree* of G is obtained from a spanning tree T of G by giving each edge in T a direction such that only one vertex in T has indegree 0 (this is called the root of T) and the rest of the vertices have indegree 1. The vertices that have outdegree 0 are called the *leaves*.

Definition 11 Let T be a rooted spanning tree of G. If $< v, w >$ is in T, then v is the *parent* of w and w is a *child* of v. If there exist $v_1, v_2, ..., v_k$ such that $< v_i, v_{i+1} >$ is in T for every $i, 1 \leq i < k$, v_1 is an *ancestor* of v_k and v_k is a *descendant* of v_1. A vertex is an *ancestor* and *descendant* of itself unless otherwise stated.

Definition 12 Let T be a rooted spanning tree of G. For any vertex v in G , the *depth*

of v in T is the number of edges on the unique path from the root to v. For any two vertices v and w in G, the *nearest common ancestor* of v and w (denoted as $nca(v,w)$) is the ancestor of both v and w that has the largest *depth.*

Definition 13 A *pre* order numbering of a rooted spanning tree T of G is generated by a depth-first search on T. Each vertex is numbered when we first encounter it during the search. A *post* order numbering of T is also generated by a depth-first search on T. Each vertex is numbered after all its descendants are searched and numbered.

Definition 14 A *rooted spanning* T of G is a depth-first tree, if for every edge $< x, y >$ in G, x is an ancestor of y or vice-versa.

Definition 15 Let $G = (V, E)$ be a graph, we say graph $(V, \{(v,w) \mid v,w \in V\} - E)$ is the *complement* of G.

Definition 16 A digraph G is *acyclic* if there is no cycle in G.

Definition 17 Let $G = (V, E)$ be an acyclic digraph. Let $\mid V \mid = n$. We can assign a distinct integer $n(v)$ between 1 and n to each vertex v in V such that $< v, w > \in E$ iff $n(v) > n(w)$. We call such an assignment a *topological ordering* of G.

Definition 18 Let $G = (V, E)$ be a graph. G is *2-connected* if for every vertex $v \in V$, $(V - \{v\}, E - \{(v,w) \mid \exists w \in V, (v,w) \in E\})$ is connected. When a graph G is not 2-connected, it can be decomposed into maximal edge-disjoint subgraphs that are 2-connected. (See [AHU].) These subgraphs are called the *2-connected components* of G. A *bridge* is a 2-connected component that has just one edge.

Definition 19 Let $G = (V, E)$ be a graph. G is *2-edge connected* if for every edge $(v, w) \in E$, $(V, E - \{(v, w)\})$ is connected. When a graph G is not 2-edge connected, it can be decomposed into maximal vertex-disjoint subgraphs that are 2-edge connected. (See [AHU].) These subgraphs are called the *2-edge connected components* of G.

Definition 20 Let $G = (V, E)$ be a graph. G is *k-connected* if for every subset $V' \subseteq V$ with $| V' | \leq k - 1$, $(V - V', E - \{(v, w) \mid v \in V', w \in V, (v, w) \in E\})$ is connected.

Definition 21 Let $G = (V, E)$ be a graph. G is *k-edge connected* if for every subset $E' \subseteq E$ with $| E' | \leq k - 1$, graph $(V, E - E')$ is connected.

Definition 22 Let $G = (V, E)$ be a k-connected graph. G is *minimal k-connected* if for every edge (v, w) in E, $(V, E - \{(v, w)\})$ is not k-connected.

Definition 23 Let $G = (V, E)$ be a k-edge connected graph. G is *minimal k-edge connected* if for every edge (v, w) in E, graph $(V, E - \{(v, w)\})$ is not k-edge connected.

Definition 24 Let $G = (V, E)$ be a k-connected (or k-edge connected) graph and $(x, y) \in E$. Edge (x, y) is a *k-essential* (or *k-edge essential*) edge for G iff $G = (V, E - \{(x, y)\})$ is not k-connected (or k-edge connected). When $k = 2$, edge (x, y) is a *2-essential* (or *2-edge essential*) edge of G.

Definition 25 Let $G = (V, E)$ be a graph. For each vertex v of G, we draw a distinct point on the plane. For each edge (v, w) in E, we draw an arc on the plane. If we can arrange the arcs in such a way that no two arcs intersect except at their end points, G is called a *planar graph* and the drawing is called a *planar embedding* of G. If G is not planar, it is a *nonplanar* graph.

Definition 26 Let $G = (V, E)$ be a graph with $\mid V \mid = n$. An *st-numbering* is an assignment $st(v)$ of distinct integers between 1 to n to the vertices in V. The assignment satisfies following condition: for each vertex v such that $st(v) \notin \{1, n\}$, there exist (v, u) and (v, w) in E such that $st(v) < st(u)$ and $st(v) > st(w)$.

Definition 27 Let $G = (V, E)$ be a graph. V' is a *clique* of G if $(v, w) \in E$ for every two vertices v and w in V'. A *maximum clique* of G is a clique with the maximum cardinality.

Definition 28 Let $G = (V, E)$ be a graph with $\mid V \mid = n$. A proper *coloring* of G is a mapping $C : V \rightarrow \{1, ..., n\}$ such that for v and w in V, $C(v) \neq C(w)$ if $(v, w) \in E$. The set of colors being used is $\{i \mid 1 \leq i \leq n \text{ and } \exists v \in V, C(v) = i\}$. The *minimum coloring* of G is a coloring that uses the fewest colors.

Definition 29 Let $G = (V, E)$ be a graph. V' is an *independent set* of G if $(v, w) \notin E$ for every two vertices v and w in V'. A *maximum independent* set of G is an independent set with the maximum cardinality.

Definition 30 Let $G = (V, E)$ be a graph. $\{V_1, ..., V_k\}$ is a *clique covering* of G if (1) $V_i (1 \leq i \leq k)$ is a clique, (2) $V_i \cap V_j = \phi$ for $1 \leq i \neq j \leq k$, and (3) $\cup_{i=1}^{i=k} V_i = V$. The size of a clique covering is $\mid \{V_1, ..., V_k\} \mid$. A *minimum clique covering* of G is a clique covering with the smallest size.

References

[**AHU**] A. Aho, J. Hopcroft, and J. Ullman, *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

[**AP**] L. Auslander and S.V. Parter, "On imbedding graphs in the plane", *J. Math. and Mech*, **10**(3) (1961) 517-523.

[**B**] B. Bollobas, *Extremal Graph Theory*, Academic Press (1978)

[**BL**] K.S. Booth and G.S. Leuker, "Testing For the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms", *J. of Comp. and Sys. Sci*, **13** (1979) 335-379.

[**BT**] G.D. Battista and R. Tamassia, "Incremental Planarity Testing", *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, (1989) 436-441.

[**CGM**] V. Chachra, P.M. Ghare, and J.M. Moore, *Applications of Graph Theory and Algorithm*, North Holland Inc. (1979).

[**Co**] S.A. Cook, "The Complexity of Theorem Proving Procedures", *3rd. Annual ACM Symposium on Theory of Computing*, (1971) 151-158.

[**CV**] R. Cole and U. Vishkin, "Approximate Parallel Scheduling. Part 1: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time", *SIAM J. on Comp*, **17**(1) (1988) 128-142.

[**D**] G.A. Dirac, "Minimally 2-Connected Graphs", *J. Reine Angew Math*, (1967) 204-216.

[**ET1**] S. Even and R.E. Tarjan, "Computing an st-numbering", *Th. Comp. Sci*, **2** (1975) 330-334.

[**ET2**] K.P. Eswaran and R.E. Tarjan, "Augmentation Problems", *SIAM J. on Comp*, **5**(4) (1976) 653-665.

[**ET3**] S. Even and R.E. Tarjan, "Network Flow and Testing Graph Connectivity", *SIAM J. on Comp*, **4**(4) (1975) 507-518.

[**F**] R.W. Floyd, "Algorithm 97: Shortest Path", *Comm. ACM*, **5**(6) (1962) 345.

[**FF**] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press (1962).

[**G**] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press (1980).

[**GJ**] M.R. Garey and D.S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., New York (1979).

[**GKRST**] P. Gibbons, R. Karp, V. Ramachandran, D. Soroker, and R. Tarjan, "Transitive Compaction in Parallel via Branchings", *Journal of Algorithms*, **12** (1991) 110-125.

[**GS**] M. Goldberg and T. Spencer, "A New Parallel Algorithm for the Maximal Independent Set Problem", *SIAM J. on Comp*, **18**(2) (1989) 419-427.

[**H**] B. Harris. *Graph Theory and Its Applications*, Academic Press (1970).

[**HT1**] J. Hopcroft and R.E. Tarjan, "Efficient Planarity Testing", *J. ACM*, **21**(4), (1974) 549-568.

[**HT2**] J. Hopcroft and R.E. Tarjan, "Efficient Algorithms for Graph Manipulation", *Comm. ACM*, **16**(6), (1973) 372-378.

[**HT3**] J. Hopcroft and R.E. Tarjan, "Isomorphism of Planar Graphs", *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher eds. 1973 Plenum Press, New York, 131-152.

[**HT4**] J. Hopcroft and R.E. Tarjan, "Dividing a graph into triconnected components", *SIAM J. on Comp*, **2**(3) (1973) 135-158.

[**HT5**] D. Harel and R.E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors", *SIAM J. on Comp*, **13**(2) (1984) 338-355.

[**HM**] D. Helmbold and E. Mayr, "Perfect Graphs and Parallel Algorithms", *Proceedings of the 1986 International Conference on Parallel Processing*, (1986) 853-860.

[**HS**] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press (1983).

[**JTS**] R. Jayakumar, K. Thulasiraman, and M. Swamy, " $O(n^2)$ Algorithms for Graph Planarization", *IEEE Trans. on CAD* (1989) 257-267.

[**Ka**] R. Karp, "Reducibility among Combinatorial Problems", *Complexity of Computer Computations*, , R.E. Miller and J.W. Thatcher eds. 1973 Plenum Press, New York, 131-152.

[**Kn**] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., Vol.1, (1968).

[**Kr**] J.B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. Amer. Math. Soc*, **7**(1) (1956) 48-50.

[**KR1**] P. Kelsen and V. Ramachandran, "On Finding Minimal 2-connected Subgraphs", *Department of Computer Science, University of Texas at Austin, TR-90-16,* (1990).

[**KR2**] P.N. Klein and J.H. Reif, "An Efficient Parallel Algorithm for Planarity", *Proceedings of the 27th Annual IEEE Symposium on Foundation of Computer Science*, (1986) 465-477.

[**KR3**] P. Kelsen and V. Ramachandran, "The Complexity of Finding Minimal Spanning Subgrahs", *(Preliminary Version), manuscript, Department of Computer Science, University of Texas at Austin,* (1991).

[**KVV**] D. Kozen, U.V. Vazirani, and V.V. Vazirani, "NC Algorithms for Comparability Graphs, Interval Graphs, and Testing for Unique Perfect Matching", *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Dehli, (1985)

[**L**] L. Lovasz, "A Characterization of Perfect Graphs", *J. Combin. Theory*, **B**(13) 95-98.

[LEC] A. Lempel, S. Even, and I. Cederbaum, I., "An Algorithm for Planarity Testing of Graphs", *Theory of Graphs, International Symposium*, Rome, (1966) 215-232.

[M] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-completeness*, Spring-Verlag (1984).

[NNS] J. Naor, M. Naor, and A.A. Schaffer, "Fast Parallel Algorithms for Chordal Graphs", *Proceedings of the 9th Annual ACM Symposium Theory of Computing*, (1987) 355-364.

[Pl] M.D. Plummer, "On Minimal Blocks", *Trans. Amer. Math. Soc*, (1968) 134,

[Pr] R.C. Prim, "Shortest Connection Networks and Some Generalizations", *Bell System Technical J*, (1957) 1389-1401.

[PLE] E. Pnueli, A. Lempel, and S. Even, "Transitive Orientation of Graphs and Identification of Permutation Graphs", *Canada. J. Math*, **23** 160-175.

[T1] R.E. Tarjan, "Depth-first Search and Linear Graph Algorithms", *SIAM J. on Comp*, **1**(2) (1972) 146-159.

[T2] R.E. Tarjan, "Data Structures and Network Algorithms", *SIAM* (1983).

[Te] H.N.V. Temperley, *Graph Theory and Applications*, Ellis Horwood Ltd (1981).

[Wa] S. Warshall, "A Theorem on Boolean Matrices", *J. ACM*, **9**(1) (1962) 11-12.

[**WB**] R.J. Wilson and L.W. Beineke, *Applications of Graph Theory*, Academic Press, Reading, Mass. (1979).

[**Wu**] W. Wu, "On the Planar Imbedding of Linear Graphs", *J. Sys. Sci. & Math. Sci*, **5**(4) (1985) 290-302.