



MAXPLANAR : a graphical software package for testing maximal planar subgraph algorithms
by Kedan Zhao

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

Montana State University

© Copyright by Kedan Zhao (1996)

Abstract:

We present an efficient implementation of a software package, MAXPLA-NAR, with a user-friendly interface for several algorithms for finding maximal planar subgraphs of nonplanar graphs. The algorithms include the methods of path addition, edge addition, vertex addition, and cycle packing.

MAXPLANAR is designed to facilitate graph input and output and algorithm efficiency analysis. The result is an easy-to-use software package for researchers to test the various planarization algorithms.

Extensive empirical results are given for the heuristics on several families of nonplanar graphs. Type I are random nonplanar graphs with unknown maximum planar subgraph size. Type II are sparse, planar-like graphs which are graphs that are almost planar. Type III instances are dense graphs. Results of empirical testing show that the cycle-packing algorithm found the best solution in random nonplanar graphs, but it required much more CPU time than the other heuristics. The results also show that for planar-like graphs the vertex addition method is better than the edge addition method. However, for dense random graphs the edge addition method is better than others.

MAXPLANAR: A Graphical Software Package For Testing Maximal Planar Subgraph Algorithms

by

Kedan Zhao

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

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

January 1996

N378
2615

ii

APPROVAL

of a thesis submitted by

Kedan Zhao

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.

2-7-96
Date

Robert Cimbrush
Chairperson, Graduate Committee

Approved for the Major Department

2/8/96
Date

J. Dubig Stanley
Head, Major Department

Approved for the College of Graduate Studies

4/21/96
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



Date

2-8-98

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my graduate committee members, Dr. Robert Cimikowski, Dr. Gary Harkin, and Prof. Ray Babcock, and the rest of the faculty members from the Department of Computer Science for their help and guidance during my graduate program. I would also like to thank my thesis advisor, Dr. Robert Cimikowski, for his encouragement and support while supervising this thesis.

Much of the credit for completion of this project goes to my wife Qian Cai and my son Andy. Their love, support and sacrifice gave me the opportunity to spend two years in graduate school.

Special thanks go to my father Jimin Zhao and my mother Xiouzhen Zhang, for their encouragement and support.

Contents

Table of Contents	v
List of Tables	vii
List of Figures	viii
Abstract	x
1 Introduction	1
1.1 Planarity and Graph Planarization	1
1.2 Applications of Planarization	2
1.3 Thesis Outline	4
2 Definitions and Preliminaries	5
2.1 Definitions	5
2.2 Representations of Graphs	6
2.3 Some Useful Theorems	7
3 The Vertex Addition Method	12
3.1 PQ-trees	12
3.2 The PQ-tree Planarization Algorithm	14
3.3 The Maximal Planarization Algorithm	17

4	Path Addition Method	22
4.1	Depth First Search	22
4.2	The Planarity Algorithm	24
4.3	The Maximal Planarization Algorithm	27
5	The Edge Addition Method	32
5.1	Introduction	32
5.2	Planarity Testing	34
5.3	The Maximal Planarization Algorithm	36
6	The Cycle-Packing Method	41
6.1	Introduction	41
6.2	The Cycle-Packing Algorithm	43
7	The OSF/Motif MAXPLANAR Interface	45
7.1	An Introduction to the X Window System	45
7.2	An Introduction to OSF/Motif	46
7.3	MAXPLANAR	47
7.4	Graph Edit	49
7.5	Graph Test	51
8	Computational Experience	61
8.1	Test Graph Generation	61
8.2	Analysis of Results	62
8.3	Conclusions	69
	Bibliography	71

List of Tables

8.1	Heuristics applied to the special graphs of Figures 8.1-8.6. . .	65
8.2	Heuristics applied to random nonplanar graphs.	66
8.3	Heuristics applied to sparse nonplanar graphs.	68
8.4	Heuristics applied to dense nonplanar graphs.	69

List of Figures

2.1	Four biconnected components	6
2.2	Adjacency list representation	7
2.3	Adjacency maxtrix representation	8
2.4	C_1 and C_2 are not Jordan curves but C_3 is.	8
2.5	C_1 has no hamiltonian path, C_2 has a hamiltonian path but no hamiltonian cycle, while C_3 has a hamiltonian cycle.	9
2.6	Some bipartite graphs	10
2.7	Some complete bipartite graphs	11
3.1	Examples of G , G_k , B_k and corresponding PQ-tree.	13
3.2	Two intersecting near pairs $(l, si(l))$ and $(l', si(l'))$	19
4.1	A connected graph G and a palm tree P generated from G . . .	24
4.2	Conflict between pieces. To add dotted piece S_4 on the inside of c and maintain planarity, pieces S_1 and S_3 must be moved from the inside to the outside. Piece S_2 must be moved from the outside to the inside.	27
5.1	An illustration of some of the definitions	34
5.2	Merge all the blocks of $att(e_i)$ into one intermediate block B_i .	35
5.3	Merge blocks in $att(e)$	35
5.4	Add blocks B_i into $att(e)$	36

5.5	(a) is l -planar, but (b) is not.	37
7.1	The MAXPLANAR user interface	48
7.2	The overall architecture of MAXPLANAR	49
7.3	A file selection dialog window which lists all the object files in the currently selected directory.	50
7.4	Highlight all vertex numbers 6 in the graph.	52
7.5	HT planarization test selection	53
7.6	HT maximal planar subgraph selection	54
7.7	PQ planarization test selection	55
7.8	PQ maximal planar subgraph selection	56
7.9	GT maximal planar subgraph by vertex number selection . . .	57
7.10	GT maximal planar subgraph by greedy order selection	58
7.11	CHT planarization test selection	59
7.12	CHT maximal planar subgraph selection	60
8.1	g_1 graph	62
8.2	g_2 graph	63
8.3	g_3 graph	63
8.4	g_4 graph	63
8.5	g_5 graph	64
8.6	g_6 graph	64
8.7	Performance of the heuristics on random nonplanar graphs . .	67
8.8	Performance of the heuristics on planar-like graphs	67
8.9	Performance of the heuristics on dense graphs	70

Abstract

We present an efficient implementation of a software package, MAXPLANAR, with a user-friendly interface for several algorithms for finding maximal planar subgraphs of nonplanar graphs. The algorithms include the methods of path addition, edge addition, vertex addition, and cycle packing.

MAXPLANAR is designed to facilitate graph input and output and algorithm efficiency analysis. The result is an easy-to-use software package for researchers to test the various planarization algorithms.

Extensive empirical results are given for the heuristics on several families of nonplanar graphs. Type I are random nonplanar graphs with unknown maximum planar subgraph size. Type II are sparse, planar-like graphs which are graphs that are almost planar. Type III instances are dense graphs. Results of empirical testing show that the cycle-packing algorithm found the best solution in random nonplanar graphs, but it required much more CPU time than the other heuristics. The results also show that for planar-like graphs the vertex addition method is better than the edge addition method. However, for dense random graphs the edge addition method is better than others.

Chapter 1

Introduction

1.1 Planarity and Graph Planarization

Given an undirected graph, the planarity testing problem is to determine whether the graph can be drawn in the plane without any crossing edges. It has many applications, e.g. in the design of VLSI circuits, in printed circuit board layout, automated graphical display systems, determining the isomorphism of chemical structures, and in various problems dealing with the display and readability of diagrams. A few planarity testing algorithms of different types are known. They all have linear time complexity. Of the two major algorithms, one is called a “path addition” algorithm and the other is called a “vertex addition” algorithm. These terms refer to the approaches used in the algorithms. The path addition algorithm is originally due to Auslander and Parter [1] and a linear time implementation was developed by Hopcroft and Tarjan [2]. The vertex addition algorithm, which was presented first by Lempel, Even and Cederbaum[3], and improved later to a linear algorithm by Booth and Lueker[4], uses a novel data structure called the PQ-tree.

If a graph is not planar, then we may want to delete some edges to obtain a planar subgraph. This process of removing a set of edges from a nonplanar graph G to obtain a planar subgraph is known as *planarization* of the nonplanar

graph G . On the other hand, *maximal planarization* of a nonplanar graph G refers to the process of deleting a minimal set of edges of G in order to obtain a planar subgraph.

1.2 Applications of Planarization

Maximal planarization of a nonplanar graph is an important problem encountered in the automated design of printed circuit boards. If an electronic circuit cannot be wired on a single layer of a printed circuit board, then we would like to determine the minimum number of layers necessary to wire the circuit. Since only a planar circuit can be wired on a single layer board, we would like to decompose the nonplanar circuit into a minimum number of planar circuits. In general, for a nonplanar graph, neither the set of edges to be removed to maximally planarize it nor the number of these edges is unique.

Finding the *minimum* number of edges whose deletion from a nonplanar graph gives a planar subgraph was shown to be NP-complete[5]; hence research has focused on computing a *maximal planar subgraph* G' of G , that is, a subgraph G' such that for all edges $e \in G - G'$, the addition of e to G' destroys the planarity. The first algorithm for this problem runs with $O(mn)$ worst-case time bound[6]. Here m is the number of edges and n is the number of vertices. This algorithm starts with one edge and check for every subsequent edge, whether its addition to the graph preserves the planarity (by employing a linear-time planarity tester). Recently Cai, Han and Tarjan described an $O(m \log n)$ maximal planarization algorithm[8], based on the Hopcroft-Tarjan planarity testing algorithm. Di Battista and Tamassia described an incremental algorithm to check in $O(\log n)$ amortized time whether adding an edge to the graph preserves planarity, which yields an $O(m \log n)$ time maximal planarization algorithm as well. Ozawa and Takahashi[9] proposed another

$O(mn)$ time and $O(m + n)$ space algorithm to planarize a nonplanar graph using the PQ-tree implementation[4] of Lempel, Even, and Cederbaum's planarity testing algorithm[3]. However, for a general graph this algorithm may not determine a maximal planar subgraph[10]. Moreover, in certain cases, this algorithm may terminate without considering all the vertices; in other words, it may not produce a *spanning* planar subgraph[11]. Jayakumar, Thulasiraman and Swamy[11] presented an $O(n^2)$ planarization algorithm for a special class of graphs based on PQ-trees. However, this algorithm contains some errors. Kant give a correct version that can be implemented to run in $O(n^2)$ time[12], which is better than the time bound $O(m \log n)$ for dense graphs. Moreover, instead of testing for every edge whether or not it can be added without destroying the planarity, it calculates for every vertex the minimum number of edges which must be deleted to preserve planarity. The maximal planarization algorithms were developed independently and in their original form by various reseachers, and some of them were implemented in PASCAL and some in C. Until now there was no unified software package that includes all of them. We present an efficient C language implementation of a software package with a user-friendly interface for several algorithms for finding maximal planar subgraphs of nonplanar graphs. The algorithms include the methods of path-embedding, edge-embedding, vertex-embedding, and cycle packing. The algorithms are well known but have not previously been organized and implemented as a single, cohesive, integrated software package in an efficient manner. A user-friendly interface was also developed to facilitate graph input and output and algorithm efficiency analysis. The result is an easy-to-use software package for researchers to use in applications requiring the maximal planarization of graphs.

1.3 Thesis Outline

In this thesis our objective was to develop a unified software package for all of the major graph planarization algorithms in order that empirical testing and a comparative analysis of the methods would be more easily performed. To facilitate the process, we developed a front-end graphical interface to the algorithms using Motif and X-windows.

The paper is organized as follows. In Chapter 2 we present definitions and other basic concepts. In Chapter 3 we discuss the vertex-addition algorithm. In Chapter 4 and Chapter 5 we describe the path- and edge- addition algorithms and in Chapter 6 we present with the cycle-packing algorithm. In Chapter 7 we discuss the graphical user interface, and in Chapter 8 we discuss experimental testing and offer concluding remarks.

Chapter 2

Definitions and Preliminaries

2.1 Definitions

Let $G = (V, E)$ be a graph with vertex set $V(G)$ and edge set $E(G)$. We assume that G is *simple*, that is, has no multiple edges or loops. Throughout this paper n denotes the number of vertices of G , that is, $n = |V(G)|$ and m denotes the number of edges of G , that is, $m = |E(G)|$. If there is a graph H with vertex set $V(H)$ and edge set $E(H)$ then we say that H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

A graph G is *connected* if every pair of points are joined by a path. In this paper we always assume the graph is connected. If a graph is disconnected then all of the algorithms discussed in this thesis can be applied to the individual components one by one.

A graph G is *planar* if it is embeddable in the plane without any crossing edges.

A *biconnected component* of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 2.1 illustrates this definition.

A graph G is called *k -connected*, $k \geq 3$, if G is simple, $|V(G)| \geq k + 1$, and the graph obtained from G by deleting any $k - 1$ vertices is connected.

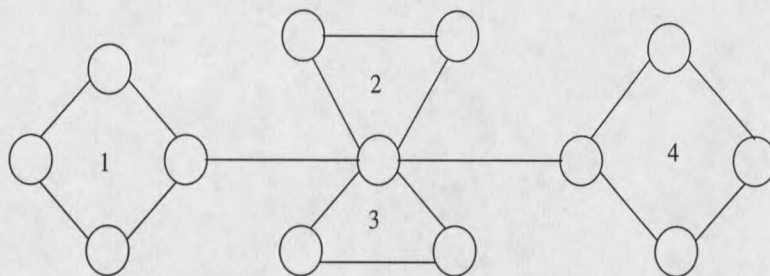


Figure 2.1: Four biconnected components

A graph is planar if and only if all of its biconnected components are planar.

We call an algorithm *randomized* if its behavior is determined not only by the input but also by value produced by a random number generator.

2.2 Representations of Graphs

There are two standard ways to represent a graph $G = (V, E)$, as a collection of adjacency lists or as an adjacency matrix. The adjacency list provides a compact way to represent *sparse* graphs, i.e., $|E| = O(n)$. All of the algorithms presented in this paper assume that an input graph is represented in adjacency list form. An adjacency matrix representation may be preferred, however, when the graph is dense, or when one needs to be able to tell quickly if there is an edge joining two vertices.

The adjacency list representation of a graph $G = (V, E)$ consists of an array $Adj[]$ of n lists, one for each vertex in V . For each $u \in V$, the adjacency list $Adj[u]$ contains (pointers to) all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G . The vertices in each adjacency list are typically stored in an arbitrary order. Figure 2.2 is an adjacency list representation of an undirected graph.

A potential disadvantage of the adjacency list representation is that there

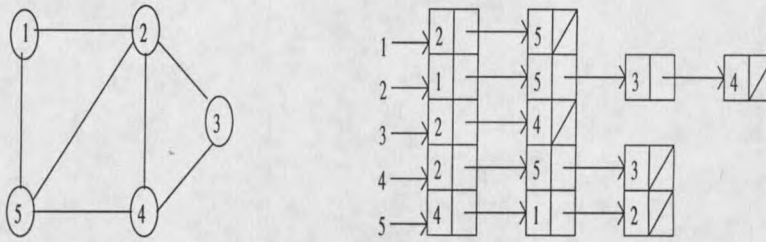


Figure 2.2: Adjacency list representation

is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency matrix representation of the graph, at the cost of using a factor of n more memory.

For the adjacency matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \dots, n$ in some arbitrary manner. The adjacency matrix representation of a graph G then consists of a $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $\{i, j\} \in E$ and $a_{ij} = 0$ otherwise.

Figure 2.3 is the adjacency matrix representation of an undirected graph. The adjacency matrix representation of a graph requires $O(n^2)$ memory, independent of the number of edges in the graph.

2.3 Some Useful Theorems

A *Jordan curve* in the plane is a continuous non-self-intersecting curve whose origin and terminus coincide. For example, in Figure 2.4, the curve C_1 is not a Jordan curve because it intersects itself, C_2 is not a Jordan curve since its origin and terminus do not coincide, i.e., its two end points do not meet but C_3 is a Jordan curve.

If J is a Jordan curve in the plane then the part of the plane enclosed by

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 2.3: Adjacency maxtrix representation

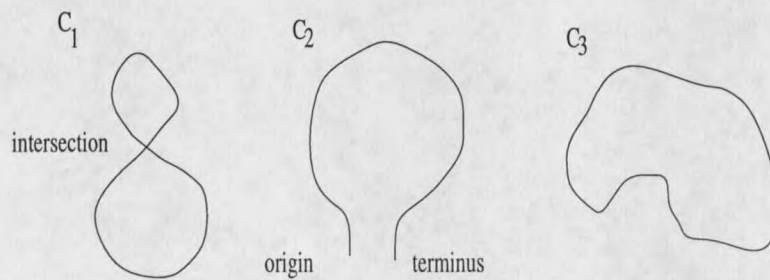


Figure 2.4: C_1 and C_2 are not Jordan curves but C_3 is.

J is called the *interior* of J and denoted by $\text{int } J$. We exclude from $\text{int } J$ the points actually lying on J . Similarly the part of the plane lying outside J is called the *exterior* of J and denoted by $\text{ext } J$.

Jordan Curve Theorem. If J is a Jordan curve, x is a point in $\text{int } J$ and y is a point in $\text{ext } J$ then any line joining x to y must meet J at some point, i.e., must cross J .

A *hamiltonian path* in a graph G is a simple path which contains every vertex of G . No vertex of a path is repeated. A *hamiltonian cycle* in G with initial vertex v contains every vertex of G precisely once and then ends up back at the initial vertex.

A graph is called *hamiltonian* if it has a hamiltonian cycle.

By simply deleting the last edge of a hamiltonian cycle we get a hamiltonian path. However a nonhamiltonian graph may possess a hamiltonian path, i.e., hamiltonian paths cannot always be used to form hamiltonian cycles. For example, in Figure 2.5, C_1 has no hamiltonian path or hamiltonian cycle; C_2 has the hamiltonian path b, a, c, d but no hamiltonian cycle, while C_3 has the hamiltonian cycle a, b, d, c, a .

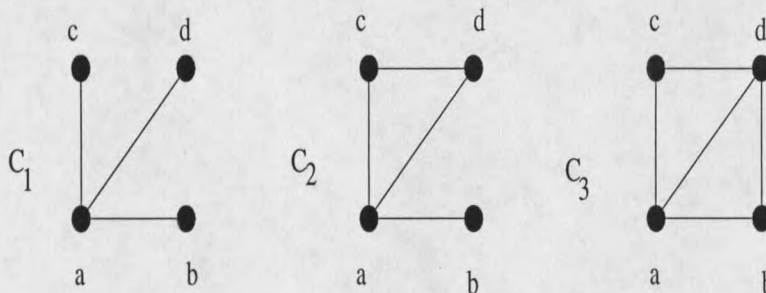


Figure 2.5: C_1 has no hamiltonian path, C_2 has a hamiltonian path but no hamiltonian cycle, while C_3 has a hamiltonian cycle.

An *independent set* of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices

such that each edge in E is incident on at most one vertex in V' .

If the vertex set V of G can be partitioned into two nonempty subsets X and Y (i.e., $X \cup Y = V$ and $X \cap Y = \phi$) in such a way that each edge of G has one end point in Y then G is called *bipartite*. The partition $V = X \cup Y$ is called a *bipartition* of G .

A *complete bipartite graph* is a simple bipartite graph G , with bipartition $V = X \cup Y$, in which every vertex in X is joined to every vertex of Y . If X has m vertices and Y has n vertices, such a graph is denoted by $K_{m,n}$. Figure 2.6 shows two bipartite graphs. They are not complete bipartite. However, the graphs of Figure 2.7 are complete bipartite.

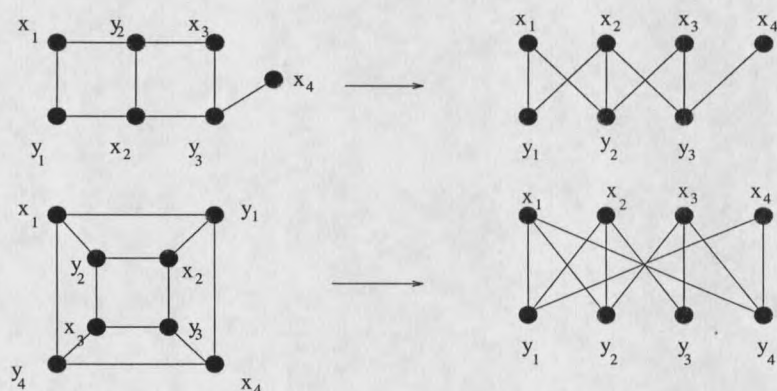


Figure 2.6: Some bipartite graphs

A *subdivision* of G is a graph obtained from G by inserting vertices (of degree 2) into one or more of the edges of G .

Kuratowski's Theorem. A graph G is planar if and only if it has no subgraph isomorphic to a subdivision of K_5 or $K_{3,3}$.

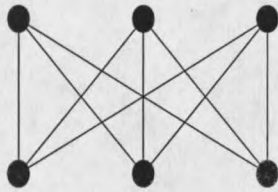
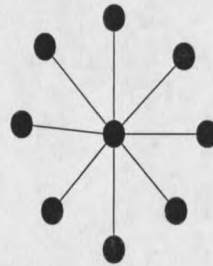
 $K_{3,3}$  $K_{1,8}$

Figure 2.7: Some complete bipartite graphs

Chapter 3

The Vertex Addition Method

In this chapter we discuss the first planarization algorithm, which is based on a novel data structure called a PQ tree.

3.1 PQ-trees

Let $G = (V, E)$ be a graph with n nodes and m edges. We assume that G is simple, that is, has no multiple edges or loops. A graph is planar if and only if it can be embedded in the plane without any crossing edges. A graph G is planar if and only if the biconnected components of G are planar. We henceforth assume that G is biconnected. The Lempel, Even and Cederbaum algorithm[3] first labels the vertices of G as $1, 2, \dots, n$ using what is called an *st*-numbering. An *st*-numbering is a numbering of the vertices of G by $1, 2, 3, \dots, n$ such that each vertex j is adjacent to two vertices i and k satisfying $i < j < k$. Let $G_k = (V_k, E_k)$ be the subgraph of G induced on the vertices $1, 2, \dots, k$. If $k < n$ then there must exist an edge of G with one endpoint in V_k and the other in $V - V_k$. Let G'_k be the graph formed by adding to G_k all of these edges. These edges are called *virtual edges* and their endpoints in $V - V_k$ are called *virtual vertices* and labelled as their counterparts in G , but they are kept separate, i.e., there may be several virtual vertices with the

same label, each with exactly one entering edge. Let B_k (the *bush* form) be an embedding of G'_k such that all the virtual vertices are placed on the outer face. It can be shown[3] that the *st*-graph G is planar if and only if for every $B_k, 2 \leq k \leq n - 2$, there exists a planar drawing B'_k isomorphic to B_k such that in B'_k all the virtual vertices labelled $k + 1$ appear consecutively.

The PQ-tree T_k corresponding to the bush form B_k consists of three types of vertices: (i) Leaves in T_k represent virtual edges in B_k , (ii) P-nodes in T_k represent cutvertices in B_k , and (iii) Q-nodes of T_k represent the maximal biconnected components in B_k . G , G_k , B_k and the corresponding PQ-tree are illustrated in Figure 3.1.

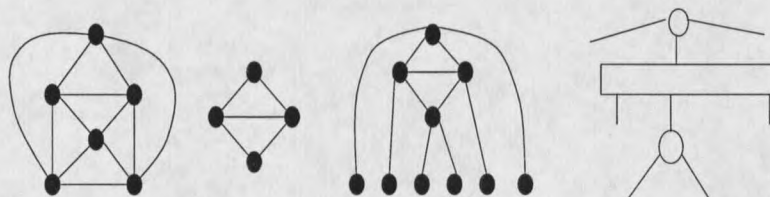


Figure 3.1: Examples of G , G_k , B_k and corresponding PQ-tree.

A few definitions are now in order. Let E_{k+1} denote the set of leaves in T_k which corresponds to the virtual vertex $k + 1$. A node X in T_k is said to be *full* if all its descendant leaves are in E_{k+1} ; X is said to be *empty* if none of its descendant leaves are in E_{k+1} ; otherwise X is *partial*. If X is full or partial, then it is called a *pertinent node*. The *frontier* of T_k is the sequence of all the descendant leaves of T_k read from left to right. The *pertinent subtree* of T_k is the smallest connected subtree which contains all the leaves in E_{k+1} . The root of pertinent subtree is called the *pertinent root*. Two PQ-trees are considered *equivalent* if one can be obtained from the other by performing one or more of the following type of operations:

- (i) Reversing the order of the children of a Q-node.
- (ii) Permuting the children of a P-node.

It can be shown[3] that B'_k exists if and only if T_k can be converted into an equivalent PQ-tree T'_k such that all the pertinent leaves appear consecutively in the frontier of T'_k . Booth and Lueker[4] have defined a set of patterns and replacements with which T'_k can be reduced into a PQ-tree T_k^* in which all the pertinent leaves appear as children of a single node. The reduction process consists of two phases. In the first phase, called the BUBBLE-UP phase, the pertinent subtree is identified. In the second phase, called the REDUCTION phase, pattern matching and corresponding replacements are carried out using the reversing and permutation operations.

To construct T_{k+1} from T_k , we first reduce T_k to T_k^* and then replace all the leaves corresponding to virtual edges of vertex $k + 1$ by a P-node whose children are all leaves, corresponding to outgoing edges of vertex $k + 1$ in G . The algorithm of Booth and Lueker, which we will refer to as PLANARITY TEST, starts with T_1 and constructs the sequence of PQ-trees $T_1, T_2, \dots$. If the graph G is planar, then the algorithm terminates after constructing T_{n-1} ; otherwise, it terminates after detecting the impossibility of reducing some T_k into T_k^* . The crucial result in the complexity analysis of PLANARITY TEST is stated in the following theorem:

Theorem 3.1[4] The sum of the sizes of all the pertinent nodes in the PQ-trees $T_1, T_2, \dots, T_{n-1}$ of a planar graph is $O(m + n)$.

3.2 The PQ-tree Planarization Algorithm

In this section we discuss the basic principle of an approach for planarization, due to Ozawa and Takahashi[9], and also studied by Jayakumar *et al.*[11] and Kant[12]. Following these papers, we classify the nodes of a PQ-tree according

to their frontier as follows:

Type W: A node is said to be *Type W* if its frontier consists of only empty leaves.

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

Type H: A node X is said to be *Type H* if the subtree rooted at X can be arranged such that all the descendant pertinent leaves of X appear consecutively at either the left end or at the right end of the frontier.

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

The central concept of the PQ-tree planarization algorithm is stated in the following theorem of which is essentially a reiteration of the principle on which PLANARITY TEST is based.

Theorem 3.2[12] A graph G is planar if and only if the pertinent roots of all subtrees in $T_2, T_3, \dots, T_{n-1}$ of G are *Type B*, *H* or *A*.

We call a PQ-tree *reducible* if its pertinent root is *Type B*, *H* or *A*; otherwise it is called *irreducible*. A graph G is planar if and only if all the T_i 's are reducible. If any T_i is irreducible, we can make it reducible by appropriately deleting some of the leaves in it. For a node X in an irreducible PQ-tree T_i , let the w -, h - and a - number be the minimum number of descendant leaves of X , which should be deleted from T_i such that X becomes *Type W*, *H* and *A*, respectively. We denote the tuple of numbers thus associated to a node by $[w, h, a]$. (Note that a partial node can not be made *Type B*, because we do not delete empty children.) When we have calculated these numbers for the root of the subtree of the pertinent nodes, we set the type of the root according

to the minimum of these numbers. If this minimum is not zero, we traverse the tree top-down and determine the type of each pertinent node. Using this information, a decision about pertinent leaves can be made, which must be thrown away to make the tree reducible.

For this, we process T_i bottom-up by the algorithm $\text{COMPUTE}(T_i)$ from the pertinent leaves to the pertinent root. When a pertinent node X is processed, the $[w, h, a]$ numbers of all its pertinent children are computed and we can compute the $[w, h, a]$ number for X , using this information.

After computing the $[w, h, a]$ number for pertinent root R of T_i , we can determine whether T_i is reducible or not. If the minimum of h and a is zero for R , then T_i is reducible, otherwise we make R Type H or A depending on which one of h and a is minimum, and make T_i reducible by deleting the corresponding pertinent leaves from T_i . The procedure which traverses the tree top-down and determines the type for each pertinent node in T_i to obtain a reducible T_i will be denoted by $\text{DELETE-NODES}(T_i)$. E_{i+1}^i denotes the corresponding set of removed incoming edges of $i + 1$.

The algorithm PLANARIZE can now be described as follows:

PLANARIZE

Construct the initial tree $T_1 = T_1^*$;

for $i:=1$ to $n-1$ do

$\text{BUBBLE-UP}(T_i)$;

$\text{COMPUTE}(T_i)$;

 if $\min\{h, a\}$ for the pertinent root R is not zero then

 make R Type H or A corresponding to the minimum of h and a ;

$\text{DELETE-NODES}(T_i)$;

 replace all full nodes of T_i by a P-node X with all

outgoing edges of node $i+1$ appearing as children of X .

end

Theorem 3.3 [12] Algorithm PLANARIZE determines a planar spanning subgraph G_p of the nonplanar graph G in $O(n^2)$ time.

3.3 The Maximal Planarization Algorithm

For a given nonplanar graph G , let G_p be a planar spanning subgraph of G obtained by the algorithm PLANARIZE. The idea to augment G_p to a maximal planar subgraph G'_p is as follows. Start with G and construct its chain of PQ-trees. After constructing a PQ-tree T_i , reducibility is obtained by deleting a minimum number of leaves representing the edges in E'_{i+1} . Note that T_i will become reducible if all the leaves from the set E'_{i+1} are deleted from T_i . This can be done by computing the $[w, h, a]$ number of the pertinent nodes in T_i . Let $T_i(G_p)$ denote the smallest subtree of T_i whose frontier contains all the pertinent leaves from G_p . Since we would like to include G_p in the final maximal planar subgraph, we take care that, during the reduction of T_i , no node in $T(G_p)$ is made Type A except its root. This ensures that the bottom-up reduction process proceeds at least up to the root of $T_i(G_p)$ and possibly beyond. While computing the $[w, h, a]$ numbers we ignore the presence of leaves from $G - G_p$, which we will call *empty leaves*. In the following, the empty leaves in T_i corresponding to the edges in E'_{i+1} will be called the *new pertinent leaves* of T_i and the other pertinent leaves of T_i (corresponding to the edges entering vertex $i + 1$ in G_p) will be called *preferred leaves*.

Again a node is full if its frontier has no empty leaf from G_p ; it is empty if its frontier has only empty leaves from G_p ; otherwise it is partial. We call node X a *preferred node* if it has some of the preferred leaves in its frontier.

This procedure leads to a construction of a sequence, here called the *preferred sequence*, containing all preferred nodes, empty nodes, and as many as possible new pertinent nodes. Hence if X is new pertinent, then X may either be retained in the reducible T_i or X may be deleted along with all its descendants to make T_i reducible. The formulas for computing the $[w, h, a]$ numbers of the pertinent nodes are similar to those in $\text{COMPUTE}(T_i)$.

In every tree T_i we again compute the maximal pertinent sequence of incoming edges of node $i + 1$ in G_p . We call only these leaves pertinent. The leaves of edges $(k, i + 1) \in G - G_p$ are not pertinent in T_i and will not be added in this step. We don't remove $(k, i + 1) \in G - G_p$ from T_i , because maybe in some later PQ-tree T_j , $j > i$, $(k, i + 1)$ can be added to G_p while preserving planarity. We call leaves of edges $(k, i + 1) \in G - G_p$ potential leaves in the PQ-tree T_j , $j > i$. As in algorithm PLANARITY TEST, we reduce the tree T_i such that the pertinent nodes form a sequence of adjacent siblings of a common parent by applying the templates and replacement patterns[4]. this sequence is replaced by a P-node X with a set of leaves, representing the outgoing edges of $i + 1$. Furthermore, to store the place of a maximal pertinent sequence in T_i , we add adjacent to X a *sequence indicator*. This special node, denoted by $< i + 1 >$, marks the place of the sequence in the PQ-tree. In [7], a related special node, a *direction indicator*, is introduced to store the place of the maximal pertinent sequence of vertex $i+1$ in the tree, with the direction of enumeration (from left to right or vice versa). This is used in [7] to compute a planar embedding of the graph using PQ-trees. We treat sequence indicators and potential leaves as empty leaves, as well as leaves of edges $\in G - G_p$, which are not potential yet.

A potential leaf l is near its sequence indicator $si(l)$, if the PQ-tree T_i can be reduced such that they are adjacent siblings, by deleting only empty nodes

and not binding partial nodes to new places. When a potential leaf l and its sequence indicator $si(l)$ are near, this is called a *near pair*, and we can reduce the PQ-tree such that they are adjacent siblings, and add the edge $e \in G - G_p$ to G_p , without binding partial nodes and leaves to new places. The following theorem is crucial for our algorithm.

Theorem 3.4[12] An edge $e \in G - G_p$ can only be added to G_p without destroying planarity, if and only if at some step the corresponding potential leaf l is near its sequence indicator $si(l)$ in the PQ-tree.

Between two elements of a near pair several potential leaves and sequence indicators may occur. Two near pairs $(l, si(l))$ and $(l', si(l'))$, are said to be *intersecting* in T_i , if either l' or $si(l')$ is between l and $si(l)$ in all equivalent PQ-trees of T_i . In Figure 3.2, an example of two intersecting near pairs is given.

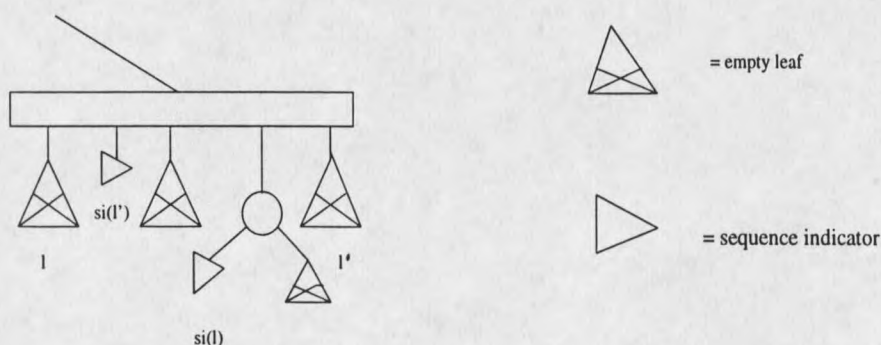


Figure 3.2: Two intersecting near pairs $(l, si(l))$ and $(l', si(l'))$

If two near pairs $(l, si(l))$ and $(l', si(l'))$ are intersecting, then only one corresponding edge of l or l' can be added to G_p , without destroying the planarity. Suppose l occurs between the element l' and $si(l')$, l will be removed from the PQ-tree; hence it does not form a near pair with $si(l)$ after reducing near pair $(l', si(l'))$. By the previous theorem, l can not be added to G_p without

destroying planarity.

Hence the order of inspecting the near pairs and applying the necessary reduction is essential for constructing the augmented planar graph G_p . All potential leaves and sequence indicators are elements of a maximal pertinent sequence; thus during the algorithm we can inspect for near pairs. The maximal planarization algorithm can now be described at a high level as follows:

MAXIMAL PLANARIZATION

assign st-numbers to all the vertices of G ;

PLANARIZE(G);

construct the PQ-tree T_1 corresponding to G'_1 ;

for $i:=1$ to $n-1$ do

begin

{compute step}

compute the maximal pertinent sequence in tree T_i

of incoming edges of node $i+1$ in G_p ;

{reduction step}

apply the template matchings in the PQ-tree,

and apply an additional reduce step to reduce near pairs

in the maximal pertinent sequence;

{vertex addition step}

for all deleted sequence indicators $\langle j \rangle$

remove the corresponding potential leaves from T_i ;

replace all the full nodes in T_i by a P-node X with all
outgoing edges of node $i+1$ appearing as children of X ;

add the sequence indicator $\langle i+1 \rangle$ as a sibling of X in T_i ;

end

Theorem 3.5 MAXIMAL PLANARIZATION can be implemented to run in $O(n^2)$ time and space.

Proof: See [12].

Chapter 4

Path Addition Method

In this chapter we describe a planarity testing algorithm based on path addition, then extend it to an algorithm for maximally planarizing a graph.

4.1 Depth First Search

Graph algorithms require a systematic way of exploring a graph. We use a common technique called depth-first search. We start from some vertex s of G and choose an edge leading from s . Traversing the edge leads to a new vertex. In general we continue the search by selecting and traversing unexplored edges. If G is connected, each edge will be traversed exactly once.

If G is undirected, a depth-first search of G imposes a direction on each edge of G given by the direction in which the edge is traversed during the search. Thus the search converts G into a directed graph G' . The search also partitions the (now directed) edges into two classes: a set of *tree arcs*, defining a spanning tree T of G' , and a set of *fronds* (v, w) which satisfy $w \rightarrow^* v$ in T [2]. A frond (v, w) is denoted by $v \rightarrow w$. A directed graph G' whose edges may be partitioned in this way is called a *palm tree*. Depth-first search is important because the structure of paths in a palm tree is very simple.

To implement a depth-first search of a connected, undirected graph, we

use a simple recursive procedure *DFS* which keeps a stack of the old vertices with possibly unexplored edges. The procedure uses a set of adjacency lists of the graph to be searched, and the exact search order depends on the order of edges in the adjacency lists. The procedure numbers the vertices from 1 to n in the order they are reached during the search, in addition to identifying tree arcs and fronds.

```

begin
    integer n;
    procedure DFS(v,u) comment: vertex u is the father of vertex v in
                                the spanning tree being constructed;
    begin
        n:=NUMBER(v):=n+1;

        for w  $\in$  A(v) do begin

            if NUMBER(w) =0 then
                begin
                    mark(v,w) as a new arc; comment w is a new vertex;
                    DFS(w,v);
                end

            else if NUMBER(w)<NUMBER(v) and  $w \neq u$  then
                begin
                    comment: this test necessary to avoid exploring
                                an edge in both directions;
                    mark(v,w) as a frond;
                end;
        end;
    end;
end;
```

```

for i:=1 until V do NUMBER(i):=0;
n:=0;
DFS(s,0);comment the search starts at vertex s;
end;

```

Theorem 4.1 The procedure above correctly carries out a depth-first search of an undirected graph and requires $O(n + m)$ time if the graph has n vertices and m edges. The vertices are numbered so that if (v, w) is a tree arc, $NUMBER(w) \geq NUMBER(v)$; and if (v, w) is a frond, $NUMBER(w) \leq NUMBER(v)$.

Proof: See [13].

Figure 4.1 shows a connected graph G and a palm tree generated from G using depth-first search.

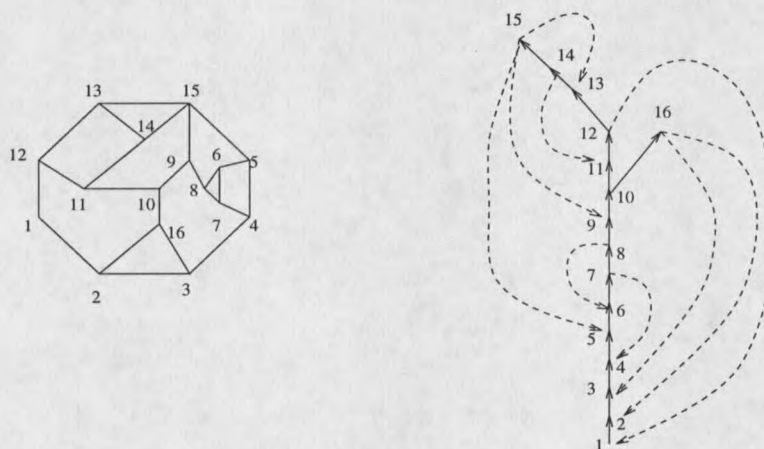


Figure 4.1: A connected graph G and a palm tree P generated from G .

4.2 The Planarity Algorithm

The first step of the algorithm gets rid of graphs with too many edges. We count the number of edges in the graph and if the count ever exceeds $3n - 3$,

we declare the graph nonplanar. Next, we divide the graph into biconnected components. Then we test the planarity of each component.

To test the planarity of a component, we apply DFS, converting the graph into a palm tree P and numbering the vertices. Now we use Auslander, Parter, and Goldstein's algorithm[1]. This algorithm finds a cycle in the graph and deletes it, leaving a set of disconnected pieces. Then the algorithm checks the planarity of each piece plus the original cycles (by applying itself recursively), and determines whether the embeddings of the pieces can be combined to give an embedding of the entire graph. Let us separately examine the cycle-finding part of this process and the planarity-testing part.

Each recursive call on the algorithm requires that we find a cycle in the piece of the graph to be tested for planarity. This cycle will consist of a simple path of edges not in previously found cycles, plus a simple path of edges in old cycles. We use depth-first search to divide the graph into simple paths which may be assembled into the cycles necessary for planarity testing. We need a second search to find paths because the search must be carried out in a special order if the planarity test is to be efficient.

Now consider the first cycle c . It will consist of a sequence of tree arcs followed by one frond in P . The numbering of vertices is such that the vertices are in order by number along the cycle. Each piece not part of the cycle will consist either of a single frond (v, w) , or of a tree arc (v, w) plus a subtree with root w , plus all fronds which lead from the subtree. We process the pieces and add them to a planar representation in decreasing order of v . Each piece can go either inside or outside c by the Jordan Curve Theorem. When we add a piece, certain other pieces must be moved from the inside to the outside or from the outside to the inside of c . (see Figure 4.2.) We continue to add new pieces and move old pieces if necessary until either a piece cannot be added or

the entire graph is embedded in the plane. Below is an outline of the entire algorithm.

```

procedure PLANARITY(G);
  begin comment: an outline of the planarity algorithm;
    integer E;
    E:=0;
    for each edge of G do
      begin
        E:=E+1;
        if E>3V-3 then go to nonplanar;
      end;
    divide G into biconnected components;
    for each biconnected component G do
      begin
        explore G to number vertices and transform G into
        a palm tree P;
        find a cycle c in P;
        construct planar representation for c;
        for each piece formed when c is deleted do
          begin
            apply algorithm recursively to determine if
            piece plus cycle is planar;
            if piece plus cycle is planar and piece may
            be added to planar representation
            then add it else goto nonplanar;
          end;
        end;
      end;
    end;
  end;

```

end;

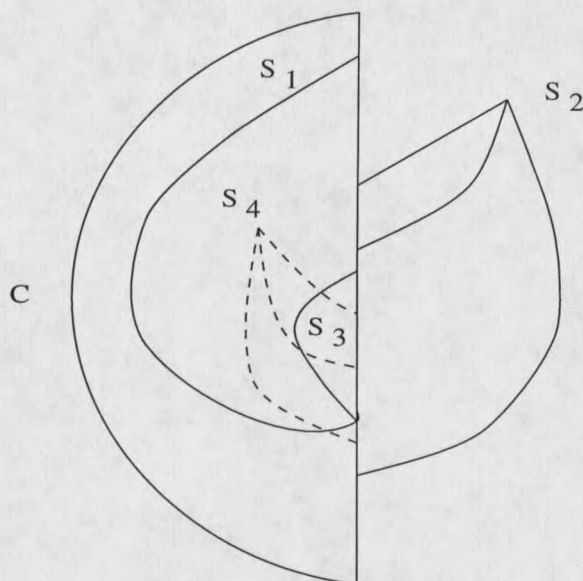


Figure 4.2: Conflict between pieces. To add dotted piece S_4 on the inside of c and maintain planarity, pieces S_1 and S_3 must be moved from the inside to the outside. Piece S_2 must be moved from the outside to the inside.

4.3 The Maximal Planarization Algorithm

Using depth-first search, an initial cycle is found in a graph G , deleted, and then embedded in the plane. The remainder of G is then decomposed into edge-disjoint paths and an attempt is made to embed each path inside or outside the cycle. If all paths can be embedded then the graph is planar; otherwise it is nonplanar. Whereas the original planarity algorithm halts when a path cannot be embedded, the planarization heuristic of Chiba *et al.*[6] deletes a frond from the path and continues embedding. The embedded paths comprise a maximal planar subgraph of G .

A depth-first search of G imposes both a numbering on the vertices and an orientation on the edges, converting G into a directed graph $D = (V, E)$ such that E is partitioned into a set E_t of tree edges and a set E_f of fronds. Paths are recursively generated by the procedure PATHFINDER.

```

PATHFINDER(v);

comment: routine to generate paths in a biconnected palm tree
        with specially ordered adjacent list A(v). Vertex s
        is a global variable, the start vertex of the current
        path, and is initialized to 0;

for w ∈ A(v) do
  if v → w then begin
    if s = 0 then begin
      s:=v;
      start new path;
    end;
    add (v,w) to current path;
    PATHFINDER(w);
  end
else begin
  comment: v →→ w;
  if s = 0 then begin
    s:=v;
    start new path
  end;
  add (v,w) to current path;
  output current path;
  s:=0;

```

```

end;
s:=0;
comment: vertex 1 is the start vertex of the search;
PATHFINDER(1);
end;

```

In order to avoid early wrong choices for embedding paths inside or outside the initial cycle C , it is necessary to generate paths systematically, to choose appropriate regions for embedding them, and, perhaps, to rearrange already embedded paths to accommodate new ones. Thus, after the initial depth-first search of G , the vertices and adjacency lists for D are reordered so that paths are generated in the desired order. First, functions $lowpt(v)$ and $nextlopt(v)$ are computed for each vertex $v \in V$. $lowpt(t)$ is the lowest numbered vertex reachable from vertex v or from any of its descendants (in the depth-first search tree) by a single frond. When it is impossible to reach below v by means of a single frond, v itself is $lowpt(v)$. Similarly, $nextlopt$ is the next lowest vertex below v , excluding $lowpt(v)$, reachable in this manner; otherwise, it is v . Also, $\phi((v, w))$ is computed for each edge $(v, w) \in E$ follows:

$$\phi((v, w)) = \begin{cases} 2w & : \text{if } (v, w) \text{ is a frond} \\ 2lowpt(w) & : \text{if } (v, w) \text{ is a tree edge and } nextlopt(w) \geq v \\ 2lowpt(w) + 1 & : \text{otherwise} \end{cases}$$

Then, for each vertex v in D , all edges (v, w) in the adjacency lists for D are sorted nondecreasingly by ϕ -value and are used to determine the precise ordering of paths generated in D . Roughly speaking, a frond entering a lower numbered vertex always precedes a frond entering a higher-numbered vertex, and tree edges (v, w) appear in nondecreasing order of their abilities to lead to a vertex lower numbered than v by a single frond. Having obtained the properly ordered adjacency list $padj()$ for D , a depth-first search is re-applied

to decompose D into a cycle C and some edge-disjoint paths p_i . Using the depth-first numbering and beginning at vertex 1, paths are built consisting of 0 or more tree edges followed by a frond (v, w) . Each tree edge continues building a path, while each frond ends a path. Each subsequent path begins with the first vertex of the last frond traversed; if this vertex has no more unexplored edges, the search backs up to the previous vertex on the last path, and this continues until all edges of D are traversed.

The first path generated by PATHFINDER is a cycle C and every other path is a simple path having exactly two vertices (its end vertices) and no edges in common with previously generated paths. After C is deleted from D , some components remain. Each component consists of one or more segments. Each segment is either (a) a single frond (v, w) not on C but with v and w on C , or (b) a subgraph consisting of a tree edge (v, w) with $v \in C$, $w \notin C$, and the directed subtree rooted at w , together with all the fronds from this subtree. Each segment S_v has a first path fp_v identified by a vertex v with the least depth-first number in S_v , called the *base vertex* of the segment. PATHFINDER generates segments in decreasing order of their base vertices, and all paths in one segment are generated before paths in the next segment. All paths in a segment must be embedded together, either inside or outside C .

After embedding C , an attempt is made to embed the segments of $D - C$ in the order they were generated by PATHFINDER. While embedding a segment, we apply the algorithm recursively and generate segments within a segment, with respect to another cycle. To embed a segment S_v , we consider the first path fp_v in S_v generated and embed it either inside or outside C . by examining previously embedded paths, we determine if fp_v can be embedded inside C . This may require that some segments already embedded on the outside be moved to the inside and so fort. If, after this rearrangement of segments, fp_v

still cannot be embedded inside C , the single frond of fp_v is deleted and the remainder of the path is embedded. If fp_v can be embedded inside C , we embed it and then try to embed the remainder of the segment S_v by applying the algorithm recursively on the segment. Hence, each segment may be broken down into several smaller segments until, finally, each segment consists of only a single frond. The process is then repeated on the remainder of the graph.

Chapter 5

The Edge Addition Method

In this chapter we describe a variation of the path addition method which adds *edges* rather than *paths* to the graph.

5.1 Introduction

The edge addition method CHT[8] is based on the path addition(HT) algorithm. The main difference is that CHT admits a more general ordering than the original HT algorithm does in processing the successors of each tree edge. Also, the HT algorithm processes one *path* at a time, while the CHT algorithm processes one *edge* at a time. In this sense, the CHT algorithm is a more recursive version of the HT algorithm.

A depth-first search will convert the undirected graph $G_0 = (V_0, E_0)$ into a directed graph $G = (V, T, B)$, where V is the set of DFS numbers of vertices in V_0 , T is the set of tree edges, and B is the set of back edges. Each edge of G_0 is converted into either a tree edge or a back edge. All the tree edges form a DFS forest. If $[a, b]$ is a tree edge, then $a < b$. If $[a, b]$ is a back edge, then $b < a$, and there is a tree path in T from b to a . In either case, a is called the *tail* of $[a, b]$, and b is called the *head* of $[a, b]$. The union of T and B will be denoted by E .

We define successors for both vertices and edges. If $[a, b]$ is a tree edge, then b is a *successor* of a . If $[a, b]$ is tree edge and $[b, c]$ is any edge, then $[b, c]$ is a successor of $[a, b]$. Back edges have no successors. We also define descendants and ancestors for both vertices and edges. A *descendant* of vertex x is defined recursively as either x itself or a successor of a descendant of x . If y is a descendant of x , then x is an *ancestor* of y .

Let $e = [a, b] \in E$, let Y be the set of vertices y such that for some x , $[x, y]$ is a back edge and also a descendant of e . If Y is not empty, we define $low_1(e)$ to be the smallest integer in Y , and $low_2(e)$ to be the second smallest integer in $Y \cup \{n + 1\}$. Otherwise, we define $low_1(e) = low_2(e) = n + 1$. The two mappings low_1 and low_2 can be computed in $O(m)$ time during depth first search on G_0 . If a is not the root of DFS tree, and $low_1(e) \leq a$, then a is an *articulation point* of G [13].

If $e = [a, b]$ is any edge in E , then we define the function ϕ on E as follows. $\phi(e) = 2low_1(e)$, if $low_2(e) \geq a$ and $\phi(e) = 2low_1(e) + 1$ otherwise.

We arrange the successors of each tree edge in increasing order on their ϕ values. This ordering can be computed in $O(m)$ time using bucket sort[8]. If $e_1, \dots, e_k$ are the successors of w ordered this way, we will call e_i the i^{th} successor of e for $i = 1, \dots, k$.

For $e = [a, b]$, we define $S(e)$, the segment of e , to be the subgraph of G that consists of all the descendants of e . We use $att(e)$ to denote the set of back edges $[c, d]$ in $S(e)$ such that d is an ancestor of a , including a itself. Each back edge in $att(e)$ is called an *attachment* of e .

For any edge $e = [a, b]$, we define $cycle(e)$ as follows: if e is a back edge, then $cycle(e) = e \cup \{e' : e' \text{ belongs to the tree path from } b \text{ to } a\}$; if e is a tree edge and $low_1(e) \geq a$, then $cycle(e) = \{\}$; otherwise, $cycle(e) = cycle(e_1)$, where e_1 is the first successor of e . We use $sub(e)$ to denote the subgraph

$S(e) \cup \text{cycle}(e)$. It is easy to see that if $\text{cycle}(e)$ is not empty, then the vertex $\text{low}_1(e)$ is always on $\text{cycle}(e)$. Also, if $\text{low}_1(e) \geq a$, then $\text{sub}(e) = S(e)$; if $\text{low}_1(e) < a$, then $\text{sub}(e) - S(e) = \{e' : e' \text{ belongs to the tree path from } \text{low}_1(e) \text{ to } a\}$.

Figure 5.1 illustrates some of these definitions, where $\text{low}_1(e) = 1$; $\text{low}_2(e) = 2$; $\text{cycle}(e) = \{[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 1]\}$; $S(e)$ contains all the edges in the graph except $[1, 2], [2, 3], [3, 4]$; $\text{sub}(e)$ is the whole graph; $\text{att}(e) = \{[8, 1], [9, 3], [12, 1], [14, 2], [13, 4]\}$.

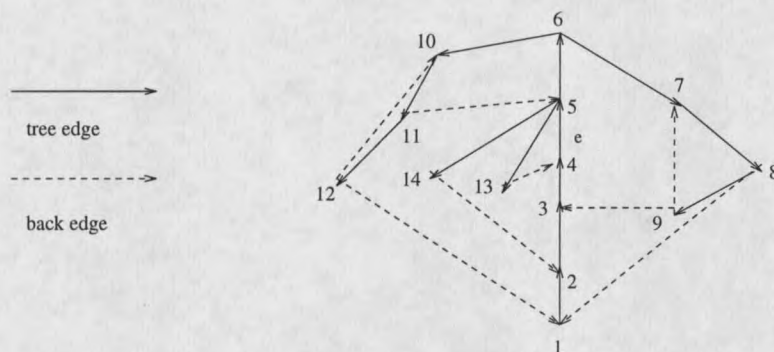


Figure 5.1: An illustration of some of the definitions

5.2 Planarity Testing

We start by computing $\text{att}(e)$. The planarity of e will be decided at the same time.

Consider any edge $e = [a, b]$. If e is a back edge, then its only attachment is e itself. therefore $\text{att}(e) = [[[b], []]]$. Otherwise, let $e_1, \dots, e_k$ be the successors of e in increasing order by their ϕ values. We first recursively compute $\text{att}(e_i)$ for each successor e_i of e . Then we compute $\text{att}(e)$ in four steps:

ALGORITHM *att*.

Step 1. For $i = 1, \dots, k$, delete all occurrences of b appearing in blocks within $\text{att}(e_i)$. Because these occurrences appear together at the end of the blocks that are contained in the last pairs of $\text{att}(e_i)$ only, a simple list traversal suffices to delete all these occurrences in time $O(k + \text{number of deletions})$. After this, initialize $\text{att}(e)$ to be $\text{att}(e_1)$.

Step 2. For $i = 2, \dots, k$, merge all the blocks of $\text{att}(e_i)$ into one intermediate block B_i . (see Figure 5.2).

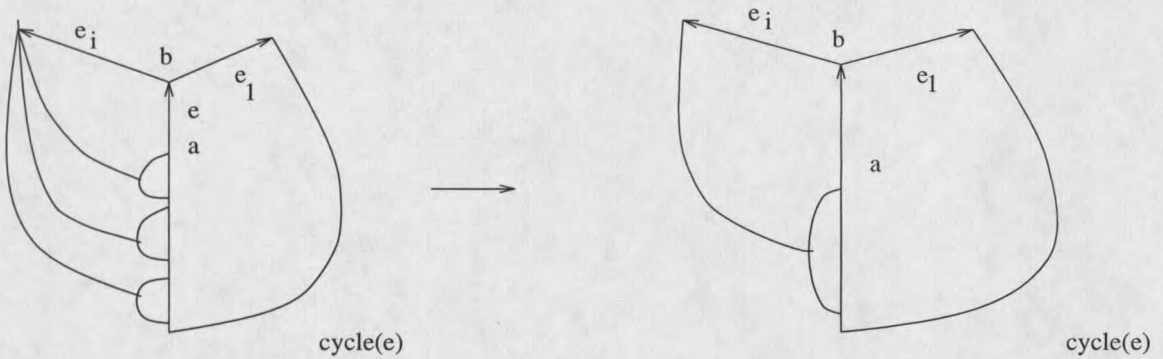


Figure 5.2: Merge all the blocks of $\text{att}(e_i)$ into one intermediate block B_i

Step 3. Merge blocks in $\text{att}(e)$. (see Figure 5.3).

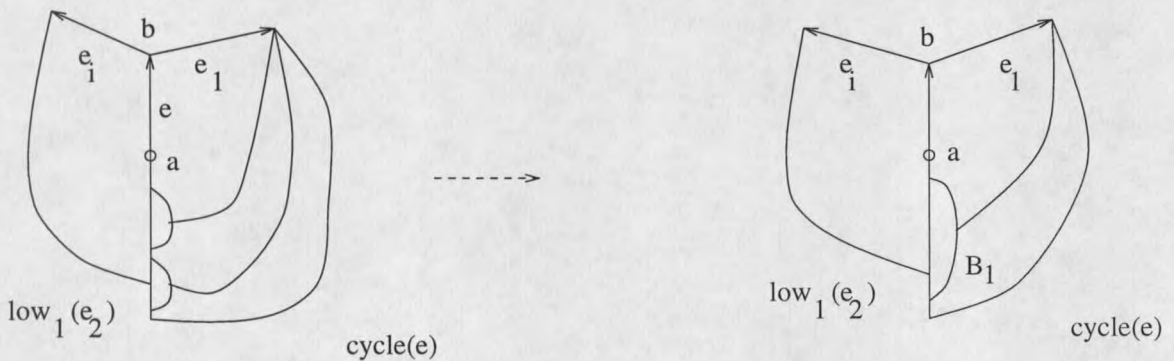
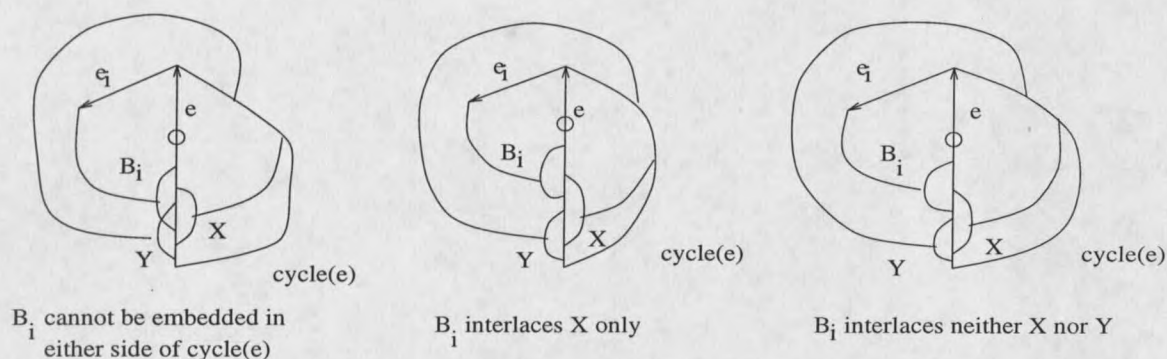


Figure 5.3: Merge blocks in $\text{att}(e)$

Step 4. For $i = 2, \dots, k$, add blocks B_i into $\text{att}(e)$. (see Figure 5.4).

Figure 5.4: Add blocks B_i into $\text{att}(e)$

5.3 The Maximal Planarization Algorithm

Now we consider the maximal planarization algorithm: find a minimal set of edges whose deletion results in a planar graph. We can always find a maximal planar subgraph of G by deleting back edges only, since all the tree edges form a forest, which is planar.

We cannot build a maximal planar subgraph of $\text{sub}(e)$ by simply putting together the maximal planar subgraphs of $\text{sub}(e_1), \dots, \text{sub}(e_k)$, and deleting those back edges causing failure in the algorithm. The reason is that after these edges are deleted, it may turn out that some other edges, which we deleted for making $\text{sub}(e_1), \dots, \text{sub}(e_k)$ planar, would not have had to be deleted at all. We avoid this difficult situation by constructing such maximal subgraph $S_1, \dots, S_k$ of $\text{sub}(e_1), \dots, \text{sub}(e_k)$ that they can be used to construct a planar subgraph S of $\text{sub}(e)$ without further deletion of edges. Two measures are taken for this purpose. Firstly, those back edges in $\text{sub}(e_i)$ that can cause failure in Step 3 or Step 4 of the algorithm are deleted before a maximal subgraph of S_i is recursively computed. Secondly, the information where blocks of $\text{sub}(e_i)$ are allowed to interlace is passed to the recursive call that computes S_i , so that

when the returned S_i is merged to $sub(e)$, Step 2 of the algorithm can also be performed successfully without deletion. Since the planar subgraph S of $sub(e)$ computed by our algorithm may be used to build a larger planar subgraph of G in the same way as we use $S_1, S_2, \dots, S_k$ to build S , we also need to know where in S blocks are allowed to interlace. This approach leads naturally to the concept of l -planar subgraphs, which is a generalization of the concept of strongly planar subgraphs.

Consider an edge $e = [a, b]$ and a vertex l on the tree path from $low_1(e)$ to a . An attachment $[u, v]$ of e is l -normal if $low_1(e) < v < l$. A block of attachments is l -normal if it contains some l -normal attachment. Let D be the list representation of a nonempty block of attachments. Define $second(D)$ to be the second smallest element in the set $\{x : x \in D\} \cup \{n+1\}$, and define $second([]) = n+1$. Then D is l -normal if and only if $low_1(e) < first(D) < l$ or $second(D) < l$. The two mappings $first$ and $second$ can be maintained during the computation of $att(e)$ in $O(1)$ time for each modification to $att(e)$.

We say that subgraph $sub(e)$ is l -planar if e is planar and l -normal blocks of $att(e)$ do not interlace. See Figure 5.5, where (a) is l -planar, but (b) is not. Edge e is l -planar if $sub(e)$ is l -planar.

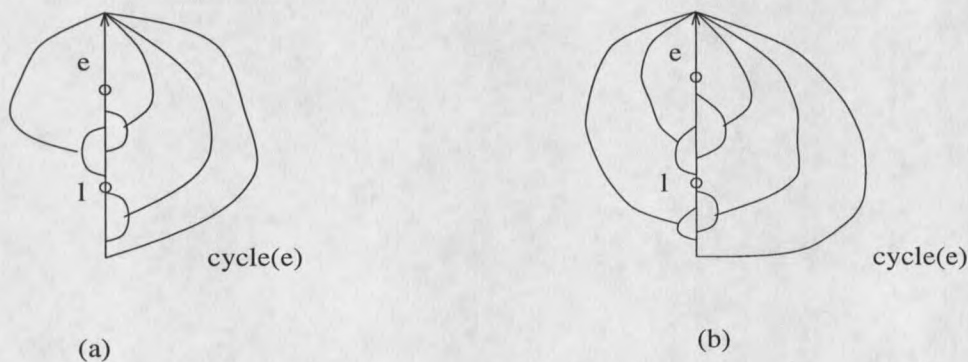


Figure 5.5: (a) is l -planar, but (b) is not.

Now we summarize the maximal planar subgraph algorithm. We take a connected undirected graph as input, and convert it into a DFS representation $G = (V, T, B)$. At some time, we compute the two mappings $succ$ and N , where, for each $e \in T$, $succ(e)$ gives the successor edges of e in increasing order of their heads, and for each $v \in V$, $N(v)$ gives the number of descendants of v . We assume that there is a dummy edge $e_0 = [0, 1]$ such that $succ(e_0)$ gives the list of tree edges leaving the root. The preprocessing takes $O(m)$ time.

We summarize the maximal planar subgraph algorithm below.

```

procedure lplanar( $e, l$ );
begin let  $e = [a, b]$ ;
    if  $e \in B$  then
return  $[[b], []]$ ;
    end if;
    if  $e$  has no successors then return  $[]$ ;
    let  $e_1, \dots, e_k$  be the successors of  $e$  not marked as 'deleted';
    split  $tree(e)$  into  $tree(e_1), \dots, tree(e_k)$ ;
    organize  $e_1, \dots, e_k$  into a heap based on their  $low_1$ 
    values, with the smallest one on the top;
    let  $e_1$  be the edge on the top of the heap;
    delete  $e_1$  from the heap;
     $att(e) := lplanar(e_1, l)$ ;
    delete all the occurrences of  $b$  from the top blocks of  $att(e)$ ;
     $j := 1$ ;
     $i := 2$ ;
    while heap is not empty do
        let  $e_i$  be the edge on the top of the heap;
        if  $low_1(e_i) \geq b$  then

```

```

    delete  $e_i$  from the heap;
    dummy :=  $lplanar(e_i, b)$ ;
elseif Condition AA is true then
     $v := tree(e_i).low_1$ ;
    while  $v = tree(e_i).low_1$  do  $tree(e_i) := delete(tree(e_i), v)$ ;
if  $e_i$  is a back edge then delete  $e_i$  from heap;
else modify heap;
elseif Condition BB is true then
     $v := tree(e_i).low_2$ ;
    while  $v = tree(e_i).low_2$  do  $tree(e_i) := delete(tree(e_i), v)$ ;
else delete  $e_i$  from the heap;
     $att(e_i) := lplanar(e_i, b)$ ;
    merge blocks of  $att(e_i)$  into one block  $B_i$ ;
    delete all occurrences of  $b$  from top blocks of  $att(e_i)$ ;
    if  $i=2$  then perform Step 3 of Algorithm 1.
    merge  $B_i$  into  $att(e)$  as described in Step 4;
     $i := i+1$ ;
endif
end while;
return  $att(e)$ ;
end;
```

Condition AA:

$(j = 1 \text{ and } low_1(e_i) < h_2)$ or $(low_1(e_j) < low_1(e_i) < h_2)$ or $(low_1(e_j) < low_1(e_i) < \min(h_3, l))$

Condition BB:

$(low_2(e_i) < \text{band}low_1(e_i) < h_2)$ or $(low_2(e_i) < \text{band}low_1(e_1) < low_1(e_i) <$

$\min(h_3, l))$ or $(low_2(e_i) < landlow_1(e_i) < h_3)$

```

procedure delete(r, v);
  begin if r is a leaf then
    mark the back edge stored in r as 'deleted';
    return null;
  else [rl, rr] := [r.lchild, r.rchild];
    if v = rl.low1 or v = rl.low2 then
      return merge(delete(rl, v), rr);
    else return merge(rl, delete(rr, v));
  end;

```

Chapter 6

The Cycle-Packing Method

In this chapter we discuss the last method of planarization involved in our study — the cycle-packing method.

6.1 Introduction

Let $G = (V, E)$ be an undirected simple graph. We may assume that G is biconnected because if G has biconnected components G_i , $i = 1, 2, \dots, m$, and if H_i is a maximum planar subgraph of G_i , then $H = H_1 \cup H_2 \cup \dots \cup H_m$ is a maximum planar subgraph of G [14].

To embed a graph H in a *book* is to order its vertices on the spine of the book and its edges on the pages, in such a way that edges residing on the same page do not cross. The first phase of the algorithm devises an ordering of the set of vertices, $O = (v_1, \dots, v_n)$ while the second phase embeds a maximal number of edges in two pages with respect to O . Therefore the planar subgraph generated by the algorithm is embeddable in a two-page book. It is known[15] that a graph H admits a two-page embedding if and only if it is the subgraph of a planar hamiltonian graph $\tilde{H}$. The ordering of the vertices on the spine must correspond to a hamiltonian cycle of $\tilde{H}$. These remarks justify the strategy used in phase 1: attempt to order the vertices of the input graph according to

a hamiltonian cycle.

A graph G is called an *overlap graph* (or *cycle graph*) if its vertices can be placed in one-to-one correspondence with a family I of intervals on a line. We say that two intervals *overlap* if they intersect and neither one is contained in the other. Two vertices of G are connected by an edge if and only if their corresponding intervals overlap. Phase 2 of the algorithm is equivalent to solving the following problem: color a maximum number of vertices of an overlap graph red or blue (this is equivalent to drawing the edges above or below the line) such that each of the two color classes forms an independent set; or, find a maximum bipartite subgraph of an overlap graph. This problem, however, is NP-complete[16].

Goldschmidt and Takvorian[20] devised the following greedy algorithm to construct a maximal bipartite subgraph of the overlap graph: find a maximum independent set, delete it from the graph, and find a maximum independent set in the remaining graph. This can be done in polynomial time since finding the maximum independent set of an overlap graph was shown to be polynomially solvable by Gavril [17]. Clearly, the two independent sets obtained induce a bipartite subgraph of the original graph. Even though this greedy algorithm does not guarantee a maximum bipartite subgraph, it is proven hereafter that the number of vertices in the subgraph obtained is at least $3/4$ the number of vertices of a maximum bipartite subgraph.

Theorem 5.1[16] Let $G = (V, E)$ be an undirected graph. Let X be a maximum independent set in G and let Y be a maximum independent set in the subgraph induced by $V \setminus X$. Let B be the set of vertices of a maximum bipartite subgraph of G . Then $|X| + |Y| \geq 3/4|B|$.

We conclude that, if the ordering O corresponds to an hamiltonian cycle in a maximum planar subgraph H of G or in some planar edge-augmentation

of H , that is, a graph obtained from H by adding zero or more edges while preserving its planarity, then the planar subgraph yielded by the algorithm has at least $3/4$ the number of edges of a maximum planar subgraph of G .

6.2 The Cycle-Packing Algorithm

Let $G = (V, E)$ be the input graph. As indicated before, in Phase 1 of the algorithm, we try to find a hamiltonian cycle in G . Even though the problem of finding a hamiltonian cycle in a graph is NP-complete, a fast randomized algorithm has been proposed that *almost certainly* finds a hamiltonian cycle in a graph on n vertices with average vertex degree $O(c \log n)$ (where $c \geq 5$ is a constant) and uniformly distributed edges[19]. *Almost certainly* is defined as "with probability $\rightarrow 1$ as $n \rightarrow \infty$ ". The algorithm attempts to find a hamiltonian cycle in G from starting vertex s , returning success if it succeeds and failure otherwise. We apply this algorithm on the input graph during the first phase of our heuristic until a hamiltonian cycle is obtained, which then determines an ordering $O = (v_1, \dots, v_n)$ of the vertices. Different hamiltonian cycles can be generated by using different starting vertices.

Additionally, we have implemented a greedy deterministic algorithm that attempts to find a hamiltonian cycle in G and thus an ordering O . The first vertex in O is the minimum degree vertex of G . After the first k vertices of the ordering have been determined, say $(v_1, \dots, v_k)$, v_{k+1} is selected from the vertices adjacent to v_k having the least adjacencies in the subgraph G_k , induced on $(V - \{v_1, \dots, v_k\})$. If v_k has no neighbors in G_k , we select v_{k+1} as a vertex of minimum degree in G_k . Even though this algorithm usually fails to obtain a hamiltonian cycle, it does actually produce a good ordering.

To start the second phase of the algorithm we place the vertices of G on a line in accordance to the ordering O . Let (v_i, v_j) and (v_k, v_l) be two edges of

the input graph, such that $v_i < v_j$ and $v_k < v_l$, and let us assume without loss of generality that $v_i < v_k$.

We define a graph $H = (E, A)$, where to each edge of the input graph G corresponds a vertex in H , and two vertices of H are connected if and only if the corresponding edges in G overlap in the ordering O obtained from phase 1 of the heuristic. Hence, H is the overlap graph corresponding to the particular given ordering. We apply a greedy approach to construct a maximal bipartite subgraph of H : using the polynomial algorithm from [17], we first find a maximum independent set in the remaining graph.

While finding a hamiltonian cycle can be done in $O(n \log^2 n)$ time[18], Gavril's algorithm[17] used in the second phase runs in $O(n^3)$ for an overlap graph of n vertices. But, since the vertices of the overlap graph that we construct in the second phase correspond to the edges of the original input graph G , the overall complexity of the second phase is $O(m^3)$. Very recently however, a new algorithm was proposed by Asano *et al.*[18]. Its corrected version [20] finds the maximum independent set of the overlap graph in $O(mn)$ time when applied to this problem. It is clear that despite this improvement, the overall complexity of the algorithm is dominated by the second phase, which was also confirmed by our empirical results.

Chapter 7

The OSF/Motif MAXPLANAR Interface

7.1 An Introduction to the X Window System

The X Window System[24] is an industry-standard software system that allows programmers to develop portable graphical user interfaces. One of the most important features of X is its unique device-independent architecture. X allows programs to display windows containing text and graphics on any hardware that supports the X protocol without modifying, recompiling, or re-linking the application. This device independence, along with X's position as industry standard, allows X-based applications to function in a heterogeneous environment consisting of mainframes, workstations, and personal computers.

One important difference between X and many other window systems is that X does not define any particular user interface style. X provides *mechanisms* to support many interface styles rather than enforcing any one *policy*. Many window systems support a particular style user interface. In contrast, X provides a flexible set of primitive window operations, but carefully avoids dictating the look or feel of any particular application's user interface components. Instead, X provides a device-independent layer that serves as a base

for a variety of interface styles. Therefore, the basic X Window System does not provide user interface components such as button boxes, menus, or dialog boxes often found in other window systems. Most applications depend on higher level libraries built on top of the basic X protocol to provide these components.

7.2 An Introduction to OSF/Motif

Programming in X can be cumbersome. To create a scroll bar, a programmer must create a thin rectangular window on the screen and then draw the scroll bar in it. The programmer must then write code that manages the scroll bar when the user manipulates it. For example, if the user clicks or drags in the scroll bar, the program must animate it appropriately. Using X, a single scroll bar might require an immense amount of code. Furthermore, when many programmers create their own scroll bars in their own way, each will look and behave slightly differently. This inconsistency annoys users as they move between applications.

Motif solves this problem[25]. It sits on top of X and provides a set of preconstructed user interface objects called *widgets*. These widgets can be placed on the screen by an application program.

When you need a scroll bar in a Motif application, you call a function that creates one for you in the desired location. The scroll bar appears on the screen as a nicely drawn object that is consistent across applications. Even better, Motif does all of the scroll bar management internally. When a user adjusts the scroll bar, Motif animates it appropriately on the screen. The slider moves, the arrow heads blink, and so on. When the user manipulates the scroll bar, Motif relays the new value chosen by the user to the program code.

The beauty of Motif is that almost all of the user interface overhead has already been programmed. You decide which widgets you need to use to create the user interface that you want. Motif functions position the widgets on the screen in the right location and at the right size. At the same time, Motif uses functions known as *callbacks* that notify the program when a user adjusts one of the widgets. The only additional code you must write is the “thinking” portion of the application, the part that makes the application respond correctly to user actions.

7.3 MAXPLANAR

The maximal planar subgraph (MAXPLANAR) application is an editing and testing program for graphs. It creates and edits graph adjacency lists. It can also be used to perform graph planarization or to find maximal planar subgraphs. The MAXPLANAR application is composed of several test tools. The tools allow users to perform graph planarization and to obtain a maximal planar subgraph using the HT, CHT, PQ and GT methods.

The MAXPLANAR user interface is composed of three components — menu bar, work area and status indicator (see Figure 7.1).

MAXPLANAR Modules:

- `msg.c` — is the core of the application code and also contains the Motif user interface code for MAXPLANAR.
- `pq.c` — contains the callback function of the vertex addition method.
- `ht.c` — contains the callback function of the path addition method.
- `gt.c` — contains the callback function of the cycle-packing method.
- `cht.c` — contains the callback function of the edge addition method.

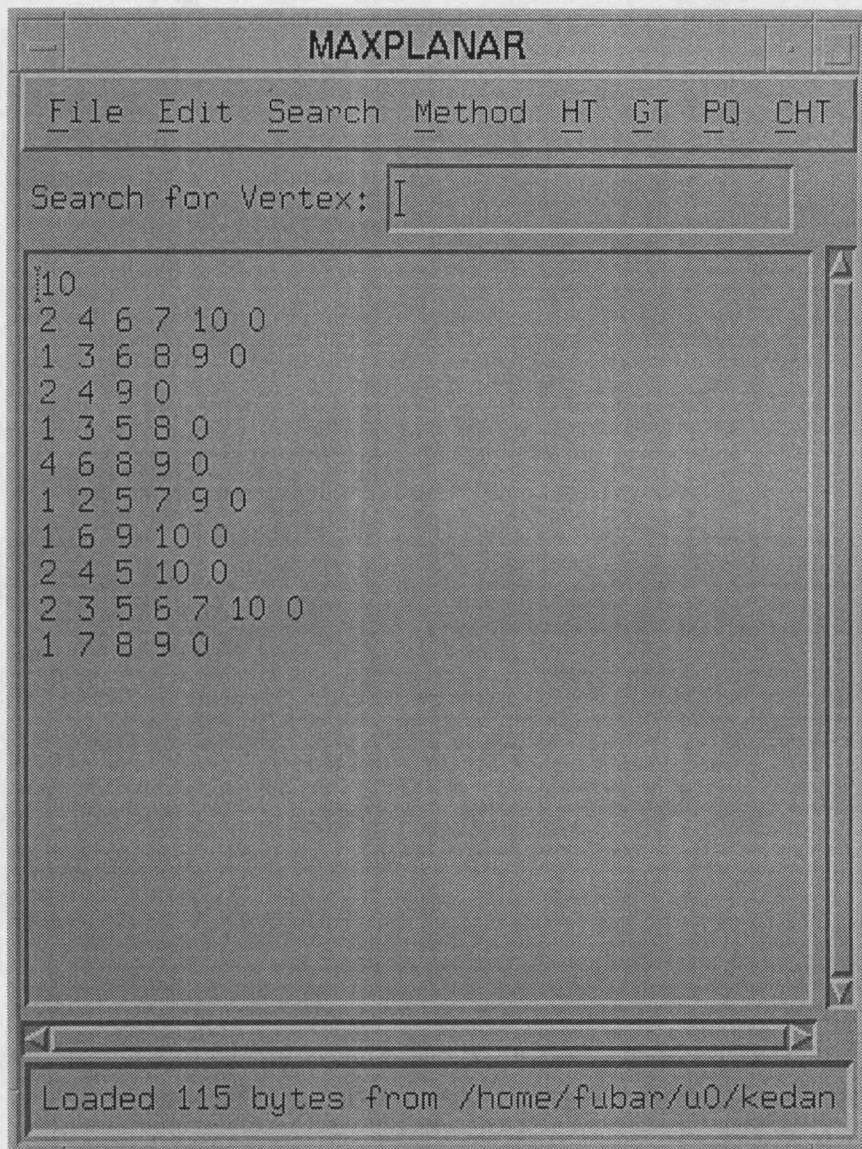


Figure 7.1: The MAXPLANAR user interface

Figure 7.2 shows the overall architecture of MAXPLANAR.

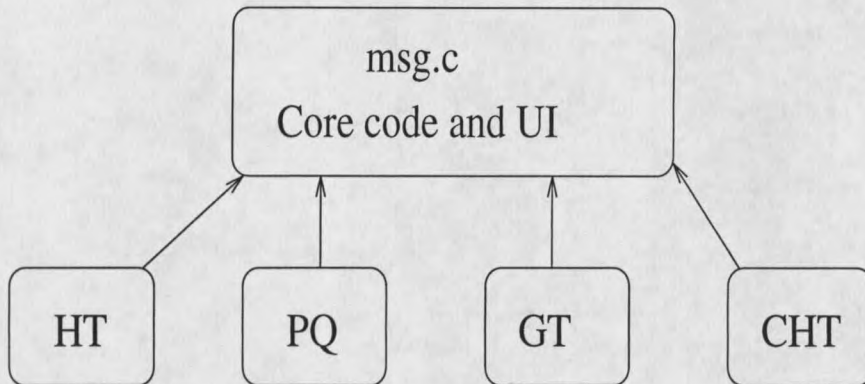


Figure 7.2: The overall architecture of MAXPLANAR

7.4 Graph Edit

When the user selects the “Open” option in the file menu, a file selection dialog appears (Figure 7.3) which lists all the object files in the currently selected directory. Currently, these graph files are designated with a “.g” extension. When the user selects the “OK” button, the box disappears and the file is loaded into the working area. If the user selects the “Cancel” button, MAXPLANAR resumes where the user left off in the currently status. If the user selects the “Filter” button, MAXPLANAR lets the user change the default directory to another directory.

If the user selects the “OK” button, the graph file is loaded into the working area. At the bottom of MAXPLANAR, the status area shows the name and the directory of the graph file.

When the user types the vertex number in the vertex window, the cursor will be automatically located in the first place of the vertex number in the graph. If the user selects the “Search all” option in the edit menu, all ver-

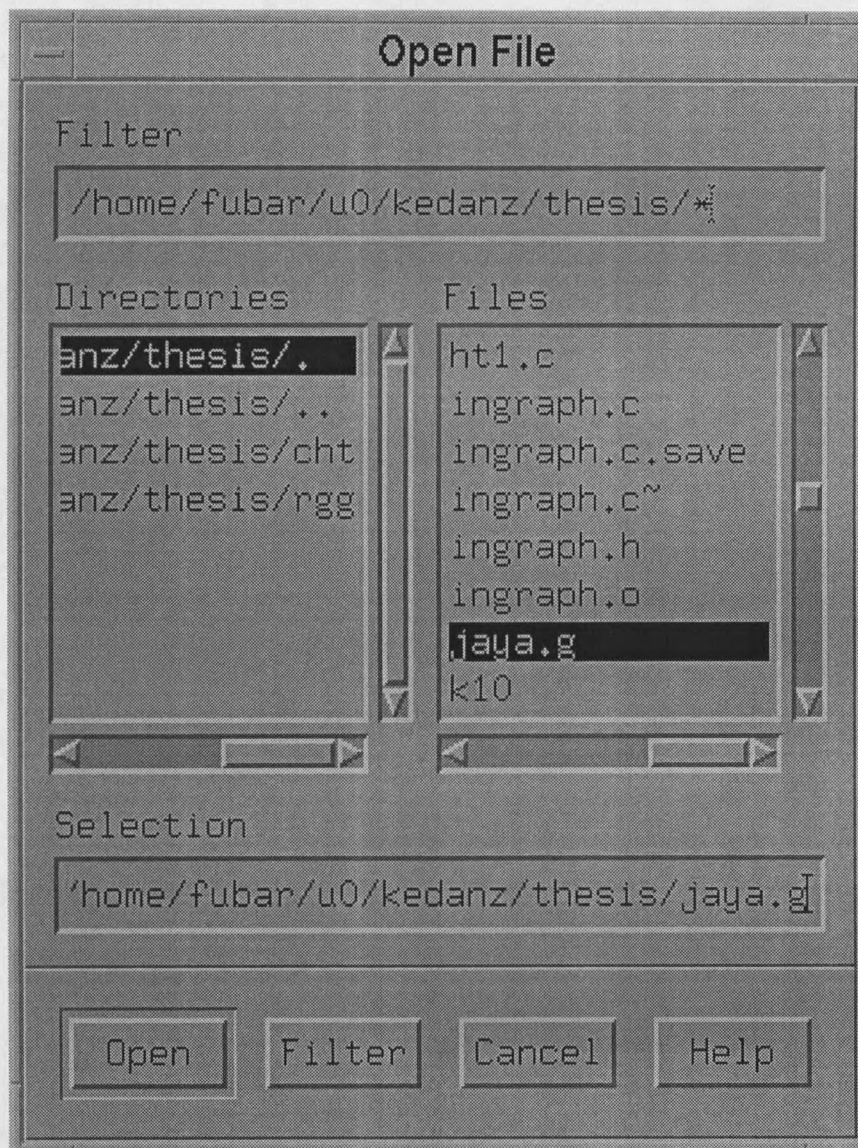


Figure 7.3: A file selection dialog window which lists all the object files in the currently selected directory.

tex numbers in the graph are highlighted. The status area shows how many highlighted vertices are counted. Figure 7.4 depicts a typical session.

If the user wants to replace one vertex number with another vertex number, he can simply select "Replace all" option from the edit menu.

The user also can select the "Cut", "Copy", "Paste", "Clear" options from the edit menu to edit a graph.

7.5 Graph Test

When the user wants to test a graph, first he needs to load the graph into the working area. The user can select different maximal planarization methods to apply to the graph. The status area shows the method the user selected. Figures 7.5 - 7.11 depict these typical operations.

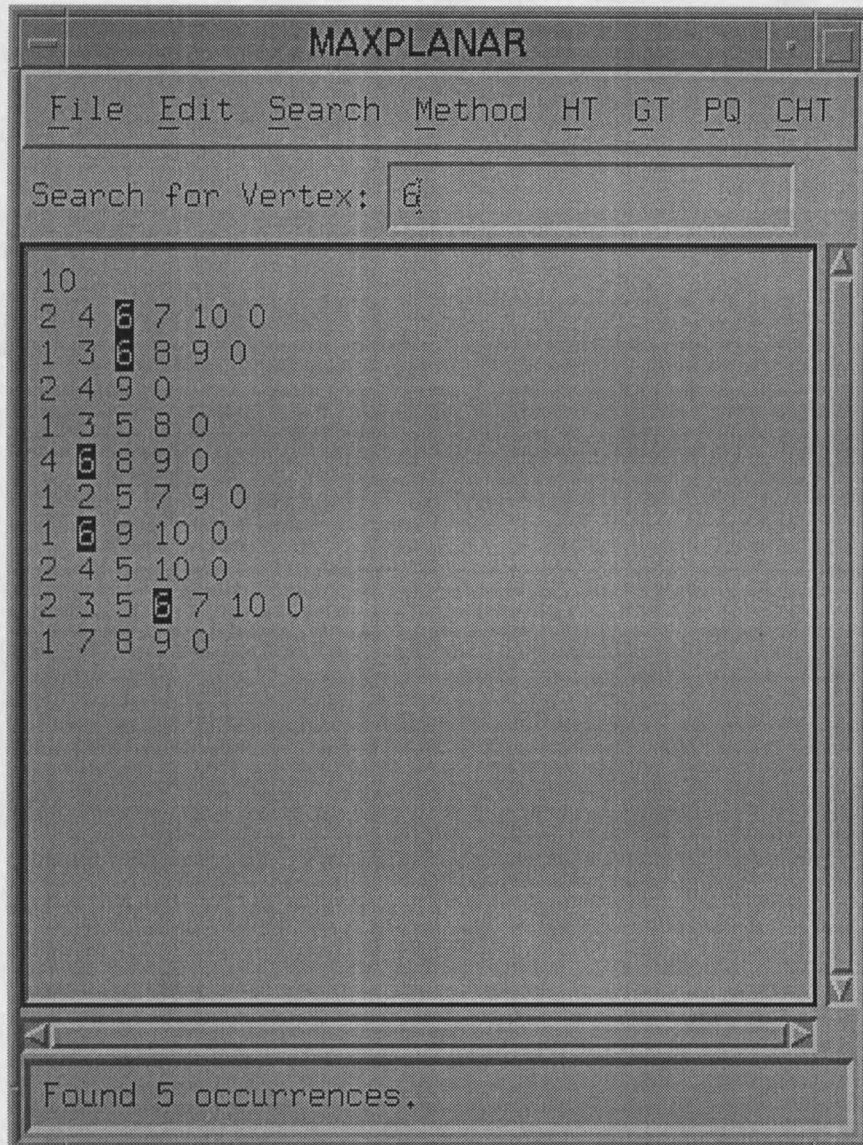


Figure 7.4: Highlight all vertex numbers 6 in the graph.

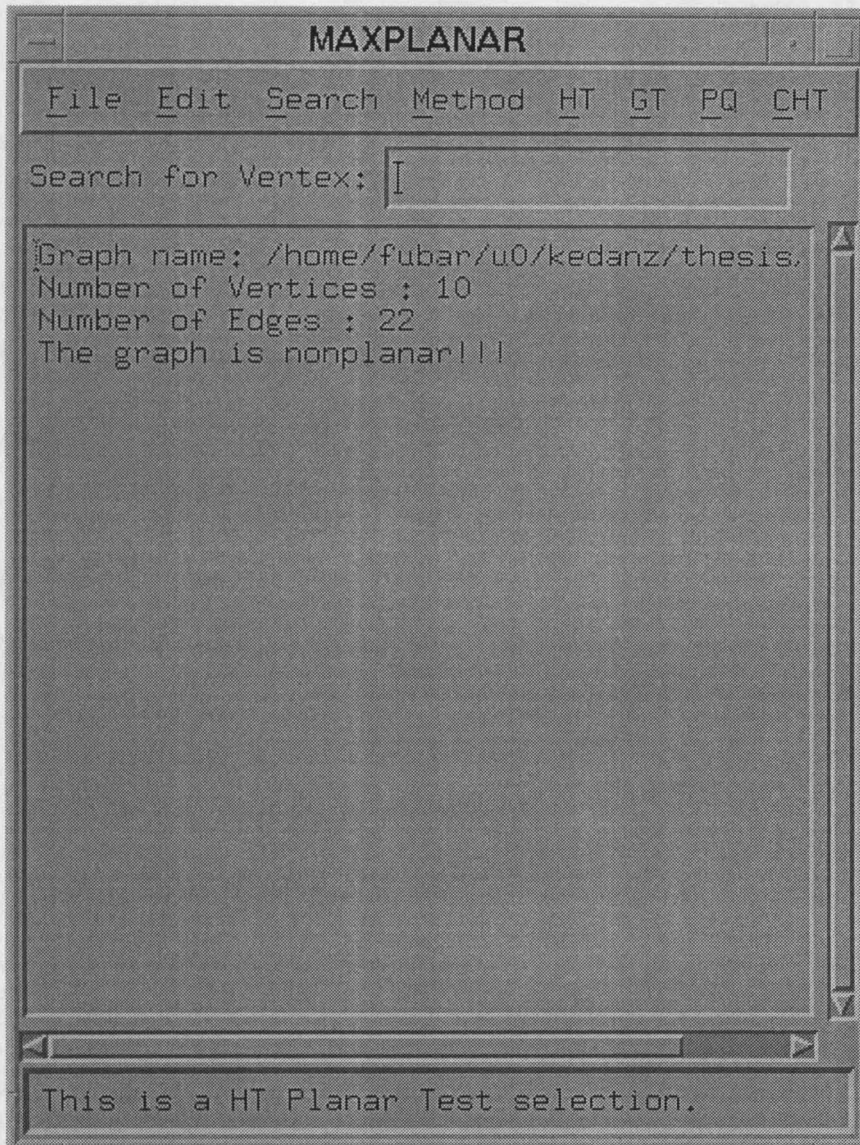


Figure 7.5: HT planarization test selection

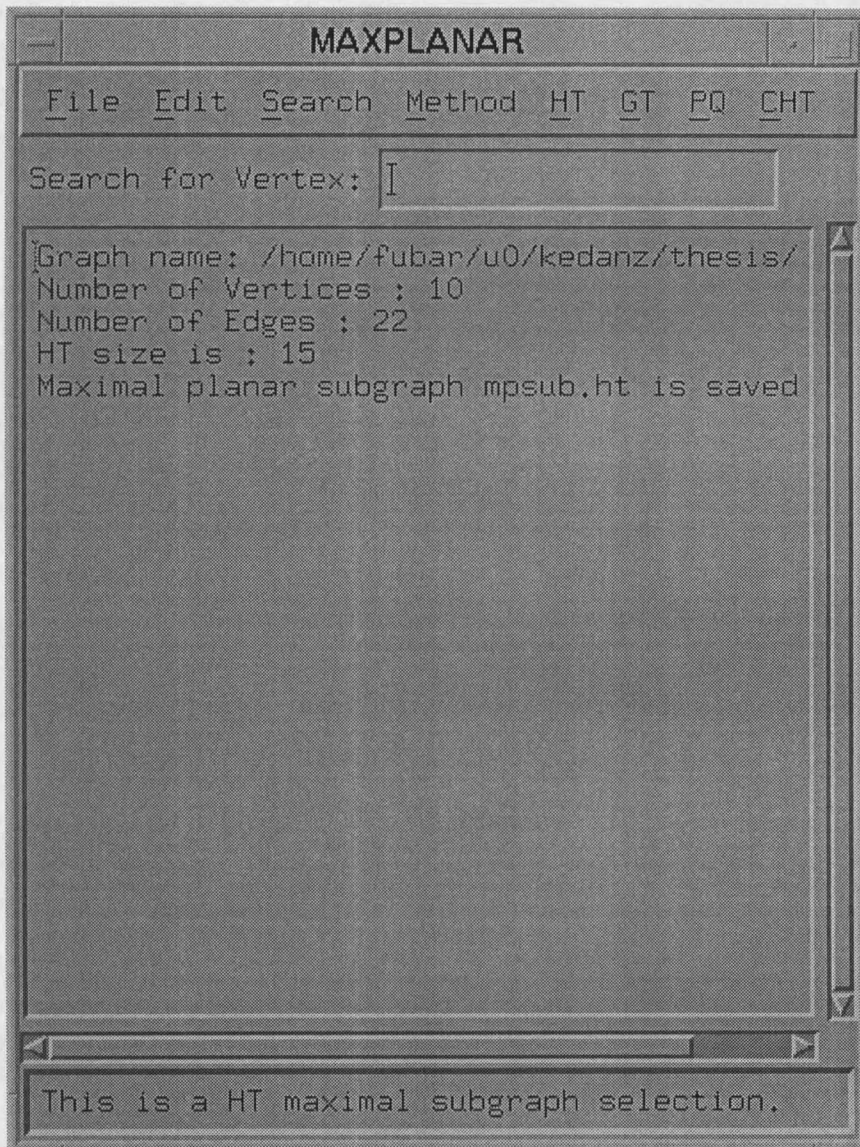


Figure 7.6: HT maximal planar subgraph selection

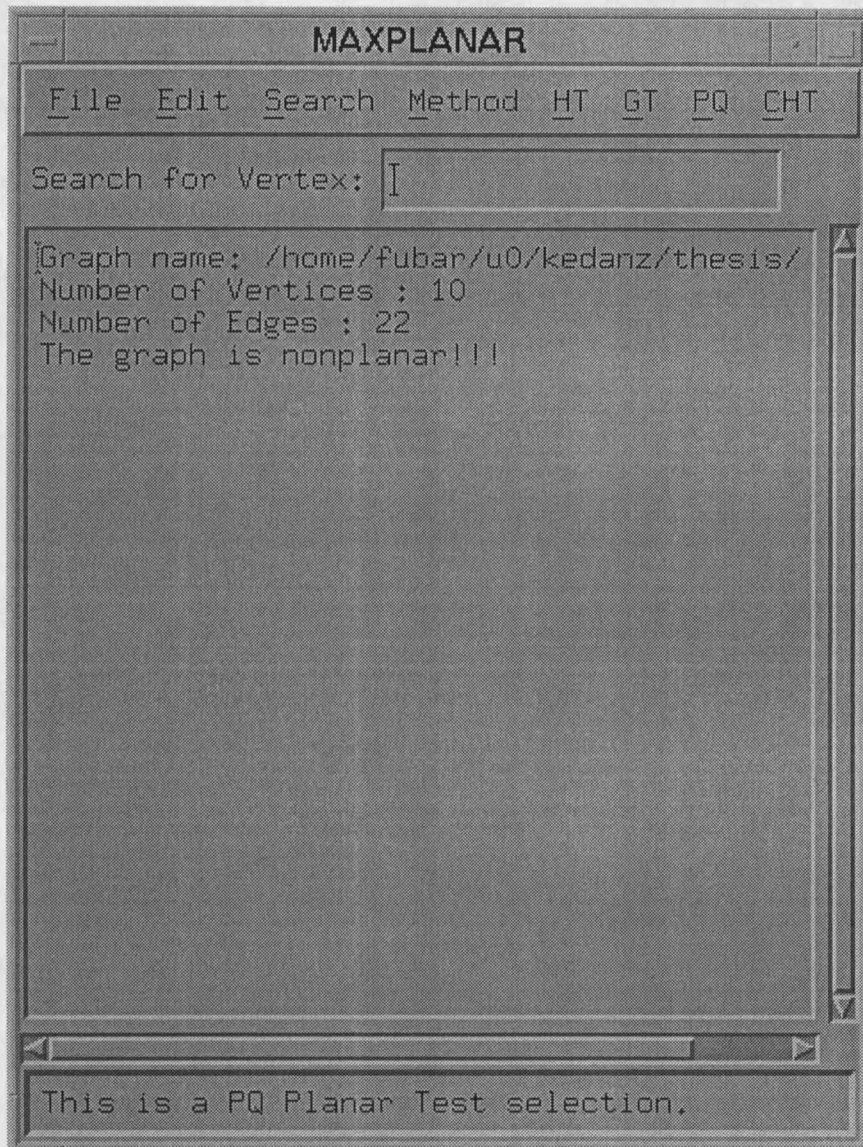


Figure 7.7: PQ planarization test selection

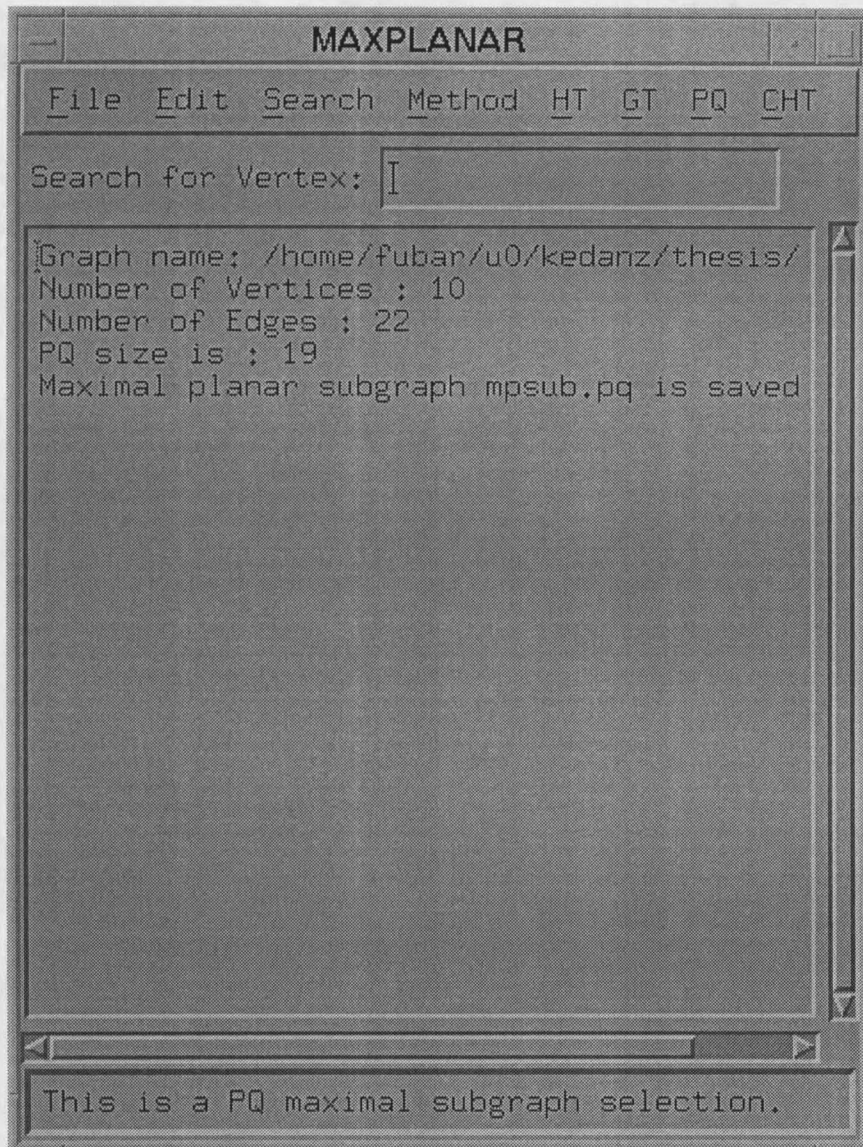


Figure 7.8: PQ maximal planar subgraph selection

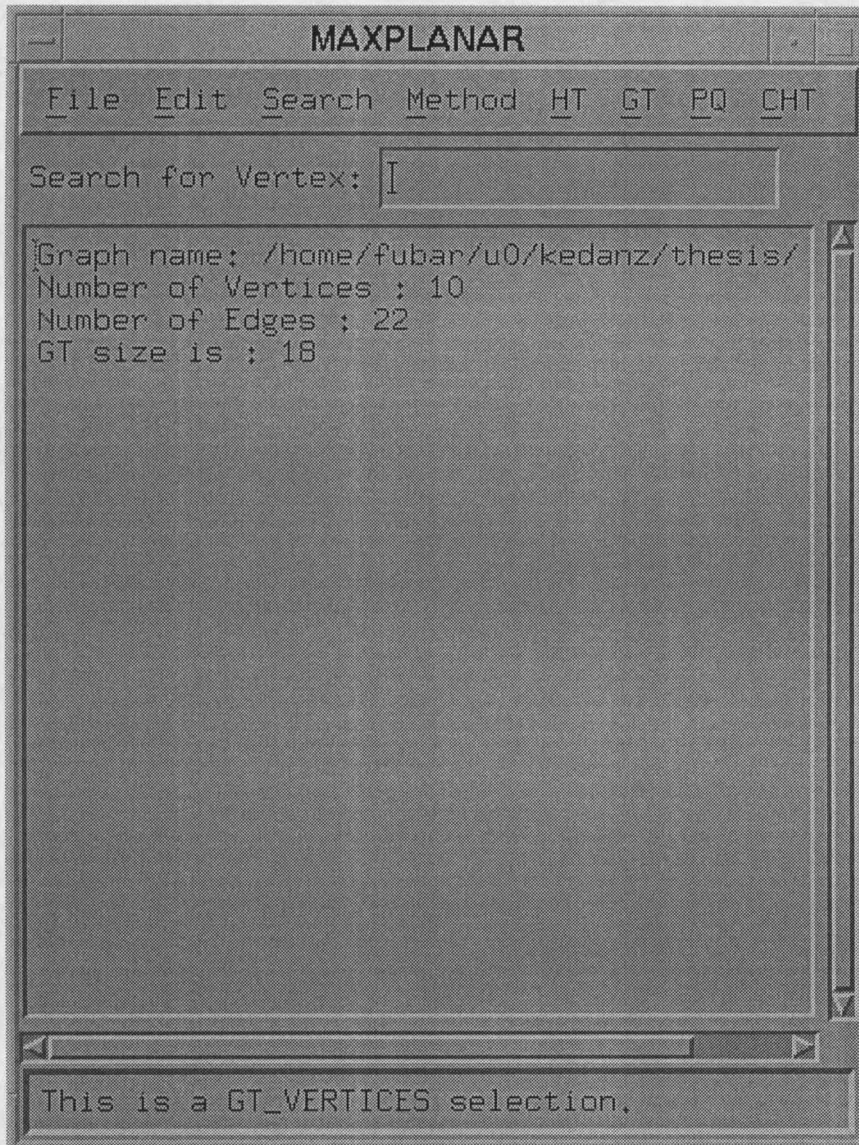


Figure 7.9: GT maximal planar subgraph by vertex number selection

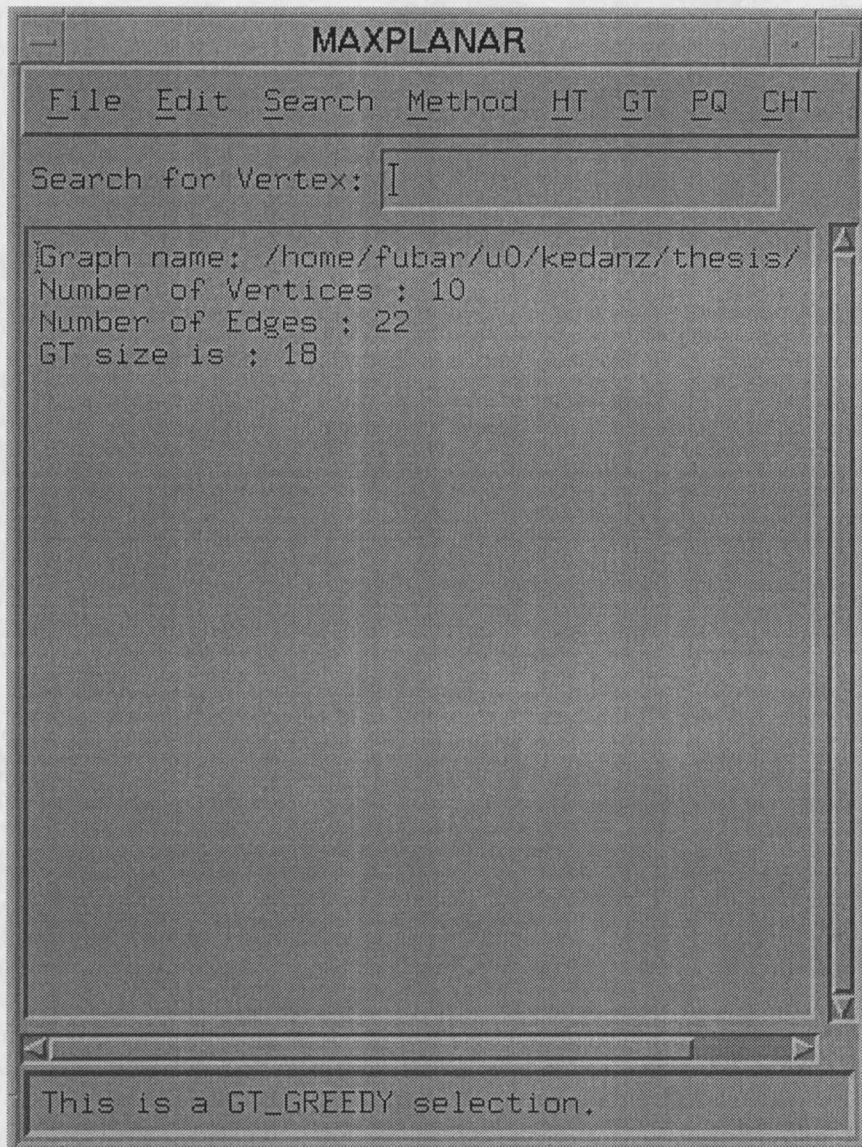


Figure 7.10: GT maximal planar subgraph by greedy order selection

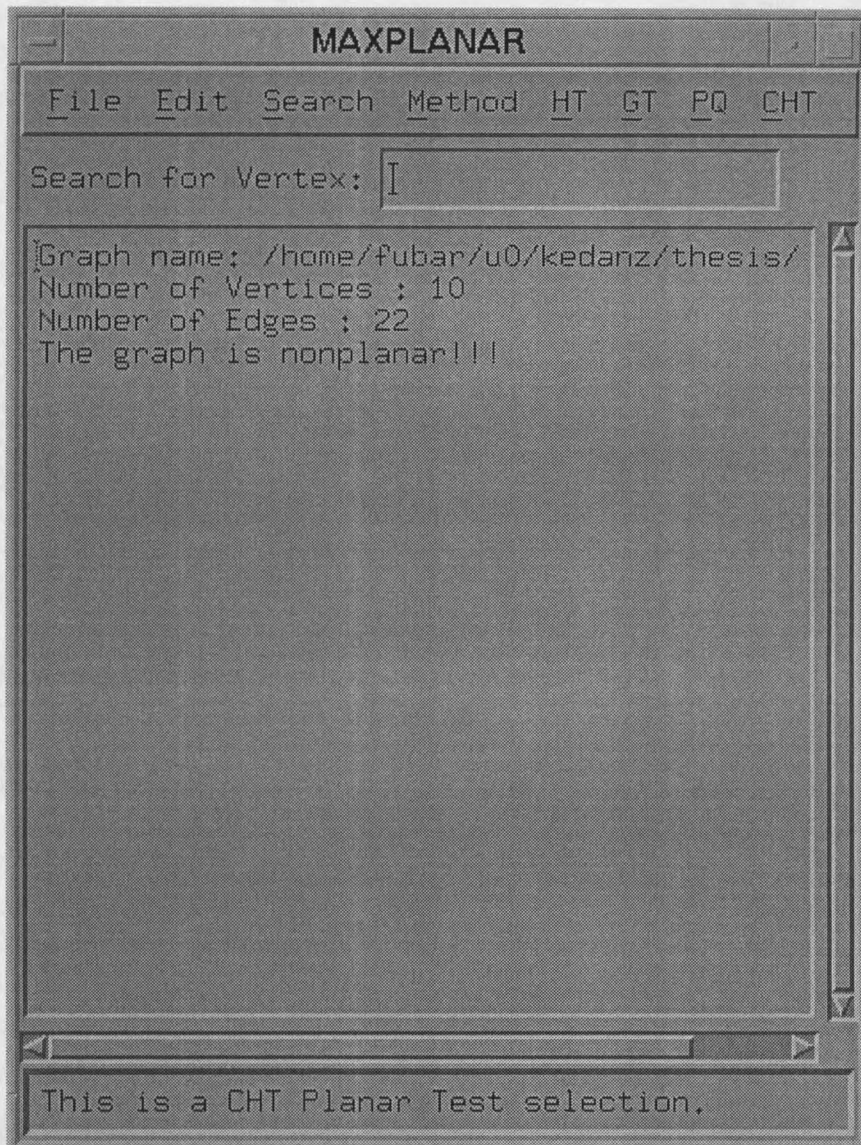


Figure 7.11: CHT planarization test selection

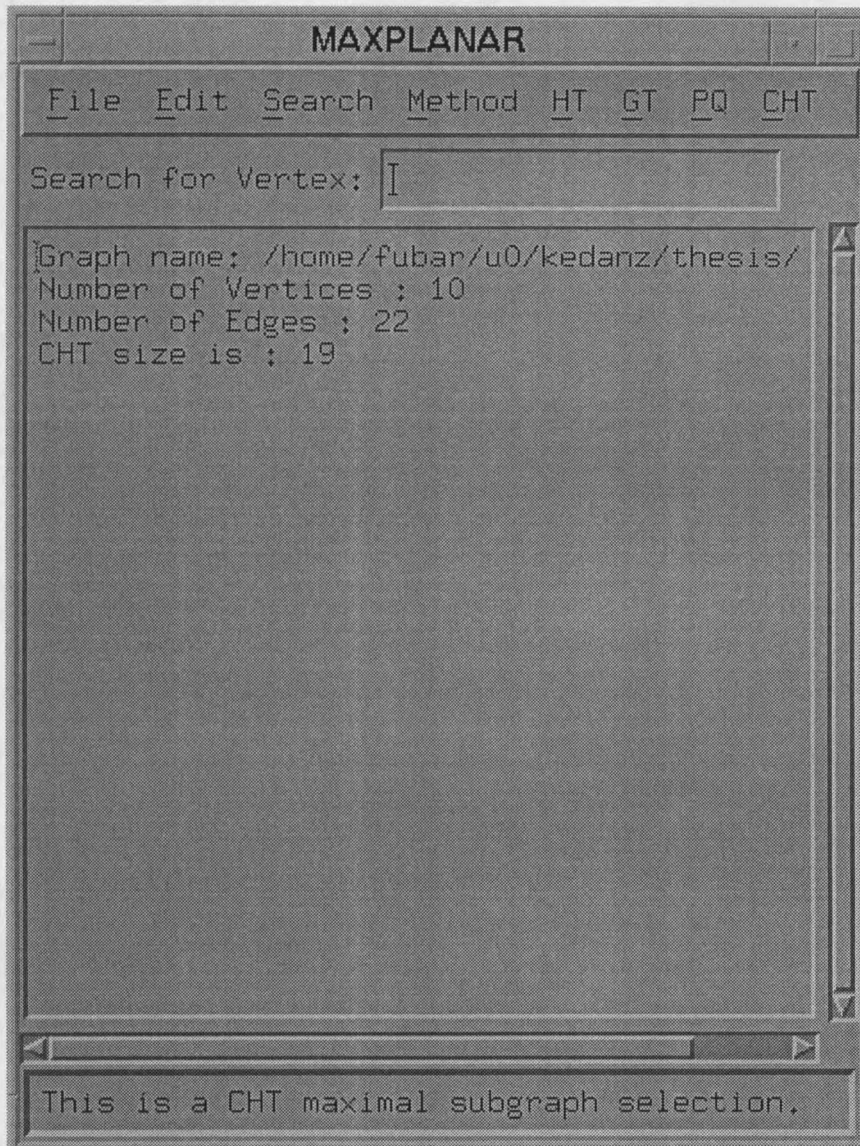


Figure 7.12: CHT maximal planar subgraph selection

Chapter 8

Computational Experience

8.1 Test Graph Generation

In addition to randomly generated instances, we selected a few special graphs, shown in Figures 8.1-8.6, either given in other papers or which have interesting adjacency structures or relevance to an application. Graph g_1 is hamiltonian and has maximum planar subgraph obtained by removing edges $\{4,8\}$ and $\{7,8\}$. g_2 has a maximum planar subgraph obtained by removing edge $\{u,v\}$. g_3 has a maximum planar subgraph obtained by removing edges $\{1,28\}$ and $\{9,12\}$. g_4 and g_5 from [11] and [12], respectively. g_4 has a structure similar to *circle graphs* which are used in 2-pin net channel routing for VLSI design[21] and in RNA secondary structure prediction in molecular biology[22]. g_5 is representative of graphs occurring in circuit layout applications. g_4 has a maximum planar subgraph formed by removing edges $\{1,4\}$ and $\{8,10\}$, while g_5 has a maximum planar subgraph obtained by deleting edges $\{1,2\}$, $\{3,4\}$ and $\{4,5\}$. g_6 is similar to program flow graphs occurring in information display systems[23] and has a maximum planar subgraph obtained by deleting edges $\{1,9\}$, $\{1,31\}$, $\{22,23\}$ and $\{23,32\}$.

Random graphs of several types were generated. A random graph $G_{n,p}$ has n vertices and an edge existing between any pair of vertices independently

with probability p .

Type I are random nonplanar graphs with unknown maximum planar subgraph size. Hence, only a relative measure of performance was possible for this class. Each graph was generated by considering a uniformly generated random number in the interval $(0,1)$ versus some chosen probability for each possible edge.

Type II are sparse planar-like graphs, *ie.*, graphs that are almost planar.

Type III instances are dense graphs in which m is much larger than n .

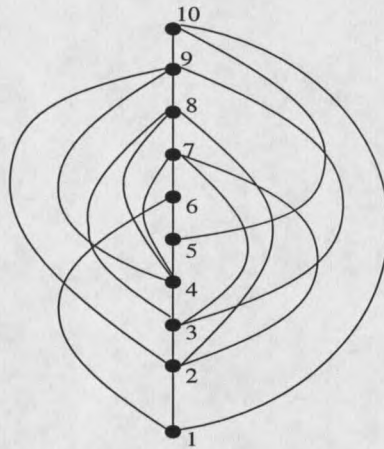
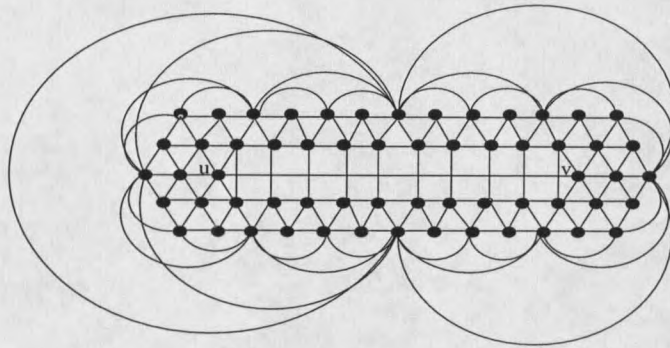
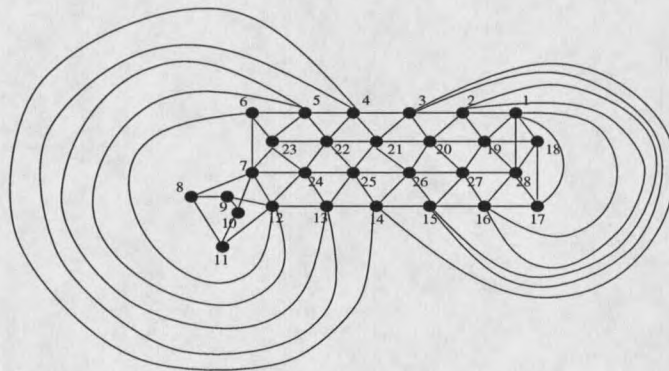
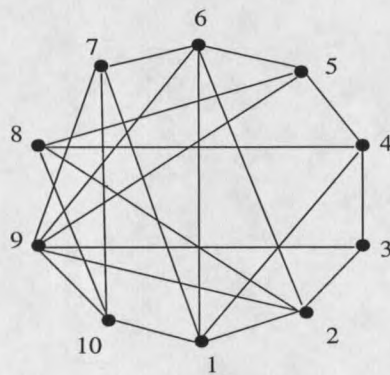


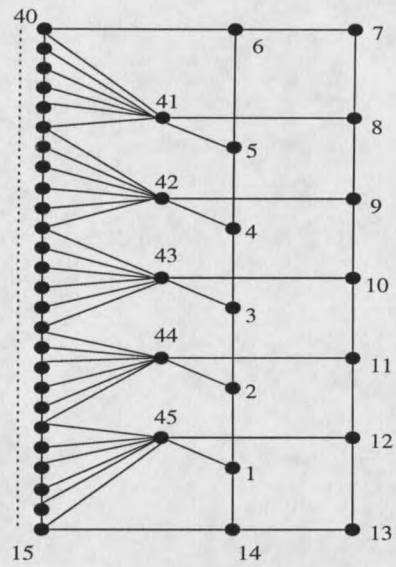
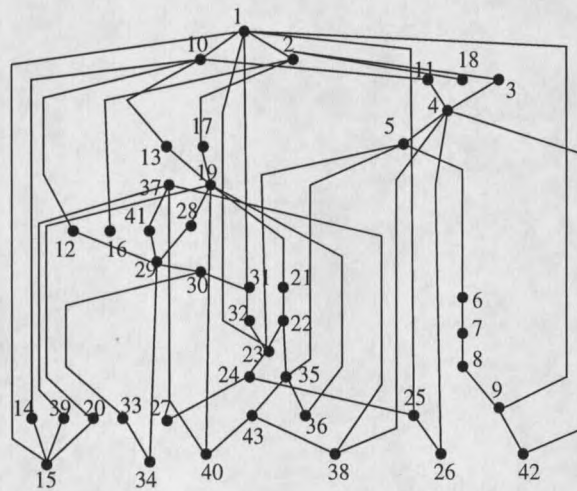
Figure 8.1: $g1$ graph

8.2 Analysis of Results

The basic performance indicator was solution quality, since in most applications computational costs are not as important as the degree of optimality.

Table 8.1 shows the performance of the heuristics on the special graphs of Figures 8.1-8.6 are shown. For those graphs, as a way of measuring relative performance, we employed the following ranking system: a heuristic was awarded k points for having the k^{th} best solution for each instance. Accordingly, the

Figure 8.2: g_2 graphFigure 8.3: g_3 graphFigure 8.4: g_4 graph

Figure 8.5: g_5 graphFigure 8.6: g_6 graph

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

¹ optimum solution

Table 8.1: Heuristics applied to the special graphs of Figures 8.1-8.6.

results were PQ(9), GT(12), HT(13), CHT(16).

Table 8.2 shows the performance of the heuristics on the Type I graphs. Under the same ranking system, the results were GT(21), PQ(39), CHT(51), HT(70). The superiority of GT is apparent on these graphs as it found the best solution in 19 of the 20 test graphs, with a minimum performance of 95%. The performance of HT is markedly worse than the other heuristics. In Figure 8.7 we depict the performance of the heuristics on Type I graphs.

Table 8.3 shows the performance of the heuristics on the Type II graphs. Under the same ranking system, the result were GT(20), PQ(43), CHT(58), HT(76). It should be noted that GT found the best solution in every case. The performance of HT is markedly worse than the other heuristics. In Figure 8.8 we depict the performance of the heuristics on Type I graphs.

Table 8.4 shows the performance of the heuristics on the Type III graphs. Under the same ranking system, the results were GT(18), CHT(23), PQ(30), HT(45). The superiority of GT is apparent on these graphs as it found the best solution in 12 of the 20 test graphs, with a minimum performance of 60%. The performance of HT is markedly worse than the other heuristics. The performance of CHT is better than PQ. In Figure 8.9 we depict the

graph	n	m	Size of planar subgraph			
			HT	PQ	CHT	GT
rg10.1	10	25	19	21	18	21
rg10.2	10	26	19	19	18	20
rg10.3	10	33	21	23	21	22
rg10.4	10	38	21	23	23	24
rg10.5	10	43	21	24	24	24
rg20.1	20	59	34	36	38	38
rg20.2	20	79	37	44	39	46
rg20.3	20	104	42	49	48	51
rg20.4	20	111	44	49	51	51
rg20.5	20	137	32	51	48	53
rg50.1	50	382	63	111	99	129
rg50.2	50	510	78	123	111	143
rg50.3	50	597	78	128	121	141
rg50.4	50	742	76	129	134	143
rg50.5	50	845	80	133	137	142
rg100.1	100	504	126	164	151	189
rg100.2	100	494	124	164	150	190
rg100.3	100	484	122	168	150	184
rg100.4	100	735	131	182	163	217
rg100.5	100	798	123	186	178	226

Table 8.2: Heuristics applied to random nonplanar graphs.

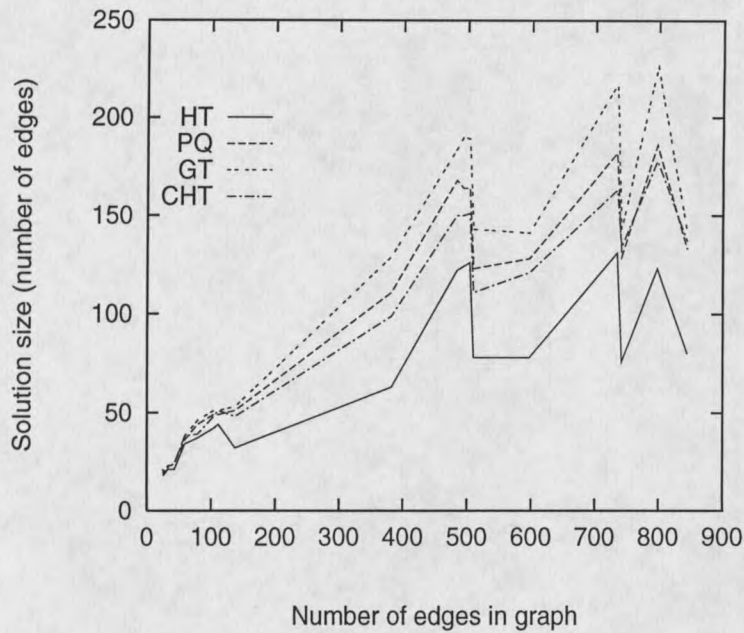


Figure 8.7: Performance of the heuristics on random nonplanar graphs

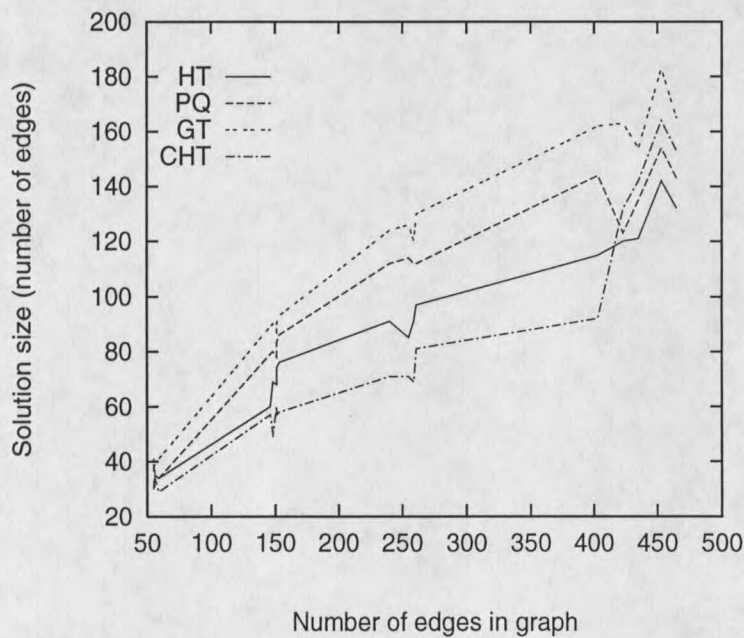


Figure 8.8: Performance of the heuristics on planar-like graphs

graph	n	m	Size of planar subgraph			
			HT	PQ	CHT	GT
pl20.1	20	56	34	34	34	39
pl20.2	20	59	34	37	36	41
pl20.3	20	55	31	39	35	41
pl20.4	20	55	31	37	36	38
pl20.5	20	56	35	38	34	39
pl50.1	50	148	69	80	74	90
pl50.2	50	151	74	78	83	87
pl50.3	50	151	68	83	80	90
pl50.4	50	146	60	79	82	89
pl50.5	50	153	76	86	77	93
pl70.1	70	259	91	112	102	120
pl70.2	70	240	91	112	101	124
pl70.3	70	255	85	114	107	126
pl70.4	70	240	91	112	101	124
pl70.5	70	261	97	112	106	130
pl90.1	90	403	115	144	131	162
pl90.2	90	423	120	123	130	163
pl90.3	90	435	121	135	124	154
pl90.4	90	465	132	143	137	165
pl90.5	90	453	142	154	127	183

Table 8.3: Heuristics applied to sparse nonplanar graphs.

graph	n	m	Size of planar subgraph			
			HT	PQ	CHT	GT
ds10.1	10	39	22	23	24	23
ds10.2	10	40	18	24	24	24
ds10.3	10	42	23	24	24	24
ds10.4	10	42	20	24	24	24
ds10.5	10	40	24	24	24	23
ds30.1	30	404	62	84	84	84
ds30.2	30	394	47	84	83	84
ds30.3	30	389	52	80	81	84
ds30.4	30	388	56	83	83	84
ds30.5	30	397	60	81	82	84
ds50.1	50	1105	86	141	144	143
ds50.2	50	1109	90	137	140	146
ds50.3	50	1119	81	144	144	165
ds50.4	50	1118	78	139	144	178
ds50.5	50	1110	81	139	142	154

Table 8.4: Heuristics applied to dense nonplanar graphs.

performance of the heuristics on Type III graphs.

8.3 Conclusions

In this paper, we presented an efficient implementation of a software package with a user friendly interface for several algorithms for finding maximal planar subgraphs of nonplanar graphs. The algorithms include the methods of path-embedding, edge-embedding, vertex-embedding, and cycle-packing.

From Figure 8.8 we conclude that for planar-like random graphs the vertex-addition method (based on PQ-trees) is better than the edge-addition method (CHT). However, for dense random graphs the edge-addition method seems to be better than others. (see Figure 8.9).

It should be indicated that the cycle-packing algorithm found the best solution in most cases, but it needed much more CPU time than the other

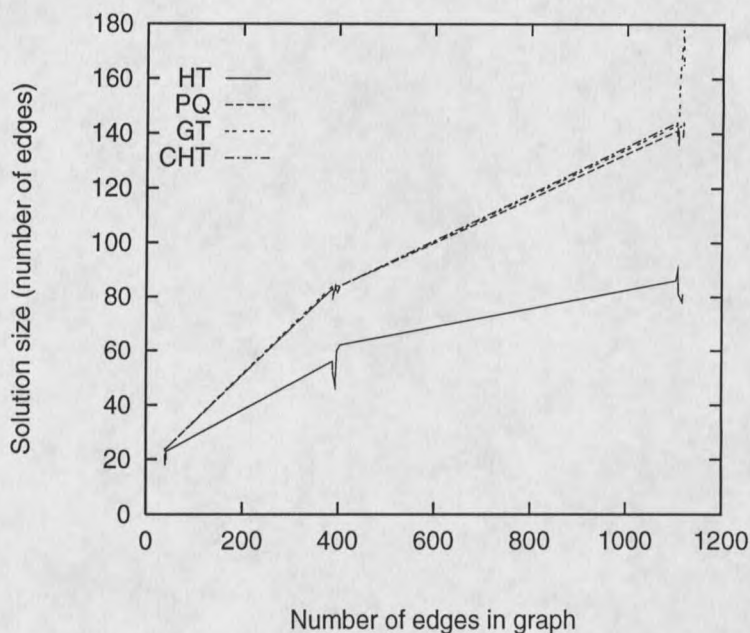


Figure 8.9: Performance of the heuristics on dense graphs

heuristics.

The performance of the path addition algorithm is worse than the other heuristics for some random graphs. (see Figure 8.7).

The best time bound for maximal planarization is $O(m \log n)$ by the edge addition algorithm. It is an open problem whether there exists a maximal planarization algorithm based on PQ-trees which also works in $O(m \log n)$ time, or whether there is a general maximal planarization algorithm that is faster than $O(m \log n)$.

Bibliography

- [1] Auslander, L., and S.V. Parter, On embedding graphs in the plane, *J. Math. Mech.* 11 (1961), pp. 517-523.
- [2] Hopcroft, J., and R.E. Tarjan, Efficient planarity testing, *J. ACM* 21 (1974), pp. 549-568.
- [3] Lempel, A., S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, *Theory of Graphs, Int. Symp. Rome* (1966), pp. 215-232.
- [4] Booth, K.S., and G.S. Lueker, Testing for the consecutive ones property, interval graphs and graph planarity testing using PQ-tree algorithms, *J. of Computer and System Sciences* 13 (1976), pp. 335-379.
- [5] Garey, M.R., and D.S. Johnson, *A Guide to the Theory of NP-completeness*, Freeman Co., San Francisco, 1979.
- [6] Chiba, T., I. Nishicka and I. Shirakawa, An algorithm of maximal planarization of graphs, *Proc. 1979 IEEE Int. Symp. on Circuits and Systems* (1979), pp. 649-652.
- [7] Chiba, N., T. Nishizeki, S. Abe and T. Ozawa, A linear algorithm for embedding planar graphs using PQ-trees, *J. of Computer and System Sciences* Vol. 30 (1985), pp. 54-76.
- [8] Cai, J., 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.
- [9] Ozawa, T. and Takahashi H., A graph-planarization algorithm and its application to random graphs, *Graph Theory and Algorithms*, Springer-Verlag Lecture Notes in Computer Science, Vol. 108, (1981), pp. 95-107.

- [10] R.Jayakumar, K.Thulasiraman and M.N.S.Swamy, On maximal planarization of nonplanar graphs, *IEEE Trans. Circuits & Systems CAS-33*, Vol. 8 (1986), pp. 843-844.
- [11] R.Jayakumar, K.Thulasiraman and M.N.S.Swamy, $O(n^2)$ algorithms for graph planarization, *IEEE Trans. on Computer-aided Design*, Vol. 8 (1989), pp. 257-267.
- [12] G.Kant, An $O(n^2)$ maximal planarization algorithms based on PQ-trees, *Tech. Report RUU-CS-92-03, CS Dept., Univ. Utrecht, Netherlands*, Jan., 1992.
- [13] Tarjan, R., Depth-first search and linear graph algorithms, *SIAM J. Comput.*, Vol. 2 (June, 1979), pp.146-159.
- [14] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD (1979).
- [15] F. Bernhart and P. C. Kaine, The book thickness of a graph, *J. Combin. Theory, Ser.*, Vol. 27 (1979), pp. 320-331.
- [16] M. Sarrafzadeh and D. Lee, A new approach to topological via minimization, *IEEE Trans. on Computer-aided Design*, Vol. 8 (1989), pp. 890-900.
- [17] F. Gavril, Algorithms for a maximum clique and a maximum independent set of a circle graph, *Networks*, Vol. 3 (1973), pp. 261-273.
- [18] T. Asano, H. Imai, and A. Mukaiyama, Finding a maximum weight independent set of a circle graph, *IEEE Trans.*, Vol. 74 (1991), pp. 681-683.
- [19] D. Angluin and L. G. Valiant, Probabilistic algorithms for Hamiltonian circuits and matchings, *J. Comput. Syst. Sci.*, Vol. 18 (1979), pp. 155-190.
- [20] O. Goldschmidt and A. Takvorian, An efficient two-phase planarization heuristic, *Networks*, Vol. 24 (1994), pp. 69-73.
- [21] K.J. Supowit, Finding a maximum planar subset of a set of nets in a channel, *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6 (1987), pp. 93-94.

- [22] Y. Takfuji, K. Lee and J. Huffman, Parallel algorithms for finding a near maximum independent set of a circle graph, *IEEE Trans. on Neural Networks*, Vol. 1 (1990), pp. 263-267.
- [23] R. Tamassia, G. Di Battista, and C. Batini, Automatic graph drawing and readability of diagrams, *IEEE Trans. of Sys., Man. and Cyber.*, Vol. 18 (1988), pp. 61-79.
- [24] N. Barkakati, *X Window System Programming*, SAMS, Carmel, IN (1991).
- [25] D. A. Young, *The X Window System Programming and Applications with Xt, OSF/Motif Edition*, Prentice-Hall, Englewood-Cliffs, NJ (1990).

MONTANA STATE UNIVERSITY LIBRARIES



3 1762 10228627 3