



An efficient implementation of a planarity testing and maximal planar subgraph algorithm
by Zhongyuan Li

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in
Computer Science

Montana State University

© Copyright by Zhongyuan Li (1996)

Abstract:

This thesis presents efficient implementations of a planarity testing and a maximal planar subgraph algorithm. The algorithms are based on the methods of Shih and Hsu [10] and Hsu [9]. The algorithms use the vertex addition method. The key of the algorithms is to add vertices according to a postordering obtained from a depth-first search tree. The algorithms run in linear time.

The implementations were developed based on Shih and Hsu's methods, and were implemented in the C language. Empirical analysis shows that the programs run in linear time.

The maximal planar subgraph algorithm was also compared with several other algorithms. In two tables, the sizes of the maximal planar subgraphs obtained from several special graphs and twenty random graphs are listed.

An Efficient Implementation of a Planarity Testing and Maximal Planar Subgraph Algorithm

by

Zhongyuan Li

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

July 1996

N378
L61392

ii

APPROVAL

of a thesis submitted by

Zhongyuan Li

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

7-16-96
Date

Robert C. Cullen
Chairperson, Graduate Committee

Approved for the Major Department

7/16/96
Date

J. D. Stanley
Head, Major Department

Approved for the College of Graduate Studies

8/4/96
Date

R. Brown
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature

Li Zhongyuan

Date

July 19, 1996

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my graduate committee members, Dr. Robert Cimikowski, Dr. Denbigh Starkey, and Prof. Anne DeFrance, and the rest of the faculty members from the Department of Computer Science for their help and guidance during my graduate program. I would especially like to thank my advisor, Dr. Robert Cimikowski, for his encouragement and support during the whole work.

Contents

Table of Contents	v
List of Tables	vii
List of Figures	viii
Abstract	x
1 Introduction	1
2 Preliminaries	3
2.1 Definitions	3
2.2 Some Useful Theorems	7
3 Planarity Testing and Maximal Planar Subgraph Algorithms	8
3.1 Overview	8
3.2 Theorems	13
4 Data Structures	25
4.1 Representations of Graphs	25
4.2 Representation of the Depth-First Search Tree	26
5 Shih and Hsu's Planarity Testing Algorithm	28
5.1 Outline of the Algorithm	28

5.2	Time Complexity of the Algorithm	30
6	Hsu's Maximal Planar Subgraph Algorithm	32
6.1	Outline of the Algorithm	32
6.2	Time Complexity of the Algorithm	34
7	Test Results	35
7.1	Test Graphs	35
7.2	Comparison with Other Algorithms	42
	Bibliography	45

List of Tables

7.1	Heuristics applied to dense nonplanar graphs	44
7.2	Heuristics applied to random graphs	44

List of Figures

2.1	A directed graph and an undirected graph	3
2.2	A multigraph and a pseudograph	4
2.3	A cut vertex and separation pair	5
2.4	Graphs K_5 and $K_{3,3}$	5
2.5	A planar graph and an embedding	6
2.6	A graph G and the subdivision graph of G	6
3.1	A graph, its DFS tree and 4th iteration	9
3.2	Cut vertex i and the subtrees	11
3.3	A tree before contraction and after contraction	12
3.4	A terminal v-node and a terminal c-node	13
3.5	An intermediate c-node, h_1 and h_2 , upward and downward di- rection	14
3.6	A forbidden structure	15
3.7	A forbidden structure	16
3.8	A forbidden structure	16
3.9	A forbidden structure	17
3.10	A forbidden structure	18
3.11	An inner cycle C_1 and an outer cycle C_2	19
3.12	Border of cycle C_i with one terminal v-node	20
3.13	Border of cycle C_i with one terminal c-node	21

3.14	Two terminal v-nodes u_1, u_2 and v-node u	22
3.15	Two terminal v-nodes u_1, u_2 and u is a c-node.	23
3.16	Two terminal nodes, u_1, u_2 , in different c-node and u is a c-node.	24
4.1	The input file of K_5 and its adjacency list representation	26
4.2	Representation of a depth-first search tree	27
7.1	Graph g_1	36
7.2	A maximal planar subgraph of g_1	36
7.3	Graph g_2	37
7.4	A maximal planar subgraph of g_2	37
7.5	Graph g_3	37
7.6	A maximal planar subgraph of g_3	38
7.7	Graph g_4	38
7.8	A maximal planar subgraph of g_4	39
7.9	Graph g_5	40
7.10	A maximal planar subgraph of g_5	40
7.11	Graph g_6	41
7.12	A maximal planar subgraph of g_6	41

Abstract

This thesis presents efficient implementations of a planarity testing and a maximal planar subgraph algorithm. The algorithms are based on the methods of Shih and Hsu [10] and Hsu [9]. The algorithms use the vertex addition method. The key of the algorithms is to add vertices according to a postordering obtained from a depth-first search tree. The algorithms run in linear time.

The implementations were developed based on Shih and Hsu's methods, and were implemented in the C language. Empirical analysis shows that the programs run in linear time.

The maximal planar subgraph algorithm was also compared with several other algorithms. In two tables, the sizes of the maximal planar subgraphs obtained from several special graphs and twenty random graphs are listed.

Chapter 1

Introduction

A graph is *planar* if it can be drawn in the plane without any crossing edges. Planar graph problems can be found in many applications, e.g., in VLSI and printed circuit board design, in network design and analysis, and in determining the isomorphism of chemical structures.

The *planarity testing problem* is to determine whether a given graph can be drawn in the plane without any crossing edges.

Planarity testing algorithms are divided into two major types: path addition methods and vertex addition methods. The path addition algorithm was originally developed by Auslander and Parter [1]; Hopcroft and Tarjan [2] presented a linear time implementation. The vertex addition algorithm was presented first by Lempel, Even and Cederbaum [3], and improved to a linear algorithm by Booth and Lueker [4].

If a graph is not planar, then the next problem that arises is how to find a planar subgraph that is as close as possible to the given graph. For a given graph G , a *maximal planar subgraph* G' is a planar subgraph of G such that adding any edge of $G - G'$ to G' will result in a nonplanar graph. A maximal planar subgraph which has the maximum number of edges is called a *maximum planar subgraph*.

Since finding a maximum planar subgraph has been shown to be NP-complete [5], research has focused on computing a maximal planar subgraph G' of G . For a given graph G which has m edges and n vertices, the first algorithm for this problem has an $O(mn)$ worst-case time bound [6]. Recently Cai, Han and Tarjan developed an $O(m \log n)$ algorithm [7] for the maximal planar subgraph problem, based on the Hopcroft-Tarjan planarity testing algorithm. An algorithm with the same complexity bound of $O(m \log n)$ can also be derived from the incremental planarity testing algorithm of Di Battista and Tamassia [13]. Jayakumar, Thulasiraman and Swamy [15] presented an $O(n^2)$ planarization algorithm based on PQ-trees. However, this algorithm contains some mistakes. Kant's correct version [16] of the algorithm also runs in $O(n^2)$. Djidjev [12] presented the first linear time algorithm for the maximal planar subgraph problem. O. Goldschmidt and A. Takvorian's planarization algorithm used the cycle-packing method [14]. According to Cimikowski [8], it gives the best results for these algorithms but runs very slowly for a large graph.

This implementation is based on Shih and Hsu's linear time planarity testing algorithm [10] and maximal planar subgraph algorithm [9]. This algorithm uses the vertex addition approach. The key to this algorithm is to add vertices according to a postordering obtained from a depth-first tree. This implementation is coded in the C language.

This thesis presents some useful definitions and theorems in Chapter 2. Chapter 3 gives an overview of the algorithms. Chapter 4 describes the main data structures of the implementations. Chapter 5 discusses the details of the planarity testing algorithm and its time complexity analysis. Chapter 6 discusses the details of the maximal planar subgraph algorithm. Finally, the last chapter discusses the test cases and results.

Chapter 2

Preliminaries

2.1 Definitions

A graph $G = (V, E)$ consists of a finite nonempty vertex set V and a finite edge set E . Each edge is a pair of distinct vertices. The number of vertices in the graph G is $n=|V|$, and the number of edges in the graph G is $m=|E|$. Any vertex pair $(u, v) \in E$ is called an *edge* and is said to *join* the vertices u and v .

The graph $G = (V, E)$ is a *directed graph* if E is defined as a set of ordered pairs of distinct vertices. If E is defined as a set of unordered pairs of distinct vertices, the graph G is called an *undirected graph*.

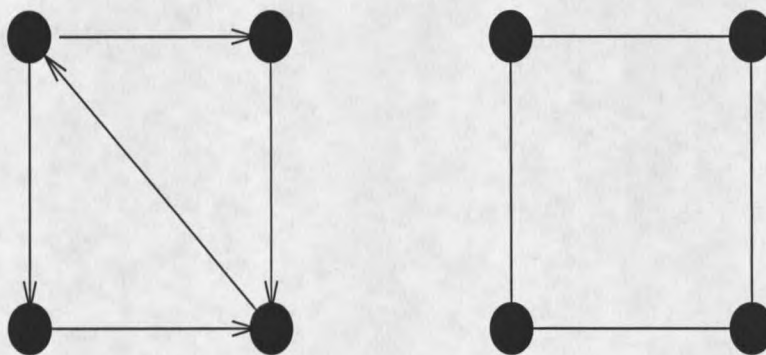


Figure 2.1: A directed graph and an undirected graph

A *simple graph* G is a graph which has no *multiple edges* or *loops*. The *multiple edges* are the edges which connect the same pair of vertices. A *loop* is the edge which joins a vertex to itself. If a graph G has some multiple edges but no loops, it is called a *multigraph*. If a graph G has multiple edges or loops, it is called a *pseudograph*.

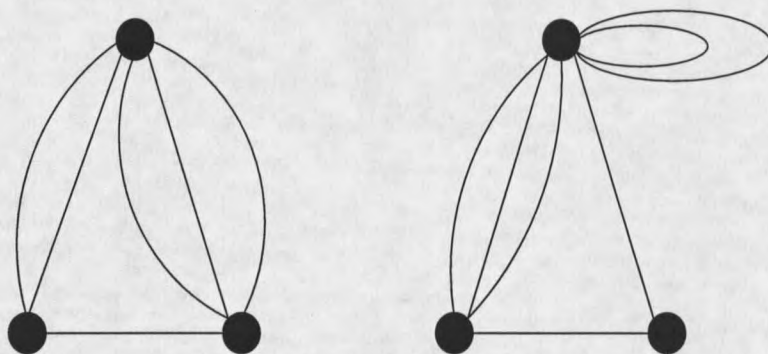


Figure 2.2: A multigraph and a pseudograph

A *walk* of a graph G is an alternating sequence of vertices and edges, beginning and ending with vertices. If all vertices on a *walk* are distinct, the walk is called a *path*. A graph G is *connected* if every pair of vertices is connected by a path.

A *connected component* of a graph is a maximal connected subgraph. A *cut vertex* is a vertex whose deletion increases the number of components. A graph G is *2-connected* if it is connected and has no cut vertex. A *separation pair* of a 2-connected graph G is a pair of vertices whose deletion disconnects the graph G . If a graph G has no cut vertex or separation pair then it is *3-connected*. Usually a 2-connected graph is called *biconnected*, and a 3-connected graph is called *triconnected*. Figure 2.3 shows a cut vertex and separation pair.

A graph G in which every pair of distinct vertices is adjacent is called a *complete graph*. The complete graph on n vertices is denoted by K_n .

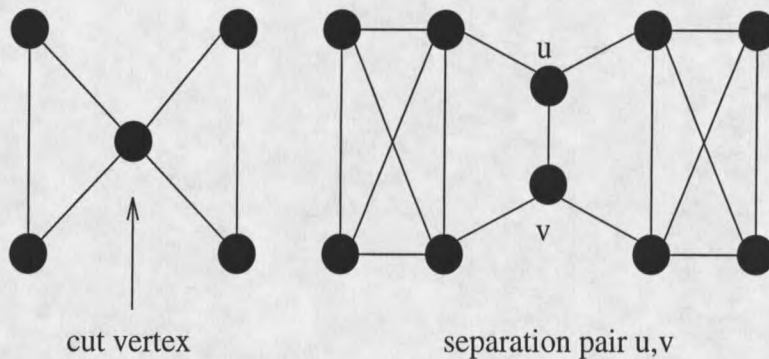
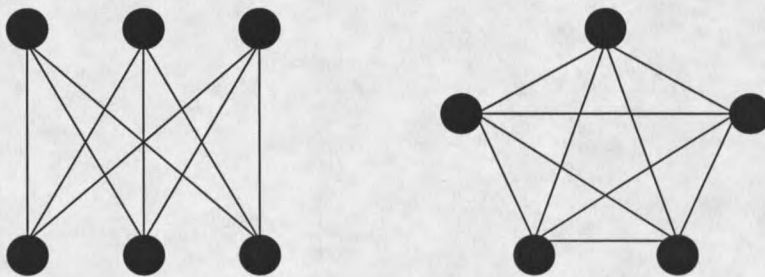


Figure 2.3: A cut vertex and separation pair

A graph G is called *bipartite* if its vertex set V can be partitioned into two subsets, V_1 and V_2 , such that every edge of G joins V_1 with V_2 . If the graph G contains every edge joining V_1 with V_2 , then it is called a *complete bipartite graph*. If V_1 and V_2 have m and n vertices, the graph G is denoted by $K_{m,n}$.

Figure 2.4: Graphs K_5 and $K_{3,3}$

A graph G is said to be *embedded* in a surface S when it is drawn on S so that no two edges intersect. A graph G is *planar* if it can be embedded in the plane.

A *subdivision* of G is a graph obtained from G by subdividing some of the edges, that is, by replacing the edges by paths having at most their end vertices in common.

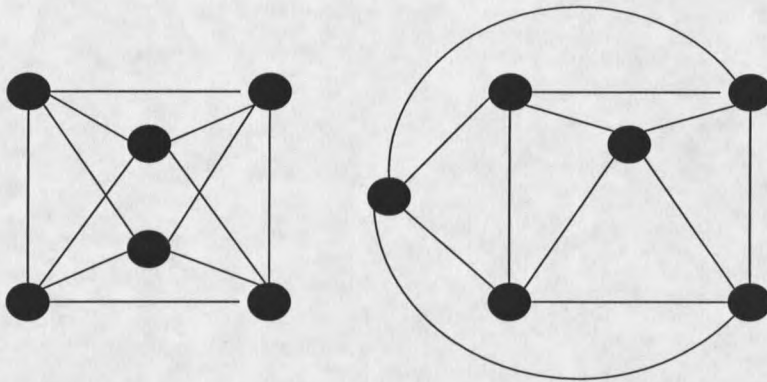


Figure 2.5: A planar graph and an embedding

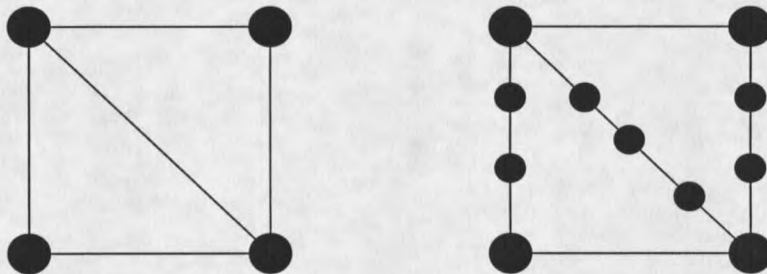


Figure 2.6: A graph G and the subdivision graph of G

2.2 Some Useful Theorems

Kuratowski's Theorem. A graph is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$.

Kuratowski's Theorem can be proven by using Euler's Polyhedron Formula and its corollaries and Lemma 1.1. These corollaries and lemma are listed below. The details are given in Nishizeki and Chiba [11].

Euler's Polyhedron Formula. Let G be a connected plane graph, and let n , m and f denote, respectively, the number of vertices, edges, and faces of G . Then $n - m + f = 2$.

Corollary 1.1 If G is a planar graph with n (≥ 3) vertices and m edges, then $m \leq 3n - 6$. Moreover, the equality holds if G is a maximal planar graph.

Corollary 1.2 If G is a planar bipartite graph with n (≥ 3) vertices and m edges, then $m \leq 2n - 4$.

Lemma 1.1 If G is a 3-connected graph having five or more vertices, then G contains an edge e such that the graph G' obtained from G by contracting e is 3-connected.

Chapter 3

Planarity Testing and Maximal Planar Subgraph Algorithms

This chapter discusses the details of Shih and Hsu's planarity testing algorithm [10] and Hsu's maximal planar subgraphs algorithm [9]. These algorithms work on a given undirected and biconnected graph. First, an overview is given of the algorithms. Then several definitions and theorems used in the algorithms are discussed.

3.1 Overview

As was mentioned in Chapter 1, the planarity testing problem is to determine whether there exists a way to draw a graph in the plane without any crossing edges. Given an undirected graph G , finding a maximal planar subgraph means finding a planar subgraph G' of G such that no edge of $G - G'$ can be added to G' without destroying planarity.

Basically there are two types of planarity testing and maximal planar subgraph algorithms: one uses a *path addition* approach, while the other uses a *vertex addition* approach.

Shih and Hsu's [10] planarity testing and maximal planar subgraph algorithms are based on the *vertex addition* approach. In the Hopcroft and Tarjan

[2] and Lempel [3] algorithms, the subgraphs constructed at each iteration are connected, but Shih and Hsu's algorithms keep the subgraph induced by the vertices that have not been added connected.

In Shih and Hsu's algorithm, a depth-first search tree is first created, and each vertex is given a number based on postordering. Then the vertices are added one by one in ascending order of the postordering. This means that each vertex will not be considered for inclusion until all of its children have been considered.

By using postordering of a depth-first search tree, the graph induced by the vertices that have not been added into the subgraphs is connected.

Let T be the depth-first search tree of G . All edges in T are called *tree edges*, and the other edges of G are called *back edges*. In a depth-first search tree, if a vertex in the tree has a back edge, then this back edge must connect to one of its ancestors. Also, one back edge can create at most one biconnected component. In the algorithm, once the biconnected component is embedded, it will never change.

Figure 3.1 shows a graph G , its depth-first search tree, and the subgraphs after inserting the vertex i .

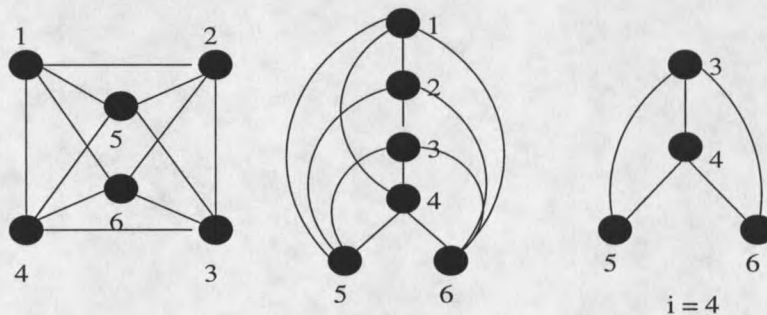


Figure 3.1: A graph, its DFS tree and 4th iteration

The graph created in iteration i is denoted as G_i ; the forest which is generalized in iteration i is denoted as F_i ; the trees in the forest F_i is denoted as $T_1, T_2, \dots, T_k$; and their roots is denoted as $r_1, r_2, \dots, r_k$.

Now consider the i^{th} iteration of the algorithm. In this iteration, we will insert the vertex i into graph G_{i-1} . As was mentioned before, if there is a vertex in the tree T_j of forest F_{i-1} which has a back edge to i then vertex i has a tree edge to the vertex r_j , the root of T_j . Now we list the trees of F_{i-1} whose roots are adjacent to vertex i as $T_1, T_2, \dots, T_l$.

For any subset V_1 of V , the subgraph G' induced on V_1 is the maximal subgraph of G with vertex set V_1 . Denote the graph G' induced on V_1 as $G[V_1]$.

Clearly the vertex i is a cut vertex of the induced subgraph $G_i = G[\{i\} \cup T_1 \cup T_2 \cup \dots \cup T_l]$. This means G_i is planar if and only if each $G[\{i\} \cup T_j]$ is planar, $j = 1, 2, \dots, l$. So the problem can be solved by determining the planarity of each $G[\{i\} \cup T_j]$. Figure 3.2 shows the situation at iteration i .

For a given undirected and biconnected graph G , if every G_i , $i = 1, 2, \dots, n$, is planar, then the given graph G is planar. If there exists a G_i which is not planar then the given graph G is not planar. Some edges (determined by the theorems) can be removed to make the G_i planar. After all the vertices are inserted, the final graph will be a maximal planar subgraph of the given graph G .

For each graph G_i created at the i^{th} iteration, the vertex in G_i is defined as a *v-node* if the vertex is an original vertex of the given graph G . And a biconnected component of G_i is defined as a *c-node*. At the time a c-node is created, the embedding of the biconnected component of G_i is determined and will never change.

A *marked* vertex in F_i is a vertex which is adjacent to vertex i . At the

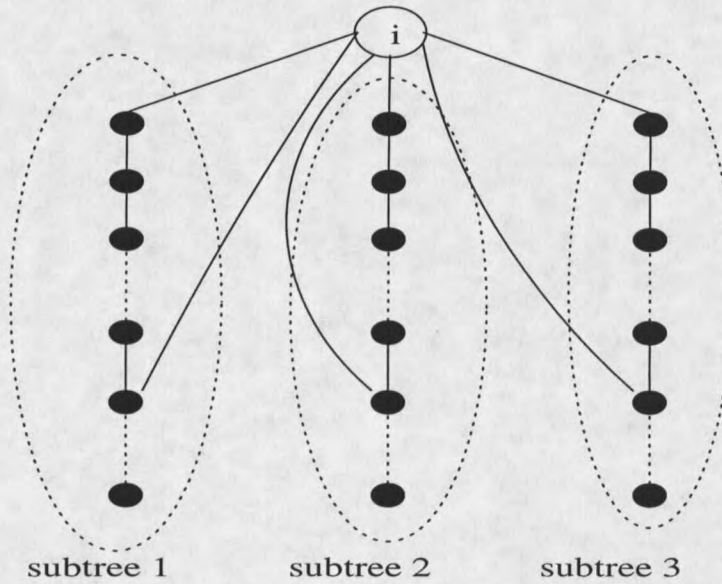


Figure 3.2: Cut vertex i and the subtrees

beginning of each iteration, the vertices which are adjacent to vertex i are marked. If a marked vertex is a v -node, then the v -node is called a *marked v -node*. If the vertex is in a c -node then the c -node is called a *marked c -node*.

The *external degree* of a vertex v at the i^{th} iteration is the number of its neighbors among $\{i+1, \dots, n\}$. A vertex with an external degree of 0 has no back edges to vertices $\{i+1, \dots, n\}$.

A *vertex contraction procedure* eliminates the vertices which are leaves and have an external degree of 0. At the beginning of each iteration, the external degree of each vertex which is adjacent to vertex i will be reduced by one. Then the vertex contraction procedure is performed to contract the vertices which have external degree 0. During the vertex contraction procedure, if we find a leaf vertex u with an external degree of 0 then we delete the vertex and connect its parent node with the vertex i . This new edge will represent the path $parent(u), u, i$. During the vertex contraction procedure, if any vertex

with an external degree of 0 becomes a leaf then the vertex can be contracted. Because each *marked* vertex is checked in ascending order, it can be guaranteed that every vertex will not be checked before all of its marked children have been checked.

Figure 3.3 shows how the vertex contraction procedure works.

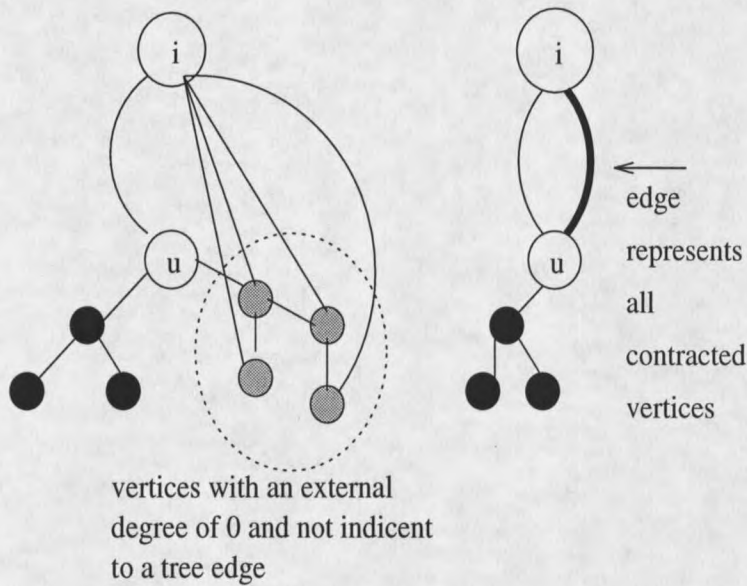


Figure 3.3: A tree before contraction and after contraction

A marked v-node is called a *terminal node* if none of its descendants is marked. A marked c-node is called a *terminal node* if none of its descendants is marked. Figure 3.4 shows a terminal v-node and a terminal c-node.

A c-node is called an *intermediate c-node* if the c-node appears on the tree path from the last inserted vertex to one of the marked nodes. In each intermediate c-node, there exist exactly two vertices adjacent to the two tree edges. We call these two vertices *high P-vertex* h_1 and *low P-vertex* h_2 , respectively, according to their postordering number. Both the *high P-vertex* h_1 and the *low P-vertex* h_2 are referred to as *P-vertex*. All other vertices in the c-node are

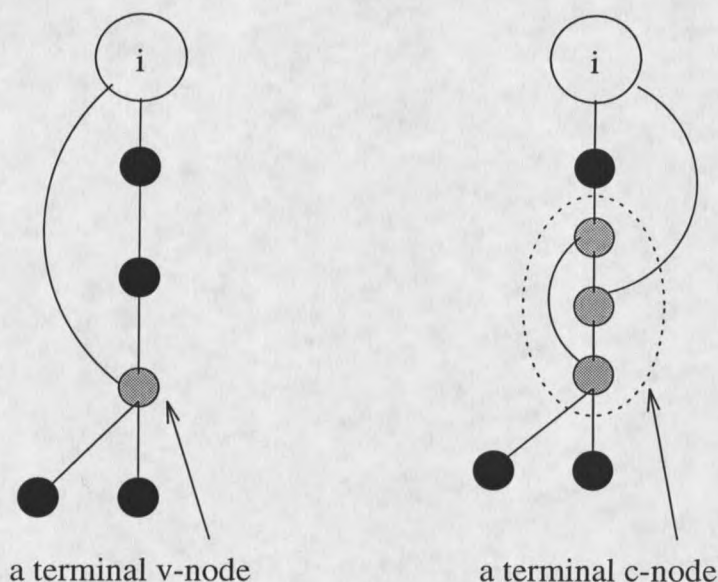


Figure 3.4: A terminal v-node and a terminal c-node

called *non-P-vertex*. For every non-P-vertex in an intermediate c-node, the *upward direction* is defined as the direction which leads from the non-P-vertex to the h_1 first, and the *downward direction* is defined as the direction which leads from the non-P-vertex to the h_2 first. Any tree edge, once it is included into a c-node, will never be called a tree edge again. Figure 3.5 shows an intermediate c-node, h_1 and h_2 , and the upward and downward directions.

3.2 Theorems

This section discusses several theorems. These theorems are the basis for this implementation.

Theorem 1. If G is planar, then there exist at most two terminal nodes in T_j .

Proof: Let G be a graph which has n vertices. Suppose at the i^{th} iteration, there exist more than two terminal nodes. Pick three of them and denote them

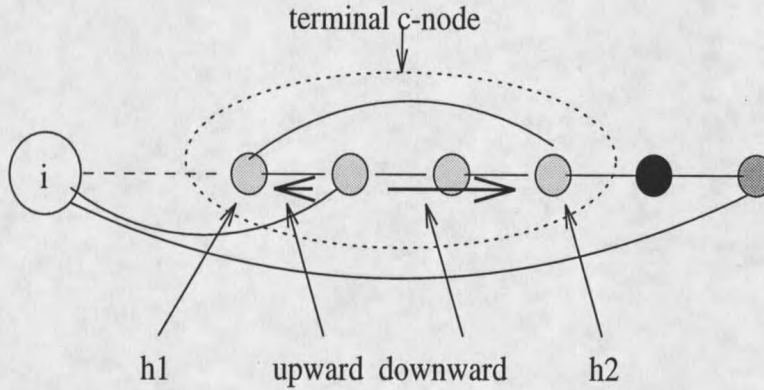


Figure 3.5: An intermediate c-node, h_1 and h_2 , upward and downward direction

as 1, 2, and 3, respectively. By the definition of the terminal nodes, it is known there are some children adjacent to the vertices which are among the vertices $\{i+1, \dots, n\}$. Choose three from these vertices for 1, 2, and 3, respectively, and label the vertex which contains the medium postordering number as t . Let the least common ancestor of 1, 2, and 3 be r . Then a subgraph homeomorphic to $K_{3,3}$ can be found. By *Kuratowski's Theorem*, it is known that this graph is not planar. Figure 3.6 shows the forbidden structure. $\square$

Theorem 2.1. Each marked non-P-vertex u in a c-node must either be adjacent to the *high P-vertex* h_1 , or every vertex between u and h_1 must have an external degree of 0 and must not be incident to a tree edge.

Proof: Let G be a graph which has n vertices. At the i^{th} iteration, suppose the vertex u is a marked vertex in the c-node. Consider two cases: (1) the c-node is an intermediate c-node, or (2) the c-node is an end c-node.

Case (1): Because the c-node is an intermediate c-node, from vertex u in the downward direction, the *low-P-vertex* h_2 will be met first. This means u cannot be adjacent to the *high-P-vertex* h_1 from the downward direction. Now suppose there is a vertex u' between the vertex u and the *high-P-vertex* h_1

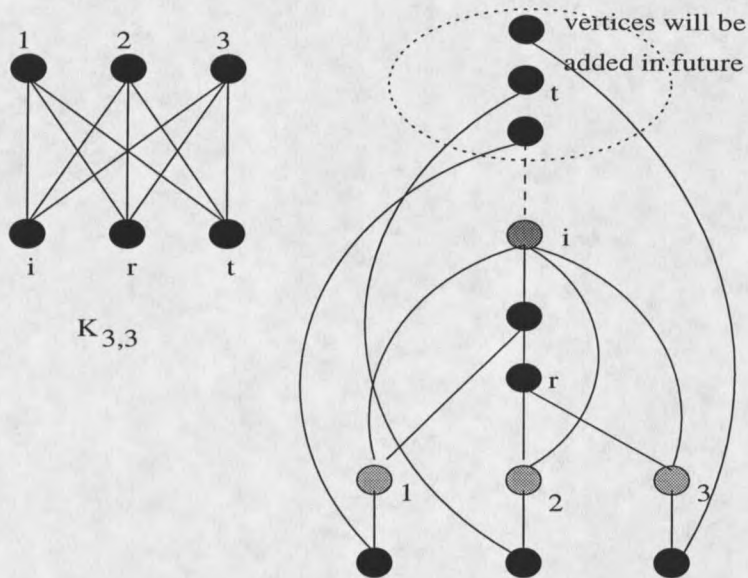


Figure 3.6: A forbidden structure

in the upward direction. Figure 3.7 shows that a subgraph homeomorphic to $K_{3,3}$ can be found.

Case (2): In this case, because the c-node is an end c-node, there is no *low-P-vertex* in the c-node. The marked non-P-vertex u may be adjacent to the *high P-vertex* in either direction. Now suppose there are two vertices, u' and u'' , between the non-P-vertex u and the *high-P-vertex* h_1 in either direction, respectively. Figure 3.8 shows that a subgraph homeomorphic to $K_{3,3}$ can be found. $\square$

Theorem 2.2. If a c-node is an intermediate c-node, then h_1 and h_2 must either be adjacent to each other or all vertices between them must have an extern degree of 0 and must not be incident to a tree edge.

Proof: Suppose there is an intermediate c-node, and h_1, h_2 are not adjacent to each other. Then there must exist two non-P-vertices, u_1 and u_2 , arranged as $h_1 u_1 h_2 u_2$ in the c-node, and both u_1 and u_2 have either a non-zero extern

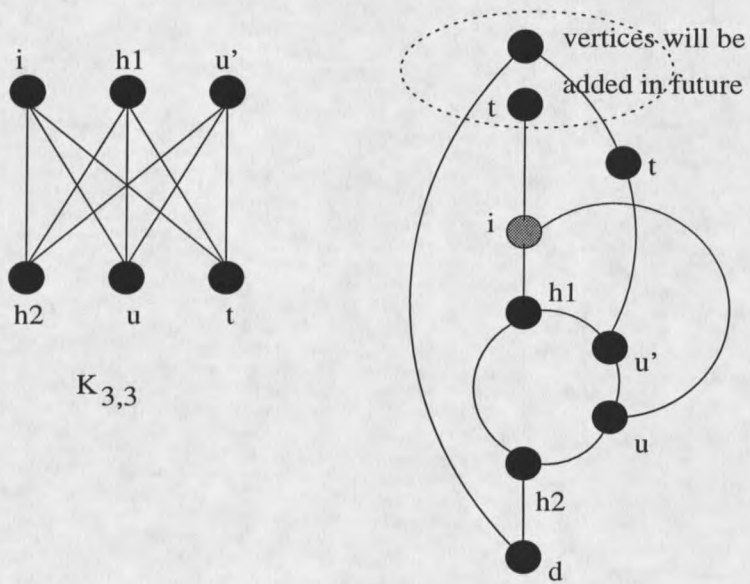


Figure 3.7: A forbidden structure

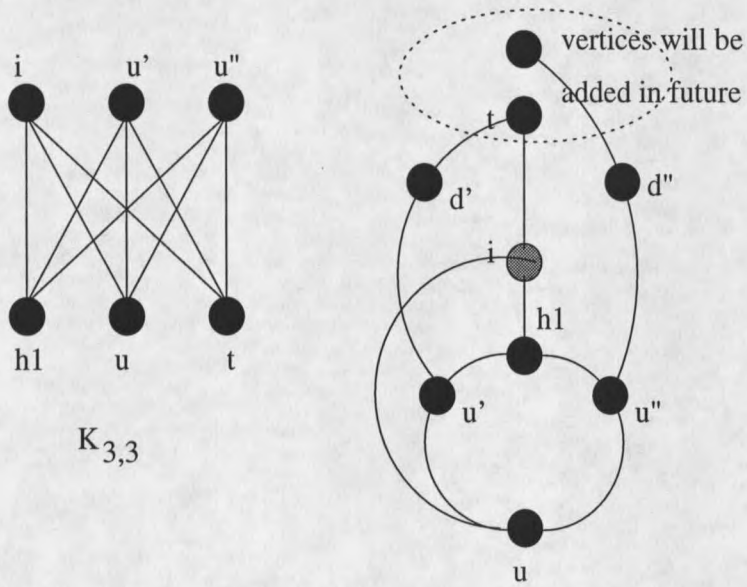


Figure 3.8: A forbidden structure

degree or induce to a tree edge. That means both u_1 and u_2 have a path to one of the vertices $\{i+1, \dots, n\}$. Figure 3.9 shows that a subgraph homeomorphic to $K_{3,3}$ can be found. $\square$

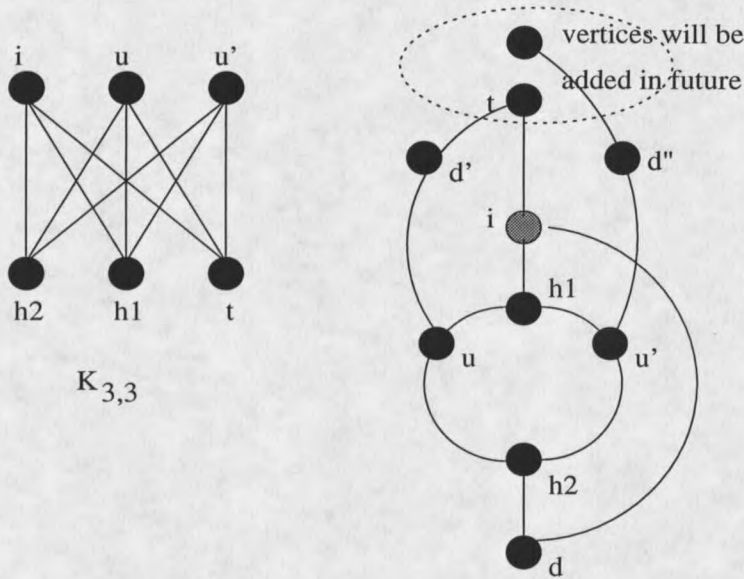


Figure 3.9: A forbidden structure

Theorem 3. At the i^{th} iteration, if there are two terminal nodes u_1 and u_2 , let the least common ancestor be u_3 . Then all vertices between vertex u_3 and i must have an external degree of 0 and must not be incident to a tree edge.

Proof: Suppose there is a vertex v between vertex u_3 and i with a non-zero external degree. This means vertex v is adjacent to one of the vertices $\{i+1, \dots, n\}$. Figure 3.10 shows that a subgraph homeomorphic to $K_{3,3}$ can be found.

Similarly, if vertex v is incident to a tree edge then vertex v connects to one of the vertices $\{i+1, \dots, n\}$ via a child vertex. We can get the same subgraph of Figure 3.10. $\square$

At the i^{th} iteration, if there are two terminal nodes u_1 and u_2 , and their

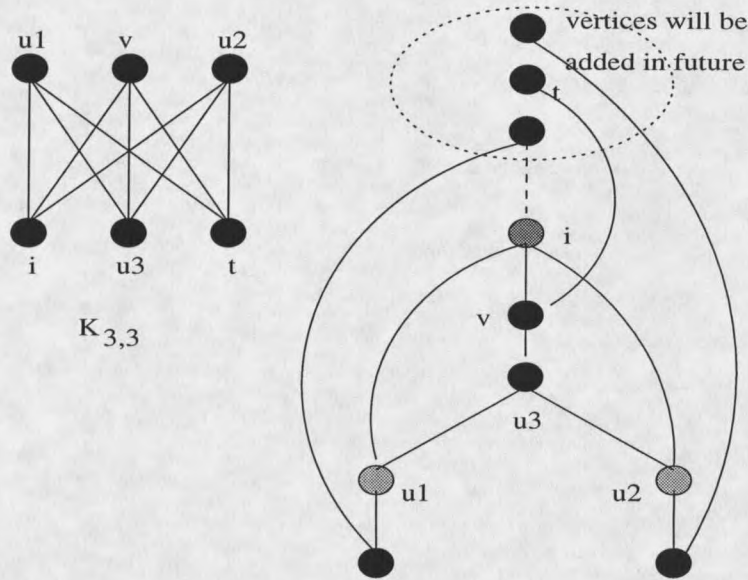


Figure 3.10: A forbidden structure

least common ancestor is a c-node, then two cycles will be found. In the c-node there exist two vertices, v_1 and v_2 , which connect with u_1 and u_2 , respectively. The *inner cycle* C_1 is defined as the cycle which contains the path $u_1v_1v_2u_2$ with the back edges (u_1, i) , (u_2, i) , and not passing through h_1 . The *outer cycle* C_2 is defined as the cycle which contains the path $u_1v_1h_1v_2u_2v_2$ with the back edges (u_1, i) , (u_2, i) . Figure 3.11 shows an example.

Theorem 4. At the i^{th} iteration, if cycles C_1 and C_2 are found, and if there is a vertex v which is on C_1 but which does not appear on C_2 , then the vertex v must have an external degree of 0 and must not be incident to a tree edge.

The proof of Theorem 4 is similar to the proof of Theorem 3.

Theorem 5. For a given graph G , if G_i (the graph which contains vertices $\{1, 2, \dots, i\}$) satisfies the previous four theorems at each vertex-adding iteration, then the graph G can be embedded in the plane.

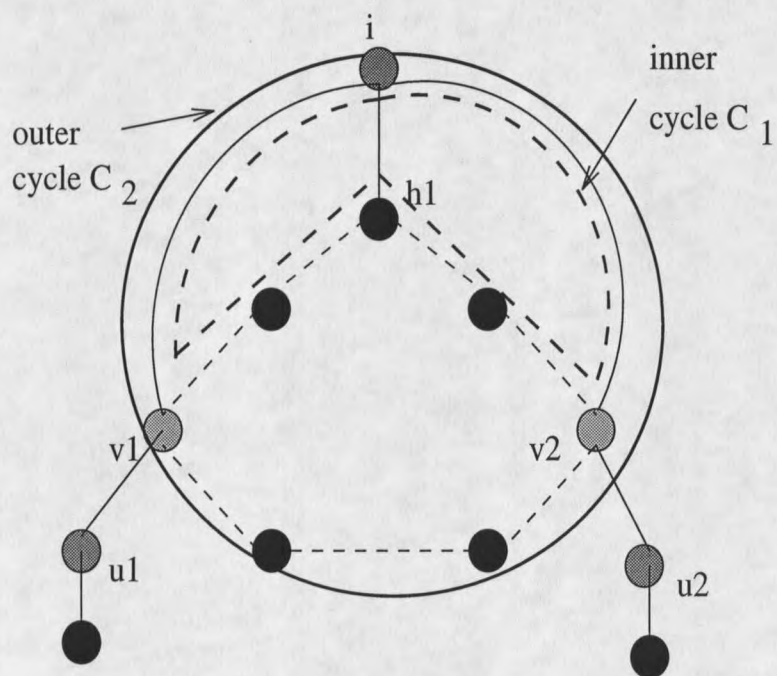


Figure 3.11: An inner cycle C_1 and an outer cycle C_2

Proof : If an embedding for each G_i can be found, such that all vertices which are adjacent to vertices $\{i+1, \dots, n\}$ are kept on the border of the cycle C_i which is created at the i^{th} iteration, then the graph G can be embedded in the plane.

Now consider two cases: (1) there is only one terminal node; (2) there are two terminal nodes.

(1) There is only one terminal node u .

First assume u is a v-node. Let P be the path from root r_j to u . The path P is arranged with the back edge (i, u) and tree edge (i, r_j) functioning as the border of the cycle. If, on the path P , there is some c-node, then by Theorem 2.2, h_1 and h_2 must be adjacent. Therefore, all vertices of the c-node can be arranged on the border of C_i . See Figure 3.12.

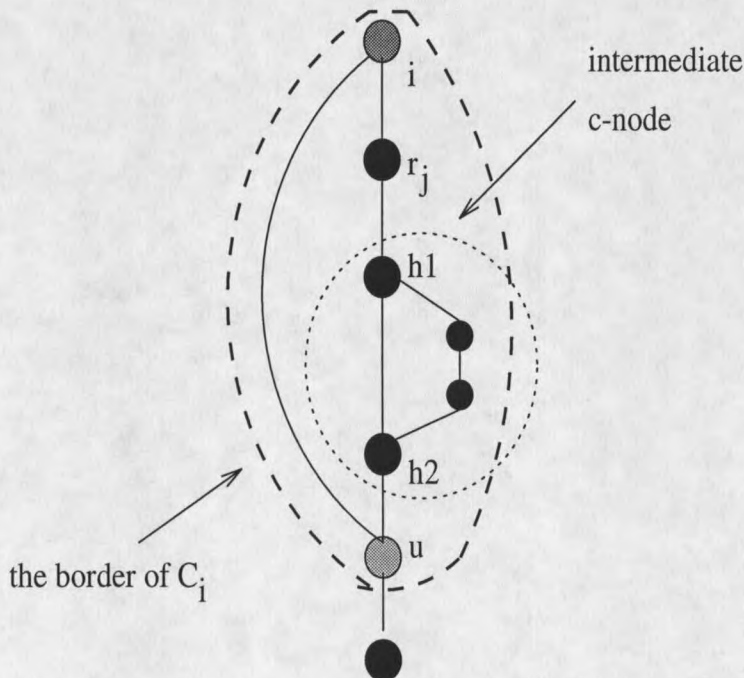


Figure 3.12: Border of cycle C_i with one terminal v-node

Otherwise, if u is in a c-node, then according to Theorem 2.1, u must be adjacent to the vertex h_1 . Path uh_2h_1 is chosen with the path from r_jh_1 and the back edge (i,u) and the tree edge (i,r_j) to be the border of C_i . See Figure 3.13.

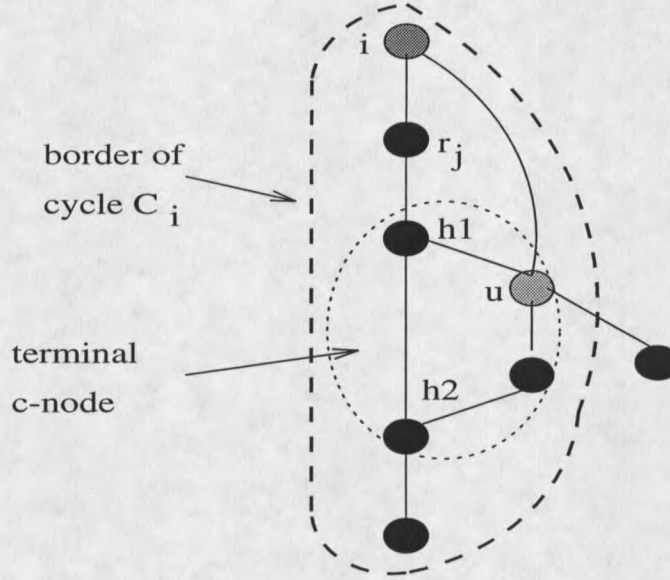


Figure 3.13: Border of cycle C_i with one terminal c-node

(2) There are two terminal nodes u_1 and u_2 .

(a) If both terminal nodes are v-nodes and the least common ancestor u is a v-node, then the border of the cycle will contain the path u_1, u, u_2 and back edges (u_1, i) , (u_2, i) . From Theorem 3, it is known that every vertex between i and u will have an external degree of 0 and must not be incident to a tree edge. Therefore, every vertex which is adjacent to vertices $\{i+1, \dots, n\}$ will be on the border of C_i . See Figure 3.14.

(b) If both terminal nodes are v-nodes and the least common ancestor u is a c-node, then two cycles C_1 and C_2 will be formed. C_2 is used as the border of C_i . From the definition of C_2 and the Theorem 4, every vertex which is

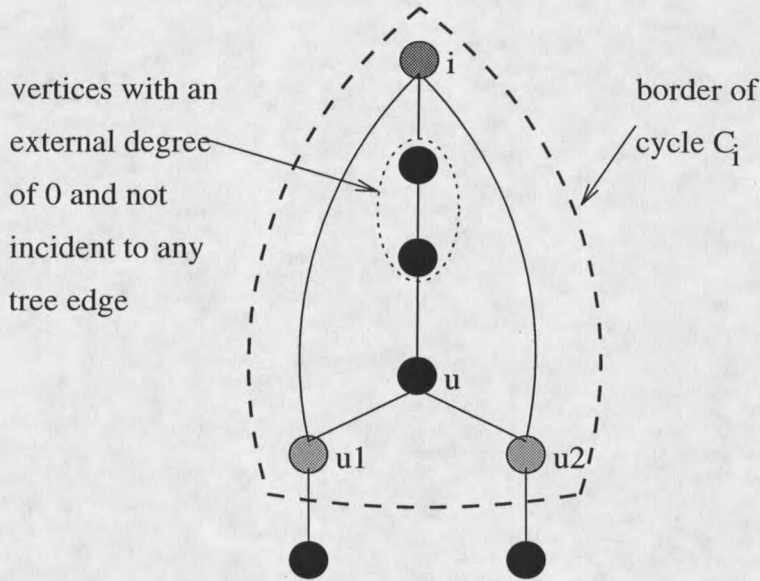


Figure 3.14: Two terminal v-nodes u_1, u_2 and v-node u .

adjacent to vertices $\{i+1, \dots, n\}$ will be on the border of C_i . See Figure 3.15.

The case of both terminal nodes u_1, u_2 in the same c-node is similar to the above case.

(c) If two terminal nodes, u_1, u_2 , are in different c-nodes and the least common ancestor u is a c-node, then cycle C_2 is used as the border of C_i . From the definition of C_2 and Theorem 4, it is known that every vertex which is adjacent to vertices $\{i+1, \dots, n\}$ will be on the border of C_i . See Figure 3.16.

Finally, there could exist two terminal nodes, u_1, u_2 , in different c-nodes and the least common ancestor is a v-node. This case is similar to the above case. $\square$

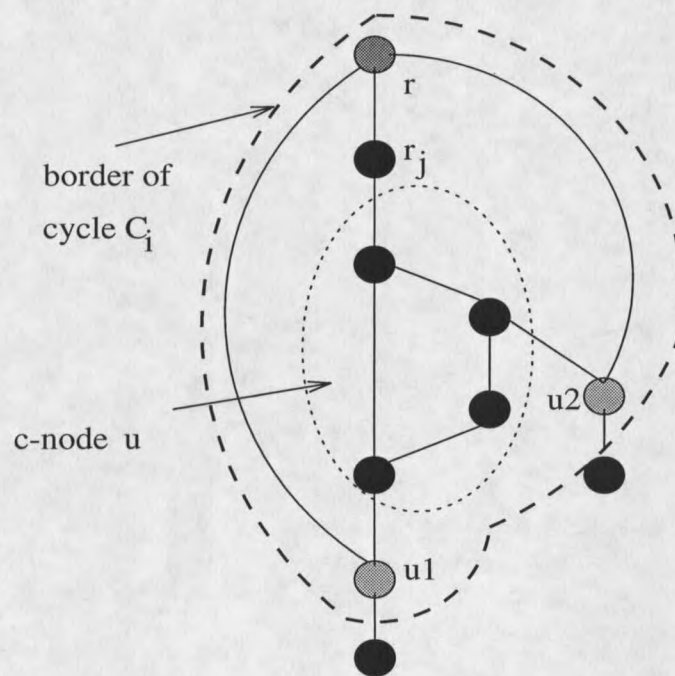


Figure 3.15: Two terminal v-nodes u_1, u_2 and u is a c-node.

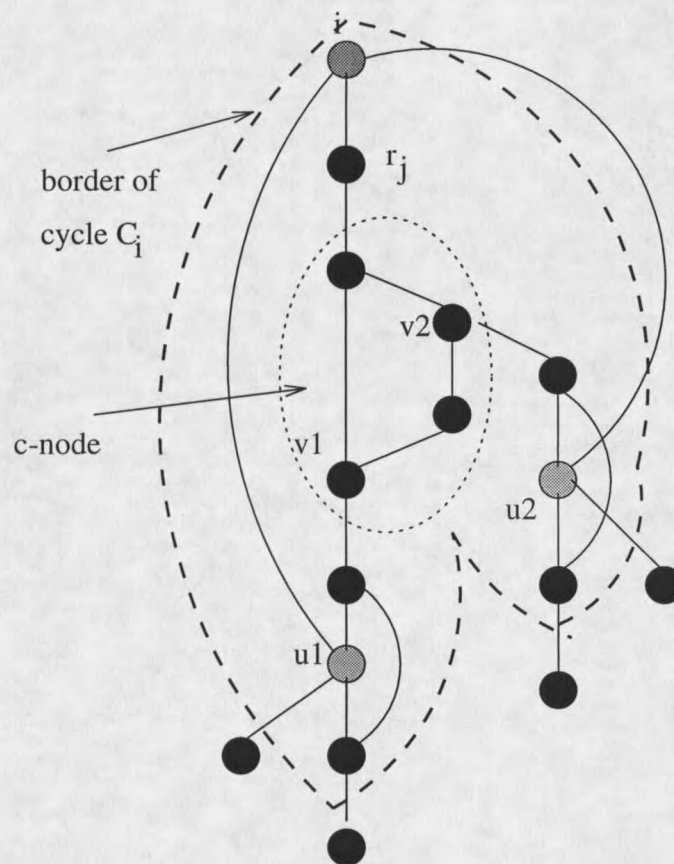


Figure 3.16: Two terminal nodes, u_1 , u_2 , in different c-node and u is a c-node.

Chapter 4

Data Structures

This chapter describes the data structures used in the implementation. Like other implementations of planarity testing and maximal planar subgraph algorithms, the data structures are complex and play an important role in the efficiency of the implementation.

4.1 Representations of Graphs

In this implementation, an adjacency list is used to represent the graph. Let $G = (V, E)$ be a graph with n vertices and m edges. The adjacency list representation is an array Adj of pointers. The array Adj has n members. Each vertex u in the graph G has an entry $Adj[u]$ which contains a pointer to a chain of all the vertices which are adjacent to vertex u .

The input to this program is a text file which contains the number of vertices n in the graph on the first line; each subsequent line represents a vertex in the graph. Vertex u is represented by the $(u + 1)^{th}$ line. This line contains the vertex numbers of the vertices which are adjacent to vertex u . Those numbers are separated by spaces, and the end of the list is marked with a 0.

Figure 4.1 shows an example of the input file and adjacency list represen-

tation of K_5 .

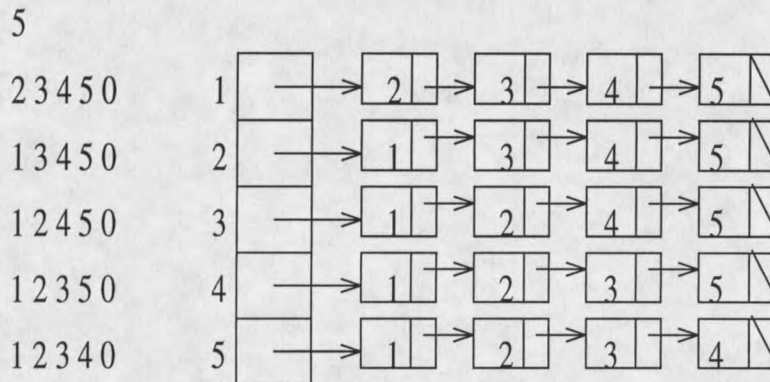


Figure 4.1: The input file of K_5 and its adjacency list representation

Another common way to represent a graph is using an adjacency matrix representation. The adjacency matrix representation of a graph $G = (V, E)$ is an $n \times n$ matrix A . For each a_{ij} , if $(i, j) \in E$ then $a_{ij} = 1$ else $a_{ij} = 0$.

When the graph is a sparse graph, the adjacency list representation is more efficient in memory usage than the adjacency matrix representation. The major disadvantage of the adjacency list representation is that there is no quick way to determine if a given edge (u, v) is present in the graph. We have to search $Adj[u]$ to determine that.

4.2 Representation of the Depth-First Search Tree

Figure 4.2 shows the basic structure of the representation of the depth-first search tree.

In each tree node, a chain for tree edges and a chain for back edges is kept. Each chain is sorted in ascending order of the postordering numbers of the vertices. Each node in the tree can be visited by the tree edge chain.

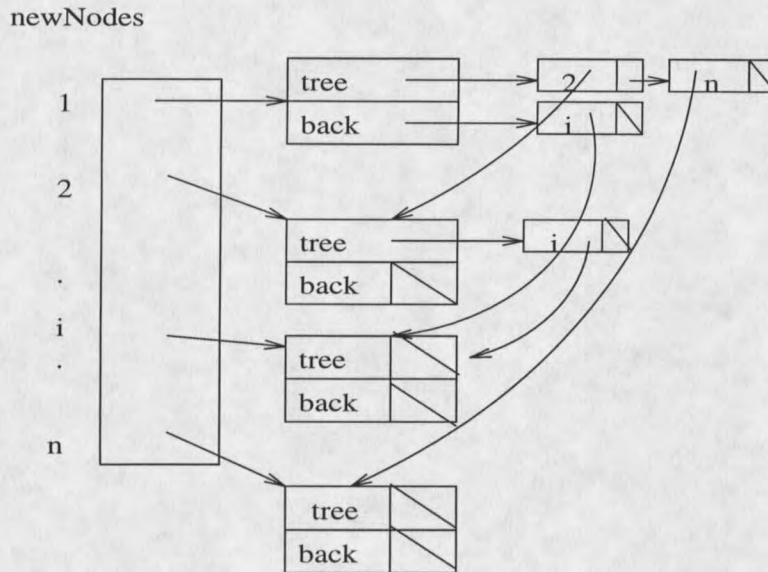


Figure 4.2: Representation of a depth-first search tree

A pointer array, *newNodes*, is used to store the pointers to each tree node. The index into the pointer array *newNodes* is the postordering number of the vertex. Thus, if the postordering number of the vertex which the program wants to visit is known, the program can visit the vertex directly by using the array *newNodes*. By using the *newNodes* array, we can avoid travel along the tree edges when finding a vertex.

In each tree node we also store the external degree, the parent's postordering number, the node type, and the "removed" mark.

Chapter 5

Shih and Hsu's Planarity Testing Algorithm

This chapter describes the details of the planarity testing algorithm and discusses the time complexity of the algorithm. The major goal of this implementation is to have the algorithm run in linear time. This means that for each edge, the algorithm can only traverse the edge a constant number of times.

5.1 Outline of the Algorithm

Following is the outline of the algorithm:

1. Create a depth-first search tree;
2. Generate a postordering of the vertices in the depth-first search tree;
3. Create a sorted tree edge chain and a sorted back edge chain for each vertex;
4. Set a *contraction step number* by a preorder traversal of the depth-first search tree. If the *contraction step number* of a vertex v is k , v cannot be contracted before vertex k added to the subgraph;
5. Add vertices one by one according to the postordering;
- 5.1 Mark every vertex which is adjacent to the newly-added vertex i ;

5.2 Perform the vertex contraction procedure on each marked vertex according to the postordering;

5.3 Check each vertex v which is adjacent to the newly-added vertex i by a back edge according to the postordering;

5.3.1 Choose an unchecked *marked* vertex v which has the smallest postordering number;

5.3.2 If vertex v is in a c-node, check if it is adjacent to the *high P-vertex* of the c-node. If it is not adjacent to the *high P-vertex* of the c-node then, output "nonplanar" (Theorem 2.1) and halt;

5.3.3 If vertex v is adjacent to the *high P-vertex* of the c-node and at least one of its children has not been contracted, check $tNodes$. If $tNodes = 2$, then output "nonplanar" (Theorem 1 and the definition of the terminal nodes) and halt else set $tNodes = tNodes + 1$, remember the vertex v as one of the two terminal nodes, and travel from vertex v to i along the tree path, marking the label of every vertex encountered in the path to i . If any c-node is found along this path then check if the *high P-vertex* and *low P-vertex* are adjacent to each other. If it is not then output "nonplanar" (Theorem 2.2) and halt, else jump to the *high P-vertex* to continue the traversal;

5.3.4 If the label of the vertex v is not i , then check the number of terminal nodes $tNodes$. If $tNodes = 2$, then output "nonplanar" (Theorem 1), else set $tNodes = tNodes + 1$, remember the vertex v as one of the two terminal nodes, and travel from vertex v to vertex i along the tree path, marking the label of every vertex encountered in the path to vertex i . If any c-node is found during the traversal, then check if the *high P-vertex* and *low P-vertex* are adjacent to each other. If not then output "nonplanar" (Theorem 2.2) and halt, else jump to the *high P-vertex* to continue the traversal;

5.3.5 Create a new c-node by using the terminal nodes found in previous

checking;

If there is only one terminal node, then create the c-node by including all vertices along the tree path from v to i ;

If there are two terminal nodes u_1, u_2 and the least common ancestor u_3 is a v-node, then create the c-node from the two terminal nodes and the least common ancestor. The c-node will include all vertices along the tree path u_1 to u_2 . If there are any uncontracted vertices between the least common ancestor u_3 and vertex i , then output "nonplanar" (Theorem 3) and halt;

If there are two terminal nodes u_1, u_2 and the least common ancestor u_3 is a c-node, then create the c-node from the two terminal nodes and the least common ancestor. The c-node will include all vertices along the tree path u_1 to v_1, u_2 to v_2 , where v_1, v_2 are the vertices in the c-node and are connected with u_1, u_2 , respectively. If there are any uncontracted vertices on cycle $C_1 - C_2$, then output "nonplanar" (Theorem 4) and halt.

5.3.6 If all vertices have been added to the subgraph then output "The graph is planar" else go to 5.3.1.

5.2 Time Complexity of the Algorithm

Suppose a given graph $G = (V, E)$ has n vertices and m edges. *Step 1*, creating a depth first tree, will take $O(m + n)$ steps. *Step 2*, postordering each vertex in the depth-first search tree by using a depth-first search, will take $O(m + n)$ steps.

Step 3, creating the sorted tree and back edge chain, means the vertices must be visited in ascending order of postordering. Each vertex is visited and each edge is traversed once. Therefore, the time complexity of this step is $O(m + n)$.

Step 4 can be done by a preorder traversal on the depth-first search tree, which means the time complexity of this step is $O(m + n)$.

Step 5 is the most important step in the algorithm. The vertex contraction process will take at most $O(n)$ steps because each vertex can be contracted at most once. *Step 5.3.2* always requires a constant number of steps. From *step 5.3.3* and *5.3.4*, it can be seen that all tree edges will be traversed at most a constant number of times because any tree edge which is included in a c-node will never be traversed again. In *step 5.3.5*, checking if the least common ancestor u_3 is adjacent to the newly-added vertex i or checking if there is any vertex in $C_1 - C_2$ is only done a constant number of times. The c-node creating phase will traverse each tree edge between the root and the marked vertex once, so the number of times a tree edge is traversed is a constant.

Summing up all the steps, this program has the time complexity $O(m + n)$.

Chapter 6

Hsu's Maximal Planar Subgraph Algorithm

This chapter describes the details of the maximal planar subgraph algorithm. This algorithm is based on the planarity testing algorithm. The major difference is that whenever a back edge whose addition causes the planarity testing algorithm to halt, instead of stopping, the edge is marked as “removed”. When the final planar subgraph is output, these edges will be marked as removed edges.

6.1 Outline of the Algorithm

Following is the outline of the algorithm:

1. Create a depth-first search tree;
2. Generate a postordering of the vertices in the depth-first search tree;
3. Create a sorted tree edge chain and a sorted back edge chain for each vertex;
4. Set a *contraction step number* by a preorder traversal of the depth-first search tree. If the *contraction step number* of a vertex v is k , v cannot be contracted before vertex k has been added to the subgraph;
5. Add vertices one by one according to the postordering;

5.1 Mark every vertex which is adjacent to the newly-added vertex i ;

5.2 Perform the vertex contraction procedure on each marked vertex according to the postordering;

5.3 Check each vertex v which is adjacent to the newly-added vertex i by a back edge according to the postordering.

5.3.1 Choose an unchecked *marked* vertex v which has the smallest postordering number;

5.3.2 If vertex v is marked "removed" then mark the back edge (v,i) as "removed" in the back edge chain of vertex i , and go to 5.3.1;

5.3.3 If vertex v is in a c-node, then check if it is adjacent to the *high P-vertex* of the c-node. If it is not adjacent to the *high P-vertex* of the c-node, then mark the back edge (v,i) as "removed" (Theorem 2.1) in the back edge chain of vertex i , and go to 5.3.1;

5.3.4 If vertex v is adjacent to the *high P-vertex* of the c-node and all of its children have not been contracted, check $tNodes$. If $tNodes=2$, then mark the back edge (v,i) as "removed" (Theorem 1 and the definition of the terminal nodes) and go 5.3.1, else set $tNodes=tNodes+1$, remember the vertex v as one of the two terminal nodes and travel from vertex v to i along the tree path, marking the label of every vertex encountered in the path to i . If any c-node is found along this path then check if the *high P-vertex* and *low P-vertex* are adjacent to each other. If they are not, then mark the back edge (v,i) as "removed" (Theorem 2.2) and go to 5.3.1;

5.3.5 If the label of the vertex v is not equal to i then check $tNodes$. If $tNodes = 2$, then mark the back edge (v,i) as "removed" (Theorem 1) and go to 5.3.1, else set $tNodes=tNodes+1$, remember the vertex v as one of the two terminal nodes and travel from vertex v to vertex i along the tree path, marking the label of every vertex encountered in the traversal to vertex i . If

any c-node is found during the traversal then check if the *high P-vertex* and *low P-vertex* are adjacent to each other. If they are not then mark the back edge (v, i) as "removed" (Theorem 2.2) and go to 5.3.1;

5.3.6 Create a new c-node by using the terminal nodes found in previous checking;

If there is only one terminal node, then create the c-node by including all vertices along the tree path from v to i ;

If there are two terminal nodes, u_1, u_2 , and the least common ancestor u_3 is a v-node, then create the c-node from the two terminal nodes and the least common ancestor. The c-node will include all vertices along the tree path u_1 to u_2 . Mark all vertices between the least common ancestor u_3 and the vertex i as "removed" (Theorem 3) and go to 5.3.1;

If there are two terminal nodes, u_1, u_2 , and the least common ancestor u_3 is a c-node, then create the c-node from the two terminal nodes and the least common ancestor. The c-node will include all vertices along the tree path u_1 to v_1, u_2 to v_2 , where v_1, v_2 are the vertices in the c-node and connected with u_1, u_2 , respectively. Mark all vertices on cycle $C_1 - C_2$ as "removed" (Theorem 4) and go to 5.3.1;

6. Output the final graph. In this step, the final maximal planar subgraph will be output. The vertices will be printed in ascending order of postordering number. Every tree edge will be included in the final graph. Every back edge removed by the algorithm will be output with a mark 'r'.

6.2 Time Complexity of the Algorithm

This analysis is similar to the analysis of the time complexity of the planarity testing algorithm. The maximal planar subgraph algorithm also runs in linear time.

Chapter 7

Test Results

7.1 Test Graphs

A few special graphs were used for testing and the results were compared with other algorithms.

The first test case is graph g_1 , shown in Figure 7.1. Graph g_1 has a maximum planar subgraph obtained by removing edges (4,8) and (7,8). This maximal planar subgraph program obtained a maximal planar subgraph by removing edges (2,7), (2,8), (2,9), (3,7), (3,8), (3,9), (4,7), (4,8), and (4,9). Figure 7.2 shows the result.

The second test case is graph g_2 , shown in Figure 7.3. Graph g_2 has a maximum planar subgraph obtained by removing edge (u,v). This maximal planar subgraph program obtained a maximal planar subgraph by removing the ten edges which are crossing with edge (u,v). Figure 7.4 shows the result.

The third test case is graph g_3 , shown in Figure 7.5. Graph g_3 has a maximum planar subgraph obtained by removing edges (1,28) and (9,12). This maximal planar subgraph program obtained a maximal planar subgraph by removing edges (2,19), (2,20), (3,20), (3,21), (4,21), (4,22), (5,22), (5,23), (6,23), (7,23), (7,24), (8,11), (12,24), (13,24), (13,25), (14,25), (14,26), (15,26), (15,27), and (16,27). Figure 7.6 shows the result.

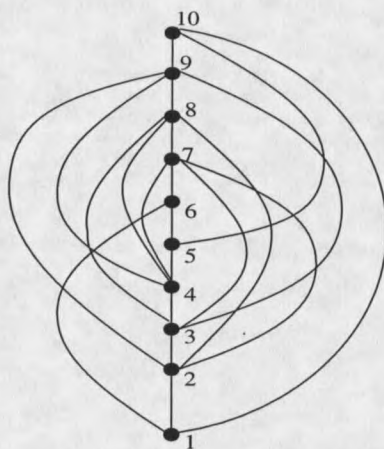


Figure 7.1: Graph g_1

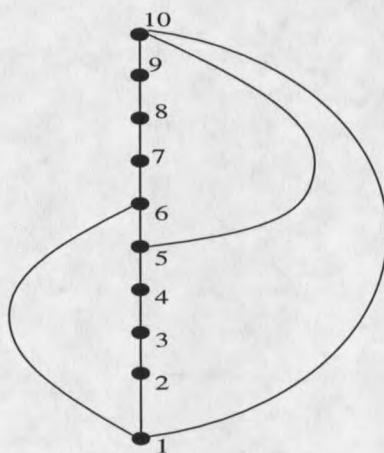
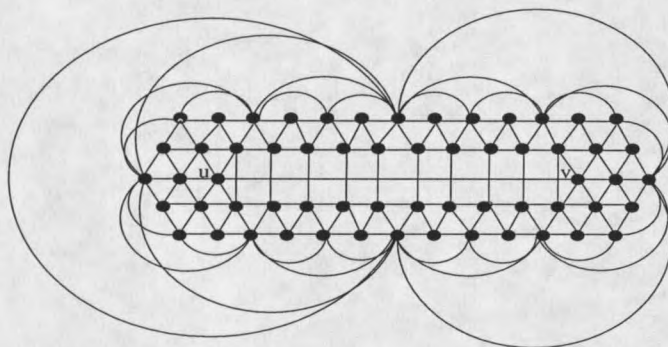
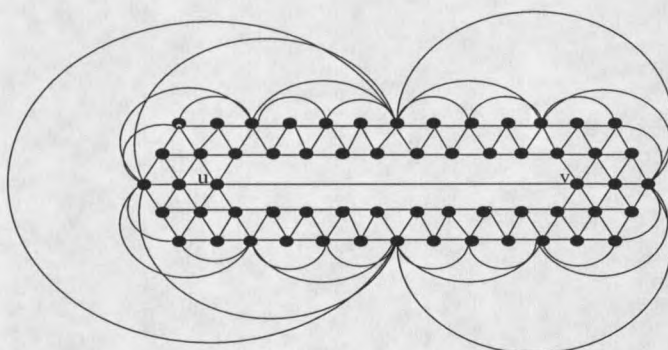
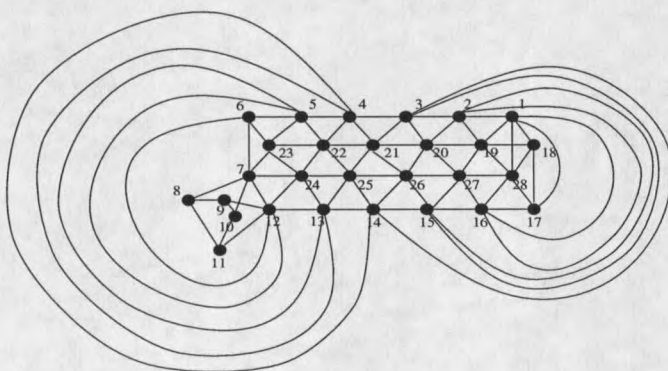
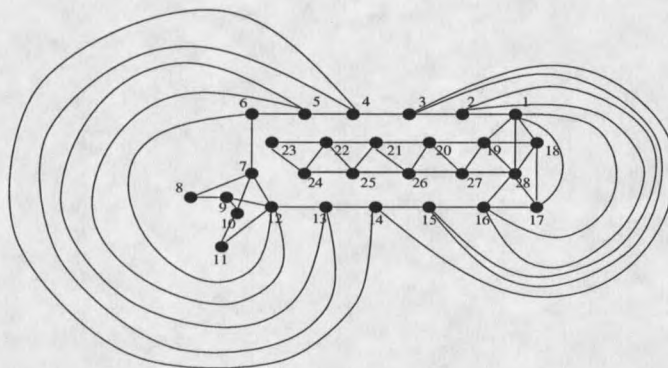
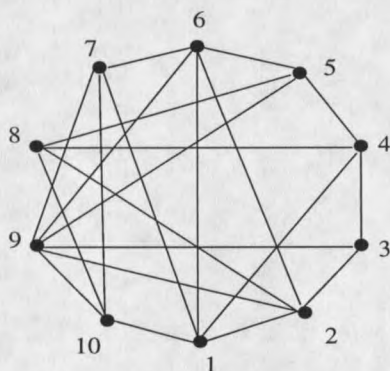


Figure 7.2: A maximal planar subgraph of g_1

Figure 7.3: Graph g_2 Figure 7.4: A maximal planar subgraph of g_2 Figure 7.5: Graph g_3

Figure 7.6: A maximal planar subgraph of g_3

The fourth test case is graph g_4 , shown in Figure 7.7. Graph g_4 has a maximum planar subgraph formed by removing edges $(1,4)$ and $(8,10)$. This maximal planar subgraph program obtained a maximal planar subgraph by removing edges $(2,8)$, $(2,9)$, $(3,9)$, and $(4,8)$. Figure 7.8 shows the result.

Figure 7.7: Graph g_4

The fifth test case is graph g_5 , shown in Figure 7.9. Graph g_5 has a maximum planar subgraph obtained by removing edges $(1,2)$, $(3,4)$ and $(4,5)$. This maximal planar subgraph program obtained a maximal planar subgraph by removing edges $(2,44)$, $(3,43)$, $(4,42)$, $(5,41)$, $(6,40)$, $(8,41)$, $(9,42)$, $(10,43)$,

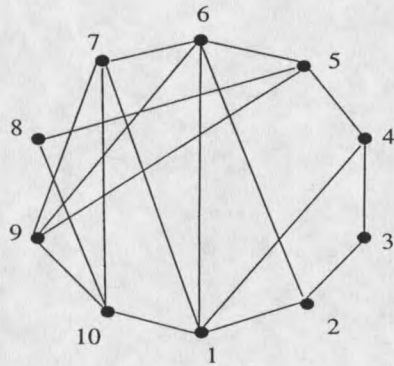
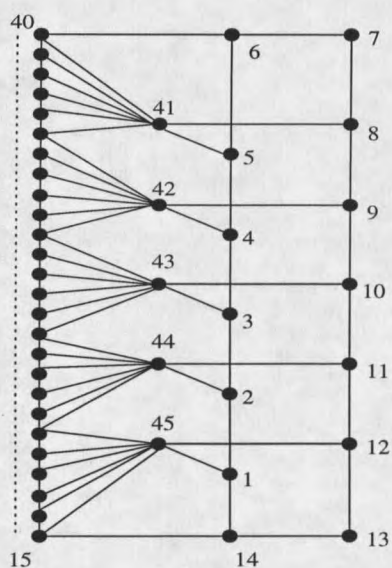
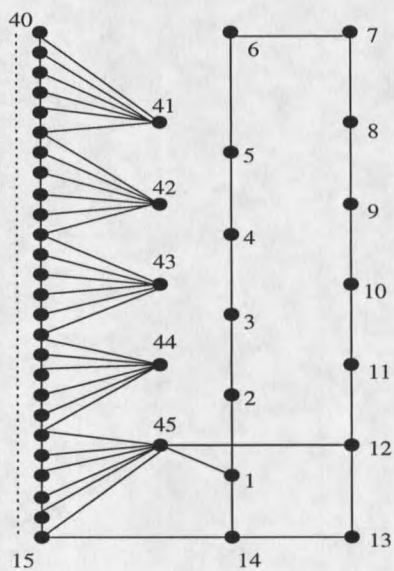
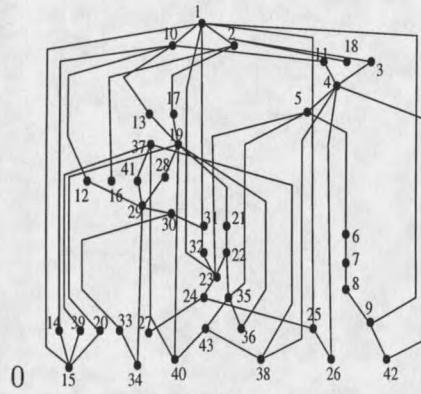
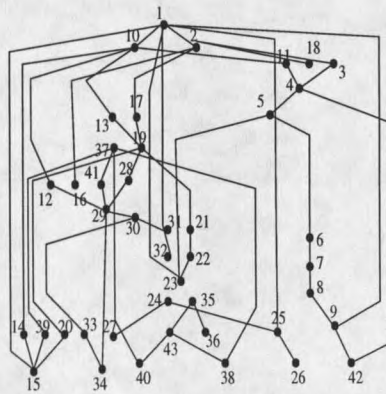


Figure 7.8: A maximal planar subgraph of g_4

and (11,44). Figure 7.10 shows the result.

The sixth test case is graph g_6 , shown in Figure 7.11. Graph g_6 has a maximum planar subgraph obtained by removing edges (1,9), (1,31), (22,23) and (23,32). This maximal planar subgraph program obtained a maximal planar subgraph by removing edges (19,40), (19,36), (22,35), (23,32), (4,38), (4,26), and (5,35). Figure 7.12 shows the result.

Figure 7.9: Graph g_5 Figure 7.10: A maximal planar subgraph of g_5

Figure 7.11: Graph g_6 Figure 7.12: A maximal planar subgraph of g_6

7.2 Comparison with Other Algorithms

The maximal planar subgraph solution of our algorithm *HSU* were compared against solutions obtained by other algorithms. Two kinds of graphs were used: the special graphs g_1 to g_6 and a set of 20 random graphs.

The random graphs are generated from K_3 . We start from K_3 and at each step we insert a new vertex and three new edges inside an arbitrary face. We created ten 100-vertex graphs and ten 200-vertex graphs. Each of them contained a maximum planar subgraph of size $3n - 6$. Then k (nonplanar) edges were added randomly, with k ranging from 10 to 100.

This section is based on the results of Cimikowski [8]. Table 7.1 shows the size of the maximal planar subgraphs found from special graphs g_1 to g_6 by each algorithm. Table 7.2 shows the size of the maximal planar subgraphs found from the random graphs.

From the table, we can see that the Goldschmidt-Takvorian algorithm obtained the best solutions for these algorithms. *HSU* did not perform as well as the PQ-tree heuristic and the Goldschmidt-Takvorian heuristic but our implementation has a faster running time.

In comparing the running times of these algorithms, we tried a 300-vertex, 1507-edge graph. We used a DEC Alpha 3000/500 system for our testing. We ran each program 5 times and computed the average running time. *HSU* found a solution in 0.12 CPU seconds. *CHT* found a solution in 0.135 CPU seconds. The remaining algorithms took a very long time to find solutions.

KEYS :

HT = Hopcroft-Tarjan heuristic

CHT = Cai-Han-Tarjan heuristic

PQ = PQ-tree heuristic

INC = Incremental heuristic

GT = Goldschmidt-Takvorian heuristic

HSU = Hsu heuristic

graph	n	m	opt. ¹	Size of planar subgraph					
				<i>HT</i>	<i>PQ</i>	<i>CHT</i>	<i>INC</i>	<i>GT</i>	<i>HSU</i>
g1	10	21	19	16	18	12	17	19	12
g2	60	166	165	160	164	154	165	149	156
g3	28	75	73	57	71	55	73	73	55
g4	10	22	20	16	19	19	18	19	19
g5	45	85	82	77	80	76	55	80	76
g6	43	63	59	59	56	55	50	54	56

¹ optimum solution

Table 7.1: Heuristics applied to dense nonplanar graphs

k^1	100 vertices ²						200 vertices ³					
	Size of planar subgraph obtained						Size of planar subgraph obtained					
	<i>HT</i>	<i>CHT</i>	<i>PQ</i>	<i>INC</i>	<i>GT</i>	<i>HSU</i>	<i>HT</i>	<i>CHT</i>	<i>PQ</i>	<i>INC</i>	<i>GT</i>	<i>HSU</i>
10	196	238	237	219	235	203	401	475	479	481	461	412
20	203	206	224	207	223	208	410	490	446	487	439	414
30	227	190	224	209	224	188	405	449	444	445	451	409
40	177	184	212	213	228	183	429	448	372	460	464	371
50	169	192	221	207	234	181	416	420	375	433	439	376
60	162	171	194	217	242	168	422	452	360	430	411	334
70	174	186	201	217	239	168	403	423	358	408	408	345
80	171	177	198	202	221	164	397	416	345	417	423	328
90	164	184	206	215	235	180	401	414	358	409	417	357
100	161	174	194	187	224	173	402	396	324	416	461	323

¹ the number of edges added to the maximum planar graph

² the optimum solution of the graph is 294

³ the optimum solution of the graph is 594

Table 7.2: Heuristics applied to random graphs

Bibliography

- [1] L. Auslander, and S. V. Parter, On embedding graphs in the plane, *J. Math. Mech.* 11 (1961), pp. 517-523.
- [2] J. Hopcroft, and R. E. Tarjan, Efficient planarity testing, *J. ACM* 21 (1974), pp. 549-568.
- [3] A. Lempel, S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, *Theory of Graphs, Int. Symp. Rome* (1966), pp. 215-232.
- [4] K. S. Booth, and G. S. Lueker, Testing for the consecutive ones property, interval graphs and graph planarity testing using PQ-tree algorithms, *J. Computer & System Sciences* 13 (1976), pp. 335-379.
- [5] M. R. Garey, and D. S. Johnson, *A Guide to the Theory of NP-completeness*, Freeman Co., San Francisco, 1979.
- [6] T. Chiba, I. Nishioaka and I. Shirakawa, An algorithm of maximal planarization of graphs, *Proc. 1979 IEEE Int. Symp. on Circuits and Systems* (1979), pp. 649-652.
- [7] J. Cai, X. Han and R. E. Tarjan, An $O(m \log n)$ -time algorithm for the maximal planar subgraph problem, *SIAM J. Comput.* Vol. 22 (6)(1993), pp. 1142-1162.
- [8] R. Cimikowski, An analysis of heuristics for graph planarization, *manuscript*, 1995.
- [9] W.-L. Hsu, A linear time algorithm for finding maximal planar subgraphs, *Proc. Sixth Annul Int. Symp. on Algorithms & Computation*, Cairns, Australia, 1995.
- [10] W.-K. Shih and W.-L. Hsu, A simple test for planar graphs, *manuscript*, 1993.

- [11] T. Nishizeki and N. Chiba, *Planar graphs : theory and algorithms*, North-Holland 1988.
- [12] H. N. Djidjev, A linear algorithm for the maximal planar subgraph problem, *Tech. Report TR95-247*, Dept of Computer Science, Rice Univ. 1995.
- [13] G. Di Battista and R. Tamassia, Incremental planarity testing, *Proc. IEEE Symp. Found. Comp. Sci.* (1989), pp. 598-611.
- [14] O. Goldschmidt and A. Takvorian, An efficient graph planarization two-phase heuristic, *Networks*, vol. 24, pp. 69-73, 1994.
- [15] R. Jayakumar, K. Thilasiraman and M. N. S. Swamy, $O(n^2)$ algorithms for graph planarization, *IEEE Trans. on Comput.-Aided Design*, vol. 8, pp. 257-267, 1989.
- [16] G. Kant, An $O(n^2)$ maximal planarization algorithm based on PQ-trees, *Tech. Report RUU-CS-92-03*, CS Dept., Univ. Utrecht, Netherlands, Jan., 1992.

MONTANA STATE UNIVERSITY LIBRARIES



3 1762 10240926 3