AD 738027

AN EFFICIENT PLANARITY ALGORITHM

BY

ROBERT E. TARJAN

STAN-CS-244-71
NOVEMBER, 1971

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Stanford University	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

AN EFFICIENT PLANARITY ALGORITHM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical, December 1971

5. AUTHOR(S) (First name, middle initial, last name)

Robert E. Tarjan

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1971	154	36

8a. CONTRACT OR GRANT NO. ONR 00014-67-A-0112-0057 b. PROJECT NO. NR 044-402 and NSF c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) STAN-CS-71-244 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Office of Naval Research Washington, D. C.

13. ABSTRACT

An efficient algorithm is presented for determining whether a graph G can be embedded in the plane. Depth-first search, or backtracking, is the most important of the techniques used by the algorithm. If G has V vertices, the algorithm requires O(V) space and O(V) time when implemented on a tandom access computer. An implementation on the Stanford IBM 360/67 successfully analyzed graphs with as many as 900 vertices in less than 12 seconds.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Graph, Connectivity, Biconnectivity, Search, Backtracking, Depth-first.						

AN EFFICIENT PLANARITY ALGORITHM

Robert E. Tarjan
Stanford University

Abstract: An efficient algorithm is presented for determining whether a graph G can be embedded in the plane. Depth-first search, or backtracking, is the most important of the techniques used by the algorithm. If G has V vertices, the algorithm requires O(V) space and O(V) time when implemented on a random access computer. An implementation on the Stanford IBM 360/67 successfully analyzed graphs with as many as 900 vertices in less than 12 seconds.

This research was supported by the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR 044-402.

I. In the Beginning

1. Introduction

Graph theory is an endless source of easily stated yet very hard problems. Many of these problems require algorithms; given a graph, one may ask if the graph has a certain property, and an algorithm is to provide the answer. Since graphs are widely used as models of real phenomena, it is important to discover <u>efficient</u> algorithms for answering some graph-theoretic questions.

This work presents an algorithm for determining whether an arbitrary graph G can be embedded (without any crossing edges) in the plane. If V is the number of vertices and E the number of edges in the graph G, then the method requires amounts of space and time bounded by a linear function of V and E. The algorithm is optimal (to within a constant factor), because it is possible to show within a suitable theoretical framework that each edge of a graph must be examined at least once to resolve the planarity question.

The planarity algorithm is based upon a depth-first search, or backtracking, technique for exploring a graph. Backtracking has been widely used for finding solutions to problems in combinatorial theory and artificial intelligence [Gol 65, Nil 71]. Analysis reveals that by depth-first examination of a graph, we may simplify the graph and collect enough information to determine planarity rapidly. Besides planarity, several other problems have been solved using depth-first search.

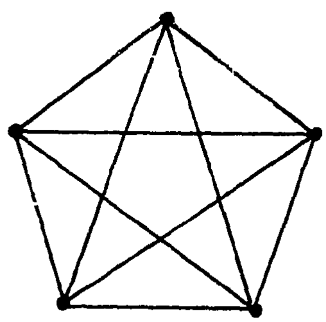
In order to analyze the efficiency of an algorithm, we use a random-access computer model. Data storage and retrieval, arithmetic operations, comparisons, and logical operations are assumed to require fixed times. A memory cell is allowed to hold integers whose absolute value is bounded by $k \max(V,E)$, where V is the number of vertices and E is the number of edges of the graph being processed, and k is some constant. An exact computer model will not be specified; see Cook [Coo 71]. To express the time and space bounds of algorithms, we shall use an extended version of the big O notation. Of functions of $x_1,\ldots,x_m$ we say f is $O(f_1,\ldots,f_n)$ if, for some constants k_i , $|f(x_1,\ldots,x_m)| \leq k_0 + k_1|f_1(x_1,\ldots,x_m)| + \ldots + k_n|f_n(x_1,\ldots,x_m)|$ for all values of x_i .

2. Previous Research on Planarity Algorithms

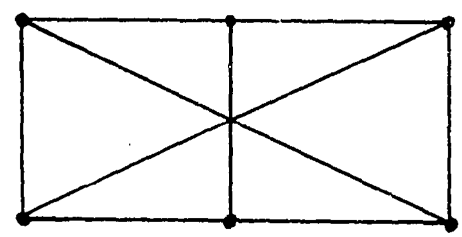
Embedding a graph in a plane has several applications. The design of integrated circuits requires knowing when a circuit may be embedded in a plane. Determining isomorphism of chemical structures is simplified if the structures are planar [Led 65, Hop 71b, Wei 65a, 65b, 66]. The importance of the problem is suggested by the number of published planarity algorithms. Examples include [Aus 61, Bru 70, Chu 70, Fis 66, Gold 63, Hop 71c, Lem 67, Mei 70, Mon 71, Shi 69, Tut 63, Win 66, You 63]. Surprisingly little work has been directed toward a rigorous analysis of their running times, however, and algorithms continue to appear which are obviously inferior to previously published ones. We shall examine several of the best algorithms here; a more complete history of the planarity problem may be found in Shirey's dissertation [Shi 69], which contains an extensive bibliography.

The earliest characterization of planar graphs was given by Kuratowski [Kur 30]. He proved that every non-planar graph contains a subgraph which upon removal of degree two vertices is isomorphic either to the complete graph on five vertices or to a complete bipartite graph on six vertices. (See Figure 2.1.) Conversely, no planar graph contains either of these graphs. Although elegant, Kuratowski's condition is useless as a practical test of planarity; testing for such subgraphs directly may require an amount of time proportional to at least V^6, if not much worse, where V is the number of vertices in the graph.

The best approach to the planarity problem seems to be an attempt to actually draw the graph in the plane. If such a drawing can be completed, then the graph is planar; if not, then the graph is non-planar.



K_5



$K_{3,3}$

Figure 2.1: The Kuratowski subgraphs.

The first such algorithm was proposed by Auslander and Parter [Aus 61]. First, a cycle is found in the graph. When this cycle is removed, the graph falls into several pieces. The algorithm is called recursively to embed each piece in the plane with the original cycle. Then the embeddings of the pieces are combined, if possible, to give an embedding of the entire graph. Unfortunately, Auslander and Parter's paper contains an error; the proposed method may loop indefinitely. Goldstein [Gold 63] correctly formulated the algorithm, using iteration instead of recursion. Shirey [Shi 69] implemented this method using a list structure representation for graphs, and proved an asymptotic time bound of $O(V^3)$ for his variation of the algorithm.

Lempel, Even, and Cederbaum [Lem 67] have presented another method for building a graph in the plane. They start with a single vertex, and add all edges incident to that vertex. They then add all edges incident to one of the new vertices, and continue in this way until the entire graph is constructed. Vertices must be selected in a special order if the algorithm is to work correctly. Lempel, Even, and Cederbaum give no implementation or time bound for their method; however, Tarjan [Tar 69] has implemented the algorithm in a way which requires $O(V)$ space and $O(V^2)$ time.

Mondshein [Mon 71] has recently proposed another constructive algorithm. He adds one vertex at a time until the entire graph is constructed. The order of vertex selection is again crucial. Mondshein's implementation requires $O(V^2)$ time. Hopcroft and Tarjan [Hop 71c], using depth-first search in a complicated program, have devised a variant of Goldstein's algorithm with a time bound of $O(V \log V)$. This method, although ponderous, is asymptotically the most efficient previously known.

A few algorithms deserve mention because of their novel approach. Fisher [Fis 66] gives an algorithm which works directly from the incidence matrix of a graph. This method, however, is not very efficient, nor is any method which uses incidence matrices. (See Chapter 4.) Bruno, Steiglitz, and Weinberg [Bru 70] present an algorithm based on some theorems of Tutte relating to triconnected planar graphs. Instead of constructing a graph in the plane, they reduce it to simpler and simpler graphs. Although they give no explicit time bound, the algorithm does not compare favorably with those mentioned above.

3. Definitions from Graph Theory

This chapter outlines the graph-theoretic concepts needed to understand the planarity algorithm. We use definitions similar to those found in any text on graph theory; for instance [Ber 64, Bus 65, Har 69, Ore 62]. We shall also introduce some special terminology. Proofs are omitted in this chapter; the results are either obvious or are standard in the literature of graph theory.

Definition 3.1: A graph $\tilde{G} = (\mathcal{V}, \mathcal{E})$ is an ordered pair, consisting of a finite set $\mathcal{V}$ of vertices and a finite set $\mathcal{E}$ of edges.

We shall deal with the properties of finite graphs only; we are concerned with constructive characterization of certain properties of graphs, and computers cannot manipulate infinite objects. The vertices of a graph may also be called points or nodes. The edges of a graph may also be called arcs or links. For the moment we have left undefined the nature of the edges of a graph; there are two kinds of graphs which we shall study, with two different types of edges.

Definition 3.2: An undirected graph $G = (\mathcal{V}, \mathcal{E})$ consists of a set of vertices and a set of edges. Each edge is an unordered pair $\{v,w\}$ of distinct vertices of G. The vertices v and w are said to be incident to v and w; v and w are said to be incident to $\{v,w\}$. Vertices v and w are said to be adjacent if $\{v,w\}$ is an edge of G. The relation $v \Rightarrow w$ holds if and only if $\{v,w\}$ is an edge of G.

Definition 3.3: A directed graph $\vec{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of vertices and a set of edges. Each edge is a directed pair $\langle v,w \rangle$ of distinct vertices of $\vec{G}$. The vertex v is said to be the tail of the edge $\langle v,w \rangle$. Vertex w is said to be the head of the edge $\langle v,w \rangle$. Incidence and adjacency are defined as for undirected graphs. A directed graph is really only an irreflexive relation; as with undirected graphs, we use the notation $v \Rightarrow w$ to mean that v and w satisfy the relation " $\langle v,w \rangle$ is an edge of $\vec{G}$ ".

Notice that we do not allow loops (edges whose two endpoints are identical). Neither do we allow several identical edges. An object resembling a graph but which contains multiple edges will be called a multigraph. We shall use capital letters ("G") to denote undirected graphs and capital letters with an arrow ("$\vec{G}$") to denote directed graphs. A capital letter with a tilde ("$\tilde{G}$") will denote a graph, either directed or undirected.

Let us consider the relationship between directed graphs and undirected graphs. Given an undefined graph G , we may convert it to a directed graph in one of two ways. First, we may convert each undirected edge $\{v,w\}$ of G into two directed edges, $\langle v,w \rangle$ and $\langle w,v \rangle$.

Definition 3.4: Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph. Then $\vec{G} = (\mathcal{V}, \mathcal{E}')$ is the directed graph such that $\mathcal{E}' = \{\langle v,w \rangle \mid \{v,w\} \in \mathcal{E}\}$. $\overleftrightarrow{G}$ is called the doubly directed version of G .

The computer representations of an undirected graph G and of the doubly directed version $\overleftrightarrow{G}$ of G will be indistinguishable; each edge will appear twice in the representation, once for each of its possible directions.

Another way to convert an undirected graph G into a directed graph is to convert each edge $\{v,w\}$ of G into a single directed edge $\langle v,w \rangle$. This will give a directed graph $\vec{G}$ with the same number of edges as G, in which each edge of G is assigned one of the two possible directions.

Conversely, suppose we have a directed graph $\vec{G} = (\mathcal{V}, \mathcal{E})$. We may convert G into an undirected graph by ignoring the direction of the edges. (We may have to delete multiple copies of the same undirected edge; otherwise a multigraph will result.)

<u>Definition 3.5</u>: The function u maps directed graphs into undirected graphs. If $\vec{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph, $u(\vec{G}) = (\mathcal{V}, \mathcal{E}')$ is the undirected graph formed by ignoring the directions of all the edges of $\vec{G}$: $\mathcal{E}' = \{\langle v,w \rangle | (v,w) \in \mathcal{E}\}$. The inverse function is multivalued. If $G = (\mathcal{V}, \mathcal{E}')$ is an undirected graph, $u^{-1}(G) = (\mathcal{V}, \mathcal{E})$ will denote any directed graph formed by giving each edge of G a direction.

Henceforth, we shall use " (v,w) " to denote an edge of any graph, either directed or undirected. We then have $(v,w) = (w,v)$ in an undirected graph but not in a directed graph. The following definitions apply to both directed and undirected graphs.

<u>Definition 3.6</u>: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ and $\tilde{G}' = (\mathcal{V}', \mathcal{E}')$ be graphs. If $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$, then $\tilde{G}'$ is a <u>subgraph</u> of $\tilde{G}$. $\tilde{G}'$ is called a <u>proper subgraph</u> of $\tilde{G}$ if $\tilde{G}' \neq \tilde{G}$.

Definition 3.7: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph. A sequence of vertices v_i, $1 \leq i \leq n$, such that $e_i = (v_i, v_{i+1})$ is an edge in $\tilde{G}$ for $1 \leq i < n$, is called a path of $\tilde{G}$. If all the vertices on the path are distinct, the path is called a simple path. If $v_1 = v_n$, all the vertices v_i, $1 \leq i < n$, are distinct, and all the edges e_i, $1 \leq i < n$, are distinct, then the path is called a cycle. The vertex v_1 is called the start vertex of the path. The vertex v_n is called the finish vertex of the path. Vertices v_1 and v_n are called the endpoints of the path. If $n \neq 1$, the path is called proper. The length of a path is the number of edges it contains.

Although a path may be conceptualized as a subgraph, the order of the vertices in the path is important. We shall generally identify a path by listing its sequence of points; the edges of the path are uniquely determined by this sequence. Note that a path may contain no edges. Paths will be denoted by the small letter " p " with or without subscripts. The small letter " c " will occasionally be used to denote a cycle. We assert the existence of a path from v_1 to v_n, and name the path p, by writing $p: v_1 \overset{*}{\Rightarrow} v_n$. The notation $v_1 \overset{+}{\Rightarrow} v_n$ means that there exists a path of length one or greater between v_1 and v_n. (In general, if R is any binary relation and I is the identity relation, R^+ denotes the transitive closure of R, and R^* denotes the reflexive transitive closure of R.)

Lemma 3.1: Let $\vec{G}$ be a directed graph. Then any path (simple path, cycle) of $\vec{G}$ is a path (simple path, cycle) of $G = u(\vec{G})$.

The converse of this lemma is not true. However:

Lemma 3.2: Let G be an undirected graph. Then any path (simple path, cycle) of G corresponds to a path (simple path, cycle) of $\overleftrightarrow{G}$, the doubly directed version of G. Conversely, any path (simple path, cycle of length greater than two) of $\overleftrightarrow{G}$ corresponds to a path (simple path, cycle of length greater than two) of G.

Definition 3.8: Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph. Suppose that for each pair of vertices v and w in G, there exists a path $p: v \overset{*}{\Rightarrow} w$. Then G is connected. If $\vec{G} = u^{-1}(G)$, $\vec{G}$ is called connected if and only if G is connected.

Lemma 3.3: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Then $\tilde{G}$ may be uniquely partitioned into a set of pairwise vertex- and edge-disjoint subgraphs, each of which is connected, and each of which is not properly contained in a connected subgraph of $\tilde{G}$. These maximal connected subgraphs are called the connected components of $\tilde{G}$.

Proof: See [Ore 62].

Definition 3.9: Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph. Suppose that for each triple of distinct vertices v, w, a in $\mathcal{V}$, there is a path $p: v \overset{*}{\Rightarrow} w$ such that a is not on the path p. Then G is biconnected. If, on the other hand, there is a triple of distinct vertices v, w, a in $\mathcal{V}$ such that a is on any path $p: v \overset{*}{\Rightarrow} w$, and there exists at least one such path, then a is called an articulation point of G. If $\vec{G} = u^{-1}(G)$, then $\vec{G}$ is called biconnected if and only if G is biconnected. If a is an articulation point of G , then a is also said to be an articulation point of $\vec{G}$.

Lemma 3.4: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph. We may define an equivalence relation on the set of edges as follows: two edges are equivalent if and only if they belong to a common cycle. Let the distinct equivalence classes under this relation be $\mathcal{E}_i$, $1 \leq i \leq n$, and let $\tilde{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$, where $\mathcal{V}_i$ is the set of vertices incident to the edges of $\mathcal{E}_i$: $\mathcal{V}_i = \{v \mid \exists w((v,w) \in \mathcal{E}_i)\}$. Then:

(i) $\tilde{G}_i$ is biconnected, for each $1 \leq i \leq n$.

(ii) No $\tilde{G}_i$ is a proper subgraph of a biconnected subgraph of $\tilde{G}$.

(iii) Each articulation point of $\tilde{G}$ occurs more than once among the $\mathcal{V}_i$, $1 \leq i \leq n$. Each non-articulation point of $\tilde{G}$ occurs exactly once among the $\mathcal{V}_i$, $1 \leq i \leq n$.

(iv) The set $\mathcal{V}_i \cap \mathcal{V}_j$ contains at most one point, for any $1 \leq i,j \leq n$. Such a point of intersection is an articulation point of the graph. The subgraphs $\tilde{G}_i$ of $\tilde{G}$ are called the biconnected components of $\tilde{G}$.

Proof: See [Har 69].

Definition 3.10: Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph. Suppose that for each quadruple of distinct vertices v,w,a,b in $\mathcal{V}$, there is a path $p: v \overset{*}{\Rightarrow} w$ such that neither a nor b is on the path p . Then G is triconnected. If there is a quadruple of distinct vertices v,w,a,b in $\mathcal{V}$ such that there is a path $p: v \overset{*}{\Rightarrow} w$, and any such path contains either a or b , then a and b are a <u>biarticulation point pair</u> in G . If $\vec{G}$ is a directed version of G , then $\vec{G}$ is called <u>triconnected</u> if and only if G is triconnected. If a and b are a biarticulation point pair in G , they are also said to be a <u>biarticulation point pair</u> in $\vec{G}$.

The triconnected components of a graph may be defined in several ways (see for instance [Tut 66]), each giving an analogy to Lemmas 3.3 and 3.4. We shall not need to use triconnected components in our study of planarity. However, with a suitable definition of triconnected components, a graph is planar if and only if its triconnected components are planar, and a triconnected planar graph has an essentially unique representation in the plane.

<u>Definition 3.11</u>: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Suppose that $\tilde{G}$ may be embedded in a plane (or equivalently, in the surface of a sphere). That is, suppose there is a mapping of the edges of the graph into the plane in such a way that each edge (v,w) is mapped into a simple curve, with the points v and w mapped into the endpoints of the curve. Mappings of two different edges may have only their common endpoints in common. If such a mapping exists, the graph $\tilde{G}$ is called <u>planar</u>. If $m(\tilde{G})$ is the image of $\tilde{G}$ in the plane, and if $m(\tilde{G})^C$ is the complement of this set relative to the plane, then the connected sets of points in $m(\tilde{G})^C$ are called the <u>faces</u> $\mathcal{F}$ of $\tilde{G}$ (relative to the mapping m).

Lemma 3.5 (Euler's Theorem): Let V be the number of vertices, E the number of edges, and F the number of faces in a planar embedding of a connected graph $\tilde{G}$. Then $V+F=E+2$.

Proof: See [Har 69].

The most useful property of the plane related to graphs is the Jordan Curve Theorem:

Lemma 3.6: Let c be a simple closed curve in the plane. Removal of c from the plane divides the remaining points into exactly two topologically connected sets, called the inside and the outside of c .

Proof: Difficult. See [Hal 55, Thr 53]. However, for our purposes we need this result only for piecewise linear closed curves c . This special case is not too difficult to derive.

If $\tilde{G}$ is a planar graph and c is a cycle in G , then the image of c under a planar embedding of $\tilde{G}$ is a simple closed curve. (In fact, $\tilde{G}$ may be embedded so that all edges of c are piecewise linear. See [Bus 65].) Thus, if c is removed from $\tilde{G}$, the remaining vertices and edges fall into two sets: those embedded on the inside of the image of c and those embedded on the outside of the image of c . We base our planarity algorithm on this observation and its corollaries, all of which follow from the Jordan Curve Theorem. In particular, we need the following result:

Lemma 3.7: Let $c: x_1 \Rightarrow x_2 \Rightarrow \dots \Rightarrow x_{n-1} \Rightarrow x_1$ be a cycle in a graph G which is embedded in the plane. Let (v,x_i), (w,x_j) be two edges not on the cycle. Suppose the order of edges clockwise around vertex x_i is (x_{i-1},x_i), (v,x_i), (x_i,x_{i+1}), and that the order of edges clockwise around x_j is (x_{j-1},x_j), (w,x_j), (x_j,x_{j+1}). Then (v,x_i) and (w,x_j) are on the same side of c. If the order of edges clockwise around x_j is (x_{j-1},x_j), (x_j,x_{j+1}), (w,x_j), then (v,x_i) and (w,x_j) are on opposite sides of c.

Proof: A rigorous proof of this theorem requires knowledge of topology (see [Hal 55, Thr 53]), but the idea is simple. Suppose the order of edges clockwise around x_j is (x_{j-1},x_j), (w,x_j), (x_j,x_{j+1}). Then edges (v,x_i) and (w,x_j) may be connected by a path which follows the cycle but does not cross it, as in Figure 3.1. Thus the two edges are on the same side of the cycle.

Suppose the order of edges clockwise around x_j is (x_{j-1},x_j), (x_j,x_{j+1}), (w,x_j). Every vertex in the plane may be joined by a simple path to one of the vertices on the cycle. If (v,x_i) and (w,x_j) were on the same side of the cycle then the remark above and the first part of the Lemma would imply that every point in the plane is on one side of the cycle, contrary to Lemma 3.6. Thus the second part of the Lemma is true.

We shall need to use two special classes of directed graphs, one standard, the other new.

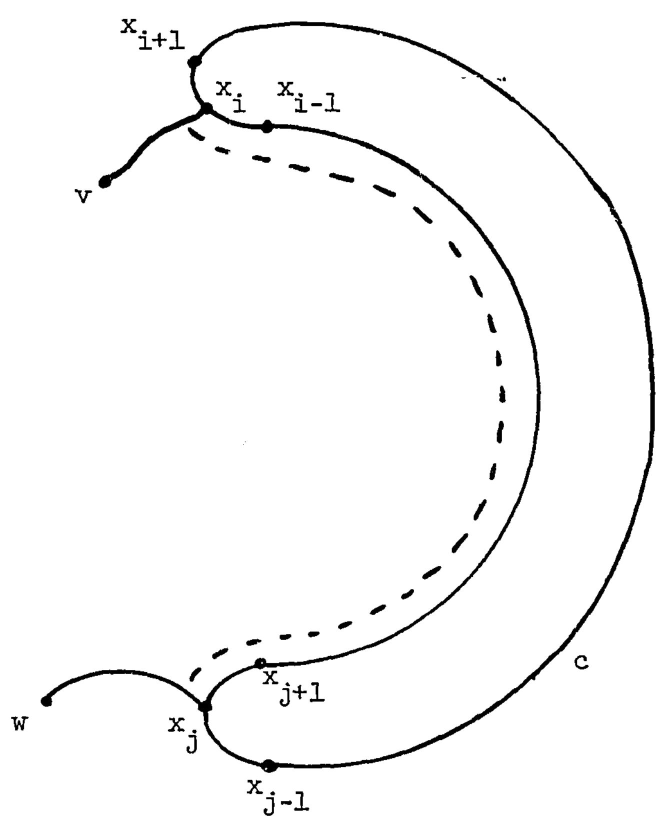


Figure 3.1: Two edges on the same side of a cycle.

Definition 3.12: Let $\vec{T}$ be a directed graph. Suppose $\vec{T}$ satisfies the following properties:

(i) $\vec{T}$ is connected.

(ii) There is a unique point in $\vec{T}$ which is the head of no edges. This point is called the root.

(iii) All other points of $\vec{T}$ are the head of exactly one edge.

Then $\vec{T}$ is called a directed rooted tree.

Since we shall deal only with trees which are directed rooted trees, we shall refer to them simply as trees. There may be simpler definitions of trees, but the one above is the most useful for our purposes.

Lemma 3.7: Let $\vec{T}$ be a tree. Then $u(\vec{T})$ contains no cycles.

Proof: An exercise for the reader.

Lemma 3.8: Let v and w be vertices in a tree $\vec{T}$. Then there exists either exactly one path p whose endpoints are v and w or no such path.

Proof: An exercise for the reader.

Definition 3.13: A path in a tree $\vec{T}$ is called a branch of $\vec{T}$.

Definition 3.14: Let $\vec{T}$ be a tree and let v and w be vertices of $\vec{T}$. If (v,w) is an edge of $\vec{T}$, then w is called a son of v, and v is called the father of w. If there is a path $p: v \overset{*}{\Rightarrow} w$, then w is called a descendant of v, and v is called an ancestor of w. If such a path is proper $(v \neq w)$, then w is called a proper descendant of v, and v is called a proper ancestor of w.

We use single-shafted arrows to denote arcs of trees, since we shall study trees which are a subgraph of a directed graph, and it will be necessary to distinguish between the tree arcs and arcs in the larger graph. We use $v \overset{*}{\rightarrow} w$ to denote the (unique) branch from v to w in a tree, and also to indicate the fact that such a path exists. (Vertices v and w satisfy the relation " v is an ancestor of w in $\vec{T}$ ".) The meaning will be clear from the context.

Definition 3.15: Let $\vec{T}$ be a tree and let v a vertex of $\vec{T}$. The subtree of $\vec{T}$ rooted at v is the tree $\vec{T}_v = (\mathcal{V}', \mathcal{E}')$ whose vertices $\mathcal{V}'$ are all the descendants of v and whose edges are all those edges with tails in $\mathcal{V}'$: $\mathcal{V}' = \{w \mid v \overset{*}{\rightarrow} w\}$; $\mathcal{E}' = \{(v,w) \mid v \rightarrow w \ \& \ v \in \mathcal{V}'\}$.

Definition 3.16: Let $\vec{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. A spanning tree $\vec{T}$ of $\vec{G}$ is a subgraph of $\vec{G}$ which is a tree and which contains all the vertices of $\vec{G}$. If $G = (\mathcal{V}', \mathcal{E}')$ is an undirected graph, any spanning tree of the doubly directed version $\overleftrightarrow{G}$ of G is also a spanning tree of G.

We now present a new class of directed graphs, upon which the planarity algorithm is based.

Definition 3.17: Let $\vec{P} = (\mathcal{V}, \mathcal{E})$ be a directed graph, consisting of two disjoint sets of edges, denoted by $v \rightarrow w$ and $v \dashrightarrow w$ respectively.

Suppose $\vec{P}$ satisfies the following properties:

(i) The subgraph containing the edges $v \rightarrow w$ is a tree $\vec{T}$ which contains all the vertices of $\vec{P}$, called the <u>spanning tree</u> of $\vec{P}$.

(ii) We have $\dashrightarrow \subseteq (\overset{*}{\rightarrow})^{-1}$, where "$\dashrightarrow$" and "$\rightarrow$" denote the relations defined by the corresponding sets of edges. That is, each edge which is not in the spanning tree $\vec{T}$ of $\vec{P}$ connects a vertex with one of its ancestors in $\vec{T}$.

Then $\vec{P}$ is called a <u>palm tree</u>. The arcs $v \dashrightarrow w$ are called the <u>fronds</u> of $\vec{P}$.

Figure 3.2 shows a palm tree and its fronds. Since the notion of a palm tree is non-standard, we shall not develop its properties until we discover the context in which it arises. Tree palms are in reality more nearly comparable in structure to overgrown cornstalks than to true trees.

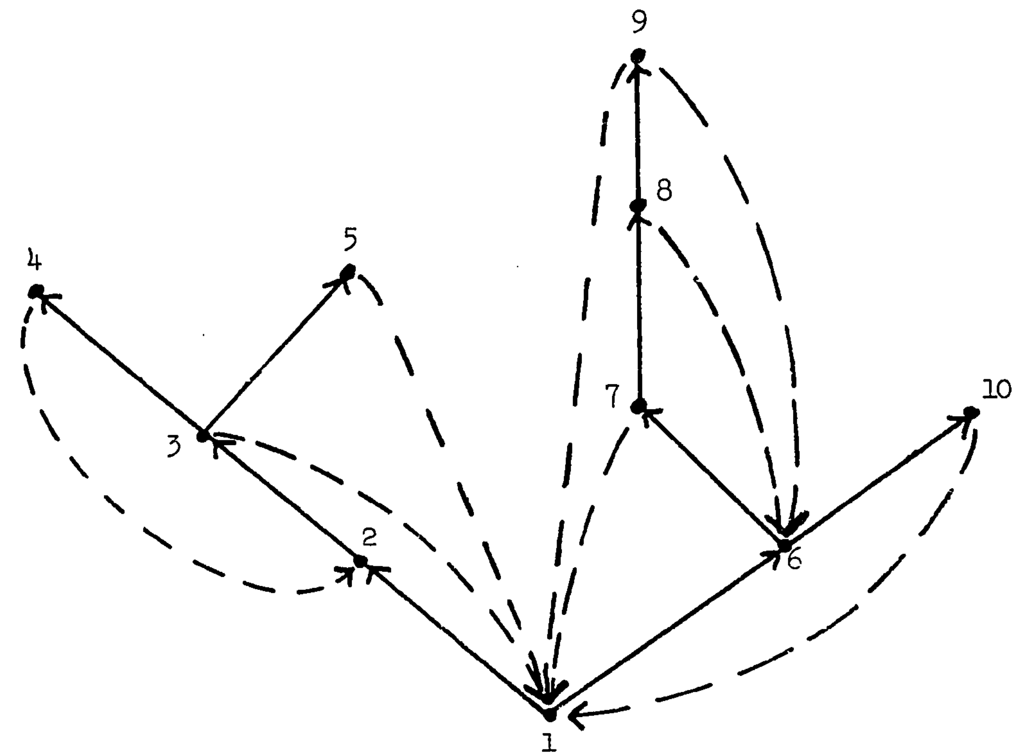


Figure 3.2: A palm tree. Fronds are dotted.

II. The Technique of Depth-first Search

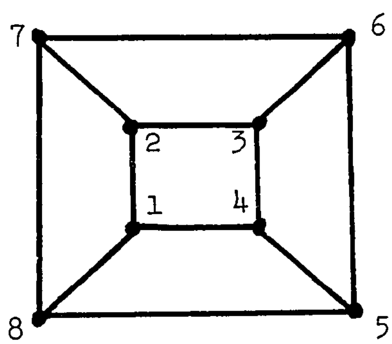
4. Data Structures Representing Graphs

Good algorithms require an appropriate data structure; we therefore look with some care at how a graph may be represented in a computer. We need a representation which will preserve the adjacency properties of the graph, which will be economical of storage, and which may easily be constructed from the original list of vertices and edges which define the graph.

Definition 4.1: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph with vertices $\{1,2,\ldots,V\}$. The adjacency matrix $A = (a_{ij})$ of $\tilde{G}$ is a $V \times V$ matrix of zeros and ones such that $a_{ij} = 1$ if $(i,j) \in \mathcal{E}$, $a_{ij} = 0$ if $(i,j) \notin \mathcal{E}$.

The adjacency matrix of a graph is a common representation. If $\tilde{G}$ is undirected and contains no loops, A will be symmetric and will have zeros on the main diagonal. If $\tilde{G}$ is directed, then A may be asymmetric. Figure 4.1 gives an example of a graph and its adjacency matrix.

The adjacency matrix of a graph has several useful features. Certain simple matrix operations correspond to simple graphical manipulations. For instance, if $(b_{ij}) = A^k$, then b_{ij} gives the number of paths of length k between vertices i and j. The zeros and ones of the adjacency matrix may be packed into machine words to save storage space; word operations such as addition and logical operations may be used to manipulate the data w bits at a time if w is the word size of the given machine. This saving is somewhat illusory, however. The amount of storage space required by an adjacency matrix is kV^2, and we may prove rigorously of most interesting graph problems that they require



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 4.1: A graph and its adjacency matrix.

examination of every bit in the matrix and thus have a computation time proportional to at least V^2 [Hol 70]. When the graph is large enough, the gain obtained by packing bits becomes insufficient. If the matrix is sparse $(E \ll V^2)$ we must use a representation which is not as wasteful as the adjacency matrix. A list structure representation of the graph is a good choice.

Definition 4.2: Let $\tilde{G} = (\mathcal{V}, \mathcal{E})$ be a graph. For each vertex $i \in \mathcal{V}$, we may construct a list L_i containing all vertices j such that $(i,j) \in \mathcal{E}$. Such a list is called an adjacency list for vertex i. A set of such lists, one for each vertex in $\tilde{G}$, is called an adjacency structure for $\tilde{G}$.

Figure 4.2 gives a graph and its adjacency structure.

A single graph $\tilde{G}$ may have many adjacency structures; each ordering of the edges around the vertices of $\tilde{G}$ gives a unique adjacency structure, and each adjacency structure corresponds to a uniqueoordering of the edges at each vertex. (An adjacency structure for an undirected graph G corresponds to an embedding of G in some orientable surface; see [You 63].)

If $\tilde{G}$ is undirected, each edge (i,j) is represented twice in an adjacency structure; once for i and once for j. If $\tilde{G}$ is directed, each edge (i,j) is represented exactly once; vertex j appears in the adjacency list of vertex i. An adjacency structure requires an amount of storage space linear in V and E. The enormous value of an adjacency structure of $\tilde{G}$ is that we may use if effectively to perform

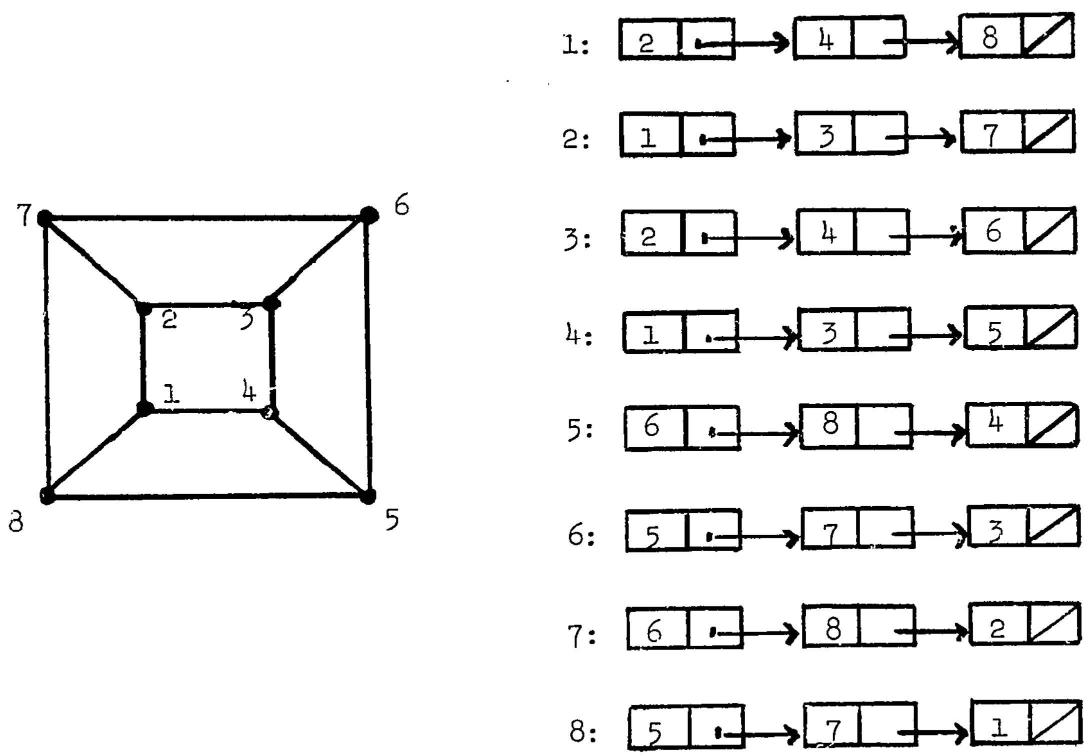


Figure 4.2: An adjacency structure for the graph in Figure 4.1.

searches of $\tilde{G}$; that is, to traverse the edges of $\tilde{G}$ in some systematic way. Such a search will require $O(V,E)$ steps.

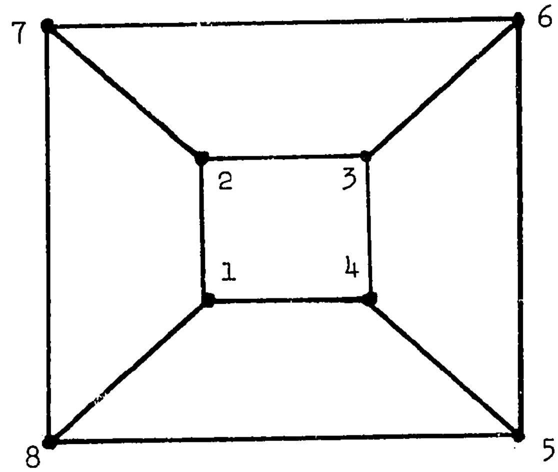
5. Searches, Spanning Trees, and Finding Connected Components

Suppose G is a connected undirected graph which we wish to explore. Consider the following procedure. Initially all the vertices of G are unexplored. We start from some vertex of G and choose an edge to follow. Traversing the edge leads to a new vertex. We continue in this way; at each step we select an unexplored edge from a vertex already reached and we traverse this edge. The edge leads to some vertex, either new or already reached. Eventually we will traverse all the edges of G, each exactly once. Such a process is called a search of G.

Any search of G imposes an orientation on the edges in G, according to the direction in which they are traversed. Thus a search converts G into a directed graph $\vec{G}$. For any starting point in G, there may be many possible searches depending upon how the edges to explore are selected. Each search generates a (possibly) different directed version $\vec{G}$ of G. Any search also produces a spanning tree $\vec{T}_G$ given by the set of edges which when traversed during the search lead to a new vertex. A graph and the results of two possible searches are illustrated in Figure 5.1.

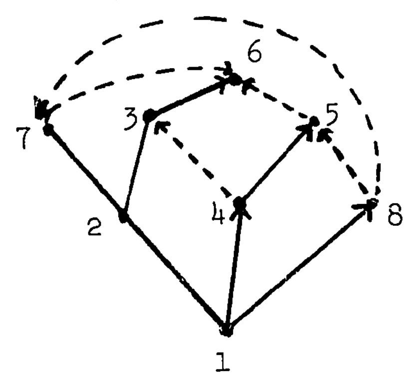
Notice that the edges of $\vec{G}$ which do not form part of the spanning tree $\vec{T}_G$ may interconnect the branches of the tree. (See the examples in Figure 5.1.) For one type of search, however, this is not true. Suppose we use the following rule for selecting an edge to traverse: Always choose an edge emanating from the vertex most recently reached which still has unexplored edges. We call a search which uses this rule a depth-first search. The set of old vertices with possibly unexplored edges may be stored on a stack; thus the search may be easily programmed

(a)



(b) (1,2)(1,4)(1,8)(2,3)(2,7)(4,3)(4,5)(8,5)(8,7)(3,6)(5,6)(7,6)

(c)



(d) (1,2)(2,3)(3,4)(4,1)(1,8)(8,5)(5,6)(6,7)(7,8)(2,7)(6,3)(4,5)

(e)

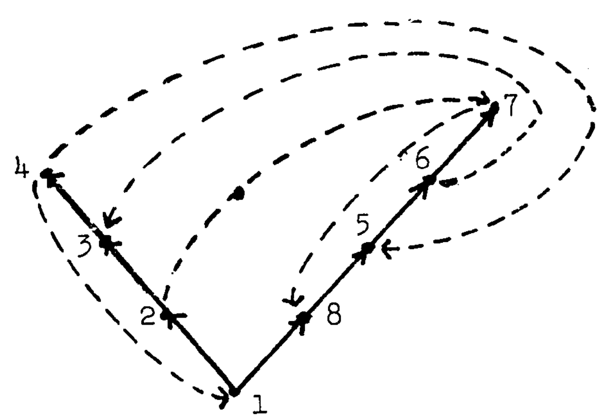


Figure 5.1: Two searches on a graph. (a) Graph. (b),(d) Search orders. (c),(e) Directed graphs generated by searches. Spanning trees indicated by solid arcs.

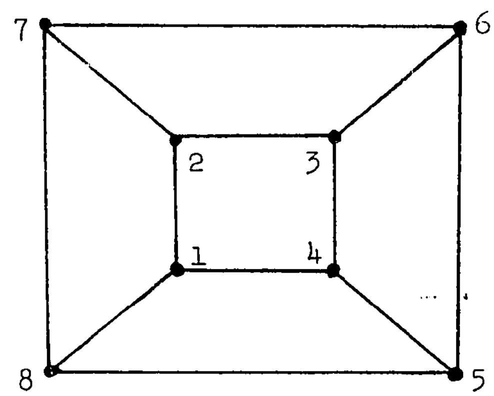
either iteratively or recursively. The program given below carries out a depth-first search of a graph G, starting at vertex s. The procedure constructs the directed graph generated by the search, and uses an adjacency structure of the graph G.

```
begin
   integer i;
   procedure DFS(v,u); comment  v is the current vertex, and u
      is the father of v in the spanning tree generated by the
      search;
         begin
            NUMBER(v) := i := i+1;
            for w in the adjacency list of v do
               begin
                  if w is not yet numbered then
                     begin
                        construct arc v → w in P;
                        DFS(w,v);
                     end
                  else if NUMBER(w) < NUMBER(v) and w ≠ u then
                     construct arc v --→ w in P;
               end;
         end;
   i := 0;
   DFS(s,0);
end;
```

Figure 5.2 gives an example of the directed graph generated by a depth-first search.

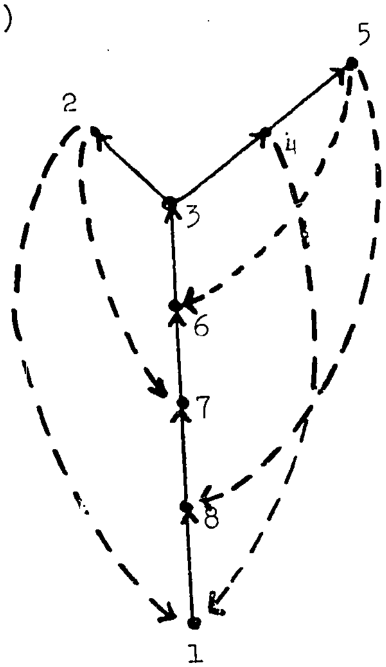
An adjacency structure gives a unique depth-first search for any starting vertex; edge selection order is fixed by the order of the adjacency lists. The search requires O(V,E) steps, where V is the number of vertices and E the number of edges of the graph. Let us characterize the directed graphs generated by depth-first searches.

(a)

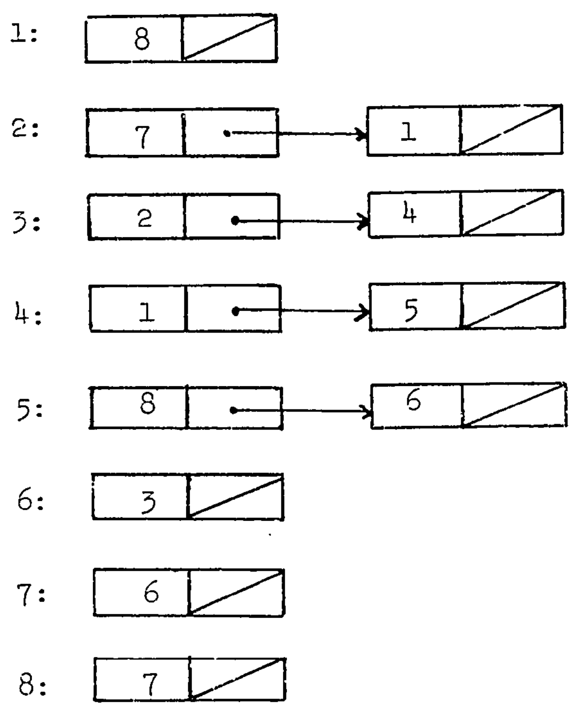


(b) (1,8)(8,7)(7,6)(6,3)(3,2)(2,1)(2,7)(3,4)(4,1)(4,5)(5,8)(5,6)

(c)



(d)



<u>Figure 5.2</u>: Depth-first search of a graph. (a) Graph. (b) Search order. (c) Generated palm tree (spanning tree indicated by solid arcs). (d) Adjacency structure of palm tree.

Recall the definition of a palm tree given in Chapter 3: $\vec{P}$ is a palm tree if $\vec{P}$ is a connected directed graph with a directed rooted spanning tree $\vec{T}$ and all arcs $(i,j) \in \vec{P}-\vec{T}$ satisfy $j \overset{*}{\rightarrow} i$ in $\vec{T}$. The edges of $\vec{P}-\vec{T}$ are called the fronds of the palm.

<u>Theorem 5.1</u>: Let $\vec{G}$ be the directed graph generated by a depth-first search of a connected graph G. Then $\vec{G}$ is a palm tree. Conversely, let $\vec{G}$ be any palm tree. Then G is generated by some depth-first search of G, the undirected version of $\vec{G}$.

<u>Proof</u>: Suppose $\vec{G} = (\mathcal{V}, \mathcal{E})$ is the directed graph generated by a depth-first search of some connected graph G, and assume that the search begins at vertex s. Examine the procedure DFS. The algorithm clearly terminates because each vertex becomes v only once and is numbered then. Furthermore, each edge in the graph is examined exactly twice. Therefore the time required by the search is linear in $\mathcal{V}$ and $\mathcal{E}$.

For any vertices v and w, let d(v,w) be the length of the shortest path between v and w in G. Since G is connected, all distances are finite. Suppose that some vertex remains unnumbered by the search. Let v be an unnumbered vertex such that d(s,v) is minimal. Then there is a vertex w such that w is adjacent to v and $d(s,w) < d(s,v)$. Thus w is numbered. But v will also be numbered, since it is adjacent to w. This means that all vertices are numbered during the search.

The vertex s is the head of no edge $w \rightarrow s$. Each other vertex v is the head of exactly one edge $w \rightarrow v$. The subgraph $\vec{T}$ of $\vec{G}$ defined by the edges $v \rightarrow w$ is obviously connected, since there is a path in $\vec{T}$ from the root s to any vertex. This may be proved by induction. Thus $\vec{T}$ is a spanning tree of $\vec{G}$.

Each arc of the original graph is directed in at least one direction; if (v,w) does not become an arc of the spanning tree $\vec{T}$, either $v \dashrightarrow w$ or $w \dashrightarrow v$ must be constructed, since both v and w are numbered whenever edge (v,w) is inspected and either NUMBER(v) < NUMBER(w) or NUMBER(v) > NUMBER(w) .

The arcs $v \rightarrow w$ run from smaller numbered points to larger numbered points. The arcs $v \dashrightarrow w$ run from larger numbered points to smaller numbered points. If arc $v \dashrightarrow w$ is constructed, arc $w \rightarrow v$ is not constructed later because both v and w are numbered. If arc $w \rightarrow v$ is constructed, arc $v \dashrightarrow w$ is not later constructed, because of the test " $w \neq u$ " in procedure DFS. Thus each edge in the original graph is directed in one and only one direction.

Consider an arc $v \dashrightarrow w$. We have NUMBER(w) < NUMBER(v) . Thus w is numbered before v . Since $v \dashrightarrow w$ is constructed and not $v \rightarrow w$, v must be numbered before edge (w,v) is inspected. Thus v must be numbered during execution of DFS(w,_) . But all vertices numbered during execution of DFS(w,_) are descendants of w . This means that $w \overset{*}{\rightarrow} v$, and G is a palm tree.

To prove the converse part of the theorem, suppose that $\vec{P}$ is a palm tree, with spanning tree $\vec{T}$ and undirected version P . Construct an adjacency structure of P in which all the edges of $\vec{T}$ appear before the other edges of P in the adjacency lists. Starting with the root of $\vec{T}$, perform a depth-first search using this adjacency structure. The search will traverse the edges of $\vec{T}$ preferentially and will generate the palm tree $\vec{P}$; it is easy to see that each edge is directed correctly. This completes the proof of the theorem.

From Theorem 5.1 we have the following interesting result:

Corollary 5.2: Let G be any undirected graph. Then G can be converted into a palm tree by directing its edges in a suitable manner.

A simple application of the concept of search is a well-known algorithm for determining the connected components of a graph G. We choose an arbitrary initial vertex and search. The search gives one connected component. We then choose some new vertex and search again. After a suitable number of searches the graph will be completely explored and all its connected components will be found. The program below will carry out these searches.

```
begin
   integer i;
   procedure CONNECT(v,u);
      begin
         NUMBER(v) := i := i+1;
         for w in the adjacency list of v do
            begin
               if w is not yet numbered then
                  begin
                     add edge (v,w) to current connected component;
                     CONNECT(w,v);
                  end
               else if NUMBER(w) < NUMBER(v) and w ≠ u then
                  add edge (v,w) to current component;
            end;
      end;
   i := 0;
   for x in V if x is not yet numbered then
```

```
        begin
            start new connected component;
            CONNECT(x,0);
        end;
end;
```

Depth-first search is convenient but not neceesary for this algorithm; any search method will do. It is easy to verify that the space and time requirements of the algorithm are linear in V and E .

As we shall see, depth-first search is an extremely useful technique. In the algorithms that follow we perform one depth-first search of a graph G to generate a palm tree $\vec{P}$ and a corresponding adjacency structure. In some cases we may reorder the lists of this adjacency structure to give a new depth-first search. The new search is performed on the directed graph $\vec{P}$; thus the edges are traversed in the same direction as during the first search but explored in a different order. The test to avoid traversing edges in the wrong direction is unnecessary, and the palm tree does not change after the initial search. We save enough information during the later search to enable us to answer interesting questions about G , aided by the simple structure of $\vec{P}$.

6. Finding Biconnected Components Using Depth-first Search

We have seen how to use a search to find the connected components of a graph. The simple structure of palm trees enables us to answer more complicated connectivity questions in linear time. Assume for example that a connected graph G has an articulation point a as illustrated in Figure 6.1. Suppose we begin a depth-first search in region G-R and enter region R by passing through vertex a . We must eventually back up through vertex a ; that is the only way to leave region R during the search. This observation allows us to efficiently calculate the biconnected components of G .

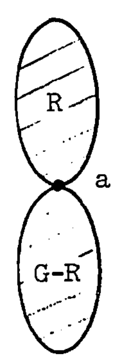


Figure 6.1: Vertex a separates region R from the rest of the graph.

Let $\vec{P}$ be the palm tree generated by a depth-first search of G and let $\vec{T}$ be its spanning tree. The procedure DFS numbers the vertices of $\vec{P}$ from 1 to V so that the numbering corresponds to the order in which they have been reached during the search. We may refer to a vertex by its number. Then an ancestor j in $\vec{T}$ of any vertex i has $j < i$. If i is any vertex of $\vec{P}$, let LOWPT1(i) be the smallest vertex in the set $S_i = \{j \mid i \overset{*}{\rightarrow} \dashrightarrow j\}$. If S_i is empty, let LOWPT1(i) $= +\infty$. The following results form the basis of an algorithm for finding biconnected components. This algorithm was discovered by Hopcroft and Tarjan [Hop 71d]. Paton [Pat 71] describes a similar algorithm.

Lemma 6.1: Let G be an undirected graph and let $\vec{P}$ be a palm tree formed by directing the edges of G. Let $\vec{T}$ be the spanning tree of $\vec{P}$. Suppose $p: v \overset{*}{\Rightarrow} w$ is any path in G. Then p contains a point which is an ancestor of both v and w in $\vec{T}$.

Proof: Let $\vec{T}_u$ with root u be the smallest subtree of $\vec{T}$ containing all vertices on the path p. If $u = v$ or $u = w$ the lemma is immediate. Otherwise, let $\vec{T}_{u_1}$ and $\vec{T}_{u_2}$ be two subtrees containing points on p such that $u \rightarrow u_1$ and $u \rightarrow u_2$. If only one such subtree exists then u is on p since $\vec{T}_u$ is minimal. If two such subtrees exist, path p can only get from $\vec{T}_{u_1}$ to $\vec{T}_{u_2}$ by passing through vertex u, since no point in one of these trees is an ancestor of any point in the other, while both $\rightarrow$ and $\dashrightarrow$ connect only ancestors in a palm tree. Since u is an ancestor of both v and w, the lemma holds.

Lemma 6.2: Let G be a connected undirected graph. Let $\vec{P}$ be a palm tree formed by directing the edges of G, and let $\vec{T}$ be the spanning tree of $\vec{P}$. Suppose a , v , w are distinct vertices of G such that $(a,v) \in \vec{T}$, and suppose w is not a descendant of v in $\vec{T}$. (That is, $\neg(v \overset{*}{\rightarrow} w)$ in $\vec{T}$.) If $\mathrm{LOWPT1}(v) \geq a$ then a is an articulation point of $\vec{P}$ and removal of a disconnects v and w . Conversely, if a is an articulation point of G then there exist vertices v and w which satisfy the properties above.

<u>Proof</u>: If $a \rightarrow v$ and $\mathrm{LOWPT1}(v) \geq a$, then any path from v not passing through a remains in the subtree $\vec{T}_v$, and this subtree does not contain the point w . This gives the first part of the Lemma.

To prove the converse, let a be an articulation point of G . If a is the root of G then at least two tree arcs must emanate from a . Let v be the head of one such arc and let w be the head of another such arc. Then $a \rightarrow v$, $\mathrm{LOWPT1}(v) \geq a$, and w is not a descendant of v . If a is not the root of $\vec{P}$, consider the connected components formed by deleting a from G . One component must be a subtree of $\vec{T}$ whose root v is a son of a . If w is any proper ancestor of a , then $a \rightarrow v$, $\mathrm{LOWPT1}(v) \geq a$, and w is not a descendant of v . Thus the converse part of the Lemma is true.

Figure 6.2 shows a graph, its LOWPT1 values, articulation points, and biconnected components. The LOWPT1 values of all the vertices of a palm tree $\vec{P}$ may be calculated during a single depth-first search, since $\mathrm{LOWPT1}(v) = \min(\{\mathrm{LOWPT1}(w) \mid v \rightarrow w\}, \{\mathrm{NUMBER}(w) \mid v \dashrightarrow w\})$.
On the basis of such a calculation, the articulation points and the

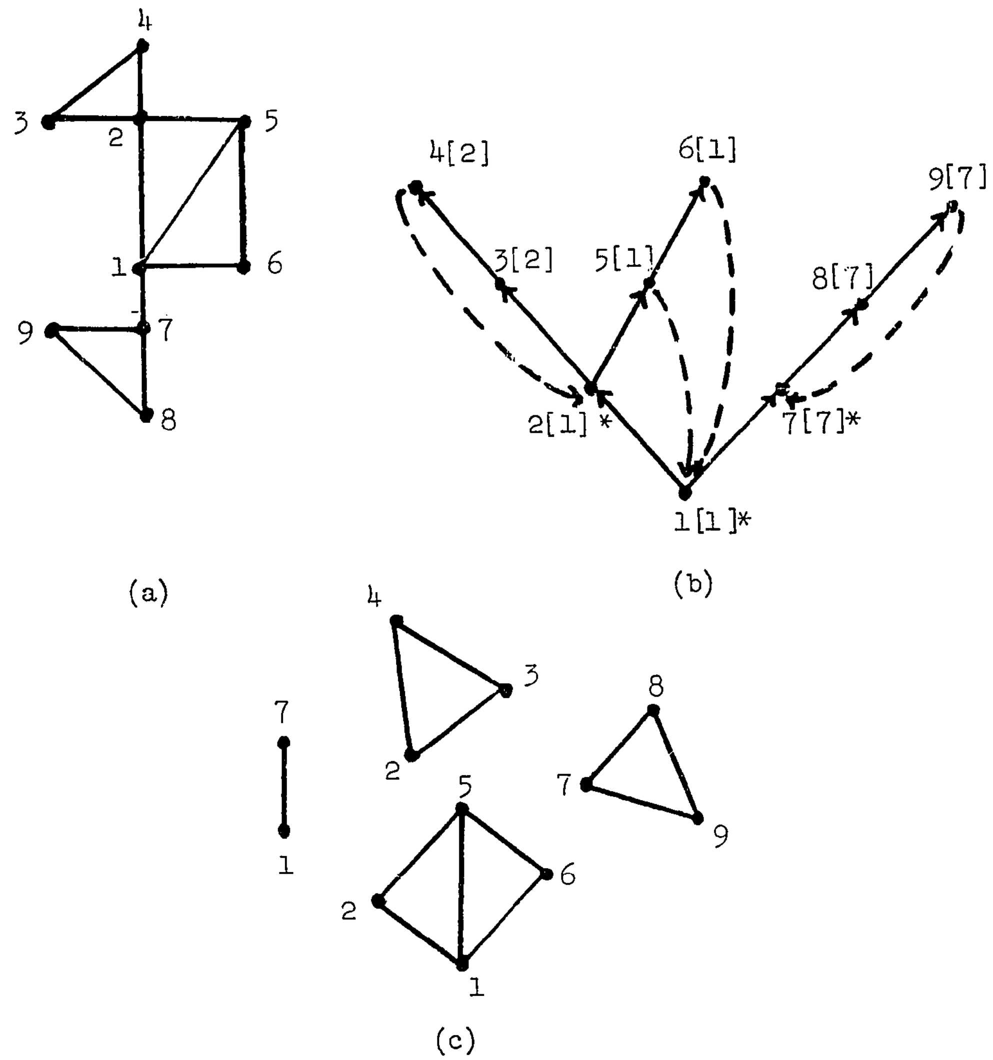


Figure 6.2: A graph and its biconnected components.

(a) Graph.

(b) A palm tree with LOWPOINT values in [ ], articulation points marked with *.

(c) Biconnected components.

biconnected components may be determined, all during one search. The biconnectivity algorithm is presented below. The program will compute the biconnected components of a graph G , starting from vertex s .

```
begin
   integer i;
   procedure BICONNECT(v,u);
      begin
         NUMBER(v) := i := i+1;
         LOWPT1(v) := +∞;
         for w in the adjacency list of v do
            begin
               if w is not yet numbered then
                  begin
                     add (v,w) to stack of edges;
                     BICONNECT(w,v);
                     LOWPT1(v) := min(LOWPT1(v),LOWPT1(w));
                     if LOWPT1(w) ≥ NUMBER(v) then
                        begin
                           start new biconnected component;
                           for (u1,u2) on edge stack with
                              NUMBER(u1) > NUMBER(v) do
                                 delete (u1,u2) from edge stack
                                    and add it to current component;
                           delete (v,w) from edge stack and add it
                              to current component;
                        end;
                  end
               else if NUMBER(w) < NUMBER(v) and w ≠ u then
                  begin
                     add (v,w) to edge stack;
                     LOWPT1(v) := min(LOWPT1(v),NUMBER(w));
                  end;
            end;
      end;
```

```
    i := 0;
    empty the edge stack;
    for x in V do if x is not yet numbered then BICONNECT(x,0);
end;
```

The edges of $\vec{P}$ are placed on a stack as they are traversed; when an articulation point is found the corresponding edges are all on top of the stack. (If $(v,w) \in T$ and $LOWPT1(w) \geq v$, then the corresponding biconnected component contains the edges in $\{(u_1,u_2) | w \overset{*}{\rightarrow} u_1\} \cup \{(v,w)\}$ which are still on the edge stack.) A single search on each connected component of a graph G will give us all the biconnected components of G.

<u>Theorem 6.3</u>: The biconnectivity algorithm requires $O(V,E)$ space and time when applied to a graph with V vertices and E edges.

<u>Proof:</u> The algorithm clearly requires space linear in V and E. The algorithm is similar to the connectivity algorithm, except that LOWPT1 values are calculated and each edge is placed on the edge stack once and removed from the edge stack once. The amount of extra time required by these operations is proportional to E. Thus BICONNECT has a time bound linear in V and E.

<u>Theorem 6.4</u>: The biconnectivity algorithm correctly gives the biconnected components of any undirected graph G.

<u>Proof:</u> The actual depth-first search undertaken by the algorithm depends on the adjacency structure chosen to represent G; we shall prove that the algorithm is correct for all adjacency structures. Notice

first that the biconnectivity algorithm contains as a part the algorithm presented in Chapter 4 for finding connected components. Each connected component is analyzed separately to find its biconnected components. Thus we need only prove that the biconnectivity algorithm works correctly on connected graphs G .

The correctness proof is by induction on the number of edges in G . Suppose G is connected and contains no edges. G either is empty or consists of a single point. The algorithm will terminate after examining G and listing no components. Thus the algorithm operates correctly in this case. Now suppose that the algorithm works correctly on all connected graphs with E-1 or fewer edges. Consider applying the algorithm to a connected graph G with E edges.

Each edge placed on the stack of edges is eventually removed and added to a component since everything on the edge stack is removed whenever the search returns to the root of the palm tree of G . Consider the situation when the first component G' is formed. Suppose that this component does not include all the edges of G . Then the vertex v currently being examined is an articulation point of the graph and separates the edges in the component from the other edges in the graph by Lemma 6.2.

Consider only the set of edges in the component. If BICONNECT(v,0) is executed, using the graph G' as data, the steps taken by the algorithm are the same as those taken during the analysis of the edges of G' when the data consists of the entire graph G . Since G' contains fewer edges than G , the algorithm operates correctly on G', and G' must be biconnected. If we delete the edges of G' from G, we get another subgraph G'' with fewer edges than G since G' is not empty. The algorithm operates correctly on G'' by the induction assumption. The behavior of the algorithm on G is simply a composite of its behavior on G' and on G''; thus the algorithm must operate correctly on G.

Now assume that only one component is found. We want to show that in this case G is biconnected. Suppose that G is not biconnected. Then G has an articulation point a. By Lemma 6.2, $LOWPT1(v) \geq a$ for some son v of a. But the articulation point test in the program will succeed when the edge (a,v) is examined, and more than one biconnected component will be generated. This contradiction shows that G is biconnected, and the algorithm works correctly in this case.

By induction, the biconnectivity algorithm gives the correct components when applied to any connected graph, and hence when applied to any graph.

III. A Linear Planarity Algorithm

7. General Description

We wish to decide whether or not a given graph G can be embedded in the plane. We can answer this question using an algorithm whose space and time bounds are linear in V , the number of vertices in the graph G . An intuitive description of the algorithm is presented here; later the various operations necessary will be discussed in detail. Figure 7.1 gives a flowchart of the overall process.

Suppose a connected graph G is embedded in a plane. When the set of points representing the edges and vertices of G is deleted from the plane, certain regions remain; these are the called the faces of G . Euler proved a relationship between the number of vertices V , faces F , and edges E of a connected planar graph: $V+F = E+2$ (Lemma 3.5). A consequence of this fact is:

Lemma 7.1: If G is a planar graph with three or more vertices then $E \leq 3V-6$.

Proof: If G is not connected, we may connect it by adding additional edges. Since G is not a multigraph the boundary of each face must contain at least three edges. Thus $3F \leq 2E$; every edge is counted twice if we sum over the facial boundaries. It follows that $3E = 3V+3F-6 \leq 3V+2E-6$, and $E \leq 3V-6$.

Because of Lemma 7.1, we may hope to determine planarity in time which is proportional to the number of vertices. The first step of the algorithm is to count the number of edges in the graph G . If the count ever exceeds $3V-6$, we stop and declare the graph non-planar. Next we may divide the graph into biconnected components, using the algorithm described

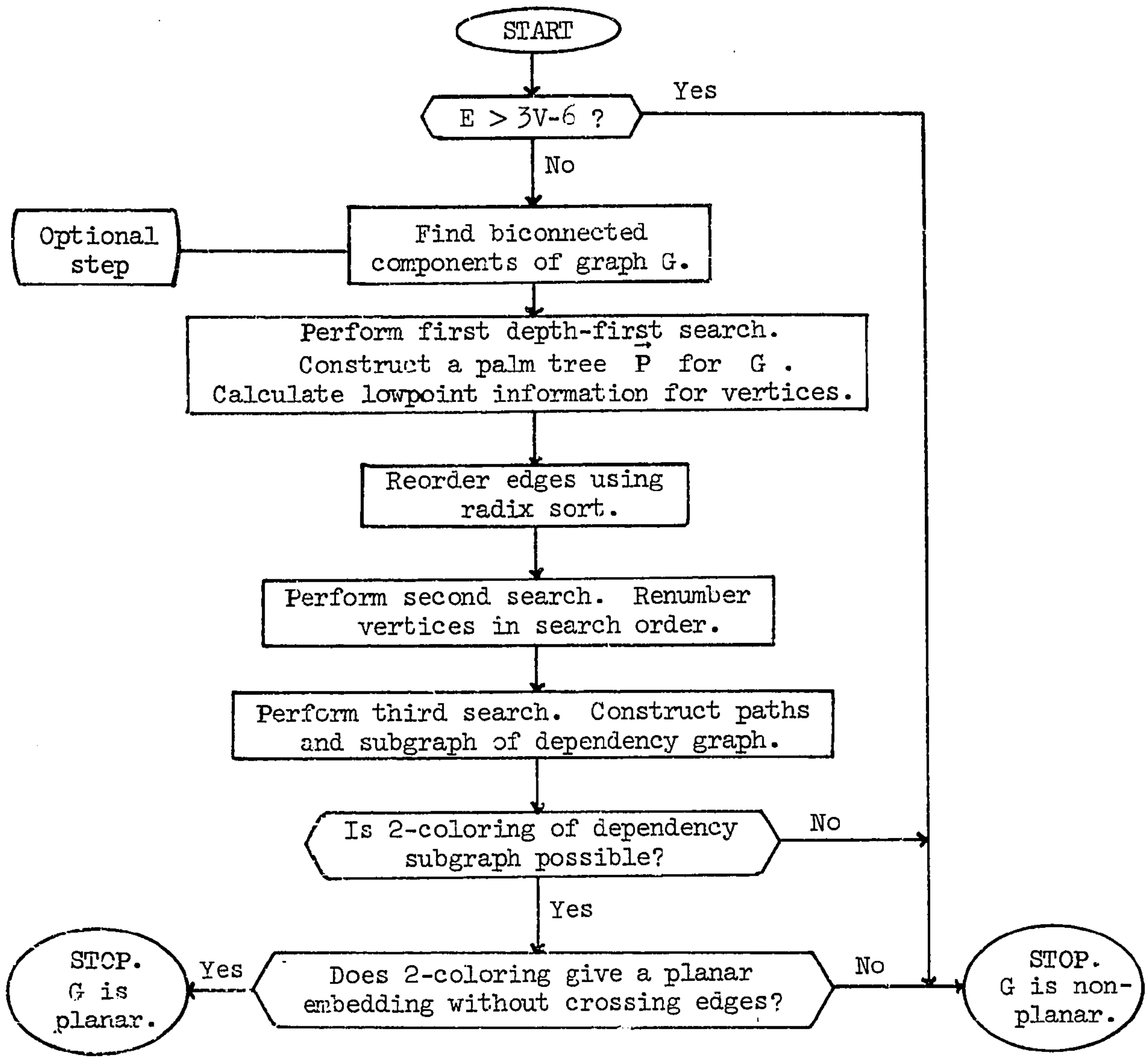


Figure 7.1: Flowchart for planarity testing algorithm.

in Chapter 6. (This step is not actually necessary, but it will simplify the presentation.)

Lemma 7.2: A graph is planar if and only if all its biconnected components are planar.

Proof: Standard. See [Ber 62].

Consider one of the biconnected components. We know that such a component may be converted into a palm tree $\vec{P}$ using a depth-first search. Suppose that $\vec{P}$ is embedded in the plane. Without loss of generality $\vec{P}$ may be embedded so that the branches of its spanning tree point "up" in the plane, and none of the fronds cross under the root of the tree. Let u be a vertex in the component, and let $(u,v_1),(u,v_2),\ldots,(u,v_n)$ be the tree arcs emanating from u, in the order they occur around u in the planar embedding. Let $T_1,T_2,\ldots,T_n$ be the subtrees whose roots are $v_1,v_2,\ldots,v_n$, respectively. Various fronds emanate from these subtrees and connect to ancestors of u, as illustrated in Figure 7.2.

For tree T_i, the lowest point of connection is $\text{LOWPT1}(v_i)$. The highest point of connection (below u) we may call $\text{HIGHPT}(v_i)$. Every subtree T_i except one (T_2 in Figure 7.2) must have all of its fronds descending on the same side of the branch $1 \overset{*}{\rightarrow} u$ in the planar embedding. The subtrees $T_1,T_2,\ldots,T_n$ must be arranged so that T_1 and T_n have the highest intervals $[\text{LOWPT1}(v_i),\text{HIGHPT}(v_i)]$ and these intervals are non-decreasing as we move in the sequence of subtrees toward the tree (if one exists) whose fronds descend on both sides of

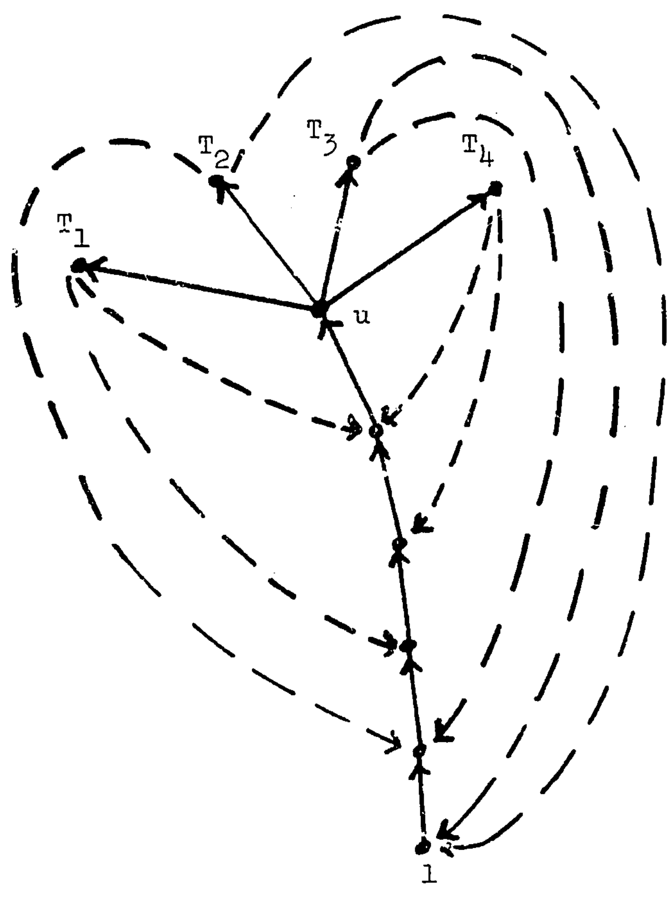


Figure 7.2: Relationship of subtrees adjacent to a single vertex in a planar embedding.

the branch $1 \overset{*}{\rightarrow} u$. Two subtrees (such as T_1 and T_4 in Figure 7.2) whose intervals overlap by more than one point cannot have their fronds descending on the same side of the branch $1 \overset{*}{\rightarrow} u$.

The value HIGHPT(v) is not easy to calculate, unfortunately, so we must resort to a bit of legerdemain to actually determine the proper arrangement of the various subtrees of a biconnected component. Instead of using subtrees, we examine paths. Each path is of the form $p: s \overset{*}{\rightarrow} \dashrightarrow f$. If (s,v) is the first edge on such a path p and $s \rightarrow v$ is a tree arc, then the interval associated with p is the same as that associated with T_v , the subtree rooted at v . If (s,v) is a frond (p is of length one), then the interval associated with p is $[v,v]$. We do not completely calculate these intervals but we do determine something about them; in particular we compute the lowest point of each interval and we determine which intervals consist of more than one point.

Using this information, we choose paths with the lowest intervals first. As the paths are selected, we may imagine adding them to a planar embedding which contains all the previously selected paths. If paths $p_1, p_2, \ldots, p_n$ pass through vertex s , then their ordering around s is restricted in the same way as the ordering of the corresponding subtrees $T_1, T_2, \ldots, T_n$, where T_i has root v_i , v_i is on path p_i , and $s \rightarrow v_i$. Thus each new path $p: s \overset{*}{\Rightarrow} f$ has at most a two-fold ambiguity in its placement; p must be placed either at the left end or at the right end of the sequence of paths around vertex s . See Figure 7.3. We call one of these possibilities the <u>left embedding</u> and the other the <u>right embedding</u>.

Using some additional information about the paths, we develop a dependency relation between paths: two paths may either constrain each

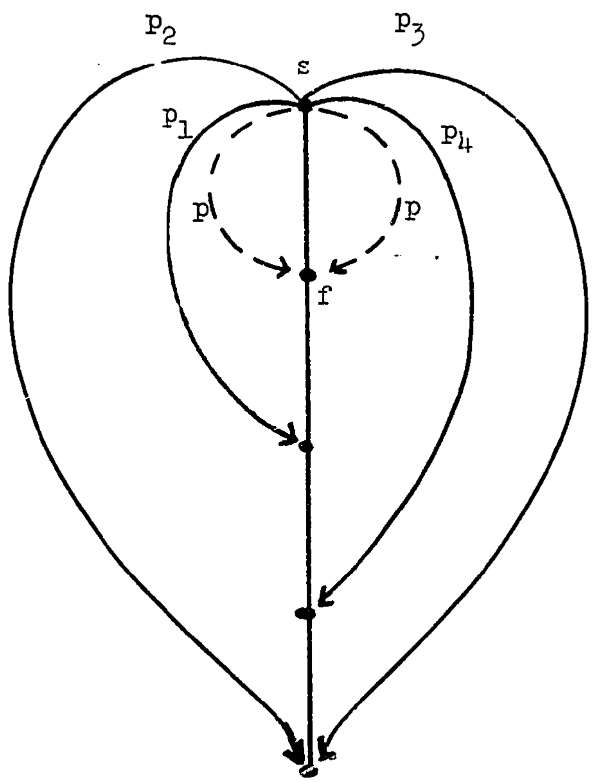


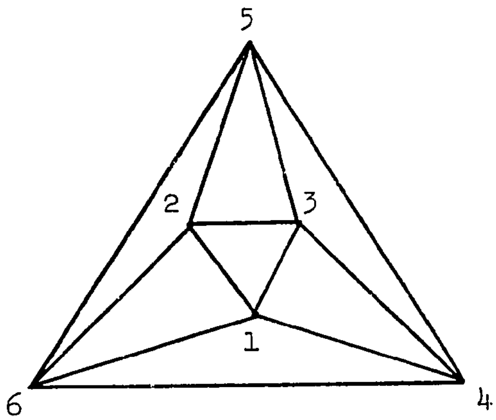
Figure 7.3: The two possible embeddings of new path p .

other to have the same embedding, or they may constrain each other to have opposite embeddings, or they may not restrict each other at all. The relation consists of a set of equalities and inequalities which must be satisfied over a two-element domain. We shall see that a graph is planar if and only if its dependency relation is satisfiable.

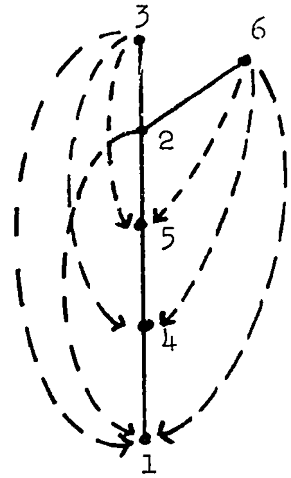
We may construct a graph corresponding to the dependency relation. The vertices in this graph are the paths in the original graph. Two paths are joined by an ELINK if they must have equal embeddings, and two paths are joined by an ILINK if they must have unequal embeddings. The resulting graph is called a dependency graph D ; this graph is colorable using two colors if and only if the original graph G is planar. In order to test planarity, then, we convert each biconnected component of the graph into a palm tree, we partition each palm tree into a set of edge-disjoint paths, we construct the corresponding dependency graph D , and we attempt to color D using two colors.

In order to get a fast algorithm, we must use another bit of cleverness. We shall see that the number of paths generated is $E-V+1$. The dependency graph may a priori contain up to $(E-V+1)(E-V)/2$ edges. We do not actually find all links in the dependency graph, but only enough to connect the connected components of this graph. Since a two-coloring of any connected component is essentially unique, the selected links provide enough information to give only one coloring. (We may permute colors in the various connected components arbitrarily.) We then test this coloring to see if it is a coloring of the entire dependency graph. If so, the original graph is planar and the coloring gives a planar embedding. If not, the graph is non-planar.

Each step of this process may be carried out in time proportional to the number of vertices. (The subgraph of the dependency graph which is actually constructed contains a number of links linear in V .) The storage space required is also proportional to the number of vertices. Thus the planarity algorithm is linear in V in both time and space; furthermore, the algorithm is optimal to within a constant factor, since any correct planarity algorithm must examine each edge of the graph at least once. Figure 7.4 gives an example of the algorithm's application. The example illustrates the general steps involved in determining planarity. In the next sections we develop the details of these steps.



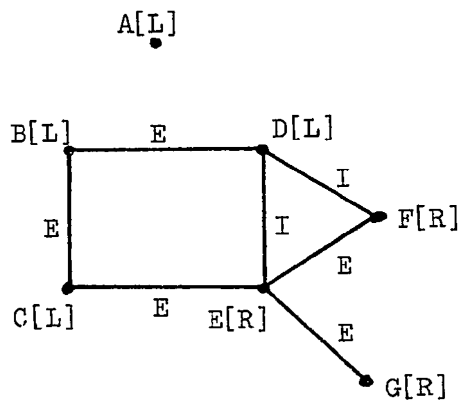
(a)



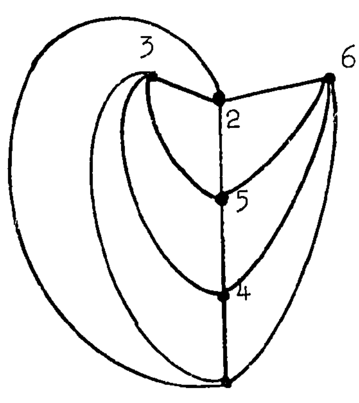
(b)

(c)

A: (1,4,5,2,1)
B: (2,3,1)
C: (3,4)
D: (3,5)
E: (2,6,1)
F: (6,4)
G: (6,5)



(d)



(e)

Figure 7.4: Application of the planarity algorithm. (a) Graph. (b) Generated palm tree. (c) Paths. (d) Dependency subgraph with 2-coloring in [ ]. (e) Planar embedding corresponding to 2-coloring.

8. Pathfinding

Assume that G is a biconnected graph with $E \leq 3V-6$. In order to decide whether G is planar, we shall perform three depth-first searches of G . The first search generates a palm tree $\vec{P}$ by directing all the edges of G . It also gives information about the fronds of $\vec{P}$. This information is used to construct an adjacency structure A for $\vec{P}$ which determines the last two searches. The second depth-first search numbers the vertices of $\vec{P}$. The third search generates paths and discovers their interrelationships. In this chapter we shall consider the three searches and the pathfinding process in detail.

If v is a point in a palm tree $\vec{P}$, we wish to know the set of points $S_v = \{w | v \overset{*}{\rightarrow} \dashrightarrow w\}$. The two lowest points in S_v adequately represent S_v for our purposes. Thus we have the following definition:

Definition 8.1: Let G be a connected undirected graph. Let $\vec{P}$ be a palm tree generated by a depth-first search of G . Suppose that the vertices of $\vec{P}$ are numbered in the order they are reached during the search. We define two numbers characteristic of a vertex v relative to the palm tree $\vec{P}$. LOWPT1(v) is the number of the lowest numbered vertex w_1 in the set $S_v = \{w | v \overset{*}{\rightarrow} \dashrightarrow w\}$. LOWPT2(v) is the number of the second lowest numbered vertex w_2 in the set S_v , if such a vertex $w_2 < v$ exists. If $|S_v| = 0$, LOWPT1(v) = LOWPT2(v) = $+\infty$. If $|S_v| = 1$, LOWPT2(v) = $+\infty$.

It is important to realize that LOWPT1(v) $\neq$ LOWPT2(v) unless LOWPT1(v) = LOWPT2(v) = $+\infty$. Figure 8.1 gives an example of a palm tree and two sets of its lowpoint values. The pair

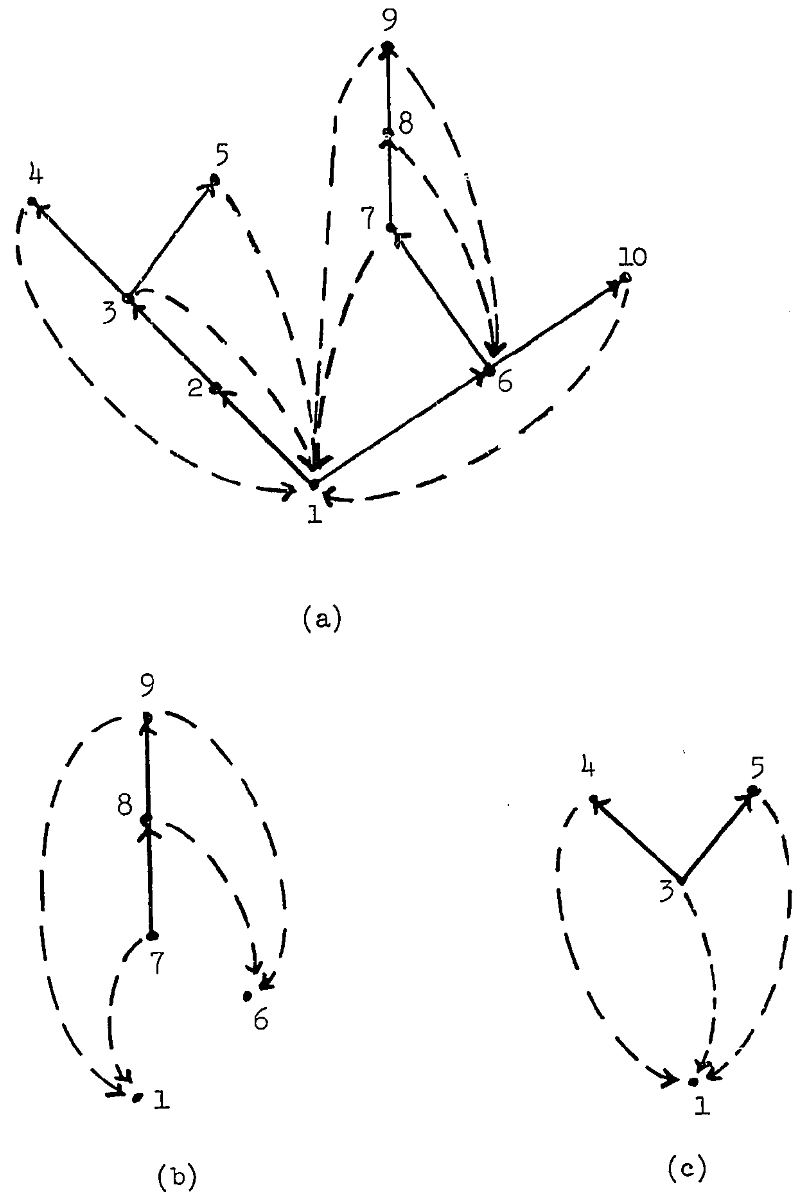


Figure 8.1: The meaning of LOWPT1 and LOWPT2.

(a) A palm tree.

(b) LOWPT1(7) = 1; LOWPT2(7) = 6.

(c) LOWPT1(3) = 1; LOWPT2(3) = $+\infty$.

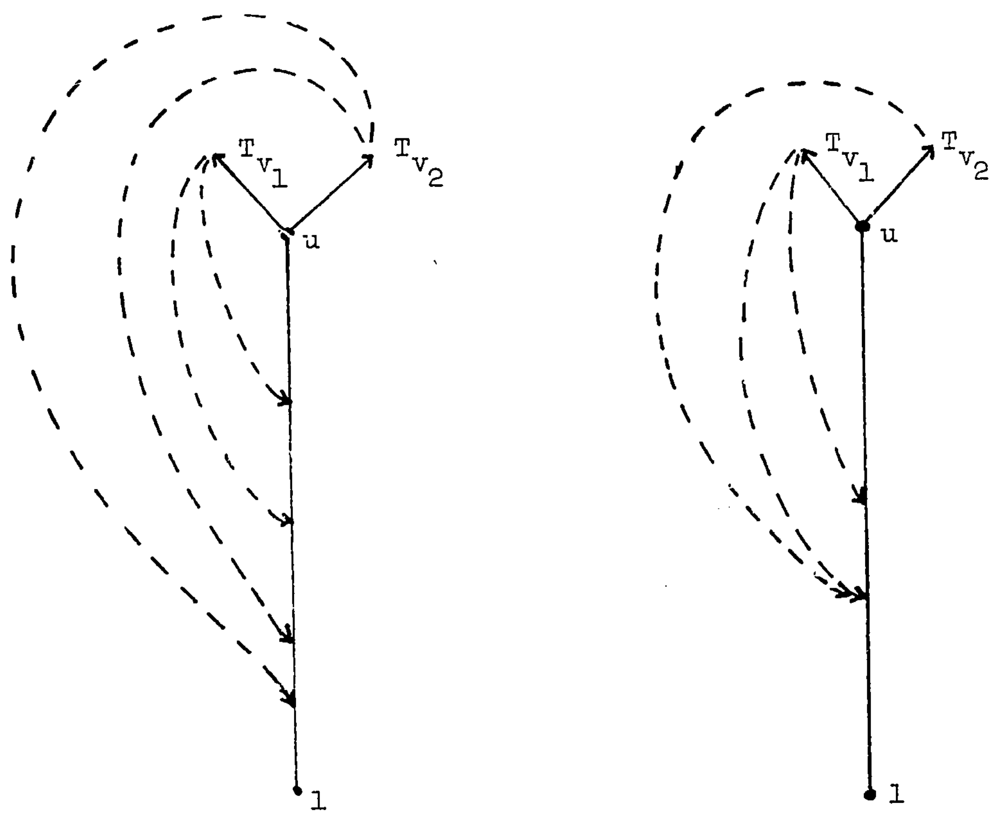
(LOWPT1(v) , LOWPT2(v)) is calculated during the initial depth-first search of G . The calculation is an extension of that in the biconnectivity algorithm. A recursive procedure for this calculation is presented below. It is easy to verify that the program correctly computes LOWPT1 and LOWPT2 , using a depth-first search which begins at vertex s .

```
begin
   integer i;
   procedure DFS1(v,u);
      begin
         NUMBER(v) := i := i+1;
         LOWPT1(v) := LOWPT2(v) + ∞;
         for w in the adjacency list of v do
            begin
               if w is not yet numbered then
                  begin
                     construct arc v → w in P⃗;
                     DFS1(w,v);
                     if LOWPT1(w) < LOWPT1(v) then
                        begin
                           LOWPT2(v) := min(LOWPT1(v),LOWPT2(w));
                           LOWPT1(v) := LOWPT1(w);
                        end
                     else if LOWPT1(w) = LOWPT1(v) then
                        LOWPT2(v) := min(LOWPT2(v),LOWPT2(w))
                     else LOWPT2(v) := min(LOWPT2(v),LOWPT1(w));
                  end
               else if NUMBER(w) < NUMBER(v) and w ≠ u then
                  begin
                     construct arc v --→ w in P⃗;
                     if NUMBER(w) < LOWPT1(v) then
```

```
                    begin
                       LOWPT2(v) := LOWPT1(v);
                       LOWPT1(v) := NUMBER(w);
                    end
                 else if NUMBER(w) > LOWPT1(v) then
                    LOWPT2(v) := min(LOWPT2(v),NUMBER(w));
              end;
          end;
      end;
   i := 0;
   DFS1(s,0);
end;
```

Figure 8.2 illustrates why we need only consider the two lowest points in the set S_v . Suppose $u \to v_1$ and $u \to v_2$ are two tree arcs in $\vec{P}$, and all fronds from T_{v_1} and T_{v_2} descend on the left in some planar embedding of $\vec{P}$. If $LOWPT1(v_2) < LOWPT1(v_1) < u$, or if $LOWPT1(v_2) = LOWPT1(v_1)$ and $LOWPT2(v_1) < u$, then v_1 must appear to the left of v_2 in the ordering of points around u . The algorithm will attempt to embed T_{v_2} before T_{v_1} .

The first search generates a palm tree $\vec{P}$. This palm tree has several possible adjacency structures, each corresponding to an ordering of the edges around the vertices of $\vec{P}$. The adjacency structures for $\vec{P}$ have one entry for each of the edges of the original graph G ; all the edges are now directed. We use the lowpoint values to choose a particular adjacency structure A , which will be used to determine the selection of paths in the graph. This adjacency structure is based upon the ordering of paths determined by their connections with ancestors of their start vertices which was described informally in Chapter 7. The



(a) (b)

Figure 8.2: Relationship of subtrees in a planar embedding.
(a) $\mathrm{LOWPT1}(v_2) < \mathrm{LOWPT1}(v_1) < u$.
(b) $\mathrm{LOWPT1}(v_2) = \mathrm{LOWPT1}(v_1)$; $\mathrm{LOWPT2}(v_1) < u$.

ordering is chosen so that a depth-first search using this adjacency structure will choose paths with lowest frond heads first. The implications of the ordering are presented in the lemmas below. We refer to vertices by the numbers assigned using DFS1 .

Definition 8.2: Let ϕ be the mapping from the edges of a palm tree $\vec{P}$ into $([1,V] \cup \{+\infty\}) \times \{0,1\}$ defined as follows:

(i) If $e = v \dashrightarrow w$, $\phi(e) = (w,0)$.

(ii) If $e = v \rightarrow w$ and $LOWPT2(w) \geq v$, $\phi(e) = (LOWPT1(w),0)$.

(iii) If $e = v \rightarrow w$ and $LOWPT2(w) < v$, $\phi(e) = (LOWPT1(w),1)$.

Definition 8.3: Let A be any adjacency structure for a palm tree $\vec{P}$. A is called acceptable if the edges e in each adjacency list L_v of A are ordered lexicographically according to the value of $\phi(e)$.

In general, a palm tree $\vec{P}$ has many acceptable adjacency structures A . It is easy to construct one by using a single radix sort. LOWPT1(v) and LOWPT2(v) are integers in the range $[1,V] \cup \{+\infty\}$. Since we may assume G is biconnected, $LOWPT1(v) < v$ for all vertices, and LOWPT1 is never $+\infty$. Thus we need 2V buckets. The following procedure gives the sorting algorithm. All vertices are identified by the number assigned to them during the initial search. It is obvious that the sorting procedure requires time proportional to V .

```
procedure SORT;
  begin
    for each arc (u,v) of P do
      if u --→ v then place (u,v) in BUCKET(2*v-1)
      else if LOWPT2(v) ≥ u then
        place (u,v) in BUCKET(2*LOWPT1(v)-1)
```

```
    else place (u,v) in BUCKET(2*LOWPT1(v));
  for i := 1 until 2*V do
    for each arc (u,v) in BUCKET(i) do
      place v at end of adjacency list of vertex u;
end;
```

Lemma 8.1: Let $\vec{P}$ be a biconnected palm tree with spanning tree $\vec{T}$. Suppose that the vertices of $\vec{P}$ are identified by distinct numbers in such a way that $v \overset{*}{\rightarrow} w$ in $\vec{T}$ implies $v < w$. Let LOWPT1 and LOWPT2 be defined as in Definition 8.1 using the given numbering. Then the acceptable adjacency structures are independent of the numbering chosen.

Proof: Since G is biconnected, LOWPT1(v) is always an ancestor of v. The value of $\phi((x,y))$ depends only on the fronds of $\vec{P}$ and the numbers of the ancestors of x. The order of the ancestors of a vertex is identical to the order of their numbers, by the hypothesis of the lemma, and this order is independent of the actual numbering selected. The property of being a frond of $\vec{P}$ is also independent of the numbering. Thus the edge order imposed by ϕ does not depend upon the numbering.

Lemma 8.1 implies that we may renumber the vertices of $\vec{P}$ in the order they are reached during any depth-first search of $\vec{P}$ without changing the adjacency structure A. (The adjacency structure specified by ϕ is not unique, but the possibilities for A are independent of the numbering.) The second depth-first search numbers the vertices in a special way in preparation for pathfinding. This search selects edges in the reverse order to that given by the adjacency structure A . The vertices are renumbered in the order they are reached during the search. This numbering is such that if vertex v appears before vertex w in the adjacency list of vertex u and $u \to v$, $u \to w$, then $v > w$. This backward numbering scheme is necessary in order to determine the interactions between the paths, as we shall see later. Henceforth we shall refer to vertices using the number assigned by the second depth-first search.

We have so far found a palm tree $\vec{P}$ for G , constructed an adjacency structure for $\vec{P}$ based upon its lowpoint values, and numbered the vertices of $\vec{P}$. We are ready to undertake the third depth-first search, which generates paths. The recursive procedure for this search appears below; PATHFINDER(1) carries out the calculation starting with the root of the palm tree. The search uses adjacency structure A (this time in the correct order) and works in the following way. The initial vertex (number one) is marked as the start vertex of the first path. The search proceeds until a frond is traversed. The sequence of edges traversed from the start vertex to this frond is the first path. When the next edge is traversed during the search, its tail vertex is marked as the start of a new path. The new path is completed when another frond is traversed. This process is repeated until the third search is completed.

```
procedure PATHFINDER(v);
   for w in the adjacency list of v do
      begin  comment  Vertex s is a global variable, the start
            vertex of the current path, and is initialized to 0;
         if v → w then
            begin
               if s = 0 then
                  begin
                     s := v;
                     start new path;
                  end;
               add (v,w) to current path;
               PATHFINDER(w);
A:             if s ≠ 0 then delete last edge on current path;
               if s = v then s := 0;
            end;
         else comment  v -→ w;
            begin
               add (v,w) to current path;
               output current path;
               s := 0;
            end;
      end;
```

The paths generated in this way have some very interesting properties which are crucial to the behavior of the remainder of the planarity algorithm.

In particular, if p: $s \overset{*}{\rightarrow} \dashrightarrow f$ is a generated path then f is the lowest vertex reachable via an unused frond from T_s. Further, if v is any intermediate vertex on path p, f is the lowest vertex reachable via any frond from T_v. A little more can be said because LOWPT2 is used in path selection. The lemmas below give the important properties. G is the original biconnected graph, having V vertices and E edges. $\vec{P}$ is the palm tree generated by the first search; $\vec{P}$ has spanning tree $\vec{T}$.

Lemma 8.2: The pathfinding algorithm generates E-V+1 paths.

Proof: One path is generated for each frond of $\vec{P}$. Since $\vec{T}$ has V-1 edges, there are E-V+1 paths.

Theorem 8.3: Let $p: s \overset{*}{\Rightarrow} f$ be a generated path. Then f is the lowest vertex reachable via an unused frond from T_s . If v is an intermediate vertex on p , f is the lowest vertex reachable via any frond from T_v .

Proof: If v is reached during the pathfinding search, then all ancestors of v have already been reached. A path terminates as soon as it reaches an ancestor of any vertex on the path. Each path contains one and only one frond, the last edge of the path. If p has length one, p consists of an unused frond leading to the lowest vertex reachable from T_s . If p has length greater than one and $s \to v$ is the first edge of p , then T_v has a frond leading to the lowest vertex reachable from T_s . This follows from the definition of ϕ . The theorem follows by induction.

Theorem 8.4: The first path generated by the pathfinding algorithm is a cycle. Each other path is a simple path having exactly two vertices (the endpoints of the path) in common and no edges in common with previously generated paths.

Proof: If a generated path p is of length one, it is obviously simple. If $p: s \to v \overset{*}{\to} \dashrightarrow f$, then f = LOWPT1(v) by Theorem 8.3.

Since G is biconnected, $f < s$ unless $s = 1$ by Lemma 6.2. Thus the initial path begins at vertex 1 and is a cycle, and all other paths are simple.

<u>Corollary 8.5</u>: If $p: s \overset{*}{\Rightarrow} f$ is one of the generated paths, then $f \overset{*}{\rightarrow} s$ in $\vec{T}$.

<u>Proof</u>: Immediate from the proof above, since LOWPT1(v) is an ancestor of v , for every vertex v in G .

<u>Lemma 8.6</u>: Let $p_1: s_1 \overset{*}{\Rightarrow} f_1$ and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be two generated paths such that $s_1 \overset{*}{\rightarrow} s_2$ in $\vec{T}$. Suppose that p_1 is found before p_2 . Then $f_1 \le f_2$.

<u>Proof</u>: Since s_2 is a descendant of s_1 in $\vec{T}$, path p_1 could have reached f_2 , but instead reached f_1 . By Theorem 8.3, $f_1 \le f_2$.

<u>Lemma 8.7</u>: Let $p_1: s \overset{*}{\Rightarrow} f_1$ and $p_2: s \overset{*}{\Rightarrow} f_2$ be two generated paths which have the same start vertex. Let v_1 be the second vertex of p_1 , let v_2 be the second vertex of p_2 , and suppose that p_1 is found before p_2 . Then we have:

(i) $f_1 \le f_2$.

(ii) Suppose $f_1 = f_2$. If p_1 is of length greater than one and $\mathrm{LOWPT2}(v_1) < s$, then p_2 is of length greater than one and $\mathrm{LOWPT2}(v_2) < s$.

<u>Proof</u>: Vertex v_1 appears before vertex v_2 in the adjacency list of vertex s , because path p_1 is generated before path p_2 . The lemma follows immediately from Definitions 8.2 and 8.3.

Lemma 8.8: Let $p_1: s_1 \overset{*}{\Rightarrow} f_1$ and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be two generated paths. Suppose that $s_1 \leq s_2$ and that p_1 is found before p_2 during the pathfinding process. Then $s_1 \overset{*}{\rightarrow} s_2$.

Proof: This result is immediate. The vertex numbering is such that the only vertices v which are examined after s_1 is first reached during pathfinding and such that $v \geq s_1$ are the descendants of s_1. Remember, the numbering scheme is backwards.

We know that a single depth-first search of a possibly planar graph G requires time proportional to V. (Remember, we have checked that $E \leq 3V - 6$.) The machinations performed during the three searches necessary to find paths all require only $O(1)$ time per step of the search process. Thus the total time spent on the three searches is $O(V)$. We have also seen that the sorting used to construct the adjacency structure A requires $O(V)$ time. Therefore the complete path generation process has a time bound linear in V. The space required is also obviously linear in V.

If G is not biconnected, the paths generated will not all be simple. In fact, any path passing from one biconnected component to another cannot possibly be simple. There are two ways in which simple paths may not be generated. One way is illustrated in Figure 8.3. The path in the figure consists of a simple path from v to w followed by a cycle which loops at w. Vertex w is an articulation point of the graph. The region R is separated from the rest of the graph by vertex w. The planarity testing algorithm will handle paths of the type "automatically"; the paths in region R do not interact with those in the rest of the graph. Figure 8.4 illustrates the only other possibility. Vertex w is a dead end (a vertex of degree one). If such a vertex is reached during path generation, the edge leading to it is deleted from the graph and ignored. (This is accomplished by test A in procedure PATHFINDER.) The presence or absence of the deleted edge does not affect the planarity of the graph. Although this is only an intuitive justification for the dispensability of the biconnectivity assumption, one may easily verify this fact using the results below.

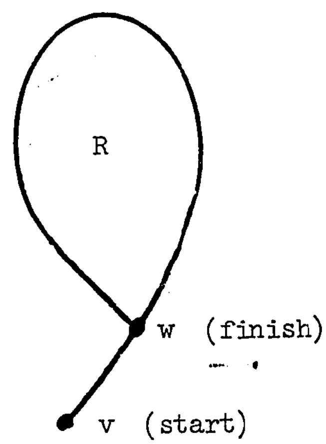


Figure 8.3: A non-simple path.

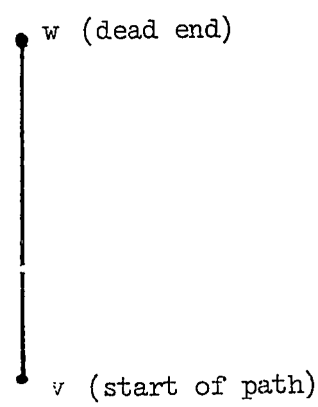


Figure 8.4: A dead-end branch.

9. Embedding of Paths

We have learned how to partition a biconnected graph G into a set of simple paths, such that each path has only its endpoints in common with previous paths, and each edge occurs in exactly one path. In this chapter we discover how to embed these paths in the plane. Every path, when it is placed, has at most two possible embeddings with respect to paths placed earlier, and we shall characterize these possibilities.

Assume that the paths found in G are numbered from 1 to $E - V + 1$ in the order they have been generated; path one is the initial cycle. We may associate a unique path with each vertex; namely the lowest numbered path to contain that vertex. We shall distinguish three types of paths; these paths interact in different ways. The first type of path is the initial cycle; it is unique. The other two types are given by the following definition.

<u>Definition 9.1</u>: Let $p: s \overset{*}{\Rightarrow} f$ be a simple path generated by the pathfinding algorithm. Let $p_0: s_0 \overset{*}{\Rightarrow} f_0$ be the earliest generated path containing s. If $f_0 < f$, then p is called a <u>normal path</u>. $f_0 = f$ then p is called a <u>special path</u>. The case $f_0 > f$ cannot occur by Lemma 8.6.

Let us imagine embedding the paths in the plane one at a time in the order they are generated. The results which follow give a specification of the possible placements of a path, in relation to previously embedded paths.

Theorem 9.1: Let p be a generated path in a biconnected planar graph G . Suppose the previously generated paths have been embedded in the plane. Then there are two possible ways to add p to the embedding, at least one of which may be extended to give a planar embedding of the entire graph.

Proof: The theorem does not claim that there are only two possible ways to insert the path p . It merely asserts that there are two placements of p to which we may restrict our attention without affecting the planarity of G . The proof requires consideration of the three different types of paths and follows from the next three lemmas, which characterize the two embeddings for each of the three types of paths. Without loss of generality we may assume that G is embedded in the plane in such a way that the arcs of the spanning tree $\vec{T}$ of G point "up" in the plane and no frond passes under the root of $\vec{T}$.

Definition 9.2: Let $\vec{P}$ with root 1 be a palm tree embedded in the plane, with tree arcs pointing "up" and no frond passing under vertex 1 . Let (v,w) be a frond of $\vec{P}$, with $x \to w \to y \overset{*}{\to} v$. (If $w = 1$, add an extra tree arc $x \to 1$ to the embedding, with vertex x directly below vertex 1 .) Frond (v,w) is said to descend on the right (of branch $1 \overset{*}{\to} v$) if the order of edges clockwise around w is (x,w) , (w,y) , (v,w) . Frond (v,w) is said to descend on the left (of branch $1 \overset{*}{\to} v$) if the order of edges clockwise around w is (x,w) , (v,w) , (w,y) . Figure 9.1 illustrates this definition.

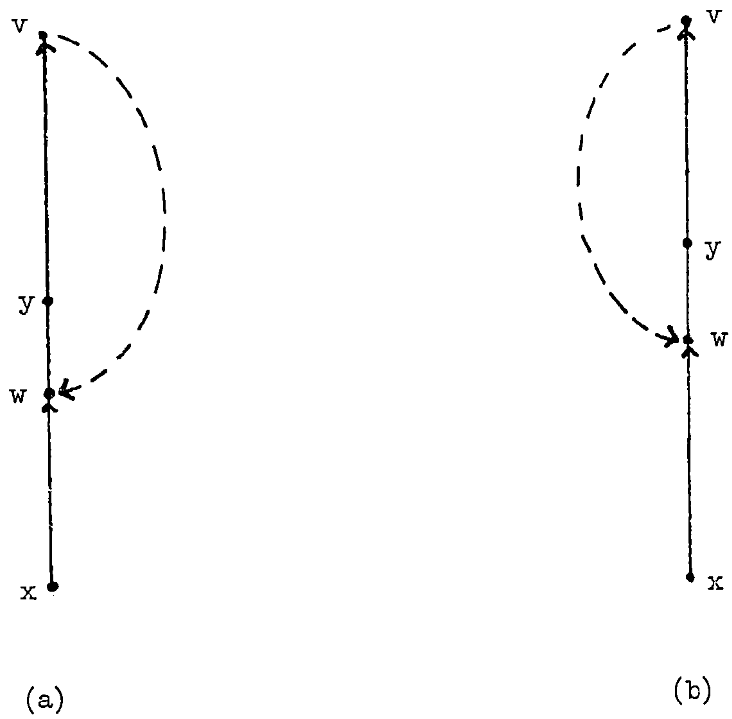


Figure 9.1: Position of fronds in a planar palm tree.

(a) Frond descends on right.

(b) Frond descends on left.

Lemma 9.2: Let $p: 1 \overset{*}{\Rightarrow} 1$ be the initial cycle of G . Then Theorem 9.1 is true for p .

Proof: Figure 9.2 illustrates the two possible embeddings for the initial cycle. If the tree arcs of the cycle are drawn upwards in the plane, the frond which forms the last arc of the cycle may descend either on the left side or on the right side of the tree arcs, giving respectively the left embedding and the right embedding.

Lemma 9.3: Let $p: s \overset{*}{\Rightarrow} f$ be a normal path of G . Let $p_0: s_0 \overset{*}{\Rightarrow} f_0$ be the earliest path containing s and suppose that $x \to s$. Then Theorem 9.1 is true for p , and without loss of generality p may be inserted into one of the two faces in the partial embedding having the edge (x,s) on its boundary.

Proof: Let $p_1, p_2, \ldots, p_n$ be the paths already embedded which contain vertex s , in the order they occur clockwise around s beginning from arc $x \to s$. We will show that without loss of generality p may be embedded either to the left of p_1 , with its frond descending on the left of branch $1 \overset{*}{\to} s$ or to the right of p_n , with its frond descending on the right of $1 \overset{*}{\to} s$. Thus suppose we wish to place p so that its frond descends on the right.

Suppose $p_n: s \overset{*}{\Rightarrow} f_n$ (path p_n starts at s). Let c_n be the cycle formed by p_n and the branch $f_n \overset{*}{\to} s$. If $f_n < f$ and the frond of p descends on the right, p must be placed to the right of p_n by Lemma 3.7 and Definition 9.2, since the frond of p and the first edge of p must be on the same side of c_n (Figure 9.3(a)). This argument also shows that p must be to the

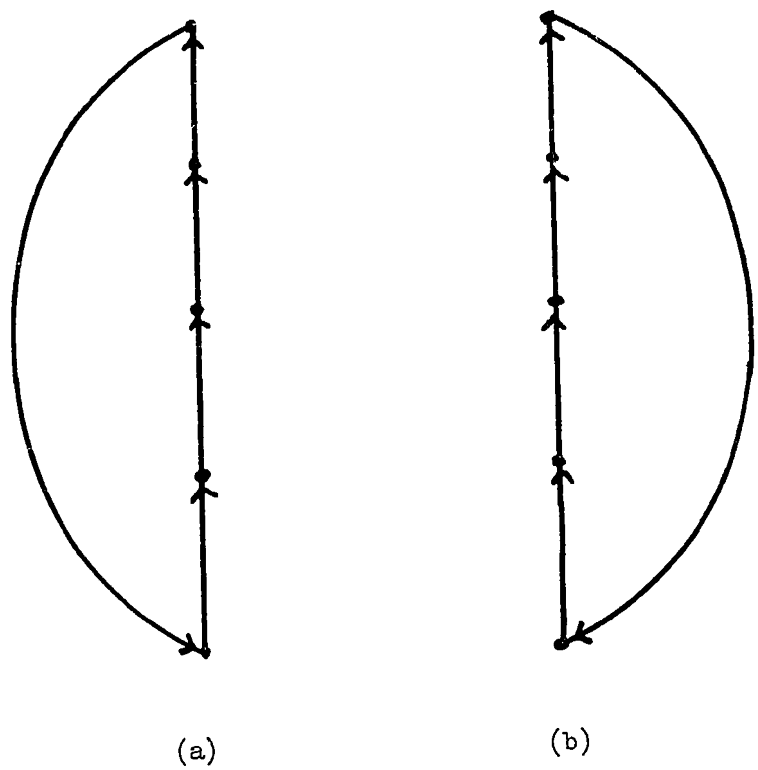


Figure 9.2: Embedding of a cycle.
(a) Left embedding.
(b) Right embedding.

right of p_0 in the ordering of paths about s .

Suppose $f_n = f$, $p: s \to v \overset{*}{\to} \dashrightarrow f$, and LOWPT2(v) < s . Consider a frond e whose tail is a descendant of v and whose head is LOWPT2(v) . If the frond of p descends on the right then so does e , applying Lemma 3.7 to cycle c_0 , e , and the frond of p . Applying Lemma 3.7 to c_n , e , and the first edge of p shows that p must be placed to the right of p_n in the ordering cf paths about s (Figure 9.3(b)).

If $f_n = f$, and either p has length 1 or $p: s \to v \overset{*}{\to} \dashrightarrow f$ with $\text{LOWPT2}(v) \geq s$, then either p_n has length 1 or $p_n: s \to v_n \overset{*}{\to} \dashrightarrow f$ with $\text{LOWPT2}(v_n) \geq s$, by Lemma 8.7. In this case (s,f) is a biarticulation point pair in G , as may be proved in the same way as Lemma 6.2. Path p may be placed either to the right or to the left of p_n without affecting the planarity of G [Har 69]. Without loss of generality we place p to the right of p_n (Figure 9.3(c)).

We must still consider what happens when $p_n: s_n \overset{*}{\Rightarrow} s$ (path p_n finishes at s). In this case some earlier path $p_k: s_k \overset{*}{\to} s \to v \overset{*}{\Rightarrow} f_k$ has $v \overset{*}{\to} s_n$. (If $p_k = p_0$, $s_k = s_0 \neq s$. Otherwise $s_k = s$.) The argument above applies to p_k . Further, if c is the cycle formed by p_n , the part of p_k following vertex v , the branch $v \overset{*}{\to} s_n$, and the branch $f_k \overset{*}{\to} s$, then both ends of p must be on the same side of c . Lemma 3.7 shows that p must be placed to the right of p_n in the ordering of paths about s .(Figure 9.4(d)).

The entire argument presented here is symmetric with respect to left and right, so without loss of generality we may embed p in

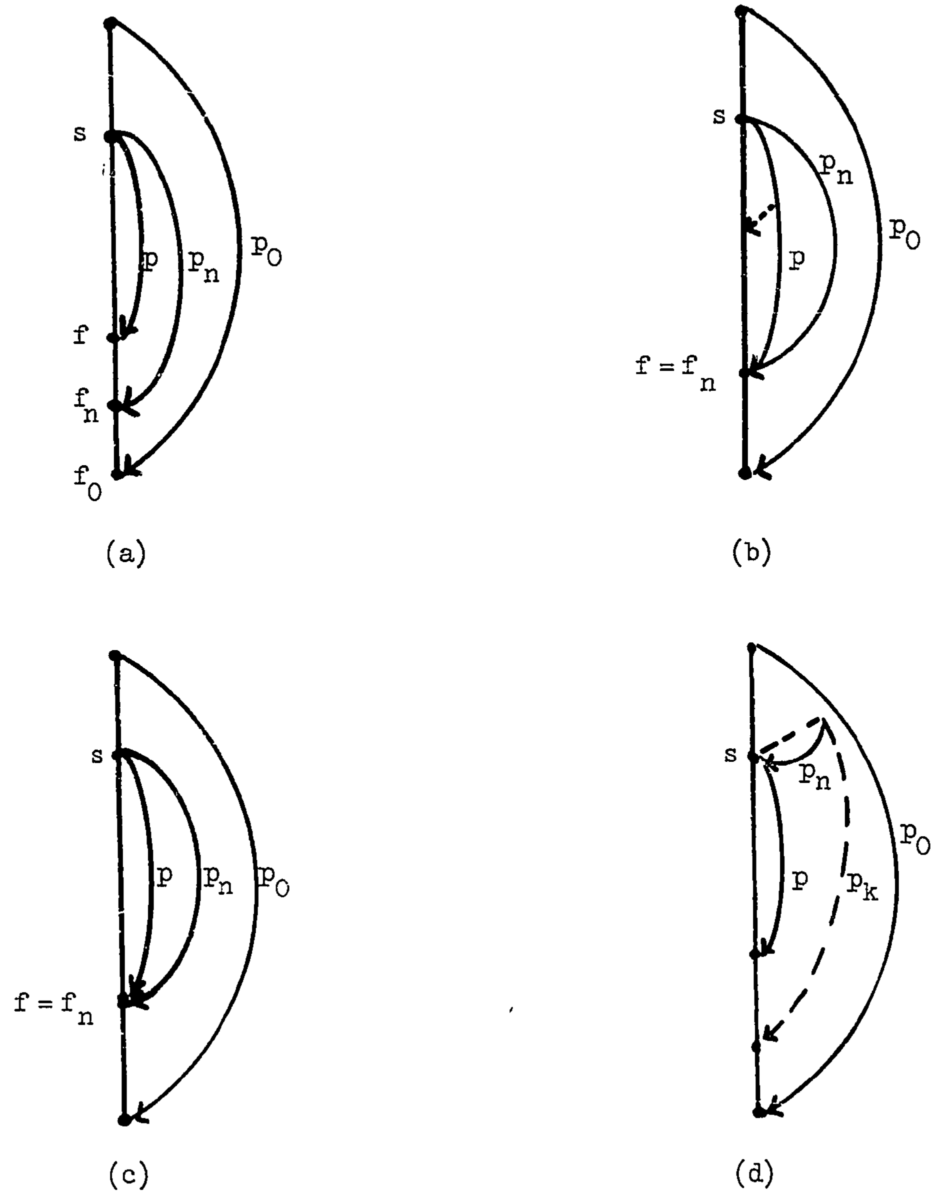


Figure 9.3: Embedding of a normal path p with start point on p_0.

(a) Interaction with path p_n, $f_n < f$.

(b) Interaction with path p_n, $f_n = f$, p has two connections.

(c) Interaction with path p_n, $f_n = f$, p has only one connection.

(d) Interaction with path p_n, $f_n = s$.

one of two places; either at the left end of the sequence of paths ordered around s, with its frond descending on the left (the <u>left embedding</u>); or, at the right end of the sequence of paths, with its frond descending on the right (the <u>right embedding</u>).

<u>Lemma 9.4</u>: Let $p: s \overset{*}{\Rightarrow} f$ be a special path of G. Let $p_0: s_0 \overset{*}{\Rightarrow} f$ be the earliest path containing s and suppose that $x \to s$. Then Theorem 9.1 is true for p, and without loss of generality p may be inserted into one of the two faces in the partial embedding having edge (x,s) on its boundary.

<u>Proof</u>: Assume that path p_0 is embedded with its frond descending on the right. An argument similar to the proof of Lemma 9.3 shows that p may be embedded at the left end (the <u>left embedding</u>) or at the right end (the <u>right embedding</u>) of the sequence of previously embedded paths ordered clockwise around s beginning from arc $x \to s$.

The location of the frond of p is not fixed by this argument; we must determine whether it descends on the left side or on the right side of the branch $1 \overset{*}{\to} s$. Figure 9.4 illustrates the three possibilities. If p has the right embedding its frond descends on the right, as in Figure 9.4(b). If p has the left embedding, its frond also descends on the right, as in Figure 9.4(a). If f = 1, this is true because the embedding 9.4(c) in which the frond descends on the left is topologically equivalent (on the sphere) to 9.4(a) and may be ignored. An induction argument shows that we may choose embedding 9.4(a) instead of 9.4(c) for all special paths with finish vertex 1.

If $f \neq 1$, then we must have $f \rightarrow v \overset{+}{\rightarrow} s$ with $LOWPT1(v) < f$, by Lemma 6.2. (G is biconnected.) Thus some path $p': s' \overset{*}{\Rightarrow} f'$, with s' on $v \overset{*}{\rightarrow} s$ and $f' < f$, has already been embedded. Applying Lemma 3.7 to the cycle c_0 formed by p_0 and $f \overset{*}{\rightarrow} s_0$, and to the cycle c' formed by p' and $f' \overset{*}{\rightarrow} s'$, shows that the embedding illustrated in Figure 9.4(c) is impossible.

Hence the frond of p descends on the same side as the frond of p_0 , independent of p's embedding. This is the difference between normal and special paths.

Definition 9.2: Let G be a biconnected planar graph. Suppose that the pathfinding algorithm is applied to G , partitioning it into a set of paths. Consider a planar representation of G such that each generated path has the left embedding or the right embedding as defined above. Such a representation is called a standard planar representation of G .

Given this definition, Theorem 9.1 becomes:

Theorem 9.4: Every biconnected planar graph G has a standard planar representation.

The proofs of the lemmas above depend heavily upon the ordering determined by ϕ and used to construct the adjacency structure A . In particular, paths would not be restricted to only two possible embeddings if LOWPT2 had not been used in the ordering. Having determined the possible path placements, we must determine how paths behave within these restrictions. This is the subject of the next chapter.

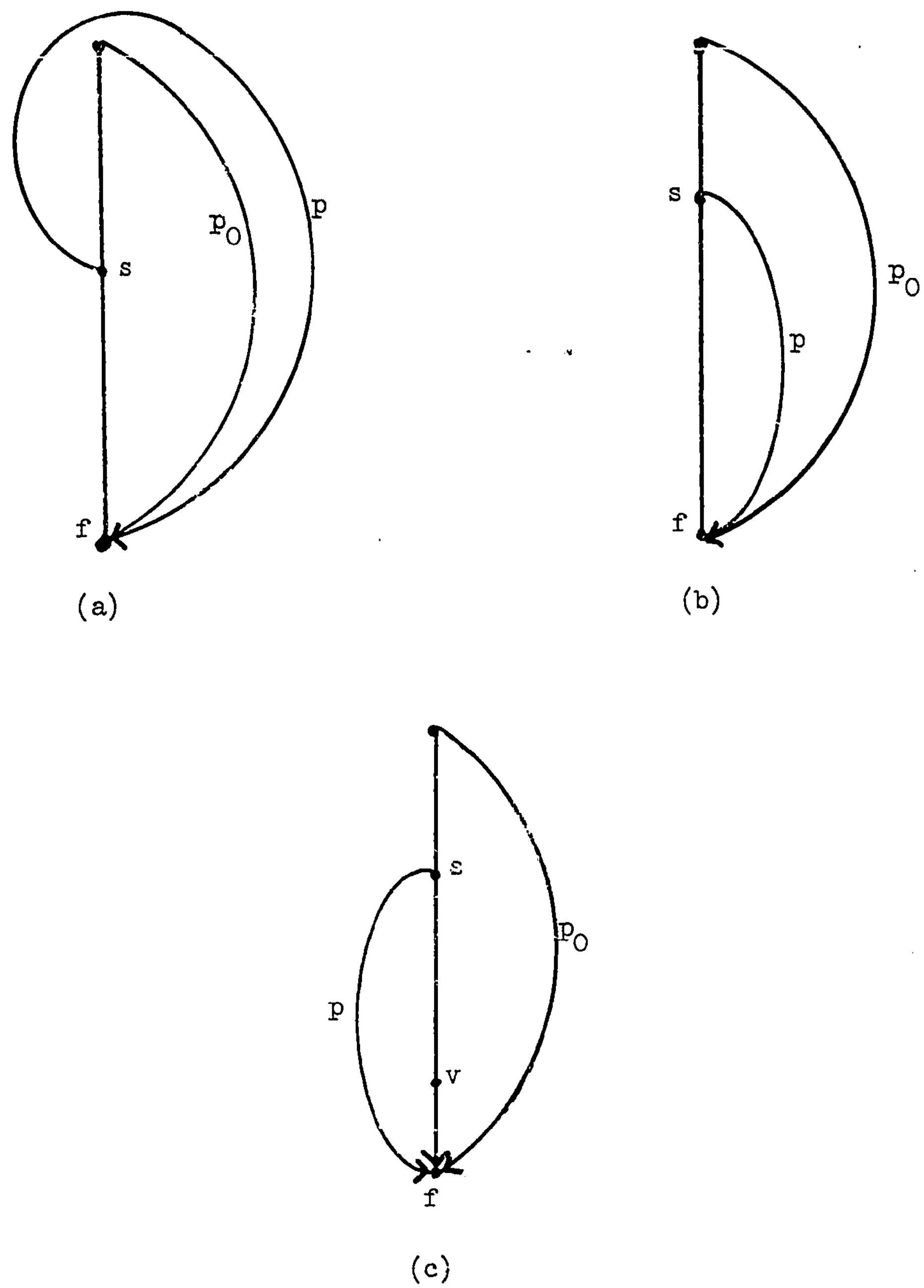


Figure 94: Embedding of special path p with start vertex on path p_0.

(a) Left embedding.

(b) Right embedding.

(c) Embedding equivalent to (a) if $f = 1$ and impossible otherwise.

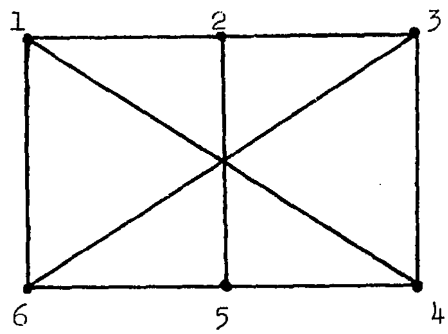
10. Dependence

Each path may be added to the planar representation we are constructing in at most two different ways. Even within these restrictions the placement of paths is not arbitrary; embedding a path in a certain way may affect the embedding of other paths. In this chapter we analyze these additional path interactions, which are in fact sufficient to determine the planarity of the graph.

Figure 10.1 shows the paths generated when the pathfinding algorithm is applied to one of the Kuratowski subgraphs. Using Lemma 3.7 it is easy to show that paths B and C must have different embeddings in any planar embedding of $K_{3,3}$. Similarly, B and D must have different embeddings, and C and D must have different embeddings. Thus $K_{3,3}$ cannot possibly be planar, since there are only two possible embeddings for each path. We wish to carry out an analysis of this sort for an arbitrary graph G.

Lemma 10.1: Let $p_1: s_1 \to v \overset{*}{\Rightarrow} f_1$ and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be two paths generated when the pathfinding algorithm is applied to a biconnected planar graph G. Suppose path p_2 is normal. If $v \overset{*}{\to} s_2$ and $f_1 < f_2 < s_1$, then p_1 and p_2 have the same embedding in any standard planar representation of G.

Proof: Path p_2 must be generated after p_1, because vertex s_2 is not reached during pathfinding until after p_1 is generated ($v \overset{*}{\to} s_2$). If w is the highest numbered ancestor of s_2 on the path p_1, $v \leq w$. Let $p_0: s_0 \overset{*}{\Rightarrow} f_0$ be the earliest path containing vertex s_1. The edge $s_1 \to v$ and the frond of p_2 must be on the same side of the cycle formed by p_0 and the



(a)

(b)

A: (1,2,3,4,5,6,1)

B: (6,3)

C: (5,2)

D: (4,1)

(c)

Figure 10.1: Relationship of paths in $K_{3,3}$.

(a) Graph.

(b) Generated palm tree.

(c) Generated paths.

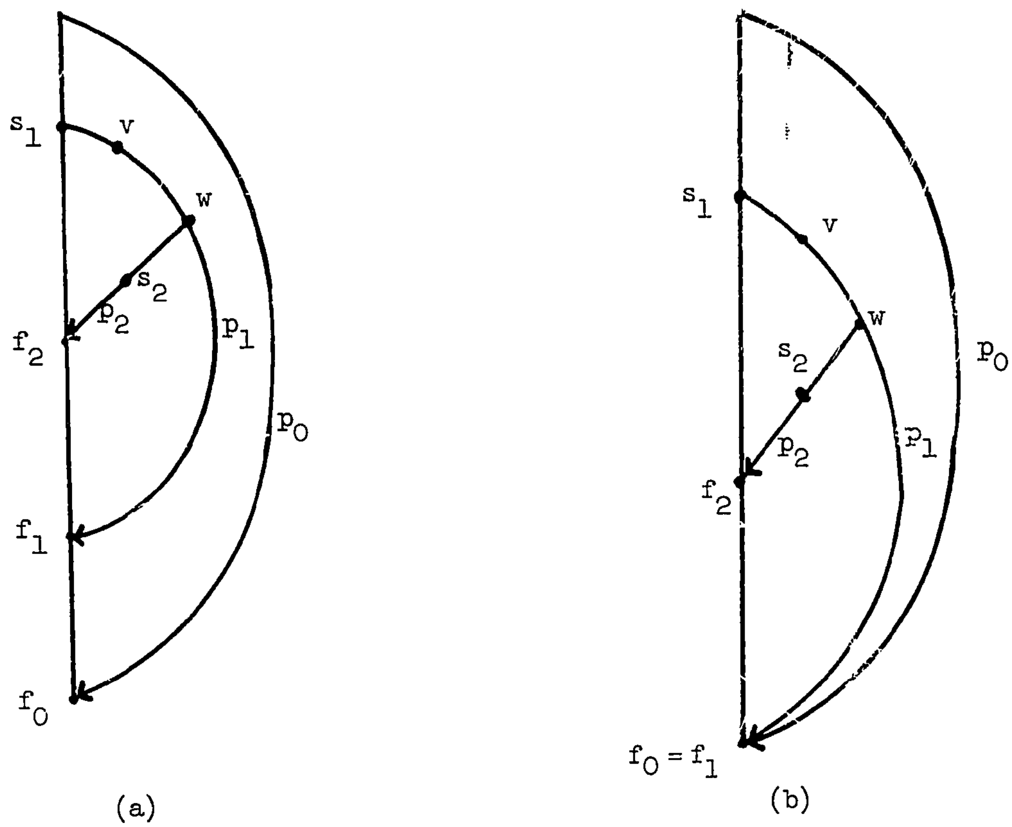
branch $f_0 \xrightarrow{*} s_0$. By Lemma 3.7, p_1 and p_2 must have the same embedding (Figure 10.2).

Lemma 10.2: Let p_1: $s_1 \overset{*}{\Rightarrow} f_1$ and p_2: $s_2 \overset{*}{\Rightarrow} f_2$ be two generated paths in G . Suppose p_1 is generated before p_2 and that p_1 is normal. If s_2 is on the branch $f_1 \xrightarrow{*} s_1$ and $f_2 < f_1 < s_2 < s_1$, then p_1 and p_2 must have different embeddings in any standard planar representation of G .

Proof: Let e be the first edge of $s_2 \xrightarrow{*} s_1$. Edge e and the frond of p_1 must be embedded on the same side of the cycle formed by p_2 and $f_2 \xrightarrow{*} s_2$. Lemma 3.7 implies that p_1 and p_2 have different embeddings (Figure 10.3).

Lemma 10.3: Let p_1: $s_1 \overset{*}{\Rightarrow} f_1$ and p_2: $s_2 \overset{*}{\Rightarrow} f_2$ be two normal paths in G generated by the pathfinding algorithm. Suppose p_1 is generated before p_2 . Let v be the second vertex on the branch $f_1 \xrightarrow{*} s_1$. If $v \leq s_2 < s_1$ and $f_2 < f_1$ then p_1 and p_2 must have different embeddings in any standard planar representation of G .

Proof: The numbers of the descendants of v form an interval (v,v+k) . Since $v \xrightarrow{*} s_1$ and $v \leq s_2 < s_1$, $v \xrightarrow{*} s_2$. Let w be the highest numbered common ancestor of s_1 and s_2 . If $s_2 = w$, the lemma follows from Lemma 10.2. Otherwise, let p: $w \rightarrow x \overset{*}{\Rightarrow} f$ be the generated path such that $x \xrightarrow{*} s_2$. Paths p and p_1 must have different embeddings by Lemma 10.2 and paths p and p_2 must have the same embedding by Lemma 10.1. This gives the lemma.



<u>Figure 10.2</u>: ELINK relation between a path p_1 and a normal path p_2.

(a) Path p_1 normal.

(b) Path p_1 special.

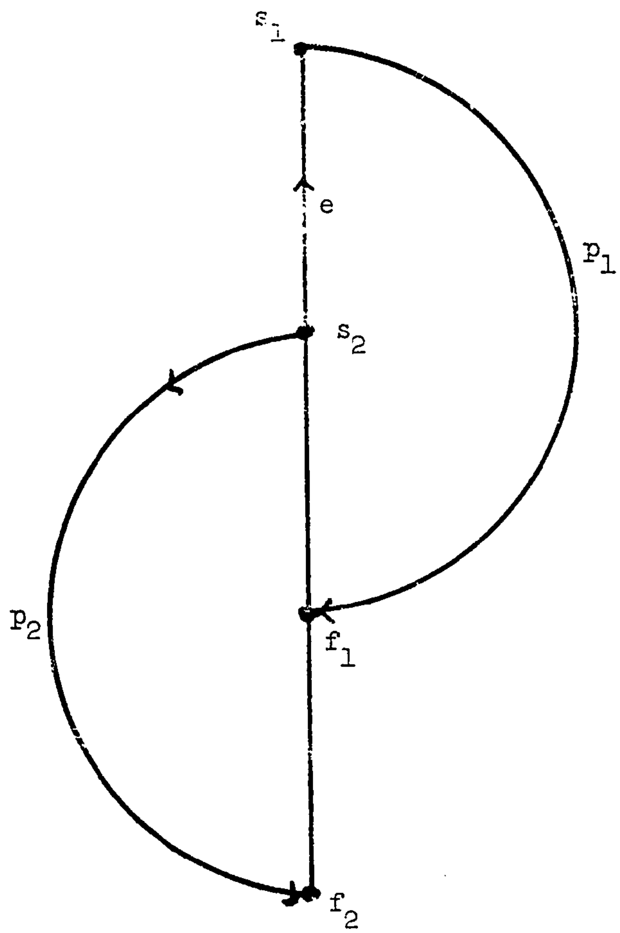


Figure 10.3: ILINK relation between a normal path p_1 and another path p_2.

Definition 10.1: Let G be a biconnected graph. Suppose the pathfinding algorithm is applied to G to yield a set of edge-disjoint paths which contain all the edges of G . Let $\{x_p\}$ be a set of variables, one for each of the paths. Let R be the smallest set of relations containing " $x_{p_1} = x_{p_2}$ " for each pair of paths p_1 , p_2 related as in Lemma 10.1, containing " $x_{p_1} \neq x_{p_2}$ " for each pair of paths p_1 , p_2 related as in Lemma 10.2, and containing " $x_{p_1} \neq x_{p_2}$ " for each pair of paths p_1 , p_2 related as in Lemma 10.3. (The inequalities based on Lemma 10.3 are redundant, but are added for convenience.) R is called the dependency relation of G . Let D be a graph having the paths of G as vertices, and having two types of edges (links). If " $x_{p_1} = x_{p_2}$ " $\in R$ then (p_1,p_2) is an ELINK in D . If " $x_{p_1} \neq x_{p_2}$ " $\in R$, then (p_1,p_2) is an ILINK in D . Then D is called the dependency graph of G .

Theorem 10.4: Let G be a biconnected graph with a dependency relation R and a dependency graph D . If G is planar, then R is satisfiable over a two-element domain. Equivalently, the vertices of D (the paths in G) may be colored with two colors so that any two paths joined by an ILINK are colored differently, and any two paths joined by an ELINK are colored the same.

Proof: This result follows from Theorem 9.4 and the three lemmas above. If G is planar, then G has a standard planar representation. We color the vertices of D with the colors "left" and "right" according to the embeddings of the corresponding paths in some standard planar representation of G . Lemmas 10.1 and 10.2

guarantee that the coloring satisfies the restrictions imposed by the links in D .

The planarity test is based upon the fact that the converse to Theorem 10.4 is true; 2-coloring the dependency graph D gives a complete test for the planarity of the original graph G . Before we verify this fact, we shall show that the structure of the dependency graph D is related not only to the planarity of the original graph G but also to the connectivity properties of G . (Since the proofs below are rather involved and are not directly related to the planarity algorithm, anyone interested only in planarity may skip the remainder of this chapter.)

Our objective is to show that the connected components of D are related in a simple way to the triconnectivity of G .

<u>Lemma 10.5</u>: Let $G = (\mathcal{V}, \mathcal{E})$ be a triconnected graph. Suppose the pathfinding algorithm is applied to G , giving a set of paths with a dependency graph D . Let $p_1: s_1 \overset{*}{\Rightarrow} f_1$ and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be two generated paths such that p_1 is the earliest path containing vertex s_2 , and p_1 is not the initial cycle. Then p_1 and p_2 are in the same connected component of D .

<u>Proof</u>: The proof of this lemma is complicated. Consider Figure 10.4. If $s_1 > f_2 > f_1$, (p_1, p_2) is an ELINK in D and there is nothing to prove. If $f_2 \geq s_1$, p_2 is normal; if $f_2 = f_1$, p_2 is special. In either of these cases, (p_1, p_2) is not a link in D . Let $S = \{p_2\}$. We prove the lemma by adding paths to S one by one. Each path added to S will be connected to p_2 in D . Eventually a path connected to p_1 in D will be added to S . We

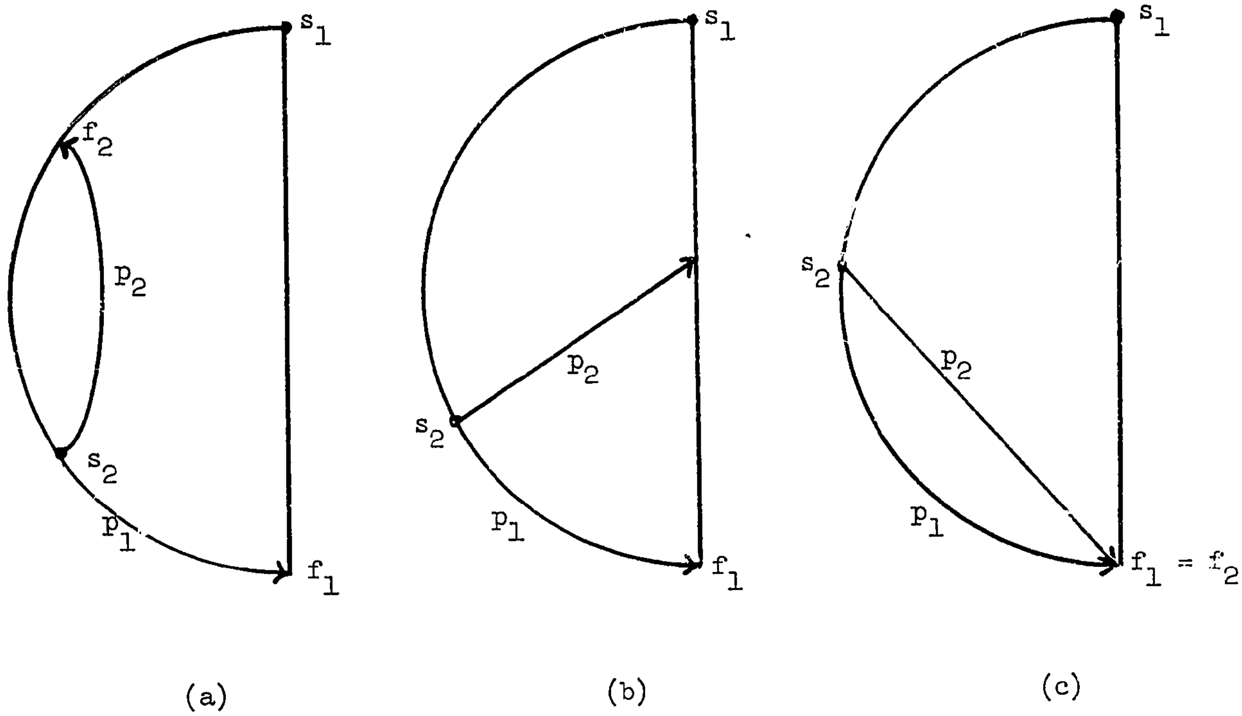


Figure 10.4: Connection in dependency graph to be proved.

(a) Second path normal, no link.

(b) Second path linked to first.

(c) Second path special.

use one extension method if S contains only normal paths and another extension method if S contains at least one special path.

Extension method 1.

Suppose S is a collection of normal paths generated by extension method 1 from the initial set $\{p_2\}$. Let s_0 be the highest numbered endpoint of a path in S and let f_0 be the lowest numbered endpoint of a path in S. Both s_0 and f_0 lie on path p_1. This may be proved by induction on the paths added to S. Let $W = W_0 \cup U_{p\in S} W_p$, where $f_0 \to v_0$ and v_0 is on path p_1, $W_0 = \{w' \mid v_0 \overset{*}{\to} w' \ \& \ \neg (s_0 \overset{*}{\to} w')\}$, and if $p\colon s \to v \overset{*}{\Rightarrow} f$, $W_p = \{w'' \mid v \overset{*}{\to} w''\}$. There must be a generated path $p_3\colon s_3 \overset{*}{\Rightarrow} f_3$ with one endpoint in W and the other endpoint in $V - W - \{s_0, f_0\}$, where V is the set of vertices of G. Otherwise G is not triconnected, since (s_0, f_0) would be a biarticulation point pair in G. Either f_3 is a proper ancestor of f_0, or s_3 is a proper descendant of s_0.

Suppose f_3 is a proper ancestor of f_0. Let w be the first common ancestor of s_3 and s_0. We have $f_0 < w < s_0$, and w is on the branch $f_0 \overset{*}{\to} s_0$. (Vertex s_3 cannot lie in any W_p because a path with start vertex in some W_p ends at a descendant of f_0 by Lemma 8.6.)

We may in fact assume that $s_3 = w$ because $p'\colon w \to x \overset{*}{\Rightarrow} f'$ with $x \overset{*}{\to} s_3$ has finish vertex at least as low as f_3 by Lemma 8.6.

We may extend the set of paths S by adding p_3. Path p_3 must be joined by an ILINK to some path p already in S, since every point on $f_0 \overset{*}{\rightarrow} s_0$ except f_0 and s_0 lies between the start and finish vertices of some path in S. This may be proved by induction on the paths added to S. If $f_3 \geq s_1$, then p_3 is normal, and we may use extension method 1 for the next step. If $f_0 < f_3 < s_0$, then p_3 is joined by an ILINK to p_1 and we are done. If $f_3 = f_0$, p_3 is special, and we use extension method 2 for the next step.

Suppose that s_3 is a proper descendant of s_0. Then vertex f_3 must lie on the branch $f_0 \overset{*}{\rightarrow} s_0$. We may assume that p_3 is normal, since some path whose start vertex is an ancestor of s_3 and whose finish vertex is f_3 must be normal, and we may select this path as p_3. Such a normal path p_3 may be chosen so that $s_3 \neq s_0$. Otherwise G is not triconnected, since (s_0, f_0) is a biarticulation point pair in G. Then path p_3 must be joined by an ILINK to some path p in S as in the case above.

Let w be the highest numbered ancestor of s_3 which lies on p_1. Let $p_4 : w \overset{*}{\Rightarrow} f_4$ be the path whose first vertex is w and whose first edge leads to an ancestor of s_3. Then p_3 and p_4 are joined by an ELINK or are identical. If $f_4 \geq s_1$ then we may add p_4 to S and apply extension method 1 for the next step (Figure 10.5(a)). If $f_1 < f_4 < s_1$ then p_4 and p_1 are joined by an ELINK and we are done (Figure 10.5(b)). If $f_4 = f_1$ then we add p_3 and p_4 to S and shift to extension method 2 for the next step (Figure 10.5(c)).

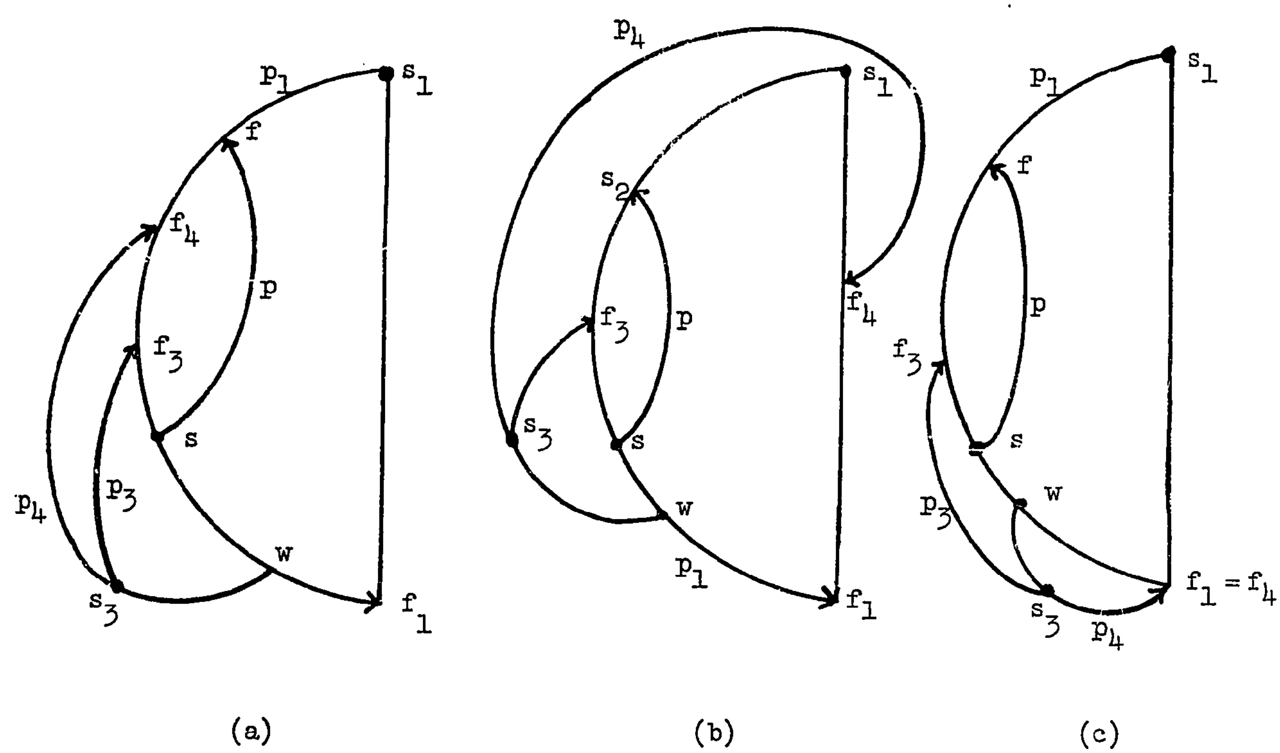


Figure 10.5: Extension of set via an ILINK with a normal path.

(a) Path p_3 normal, p_4 normal.

(b) Path p_4 linked to p_1.

(c) Path p_4 special.

Extension method 2:

We know how to extend a collection of normal paths. Suppose we add a special path to the set S. We use a variation of the method above to continue the extension process. Let $v > f_1$ be the second lowest endpoint of a path in S. Let $W = \{w' \mid \exists u(v \rightarrow u \overset{*}{\rightarrow} w'$ & u is on a path in $S \cup \{p_1\}\}$. Then there is some generated path $p_3: s_3 \overset{*}{\Rightarrow} f_3$ from a point in W to a point in $V - W - \{v, f_1\}$. Such a path must terminate on the branch $f_1 \overset{*}{\rightarrow} v$. (The point v will always be on the path p_1.) We may assume that p_3 is normal, since some normal path has finish vertex f_3 and has an ancestor of s_3 as its start vertex and we may choose this path as p_3.

Let w be the first ancestor of s_3 lying on one of the paths in S or on p_1. If vertex w is on p_1, path p_3 will either be connected with one of the normal paths in S, applying Lemma 10.3, or with one of the special paths in S, applying Lemma 10.2. In either case, p_3 is connected to p_2 in D (see Figure 10.6(a),(b)). Vertex w cannot be on one of the normal paths in S. If w is on one of the special paths p in S, then p_3 is connected by an ELINK to p as illustrated in Figure 10.6(c), and thus p_3 is connected to p_2 in D.

If $f_3 < s_1$, (p_1, p_3) is an ELINK in D and the lemma holds. If $f_3 \geq s_1$, we may add p_3 to D and apply extension method 2 again.

Extension methods 1 and 2 enable us to indefinitely enlarge the set S of paths connected to p_2 in D. Since there are only

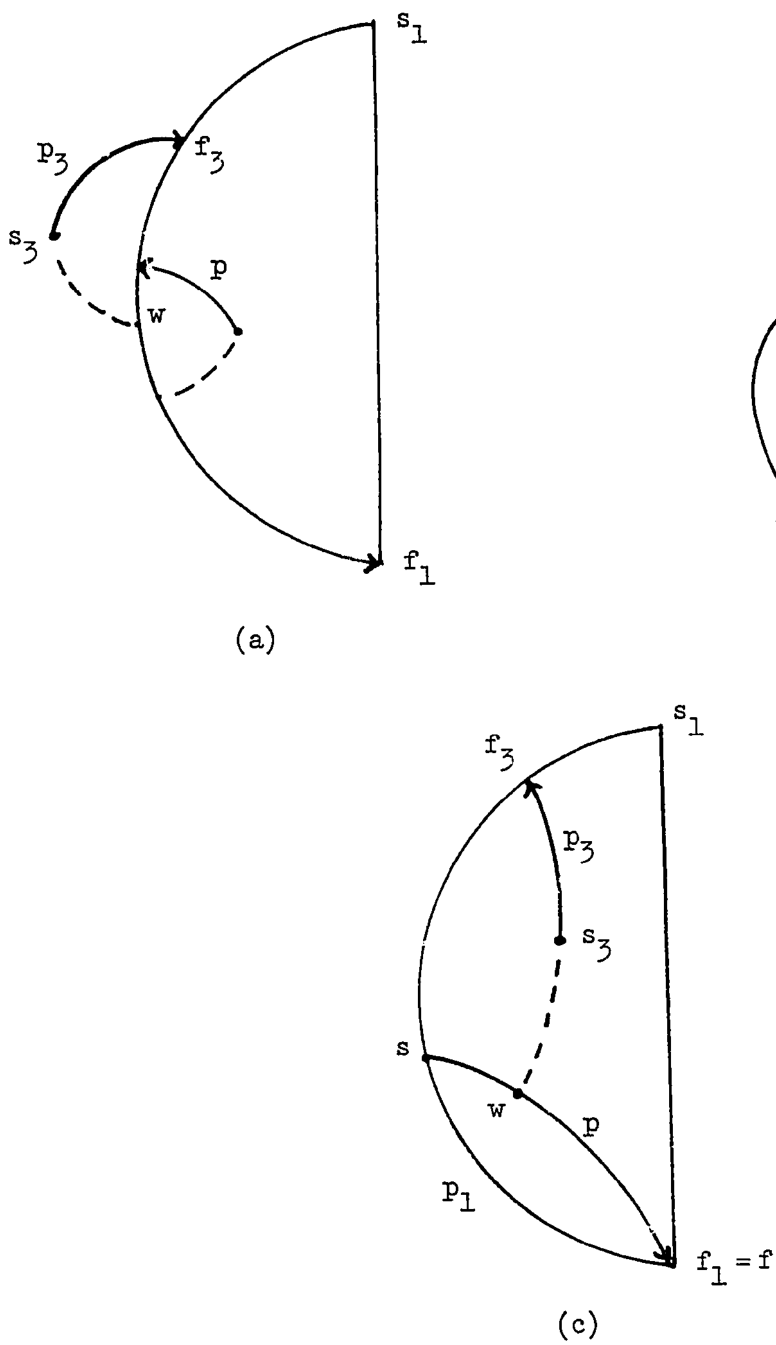


Figure 10.6: Extension of set containing a special path.

(a) Connection with normal path $p < p_3$.

(b) Connection with special path $p < p_3$.

(c) Connection via an ELINK with a special path.

a finite number of paths in the graph G, the process must stop. This can only happen when a connection between p_1 and some path in S is discovered. But then p_1 and p_2 are in the same connected component of D. This completes the proof.

Lemma 10.6: Let G be a triconnected graph. Suppose the pathfinding algorithm is applied to G, giving a set of paths. Let p_1 and p_2 be two paths whose start vertices lie on the initial cycle c. Then p_1 and p_2 lie in the same connected component of D, the dependency graph of G.

Proof: Figure 10.7 illustrates the possible interrelationships between paths p_1, p_2, and c. We use the extension methods described in the proof of Lemma 10.5 to give a set S of paths connected to one of the paths p_1 or p_2, enlarging the set until a connection in D between p_1 and p_2 is found.

In Figure 10.7(b),(e) paths p_1 and p_2 are directly linked in D. In Figure 10.7(a) we may extend the set $\{p_2\}$ using extension method 2 until a connection with p_1 is formed. In Figure 10.7(c) we may extend the set $\{p_2\}$ using extension method 1 until either a connection with p_1 is found or Figure 10.7(a) is created; this case we have already handled. In Figure 10.7(d) we may extend the set $\{p_2\}$ using extension method 1, until we either find a connection with p_2 or we create Figure 10.7(a) (already discussed). In Figure 10.7(f) we may extend the set $\{p_2\}$ using extension method 1 until we get a link with p_1. In Figure 10.7(g) we may extend the set $\{p_2\}$ using extension method 1 until we get a connection with p_1 or we

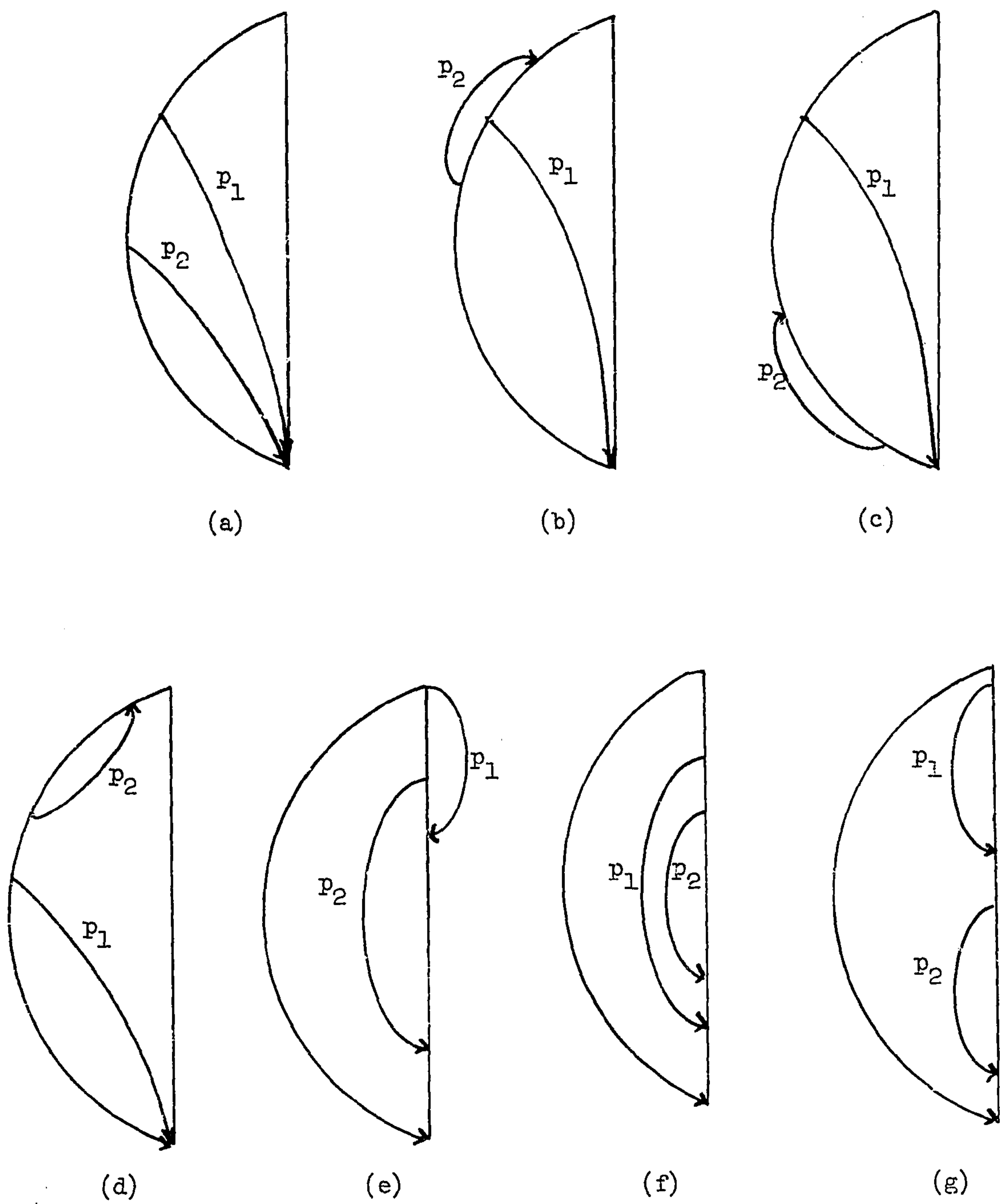


Figure 10.7 : Two paths starting on initial cycle.

(a) Paths p_1, p_2 special.

(b), (c), (d) Path p_1 special, path p_2 normal.

(e), (f), (g) Paths p_1, p_2 normal.

produce Figure 10.7(c) or 10.7(f) (both already handled). Thus p_1 and p_2 are connected in D .

Now we can prove our main result, giving a relationship between the triconnectivity of a graph G and the connectivity of its dependency graph D .

<u>Theorem 10.7</u>: Let G be a biconnected graph with four or more vertices. Suppose the pathfinding process is applied to G to give a set of paths. Let D be the corresponding dependency graph of G . Then G is triconnected if and only if G has no vertices of degree two and D consists of exactly two connected components.

<u>Proof</u>: Suppose G is triconnected. Then G must have no vertices of degree two. Examine D . The initial cycle forms a connected component of D ; it is connected to no other paths. Any two paths with start vertices on the initial cycle are in the same connected component of D by Lemma 10.6. Further, if $p: s \overset{*}{\Rightarrow} f$ is a path whose start vertex s is not on the initial cycle, then p is connected in D to the earliest path containing s , by Lemma 10.5. An induction argument shows that p is connected in D to some path with start vertex on the initial cycle. Thus all paths except the initial cycle form the second and last connected component of D .

Conversely, suppose G is not triconnected. Assume further that G does not have a vertex of degree two and that removal of vertices a and b disconnects vertices v and w in G . When a and b are removed, G falls into several connected pieces. Let R be the piece containing vertex v . We may assume without loss of generality that the first edge of the initial cycle generated by the pathfinding process does not lie in R . Add the edge (a,b) to R to form a new graph G_1 and add the edge (a,b) to G-R to form a (multi-) graph G_2 . The construction is illustrated in Figure 10.8.

It is easy to see that both G_1 and G_2 must be biconnected. Then the pathfinding process may be applied to graphs G_1 and G_2 to give a set of paths identical to those in G , with one exception. The first path found in G which has an edge in G_1 will become two paths, one being the initial cycle c_1 in G_1 and the other being a path in G_2 containing the edge (a,b) . Since both G_1 and G_2 have at least one vertex of degree 3, at least two paths are generated in each graph. Thus if D_1 is the dependency graph of G_1 , it will have at least two connected components (one being the initial cycle c_1). If D_2 is the dependency graph of G_2 it will also have at least two connected components. The dependency graph D of G must then have at least three connected components, because D is isomorphic to $D_1 \cup D_2 - \{c_1\}$. This completes the proof.

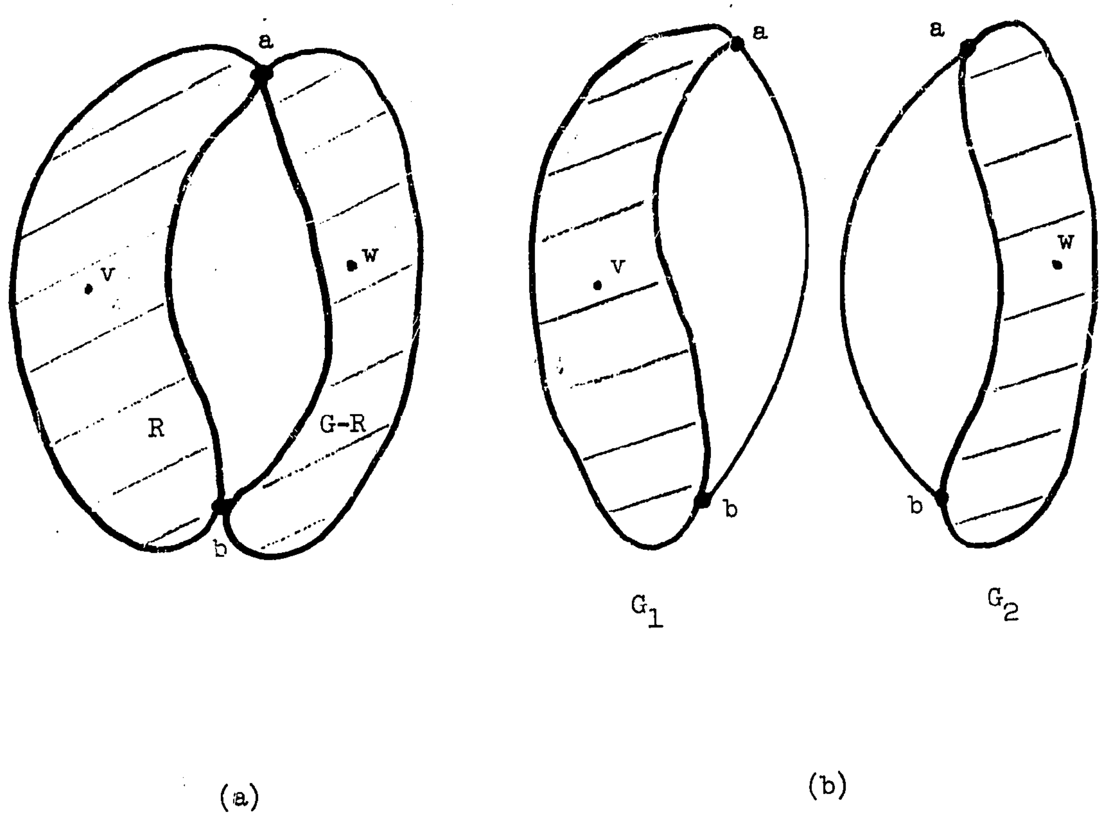


Figure 10.8: Analysis of the dependency graph of a non-triconnected graph.

(a) The original graph G.

(b) Transformation into two graphs G_1 and G_2 .

11. Constructing the Dependency Subgraph

Since the pathfinding algorithm generates $E-V+1$ paths in a biconnected graph G with V vertices and E edges, the graph D of dependencies between these paths may contain as many as $(E-V+1)(E-V)/2$ edges. If the entire planarity algorithm is to have a linear time bound, the number of dependencies must be restricted somehow. We are interested in coloring the dependency graph using two colors. If a two-colorable graph is connected, it has a unique coloring with two colors. This fact suggests that we construct only a subgraph of the entire dependency graph. If this subgraph has the same connected components as D, and if any 2-coloring of D exists, then the possible 2-colorings of the two graphs are identical. We can thus generate a single 2-coloring of the subgraph and test this coloring to discover if it is a 2-coloring of the entire dependency graph. Such a test requires only $O(V)$ time, as we shall see.

Hence our objective is to construct a subgraph D_S of the dependency graph D such that D_S has $O(V)$ edges and D_S has the same connected components as D. This is not so easy, and a detailed yet intuitive description of the process is hard to present. The basic idea is to keep track of groups of paths connected together by various types of links. Each group of paths is represented by a single path. These group representatives are stored on stacks and each new path discovered during pathfinding is compared with the top paths on the stacks to discover whether any new links should be constructed.

Four stacks are used to store paths. One stack (ASTACK) contains all the paths with an edge on the branch leading to the current vertex being examined during the pathfinding search. The other three stacks contain paths, each of which represents a connected component in the dependency subgraph of the paths found so far. Three stacks are used because three types of links in D are handled separately. A path on INSTACK represents a group of normal paths connected by ILINK's. A path on ISSTACK represents a group of normal paths connected together via ILINK's with special paths and normal paths. Only normal paths are placed on ISSTACK and INSTACK. A path on ESTACK represents a group of paths connected together by ELINK's. Procedure PATHFINDER, modified to construct the dependency subgraph as it finds paths, appears below.

```
procedure PATHFINDER(v);
  for w in the adjacency list of v do
    if v → w then
      begin
        if s0 = 0 then
          begin
            s0 := v;
          end;
        PATHFINDER(w);
        delete from ASTACK, ESTACK all paths p1 with s(p1) ≥ v;
        delete from INSTACK, ISSTACK all paths p1 with f(p1) ≥ v;
IH:     while (s(HIGHPATH(v)) > s(top of INSTACK)) and
          (v < s(top of INSTACK)) and
            (HIGHPATH(v) < top of INSTACK) do
              begin
                construct ILINK between HIGHPATH(v) and
                  top of INSTACK;
                delete top path on INSTACK;
              end;
```

```
            restore last path (if any) deleted from INSTACK by IH;
            HIGHPATH(v) := 0;
        end
    else comment v --→ w;
        begin
            p := p+1;
            s(p) := s0;
            f(p) := w;
            s0 := 0;
            add top path on ASTACK to ESTACK;
            if f(top of ASTACK) ≠ w do
                begin
E1:                 while w < s(top of ESTACK)
                        begin
                            construct ELINK between p and top of ESTACK;
                            delete top path on ESTACK;
                        end;
                    restore last path (if any) removed from ESTACK by E1;
IN:                 while f(p) ≤ f(top of INSTACK) and
                        s(p) < s(top of INSTACK) do
                            begin
                                construct ILINK between p and top of INSTACK;
                                delete top path on INSTACK;
                            end;
FIX:                while f(p) < f(top of ISSTACK) and
                        s(p) < s(top of ISSTACK) do delete top path
                        on ISSTACK;
                    add p to INSTACK;
                    add p to ISSTACK;
                    if s(p) > s(HIGHPATH(w)) then HIGHPATH(w) := p;
                end
            else
                begin comment p is special;
IS:                 while f(p) < f(top of ISSTACK) and
                        s(p) < s(top of ISSTACK) and
                        s(top of stack) ≤ s(p) + RANGE(s(p) do
```

```
                    begin
                        construct ILINK between p and top of ISSTACK;
                        delete top path on ISSTACK;
                    end
                restore last path (if any) deleted from ISSTACK by IS;
            end;
        if s(p) < v then add p to ASTACK;
    end;
```

Definition 11.1: Let G be a biconnected graph. Let D be the dependency graph corresponding to a set of paths in G generated by the pathfinding algorithm. The subgraph D_S of D which is constructed by the dependency construction algorithm given above is called the dependency subgraph D_S .

Since procedure PATHFINDER has suddenly become reasonably complicated, a few observations may be useful. Paths are numbered from 1 to E-V+1 as they are generated. The only information about a path p which is necessary to the algorithm is the start vertex s(p) and the finish vertex f(p) of the path. If v is a vertex, RANGE(v) is the number of descendants of v in the tree $\vec{T}$ of the generated palm tree. The descendants of v are all the vertices w such that $v \leq w \leq v + \text{RANGE}(v)$. The calculation of RANGE(v) is easy and may be done during the first depth-first search; we have omitted the calculation for simplicity. If $v \to w$ and w is an ancestor of the vertex currently being examined by the search procedure. HIGHPATH(v) is the normal path p with the highest start vertex s(p) such that $w \leq s(p) \leq w + \text{RANGE}(w)$ and p has finish vertex $f(p) = v$. HIGHPATH(v) depends not only upon v but upon w . However, since HIGHPATH(v,w_1) and HIGHPATH(v,w_2) are never used at the same time a single variable may be used to store both.

Consider ASTACK and ESTACK. If path p_1 occurs above path p_2 on one of these stacks, $f(p_2) \leq f(p_1)$ and $s(p_2) < s(p_1)$. Paths on INSTACK and ISSTACK are always in order according to the value of their finish vertices, highest on top. If two paths on one of these stacks have the same finish vertex, the one with the larger start vertex is lower. It is easy to verify these properties.

Statements IH and IN construct ILINK's between normal paths. Statement IS constructs ILINK's between normal and special paths. Statement E1 constructs ELINK's. Statement FIX keeps the paths on ISSTACK in the order described above. The tests indicated in these statements implement the criteria for path dependence described in Chapter 10.

Theorem 11.1: Let G be a biconnected graph and let D_S be a dependency subgraph constructed for G based upon some set of generated paths. Let D be the complete dependency graph of the same set of paths. Then D_S is a subgraph of D and the connected components of D_S and D are identical with respect to the vertices they contain.

Proof: It is easy to verify that D_S is a subgraph of D ; this follows from the fact that each link constructed in statements IH , IN , IS , and E1 is indeed a link in D . The second part of the theorem is a little more troublesome. We must show that given any link between paths in D , there is a sequence of links joining the two paths in D_S . The three types of links in D are illustrated in Figure 11.1; the next three lemmas give the proofs for these cases.

<u>Lemma 11.2</u>: Let p_0: $s_0 \overset{*}{\Rightarrow} f_0$ and p: $s \overset{*}{\Rightarrow} f$ be two paths generated by the pathfinding algorithm such that p_0 is found before p and (p_0,p) is an ELINK in the dependency graph D . Then p_0 and p are connected in the dependency subgraph D_S generated by PATHFINDER.

<u>Proof</u>: We know that $s_0 \to v \overset{*}{\to} s$, where v is the second vertex on path p_0 . The branch $s_0 \overset{*}{\to} s$ contains edges from several paths. Let these paths be $p_0, p_1, \ldots, p_n$ in the order their edges appear along $s_0 \overset{*}{\to} s$. Let $p_{n+1} = p$. If p_0 and p are joined by an ELINK in D , (p_i,p) is an ELINK in D for all $1 \leq i \leq n$, since p is normal and $s(p_i) > s_0$ for all $1 \leq i \leq n$. When path p_{i+1} is discovered, path p_i is placed on ESTACK. If an ELINK between p_i and p_{i+1} is not immediately created by statement Y , then p_{i+1} is placed just above p_i on ESTACK, since the next path placed on ESTACK is p_{i+1} . Path p_i may subsequently be removed from ESTACK only if p_i becomes linked to p_{i+1} via an ELINK in D_S . Consider the situation when p is discovered. Path p_n is placed on top of ESTACK. Let $k = \min\{i \mid p_i \text{ is on ESTACK when p is found}\}$. Then p_0 must be connected to p_k in D_S . This follows by induction from the observation above. But an ELINK between p and all paths p_i on ESTACK will be constructed by statement Y when p is found, and this includes p_k . Thus a connection between p and p_0 exists in D_S . This verifies the lemma.

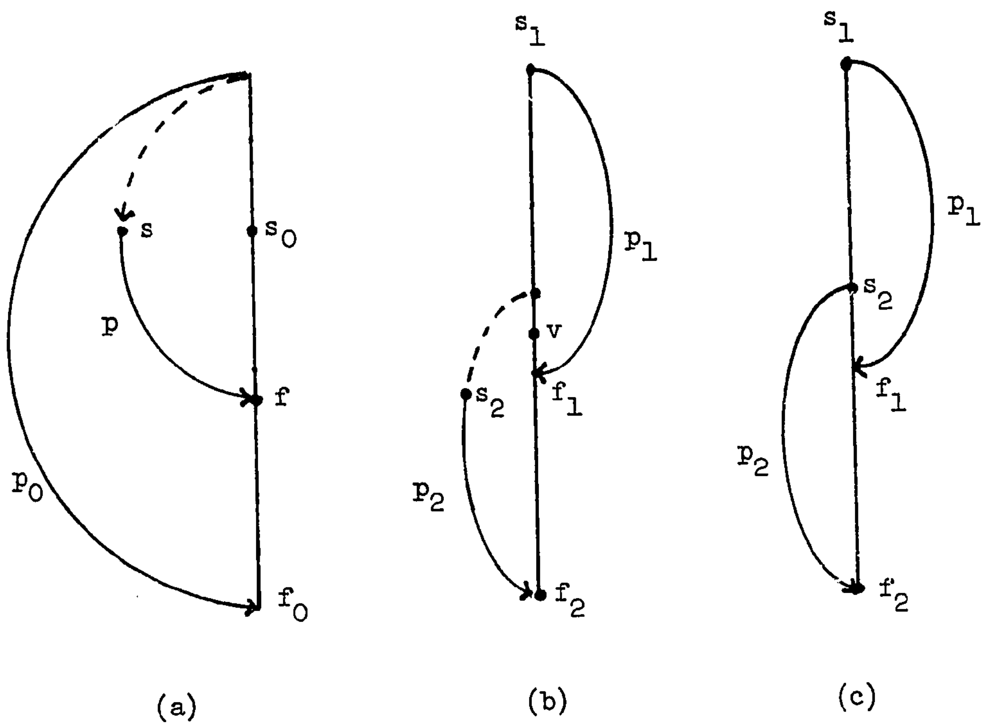


Figure 11.1: Links in D.

(a) ELINK. Path p is normal.

(b) ILINK. Paths p_1, p_2 are normal.

(c) ILINK. Path p_1 is normal, path p_2 special.

Lemma 11.3: Let $p_1: s_1 \overset{*}{\Rightarrow} f_1$ and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be two normal paths generated when the pathfinding algorithm is applied to G, such that (p_1, p_2) is an ILINK in the dependency graph D of G. Then p_1 and p_2 are connected in the dependency subgraph D_S.

Proof: Without loss of generality we may assume that $p_1 < p_2$. The proof of this lemma is complicated. We shall use induction on the number of the path p_2. The base of the induction as well as the induction step will follow from the argument below. Thus suppose that the lemma holds if $p_2 < k$. Let $p_2 = k$. Let v be the second vertex on $f_1 \overset{*}{\to} s_1$. We may assume that s_2 occurs on the branch $f_1 \overset{*}{\to} s_1$, since the links in D resulting from Lemma 10.3 are redundant.

We shall consider what happens between the time path p_1 is discovered and the time vertex f_1 is reexamined during the search. We shall assume that p_1 and p_2 do not become connected in D_S during this period. (If they do become connected, the lemma is true for $p_2 = k$.) We shall pay close attention to two paths. One, called p_3, occurs on INSTACK when p_2 is discovered and is connected to p_1 in D_S. The other, called p_4, occurs on INSTACK when vertex f_1 is reexamined and is connected to p_2 in D_S.

When p_1 is discovered it is placed on INSTACK. We may prove by induction on the path number that when p_2 is found, there is a path p_3 on INSTACK such that p_3 is connected to p_1 in D_S, $s_2 \overset{*}{\to} s(p_3)$, and $f(p_3) \leq f_1$. This follows from an examination of statements IH and IS. If $f(p_3) > f_2$, then $s(p_3) > s_2$ by

Lemma 8.6, and an ILINK between p_2 and p_3 will be formed by IN when p_2 is discovered. (This fact is easy to prove.) Hence we may assume that $f(p_3) \leq f_2$.

Path p_2 is also placed on INSTACK when it is discovered. We may prove by induction on the path number that when vertex f_1 is reexamined during the search, there is a path p_4 on INSTACK such that p_4 is connected to p_2 in D_S , $v \leq s(p_4) < s_1$ and $f(p_4) \leq f_2$. Thus let p be a path on INSTACK such that $v \leq s(p) < s_1$, $f(p) \leq f_2$, and p is connected to p_2 in D_S .

Any path p' found between the time p is found and the time f_1 is reexamined satisfies $v \leq s(p') < s_1$. Thus if p is removed from INSTACK by statement IN during this time, p becomes connected to a path p' on INSTACK with $v \leq s(p') < s_1$ and $f(p') < f(p) \leq f_2$.

Suppose path p is removed from INSTACK by statement IH . Path p will be connected in D_S to HIGHPATH(w) , for some vertex $w \geq v$. If $s(\text{HIGHPATH}(w)) \leq s_1$, then p must be connected in D_S to some path p' which remains on INSTACK and which satisfies $v \leq s(p') < s_1$ and $f(p') \leq f(p) \leq f_2$. If $s(\text{HIGHPATH}(w)) > s_1$, then $f(\text{HIGHPATH}(w))$ must lie on the branch $f_1 \overset{*}{\rightarrow} s_2$. If path p_1 is found after HIGHPATH(w) , p_1 and HIGHPATH(w) are connected in D_S by the induction hypothesis. If path p_1 is found before HIGHPATH(w) , then HIGHPATH(w) must start at a descendant of s_1 by Lemma 8.8. Path p_1 must still be on INSTACK when HIGHPATH(w) is added, since all vertices examined between the time p_1 is found and the time HIGHPATH(w) is found are descendants of s_1 . Thus both p_1 and HIGHPATH(w) must be connected to p_3 in D_S, and p_1 and p_2 are connected in D_S.

It follows by induction from the previous paragraphs that when f_1 is reexamined, there is a path p_4 on INSTACK such that p_4 is connected to p_2 in D_S, $v \leq s(p_4) < s_1$ and $f(p_4) \leq f_2$. When f_1 is again reached during the search, an ILINK will be constructed between HIGHPATH(f_1) and path p_4 by statement IH, since $s(\text{HIGHPATH}(f_1)) \geq s_1$. (This fact is easy to prove.) Thus we need only show that p_3 and HIGHPATH(f_1) are connected in D_S, since p_2, p_4, and HIGHPATH(f_1) are connected in D_S, and p_1 and p_3 are connected in D_S.

If $\text{HIGHPATH}(f_1) = p_1$ the result is immediate. Assume HIGHPATH(f_1) is found after p_1. Then $s(\text{HIGHPATH}(f_1))$ is a descendant of s_1 by Lemma 8.8. Path p_1 must still be on INSTACK when HIGHPATH(f_1) is found and added to INSTACK, since all vertices examined between the time p_1 is found and the time HIGHPATH(f_1) is found are descendants of s_1. Thus p_1 and HIGHPATH(f_1) are both connected to p_3 in D_S and the lemma holds.

Suppose that HIGHPATH(f_1) is found before p_1. If HIGHPATH(f_1) is found after p_3, then HIGHPATH(f_1) must be placed on INSTACK while p_3 is lower on INSTACK, and p_1, p_3, and HIGHPATH(f_1) must all be connected in D_S. Thus we may further assume that HIGHPATH(f_1) is found before p_3.

We have two more cases. If $s(p_3) \geq s(\text{HIGHPATH}(f_1))$, $s(p_3)$ is a descendant of $s(\text{HIGHPATH}(f_1))$ by Lemma 8.8. This means that $f_1 \leq f(p_3)$ by Lemma 8.6, which is a contradiction. If $s(p_3) < s(\text{HIGHPATH}(f_1))$, then $(\text{HIGHPATH}(f_1), p_3)$ is an ILINK in D, and HIGHPATH(f_1) and p_3 are connected in D_S by the induction hypothesis. Therefore in any case p_3 and HIGHPATH(f_1) are connected in D_S, which means that p_1 and p_2 are connected in D_S. This completes the proof of both the base of the induction and the induction step, and the lemma is true in general.

Lemma 11.4: Let $p_1: s_1 \overset{*}{\Rightarrow} f_1$ be a normal path and $p_2: s_2 \overset{*}{\Rightarrow} f_2$ be a special path generated by the pathfinding algorithm. Suppose that (p_1, p_2) is an ILINK in the complete dependency graph D. Then p_1 and p_2 are joined by a sequence of links in the dependency subgraph D_S.

Proof: We know that $p_1 < p_2$ by the definition of this type of ILINK (Lemma 10.2). Any path which starts at a descendant of s_2 must finish at a vertex not smaller than f_2, since the first path through s_2 finishes at f_2. Any path which starts at a descendant of s_2 and which finishes at f_2 must be special for the same reason. When p_1 is discovered it is placed on ISSTACK. If p_1 is removed from ISSTACK before p_2 is found, p_1 will be linked in D_S to some other path on ISSTACK with finish vertex greater than f_2 and start vertex greater than s_2, as an examination of statements IS and FIX shows. (If FIX removes paths from ISSTACK, the next path added to ISSTACK is linked in D_S to the removed paths, by Lemma 11.3.) When p_2 is discovered, an ILINK will be formed by statement IS between p_2 and all paths on top of ISSTACK with a finish vertex greater than f_2, including the path on ISSTACK to which p_1 is connected in D_S. The lemma follows.

The proof of Theorem 11.1 is immediate from the three lemmas above, because all the possible links in D have been considered.

Theorem 11.5: If G is a biconnected planar graph with V vertices and E edges, the number of edges in any dependency subgraph D_S of G is bounded by 9V .

Proof: The number of paths (ignoring the initial cycle) is E-V . Let N be the number of normal paths and let S be the number of special paths. Every time a path is found, a path may be added to ESTACK. Every time more than one ELINK is formed by statement E1 , a path is removed from ESTACK. Thus the number of ELINKs in D_S is bounded by $2(E-V) \leq 4V$. Each normal path is added to INSTACK once and to ISSTACK once. Each time a vertex is re-examined during the search one ILINK may be formed by statement IH without deleting any paths from INSTACK. Each time a special path is found an ILINK may be formed by statement IS without deleting any paths from ISSTACK. Thus the number of ILINKs formed is bounded by $2N+V+S \leq 2(E-V)+V \leq 5V$. Thus the total number of links in D_S is bounded by 9V .

Theorems 11.1 and 11.5 imply that the dependency subgraph D_S has exactly the necessary properties. Now we are almost done; we must still examine the algorithm used to check a coloring of D , and we must prove the converse of Theorem 10.4. We attend to these matters in the next chapter.

12. Coloring the Dependency Subgraph

After the dependency subgraph D_S is constructed by the pathfinding algorithm, it must still be colored using two colors. This is accomplished very simply using a depth-first search. A path is chosen and colored arbitrarily, either "left" or "right". Each time a new path is reached by traversing a link in D_S, the path is colored according to the color of the path at the other end of the link and the type of link. Each time a link between two paths already colored is traversed, the colors of the paths are checked to see if they are consistent with the type of the link. One search on each connected component of D_S will produce a coloring of D_S if such a coloring exists. A program for this purpose is presented below.

```
begin
   procedure PATHMARKER(v);
      for w in the adjacency list of v in D_S do
         if w is not yet colored then
            begin
               if (v,w) is an ELINK then COLOR(w) := COLOR(v);
               else COLOR(w) := -COLOR(v);
               PATHMARKER(w);
            end
         else if ((v,w) is an ILINK and COLOR(v) = COLOR(w))
            or ((v,w) is an ELINK and COLOR(v) ≠ COLOR(w))
               then go to nonplanarexit;
   for w a vertex in D_S if w is not yet colored then
      begin
         COLOR(w) := 1;
         PATHMARKER(w);
      end;
end;
```

If the dependency graph D_S is not colorable using two colors, then the original graph is not planar. However, the converse is not necessarily true. Given a coloring of D_S , we must discover if this coloring satisfies the constraints of the entire dependency graph D . Our test for this property uses four stacks; ALEFT, ILEFT, ARIGHT, and IRIGHT.

Imagine repeating the pathfinding process, now knowing which embedding the paths will be given as they are found. Consider a path p which is colored "left". We compare this path with the path p_1 on top of ARIGHT, which is a previously found path with the right embedding. If p and p_1 are joined by an ELINK in D , then D is not colorable using two colors. We also compare p with the path p_2 on top of ILEFT. Path p_2 is a previously found path with the left embedding. If (p,p_2) is an ILINK in D , then D is not colorable using two colors. Having performed these tests, we place p on top of ALEFT and ILEFT if it is normal, and on top of only ALEFT if it is special. Path p is treated similarly if it is colored "right".

This process is carried out for each path in the order that the paths were found. Stacks ALEFT and ARIGHT are continuously updated so that they contain only paths with edges on the tree branch leading to the start vertex of the next path. Stacks ILEFT and IRIGHT are continuously updated so that they contain only paths whose finish vertex is a proper ancestor of the start vertex of the next path. A program for the color checking process appears below.

```
procedure COLORCHECK;
   for i := 1 until E-V+1 do
      begin
         delete from ALEFT, ARIGHT all paths p with s(p) ≥ s(i);
         delete from ILEFT, IRIGHT all paths p with f(p) ≥ s(i);
         if COLOR(i) = 1 then
            begin
               if i is normal then
                  begin
                     if f(i) < s(top of ARIGHT) then go to
                        nonplanarexit;
                     if f(i) < f(top of ILEFT) and
                        s(i) < s(top of ILEFT) then go to
                        nonplanarexit;
                     put i on top of ALEFT, ILEFT;
                  end
               else comment i is special;
                  begin
                     if f(i) < f(top of ILEFT) and
                        s(i) < s(top of ILEFT) and
                        s(i) + RANGE(s(i)) ≥ s(top of ILEFT) then
                        go to nonplanarexit;
                     put i on top of ALEFT;
                  end;
            end
         else if i is normal then
            begin
               if f(i) < s(top of ALEFT) then go to nonplanarexit;
               if f(i) < f(top of IRIGHT) and
                  s(i) < s(top of IRIGHT) then
                  go to nonplanarexit;
               put i on top of ARIGHT, IRIGHT;
            end
         else if i is normal then
```

```
begin
   if f(i) < s(top of ALEFT) then go to nonplanarexit;
   if f(i) < f(top of IRIGHT) and
      s(i) < s(top of IRIGHT) then
      go to nonplanarexit;
   put i on top of ARIGHT, IRIGHT;
end
else comment i is special;
begin
   if f(i) < f(top of IRIGHT) and
      s(i) < s(top of IRIGHT) and
      s(i) + RANGE(s(i)) ≥ s(top of ILEFT) then
      go to nonplanarexit;
   put i on top of ARIGHT;
end;
end;
```

Theorem 12.1: Let G be a biconnected graph with complete dependency graph D and dependency subgraph D_S. If D is colorable using two colors, then any coloring of D_S will pass the test given by COLORCHECK. Conversely, if D is not colorable using two colors, then any coloring of D_S will fail the test given by COLORCHECK.

Proof: By Theorem 11.1, D_S and D have the same connected components. If D is colorable using two colors, then the possible two-colorings of D_S are exactly the same as the possible two-colorings of D. Thus any two-coloring of D_S must pass the test given by COLORCHECK, since COLORCHECK merely verifies that colors are consistent across certain links of D.

Conversely, suppose D is not colorable using two colors. Suppose a coloring of D_S is given. Then two paths p_1 and p_2

must be colored compatibly in D_S but incompatibly in D . There are two cases; p_1 and p_2 may be colored the same or they may be colored differently.

Suppose (p_1,p_2) is an ELINK in D and that p_1 and p_2 are colored differently. Without loss of generality we may assume that p_1 is found before p_2 , that p_1 is colored "left", and that p_2 is colored "right". When p_1 is found it is placed on ALEFT. Path p_1 will still be on ALEFT when p_2 is found. By the proof of Lemma 11.2, p_2 will be joined by an ELINK in D to all paths above and including p_1 on ALEFT. Thus the color check will fail when p_2 is tested.

Suppose (p_1,p_2) is an ILINK in D and that p_1 and p_2 are colored the same. Without loss of generality we may assume that p_1 is found before p_2 and that p_1 and p_2 are colored "left". We prove by induction on the number of paths p_2 that the color check fails. The base step and the induction step follow from the argument below. Thus suppose that the color check fails if $p_2 < k$. Let $p_2 = k$. We may assume that $s(p_2)$ lies on the branch $f(p_1) \overset{*}{\rightarrow} s(p_1)$, since the links in D resulting from Lemma 10.3 are redundant.

Path p_1 is on ILEFT when path p_2 is found. Consider the path p on top of ILEFT when p_2 is tested. Path p must have $s(p_2) \overset{*}{\rightarrow} s(p)$. If $f(p) > f(p_2)$ then $s(p) > s(p_2)$ by Lemma 8.6 and (p,p_2) is an ILINK in D . Thus the color check will fail, since p_2 is colored "left".

Hence we may assume that $f(p) \leq f(p_2)$. If $s(p) < s(p_1)$ then (p,p_1) is an ILINK in D and the color check will fail by the induction hypothesis. If $s(p) \geq s(p_1)$ then $s(p)$ is a descendant of $s(p_1)$ by Lemma 8.8. This is impossible since $f(p) \leq f(p_2) < f(p_1)$ and p was found after p_1. Thus in any case the color check fails. By induction the color check fails for all p_2. Therefore the theorem is true.

Theorem 12.2: Let G be a biconnected graph with a dependency graph D. If the vertices of D (the paths found in G) may be colored with two colors consistently with the links in D, then G is planar.

Proof: Suppose a coloring of D with the colors "left" and "right" is given. Consider building an embedding of G in the plane one path at a time in the order the paths were found, using the left embedding as defined in Chapter 9 if the path is colored "left" and the right embedding if the path is colored "right". We shall show that the embedding may be completed satisfactorily to give a planar embedding of the entire graph G.

Suppose to the contrary that some path $p: s \overset{*}{\Rightarrow} f$ may not be added to the embedding without crossing some other path. Without loss of generality we may assume that p is colored "left". Suppose p is a normal path. Path p must cross some edge (v,w) either entering or leaving the branch $f \overset{*}{\rightarrow} s$. Suppose (v,w) is on a path p_1 and leaves the branch $f \overset{*}{\rightarrow} s$ on the left as in Figure 12.1. Path p_1 is found before path p. Thus there is some path p_2 which starts at v and proceeds up the branch $v \overset{*}{\rightarrow} s$. Since p is normal, p and p_2 must be connected by an

ELINK in D . But p_2 is found after p_1 and thus p_2 cannot have the left embedding, since the edge (v,w) is to the left of the first edge of p_2 (see Figure 12.1). This contradiction shows that no such edge (v,w) exists.

Suppose some edge (v,w) on path p_1 enters the branch $f \overset{*}{\to} s$ on the left as illustrated in Figure 12.2. We may assume that p_1 is normal, since if p_1 is special some normal path whose start vertex is an ancestor of $s(p_1)$ must have finish vertex w and must enter on the left of the branch $f \overset{*}{\to} s$. (See Lemma 9.4.) Vertex $s(p_1)$ cannot lie on the branch $f \overset{*}{\to} s$ by Lemma 8.6. Thus $s(p_1) > s(p)$. But then (p,p_1) is an ILINK in D . This is impossible because p and p_1 have the same color.

Thus every normal path may be successfully embedded. Suppose $p\colon s \overset{*}{\Rightarrow} f$ is a special path whose embedding is blocked. Let $p_0\colon s_0 \to v \overset{*}{\Rightarrow} f_0$ be the normal path with highest start vertex such that $f \overset{*}{\to} s_0 \to v \overset{*}{\to} s$. If $f = 0$, let p_0 be the initial cycle. Such a path p_0 must exist since G is biconnected. Without loss of generality we may assume that both p and p_0 are embedded on the left. We know that no path blocked the placement of p_0 . Path p may only be blocked by a path p_1 starting from a descendant of s and finishing at a vertex on the branch $f \overset{*}{\to} s$ as illustrated in Figure 12.3. We may assume that p_1 is normal, since some normal path whose start vertex is a descendant of s must terminate at $f(p_1)$ on the same side of the branch $1 \overset{*}{\to} s$ as p_1 . Path p_1 must have the left embedding. Further, $s(p_1) \neq s$, since if $s(p_1) = s$, p_1 would have $f(p_1) \leq f$.

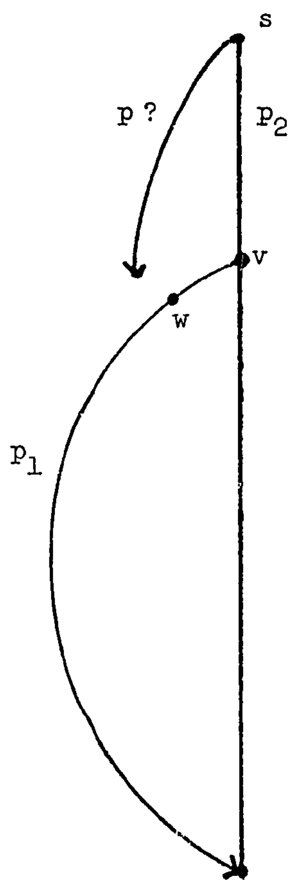


Figure 12.1: Blockage of a normal path p: $s \Rightarrow^* f$ by a path p_1 leaving $f \rightarrow^* s$.

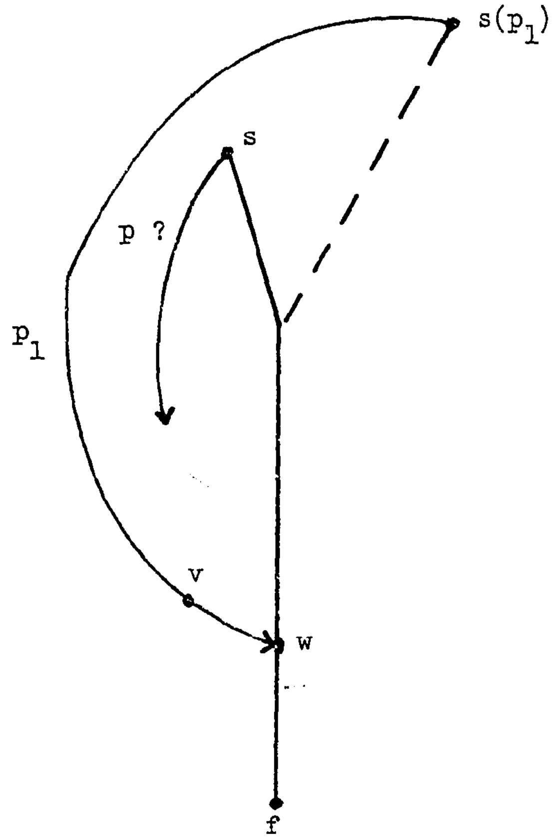


Figure 12.2: Blockage of a normal path p: s $\Rightarrow^*$ f by a path p_1 entering f $\rightarrow^*$ s.

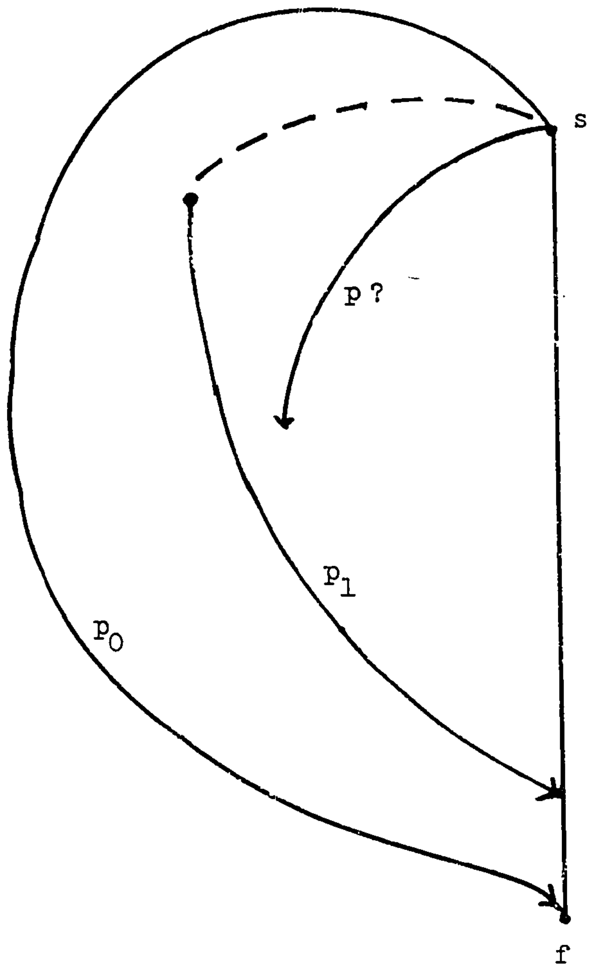


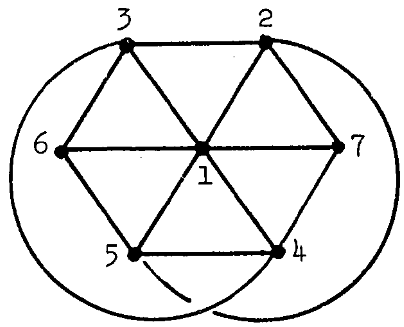
Figure 12.3: Blockage of a special path p: s $\Rightarrow^*$ f by a path entering f $\rightarrow^*$ s.

But (p,p_1) must be an ILINK in D, which is impossible since p and p_1 have the same color. Thus the placement of a special path cannot be blocked, and the entire graph G may be embedded in the plane.

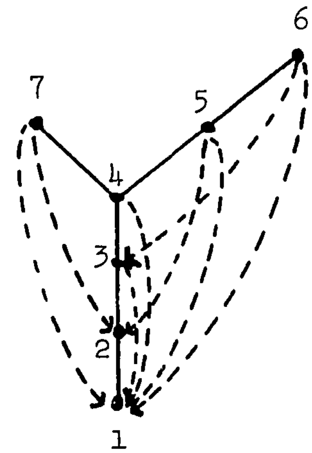
<u>Theorem 12.3</u>: Let G be a biconnected graph. Let D_S be a dependency subgraph constructed for G. Testing a two-coloring of D_S using the color checking algorithm COLORCHECK gives a necessary and sufficient condition for the planarity of G. This algorithm requires $O(V)$ time and space, if G has V vertices and E edges.

<u>Proof</u>: The correctness of the planarity test follows from Theorem 12.1, Theorem 12.2, and all the previous results. It is easy to verify that the entire algorithm requires $O(V)$ time and space, since $E \leq 3V - 6$ in a planar graph. A little extra work will show that the planarity algorithm works correctly even if the graph is not first divided into biconnected components.

With this result, we have come to the end of the line. For further enlightenment, Figure 12.4 illustrates an application of the planarity algorithm.



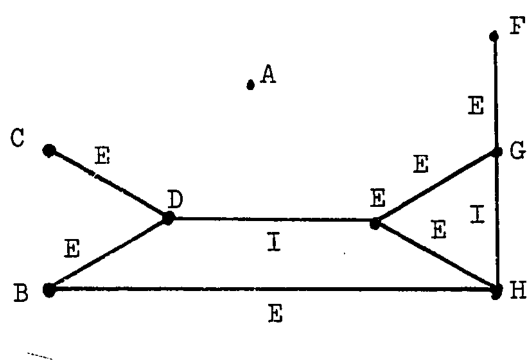
(a)



(b)

A: (1,2,3,1)
B: (3,4,1)
C: (4,7,1)
D: (7,2)
E: (4,5,1)
F: (5,6,1)
G: (6,3)
H: (5,2)

(c)



(d)

Figure 12.4: Application of the planarity algorithm.
(a) Nonplanar graph
(b) Generated palm tree
(c) Paths
(d) Dependency subgraph (not 2-colorable)

IV. From Alpha to Omega

13. Implementation and Experiments

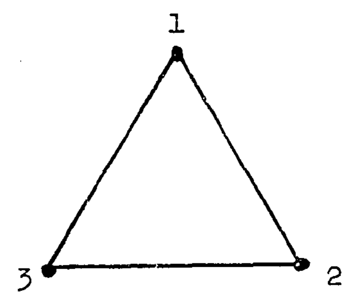
The connectivity, biconnectivity, and planarity algorithms were programmed in Algol W, the Stanford University version of Algol [Sit 71], and run on an IBM 360/67. Program listings appear in the appendix. The programs were extensively tested. The planarity algorithm was applied to a group of planar and nonplanar graphs to verify that the implementation was correct. The algorithm was also applied to a series of randomly generated complete planar graphs, in order to determine the experimental running time.

The test graphs were generated by starting with a complete graph of three vertices (Figure 13.1(a)). At each step, a triangular face of the graph was selected at random and split into three new triangular faces by adding one vertex and three edges, as in Figure 13.1(b). A graph of this type has the property that $V = 3E - 6$; no new edge may be added without destroying the planarity of the graph. Although not all complete planar graphs can be generated by dividing triangular faces in this way (see Figure 13.2 for instance), the test graphs seemed to give the planarity program a satisfactory workout.

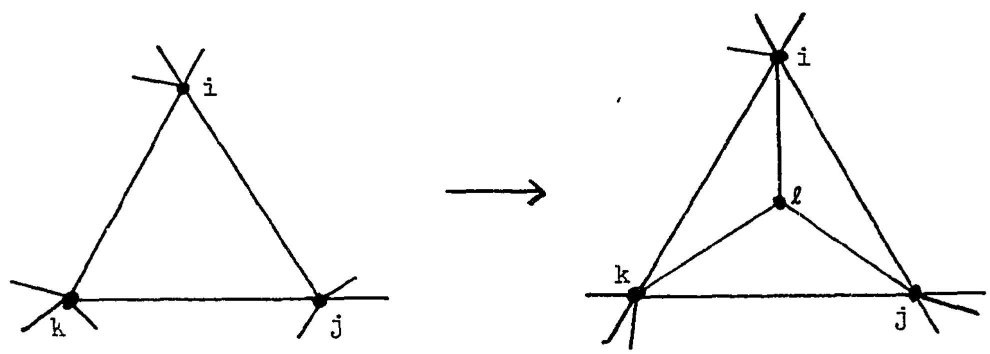
The test results are given in Figure 13.3 and plotted in Figure 13.4. A least squares fit gave:

$$(1) \qquad T = .0125V - .07$$

where T is the time in seconds and V is the number of edges in the graph. The program indeed requires time linear in the number of vertices of the graph. The data may be summarized in another way: the program will analyze a graph at the rate of 80 vertices/second (or faster, if $E < 3V - 6$). Non-planar graphs generally require less time than planar



(a)

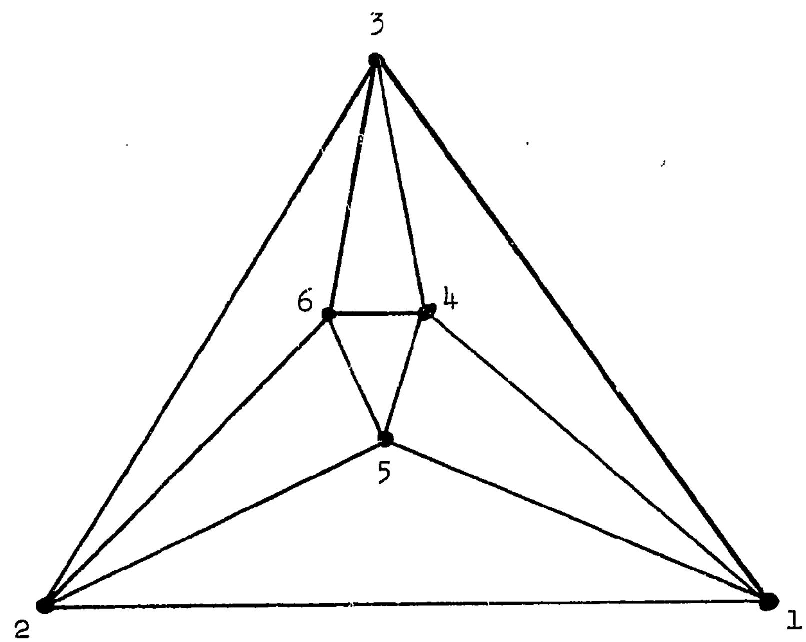


(b)

Figure 13.1: Construction of random complete planar graphs.

(a) Initial graph.

(b) Addition of a vertex by splitting a randomly selected face.



<u>Figure 13.2</u>: A complete planar graph which cannot be generated by the process in Figure 13.1.

V (vertices)	E (edges)	Time to determine planarity (seconds)
20	54	0.22
40	114	0.46
60	174	0.72
80	234	0.97
100	294	1.23
150	444	1.60
200	594	2.58
250	744	3.03
300	894	3.87
400	1194	4.62
500	1494	6.07
600	1794	7.25
700	2094	9.02
800	2394	10.28
900	2694	10.95

Figure 13.3: Results of running the planarity program on randomly generated complete planar graphs $(E = 3V-6)$.

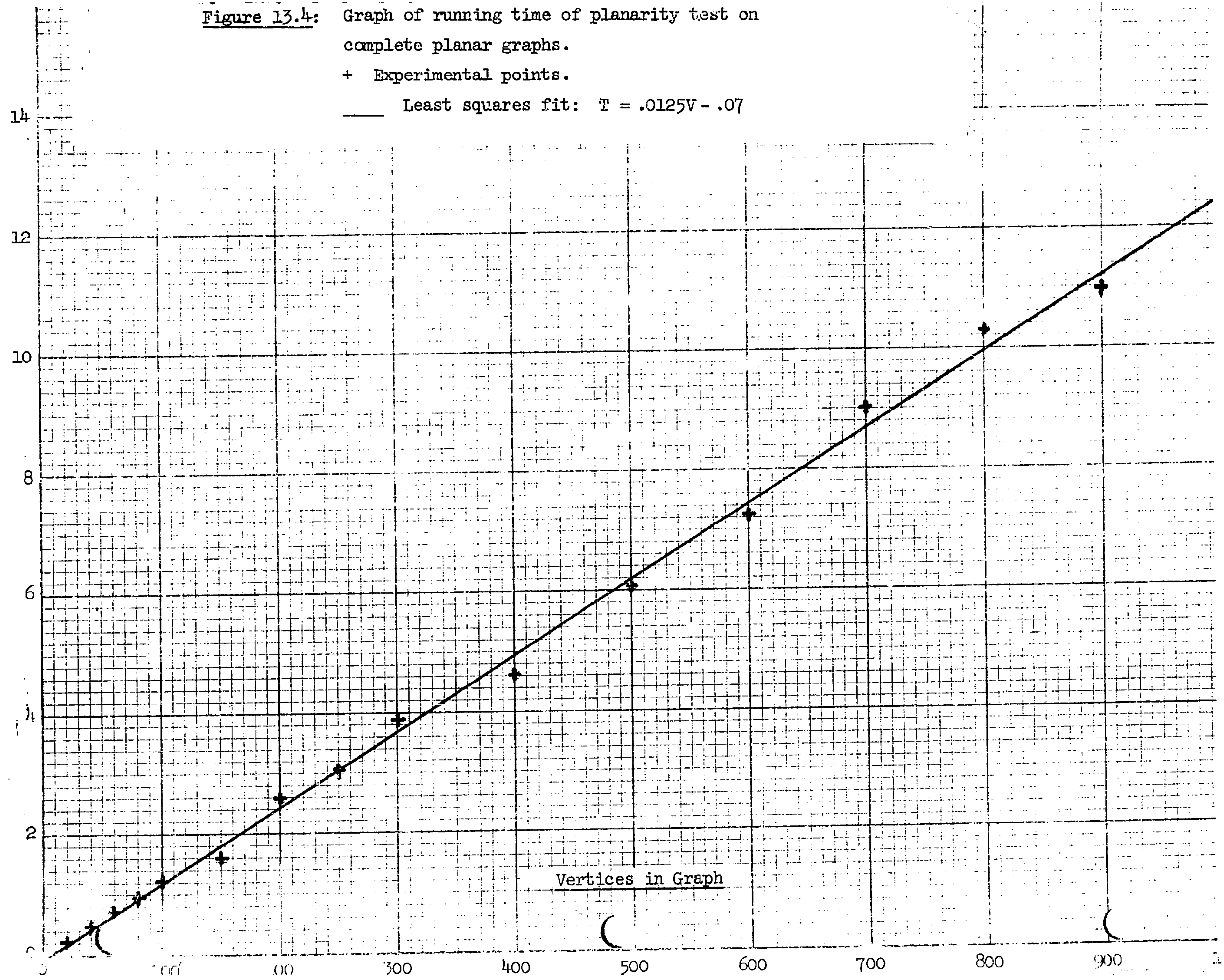


Figure 13.4: Graph of running time of planarity test on complete planar graphs.

\+ Experimental points.

____ Least squares fit: $T = .0125V - .07$

ones, since the algorithm halts as soon as the graph is found to be non-planar. The planarity program was space-limited rather than time-limited; a 1000 vertex, 2994 edge graph could not be analyzed in the space available (417,792 bytes) although no more than 12.5 seconds would be required for processing such a graph. No special care was taken in conserving storage space; careful reprogramming or use of auxiliary storage devices would allow much larger graphs to be analyzed.

It is difficult to compare the experimental running times of different algorithms, since implementations and machines vary greatly. However, an algorithm devised by Bruno, Steiglitz, and Weinberg [Bru 70] required about 30 seconds to process the 28 vertex planar graph in Figure 13.5, using an IBM 360/65. The algorithm presented here required 0.4 seconds to construct a planar representation of the same graph. The time discrepancy would be much greater on larger graphs. The experimental results were quite satisfactory, and they demonstrate that the planarity algorithm presented here is of significant practical as well as theoretical value.

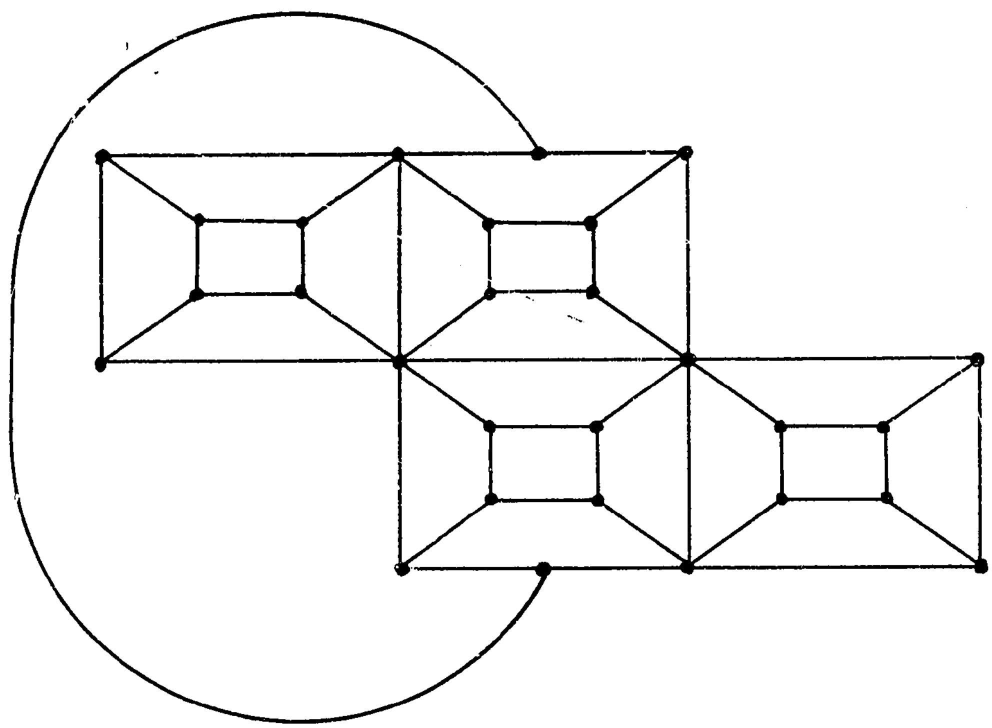


Figure 13.5: Graph analyzed using the algorithm of Bruno, et. al., and using the depth-first search method.

14. Conclusions

The depth-first search process has applications beyond those presented here. For instance, Theorem 10.7 demonstrates a relationship between the triconnectivity of a graph G and the connected components of any dependency graph D of G. Using this result it is easy to discover in O(V,E) space and time whether a graph is triconnected. Given a graph G, a dependency subgraph D_S for G is constructed. The number of connected components of D_S is found, and Theorems 10.7 and 11.1 are applied to resolve the question. An elaboration of this procedure gives an algorithm for dividing a graph into triconnected components, using O(V,E) time and space. Such an algorithm will be described in detail in a future paper.

Hopcroft [Hop 71a] has presented an algorithm for determining whether two triconnected planar graphs are isomorphic. His algorithm requires O(V log V) time. Combining this algorithm with the connectivity, biconnectivity, triconnectivity, and planarity algorithms, it is possible to construct an algorithm which determines in O(V) space and O(V log V) time whether two arbitrary planar graphs are isomorphic [Hop 71b]. This algorithm may be modified to enumerate all planar graphs of various kinds, or to construct canonical representations of planar graphs. The planar isomorphism algorithm promises to be of great value to chemists, since most molecules may be represented as planar graphs. A canonical form for molecules, which follows from the isomorphism algorithm, may greatly speed searches of the chemical literature.

We have so far considered only properties of undirected graphs. However, directed graphs may also be explored in a depth-first manner.

The structure which results, called a jungle, is more complicated than a palm tree, but it is still very useful. For example, the strongly connected components of a directed graph may be discovered in $O(V,E)$ time using depth-first search [Tar 71].

Depth-first search has been widely used by researchers in artificial intelligence and combinatorics. The algorithms presented here demonstrate the value of this technique as a systematic method of analyzing graphs.

V. Bibliography

[Aus 61] Auslander, L., Parter, S. V. "On Imbedding Graphs in the Sphere." Journal of Mathematics and Mechanics. Vol. 10, No. 3 (May, 1961), 517-523.

[Ber 64] Berge, C. The Theory of Graphs and its Applications. Translated by Alison Doig; rev. ed. London: Methuen and Co., Ltd., 1964.

[Bru 70] Bruno, J., Steiglitz, K., Weinberg, L. "A New Planarity Test Based on 3-Connectivity." I.E.E.E. Transactions on Circuit Theory. Vol. CT-17, No. 2 (May, 1970), 197-206.

[Bus 65] Busacker, R., Saaty, T. Finite Graphs and Networks: An Introduction with Applications. New York: McGraw-Hill, 1965.

[Chu 70] Chung, S. H., Roe, P. H. "Algorithms for Testing the Planarity of a Graph." Proceedings of the Thirteenth Midwest Symposium on Circuit Theory. University of Minnesota, Minneapolis, Minnesota (May 7-8, 1970), VII.4.1 - VII.4.12.

[Coo 71] Cook, S. "Linear Time Simulation of Deterministic Two-Way Pushdown Automata." IFIP Congress 71: Foundations of Information Processing. Ljubljana, Yugoslavia (August, 1971). Amsterdam: North Holland Publishing Co., 174-179.

[Fis 66] Fisher, G. J. "Computer Recognition and Extraction of Planar Graphs from the Incidence Matrix." I.E.E.E. Transactions on Circuit Theory. Vol. CT-13, No. 2 (June, 1966), 154-163.

[Gold 63] Goldstein, A. J. "An Efficient and Constructive Algorithm for Testing Whether a Graph Can Be Embedded in a Plane." Graph and Combinatorics Conference. Office of Naval Res. Logistics Proj., Contract No. NONR 1858-(21), Dept. of Math., Princeton University (May 16-18, 1963), 2 unno. pp.

[Gol 65] Golumb, S. W., Baumert, L. D. "Backtrack Programming." JACM 12, 4 (Oct., 1965), 516-524.

[Hal 55] Hall, D. W., Spencer, G. Elementary Topology. New York: Wiley, 1955.

[Har 69] Harary, F. Graph Theory. Reading, Massachusetts: Addison-Wesley, 1969.

[Hol 70] Holt, R. C., Reingold, E. M. "On the Time Required to Detect Cycles and Connectivity in Directed Graphs." Technical Report No. 70-63, Department of Computer Science, Cornell University (June, 1970).

[Hop 71a] Hopcroft, J. "An n log n Algorithm for Isomorphism of Planar Triply Connected Graphs." Technical Report No. 192, Computer Science Department, Stanford University (January, 1971).

[Hop 71b] Hopcroft, J., Tarjan, R. "A V^2 Algorithm for Determining Isomorphism of Planar Graphs." Information Processing Letters. 1 (1971), 32-34.

[Hop 71c] Hopcroft, J., Tarjan, R. "Planarity Testing in V log V Steps: Extended Abstract." IFIP Congress 71: Foundations of Information Processing. Ljubljana, Yugoslavia (August, 1971). Amsterdam: North Holland Publishing Co., 18-22.

[Hop 71d] Hopcroft, J., Tarjan, R. "Efficient Algorithms for Graph Manipulation." Technical Report No. 207, Computer Science Department, Stanford University (March, 1971).

[Kur 30] Kuratowski, C. "Sur le Probleme des Corbes Gauches en Topologie." Fundamenta Mathematicae. Vol. 15 (1930), 271-283.

[Led 65] Lederberg, J. "DENDRAL-64: A System for Computer Construction Enumeration, and Notation of Organic Molecules as Tree Structures and Cyclic Graphs, Part II: Topology of Cyclic Graphs." Interim Report to the National Aeronautics and Space Administration, Grant NsG 81-60, NASA CR 68898, STAR No. N-66-14074 (December 15, 1965).

[Lem 67] Lempel, A., Even, S., Cederbaum, I. "An Algorithm for Planarity Testing of Graphs." in P. Rosensteihl, ed., Theory of Graphs: International Symposium: Rome, July, 1966. New York: Gordon and Breach, 1967, 215-232.

[Mei 70] Mei, P., Gibbs, N. "A Planarity Algorithm Based on the Kuratowski Theorem." AFIPS Conference Proceedings, Volume 36, 1970, Spring Joint Computer Conference. Atlantic City, New Jersey (May 5-7, 1970), 91-95.

[Mon 71] Mondshein, L. "Combinatorial Orderings and Embedding of Graphs." Technical Note 1971-35, Lincoln Laboratory, Massachusetts Institute of Technology (August, 1971).

[Nil 71] Nilsson, N. Problem-Solving Methods in Artificial Intelligence. New York: McGraw-Hill, 1971.

[Ore 62] Ore, O. Theory of Graphs. American Mathematical Society Colloquium Pub., Vol. 38. Providence, Rhode Island: Amer. Math. Soc., 1962.

[Pat 71] Paton, K. "An Algorithm for the Blocks and Cutnodes of a Graph." Communications of the ACM. Vol. 14, No. 7 (July, 1971), 468-475.

[Shi 69] Shirey, R. W. "Implementation and Analysis of Efficient Graph Planarity Testing Algorithms." Ph.D. Thesis, University of Wisconsin (June, 1969).

[Sit 71] Sites, R. L. "Algol W Reference Manual." Technical Report No. 230, Computer Science Department, Stanford University (August, 1971).

[Tar 69] Tarjan, R. "Implementation of an Efficient Algorithm for Planarity Testing of Graphs." (December, 1969), unpublished.

[Tar 71] Tarjan, R. "Depth-First Search and Linear Graph Algorithms." Conference Record: Twelfth Annual Symposium on Switching and Automata Theory. (October 13-15, 1971), IEEE Computer Society, 114-119.

[Thr 53] Thron, W. J. Introduction to the Theory of Functions of a Complex Variable. New York: Wiley, 1953.

[Tut 63] Tutte, W. T. "How to Draw a Graph." Proceedings of the London Mathematical Society. Series 3, Vol. 13 (1963), 743-768.

[Tut 66] Tutte, W. T. Connectivity in Graphs. London: Oxford University Press, 1966.

[Wei 65a] Weinberg, L. "Plane Representations and Codes for Planar Graphs." Proceedings: Third Annual Allerton Conference on Circuit and System Theory. University of Illinois, Allerton House, Monticello, Illinois (Oct. 20-22, 1965), 733-744.

[Wei 65b] Weinberg, L. "Algorithms for Determining Isomorphism Groups for Planar Graphs." Proceedings: Third Annual Allerton Conference on Circuit and System Theory. University of Illinois, Allerton House, Monticello, Illinois (Oct. 20-22, 1965), 913-929.

[Wei 66] Weinberg, L. "A Simple and Efficient Algorithm for Determining Isomorphism of Planar Triply Connected Graphs." I.E.E.E. Transactions on Circuit Theory. Vol. CT-13, No. 2 (June, 1966), 142-148.

[Win 66] Wing, O. "On Drawing a Planar Graph." I.E.E.E. Transactions on Circuit Theory. Vol. CT-13, No. 1 (March, 1966), 112-114.

[You 63] Youngs, J. W. T. "Minimal Imbeddings and the Genus of a Graph." Journal of Mathematics and Mechanics. Vol. 12, No. 2 (1963), 305-315.

VI. Appendix: Program Listings

This section contains listings of the procedures needed to build the connectivity and biconnectivity algorithms, and the listing of a complete implementation of the planarity algorithm. The programs are written in Algol W. The reader may notice some differences between the programs here and the procedures discussed in the text; these are mostly a matter of convenience. Further, the comments occurring in the programs may not be completely lucid. The reader is strongly urged to implement the algorithms himself, but if he is lazy, the planarity program accepts data in the following form:

"problem name"	(a character string identifying the problem)
V	(the number of vertices in the graph)
E	(the number of edges in the graph)
V1 V2 V3 V4 - - V2E-1 V2E	(pairs of integers denoting the endpoints of the edges of the graph)

This sequence may be repeated for each graph to be analyzed.

Utility procedures for CONNECT and BICONNECT


```
PROCEDURE ADD2(INTEGER VALUE A,B;INTEGER ARRAY STACK(*);
    INTEGER VALUE RESULT PTR);
  BEGIN
    COMMENT **********************************************
    *       PROCEDURE TO ADD VALUES A, B TO STACK "STACK" AND
    *       INCREASE STACK POINTER "PTR" BY 2.
    **************************************************;
    PTR:=PTR+2;
    STACK(PTR-1):=A;
    STACK(PTR):=B
  END;
```

```
PROCEDURE NEXTLINK(INTEGER VALUE POINT,VAL);
  BEGIN
    COMMENT **********************************************
    *       PROCEDURE TO ADD DIRECTED EDGE (POINT,VAL) TO
    *       STRUCTURAL REPRESENTATION OF A GRAPH.
    *
    *       GLOBAL VARIABLES:
    *         HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
    *           REPRESENTATION OF THE GRAPH.
    *         FREENEXT: CURRENT LAST ENTRY IN NEXT ARRAY.
    **************************************************;
    FREENEXT:=FREENEXT+1;
    NEXT(FREENEXT):=NEXT(POINT);
    NEXT(POINT):=FREENEXT;
    HEAD(FREENEXT):=VAL
  END;
```

```
INTEGER PROCEDURE MIN(INTEGER VALUE A,B);
  COMMENT **********************************************
  *       PROCEDURE TO COMPUTE THE MINIMUM OF TWO INTEGERS.
  **************************************************;
  IF A<B THEN A ELSE B;
```

Recursive connectivity procedure

```
PROCEDURE CONNECT(INTEGER VALUE V,E; INTEGER RESULT CPTR;
    INTEGER ARRAY EDGELIST,COMPONENTS(*));
  BEGIN
   COMMENT *************************************************************
   *      PROCEDURE TO FIND THE CONNECTED COMPONENTS OF A
   *      GRAPH.
   *
   *      PARAMETERS:
   *        V,E: INPUT NUMBER OF VERTICES AND EDGES OF THE
   *          GRAPH.
   *        EDGELIST(1::2*E): INPUT LIST OF EDGES OF GRAPH.
   *        COMPONENTS(1::3*E): OUTPUT LIST OF EDGES OF
   *          COMPONENTS FOUND.  EACH COMPONENT IS PRECEDED BY
   *          AN ENTRY GIVING THE NUMBER OF EDGES OF THE
   *          COMPONENT.
   *        CPTR: OUTPUT POINTER TO LAST ENTRY IN COMPONENTS.
   *
   *      GLOBAL VARIABLES:
   *        HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
   *          REPRESENTATION OF THE GRAPH (UNDIRECTED, NO
   *          CROSS-LINKS).
   *        FREENEXT: LAST ENTRY IN NEXT ARRAY.
   *
   *      LOCAL VARIABLES:
   *        NUMBER(1::V+1): ARRAY FOR NUMBERING THE VERTICES
   *          DURING DEPTH-FIRST SEARCH.
   *        CODE: CURRENT HIGHEST VERTEX NUMBER.
   *        POINT: CURRENT POINT BEING EXAMINED DURING SEARCH.
   *        V2: NEXT POINT TO BE EXAMINED DURING SEARCH.
   *        OLDPTR: POSITION IN COMPONENTS TO PLACE E VALUE OF
   *          NEXT COMPONENT.
   *
   *      GLOBAL PROCEDURES:
   *        ADD2,NEXTLINK.
   *
   *      A RECURSIVE DEPTH-FIRST SEARCH PROCEDURE IS USED TO
   *      EXAMINE CONNECTED COMPONENTS OF THE GRAPH.
   **************************************************************;
   INTEGER ARRAY NUMBER(1::V+1);
   INTEGER CODE,POINT,V2,OLDPTR;
   PROCEDURE CONNECTOR(INTEGER VALUE POINT, OLDPT);
     COMMENT *********************************************************
     *      RECURSIVE PROCEDURE TO FIND A CONNECTED COMPONENT,
     *      USING DEPTH-FIRST SEARCH.
     *
     *      PARAMETERS:
     *        POINT: STARTPOINT OF SEARCH.
     *        OLDPT: PREVIOUS STARTPOINT.
     *
     *      GLOBAL VARIABLES:
     *        SEE CONNECT FOR DESCRIPTION.
     *
```

```
*       GLOBAL PROCEDURES:
*          ADD2.
*
*       EXAMINE EACH EDGE OUT OF POINT.
*************************************************************;
WHILE NEXT(POINT)>0 DO
  BEGIN
    COMMENT ************************************************
    *       V2 IS HEAD OF EDGE.  DELETE EDGE FROM
    *       STRUCTURAL REPRESENTATION.
    *********************************************************;
    V2:=HEAD(NEXT(POINT));
    NEXT(POINT):=NEXT(NEXT(POINT));
    COMMENT ************************************************
    *       HAS THE EDGE BEEN SEARCHED IN THE OTHER
    *       DIRECTION?  IF SO, LOOK FOR ANOTHER EDGE.
    *********************************************************;
    IF (NUMBER(V2)<NUMBER(POINT))AND(V2¬=OLDPT) THEN
      BEGIN
        COMMENT *******************************************
        *       ADD EDGE TO COMPONENT.
        ***************************************************;
        ADD2(POINT,V2,COMPONENTS,CPTR);
        COMMENT *******************************************
        *       HAS A NEW POINT BEEN FOUND?
        ***************************************************;
        IF NUMBER(V2)=0 THEN
          BEGIN
            COMMENT ***************************************
            *       NEW POINT FOUND.  NUMBER IT.
            **********************************************;
            NUMBER(V2):=CODE:=CODE+1;
            COMMENT ***************************************
            *       INITIATE A DEPTH-FIRST SEARCH FROM THE
            *       NEW POINT.
            **********************************************;
            CONNECTOR(V2,POINT);
          END
      END;
  END;
COMMENT ***********************************************
*       CONSTRUCT THE STRUCTURAL REPRESENTATION OF THE
*       GRAPH.
******************************************************;
FREENEXT:=V;
```

```
FOR I:=1 UNTIL V DO NEXT(I):=0;
FOR I:=1 UNTIL E DO
  BEGIN
    COMMENT ******************************************
    *     EACH EDGE OCCURS TWICE, ONCE FOR EACH
    *     ENDPOINT.
    *********************************************;
    NEXTLINK(EDGELIST(2*I-1),EDGELIST(2*I));
    NEXTLINK(EDGELIST(2*I),EDGELIST(2*I-1));
  END;
COMMENT *********************************************
*     INITIALIZE VARIABLES FOR SEARCH.
***************************************************;
CPTR:=0;
POINT:=1;
FOR I:=1 UNTIL V+1 DO NUMBER(I):=0;
WHILE POINT<=V DO
  BEGIN
    COMMENT *****************************************
    *     EACH EXECUTION OF CONNECTOR SEARCHES A
    *     CONNECTED COMPONENT.  AFTER EACH SEARCH,
    *     FIND AN UNNUMBERED VERTEX AND SEARCH AGAIN.
    *     REPEAT UNTIL ALL VERTICES ARE INVESTIGATED.
    ***********************************************;
    NUMBER(POINT):=CODE:=1;
    OLDPTR:=CPTR:=CPTR+1;
    CONNECTOR(POINT,0);
    COMMENT ******************************************
    *     COMPUTE NUMBER OF EDGES OF COMPONENT.
    ***********************************************;
    COMPONENTS(OLDPTR):=(CPTR-OLDPTR)DIV 2;
    WHILE NUMBER(POINT)¬=0 DO POINT:=POINT+1;
  END
END;
```

Recursive biconnectivity procedure

```
PROCEDURE BICONNECT(INTEGER VALUE V,E; INTEGER RESULT BPTR;
    INTEGER ARRAY EDGELIST,BICOMPONENTS(*));
  BEGIN
    COMMENT ****************************************************
    *      PROCEDURE TO FIND THE BICONNECTED COMPONENTS OF A
    *      GRAPH.
    *
    *      PARAMETERS:
    *        V,E: INPUT NUMBER OF VERTICES AND EDGES OF THE
    *          GRAPH.
    *        EDGELIST(1::2*E): INPUT LIST OF EDGES OF GRAPH.
    *        BICOMPONENTS(1::3*E): OUTPUT LIST OF EDGES OF
    *          COMPONENTS FOUND.  EACH COMPONENT IS PRECEDED BY
    *          AN ENTRY GIVING THE NUMBER OF EDGES OF THE
    *          COMPONENT.
    *        BPTR: OUTPUT POINTER TO LAST ENTRY OF BICOMPONENTS.
    *
    *      GLOBAL VARIABLES:
    *        HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL REPRE-
    *          SENTATION OF THE GRAPH (UNDIRECTED, NO CROSS-
    *          LINKS).
    *        FREENEXT: LAST ENTRY IN NEXT ARRAY.
    *
    *      LOCAL VARIABLES:
    *        NUMBER(1::V+1): ARRAY FOR NUMBERING THE VERTICES
    *          DURING DEPTH-FIRST SEARCH.
    *        CODE: CURRENT HIGHEST VERTEX NUMBER.
    *        EDGESTACK(1::2*E): STORAGE FOR LIST OF EDGES
    *          EXAMINED DURING SEARCH.
    *        EPTR: POINTER TO LAST ENTRY IN EDGESTACK.
    *        POINT: CURRENT POINT BEING EXAMINED DURING SEARCH.
    *        V2: NEXT POINT TO BE EXAMINED DURING SEARCH.
    *        NEWLOWPT: LOWPOINT FOR BICONNECTED PART OF GRAPH
    *          ABOVE AND INCLUDING V2.
    *        OLDPTR: POSITION IN BICOMPONENTS TO PLACE E VALUE
    *          OF NEXT COMPONENT.
    *
    *      GLOBAL PROCEDURES:
    *        MIN,ADD2,NEXTLINK.
    *
    *      A RECURSIVE DEPTH-FIRST SEARCH PROCEDURE IS USED TO
    *      DIVIDE THE GRAPH. THE LOWEST POINT REACHABLE FROM THE
    *      CURRENT POINT WITHOUT GOING THROUGH PREVIOUSLY
    *      SEARCHED POINTS IS CALCULATED.  THIS INFORMATION
    *      ALLOWS DETERMINATION OF THE ARTICULATION POINTS AND
    *      DIVISION OF THE GRAPH.
    *****************************************************.
```

```
*       V2 IS HEAD OF THE EDGE. DELETE EDGE FROM
*       STRUCTURAL REPRESENTATION.
*********************************************************;
V2:=HEAD(NEXT(POINT));
NEXT(POINT):=NEXT(NEXT(POINT));
COMMENT *************************************************
*       HAS THE EDGE BEEN SEARCHED IN THE OTHER
*       DIRECTION?  IF SO, LOOK FOR ANOTHER EDGE.
*********************************************************;
IF (NUMBER(V2)<NUMBER(POINT))AND(V2¬=OLDPT) THEN
  BEGIN
    COMMENT *********************************************
    *       ADD EDGE TO EDGESTACK.
    *****************************************************;
    ADD2(POINT,V2,EDGESTACK,EPTR);
    COMMENT *********************************************
    *       HAS A NEW POINT BEEN FOUND?
    *****************************************************;
    IF NUMBER(V2)=0 THEN
      BEGIN
        COMMENT *****************************************
        *       NEW POINT FOUND.  NUMBER IT.
        *************************************************;
INTEGER ARRAY NUMBER(1::V+1);
INTEGER ARRAY EDGESTACK(1::2*E);
INTEGER CODE,EPTR,POINT,V2,NEWLOWPT,OLDPTR;
PROCEDURE BICONNECTOR(INTEGER VALUE RESULT POINT,OLDPT,
    LOWPOINT);
  COMMENT ***************************************************
  *       RECURSIVE PROCEDURE TO SEARCH A CONNECTED COMPONENT
  *       AND FIND ITS BICONNECTED COMPONENTS USING DEPTH-
  *       FIRST SEARCH.
  *
  *       PARAMETERS:
  *         POINT: STARTPOINT OF SEARCH, UNCHANGED DURING
  *           EXECUTION.
  *         OLDPT: PREVIOUS STARTPOINT, UNCHANGED DURING
  *           EXECUTION.
  *         LOWPOINT: OUTPUT OF LOWEST POINT REACHABLE ON A
  *           PATH FOUND DURING SEARCH FORWARD.
  *
  *       GLOBAL VARIABLES:
  *         SEE BICONNECT FOR DESCRIPTION.
  *
  *       GLOBAL PROCEDURES:
  *         MIN,ADD2.
  *
  *       EXAMINE EACH EDGE OUT OF POINT.
  ***********************************************************;
  WHILE NEXT(POINT)>0 DO
    BEGIN
      COMMENT ***********************************************
```

```
NUMBER(V2):=CODE:=CODE+1;
COMMENT ****************************************
*      INITIATE A DEPTH-FIRST SEARCH FROM THE
*      NEW POINT.
**********************************************;
NEWLOWPT:=V+1;
BICONNECTOR(V2,POINT,NEWLOWPT);
COMMENT ****************************************
*      NOTE THAT ALTHOUGH GLOBAL VARIABLE V2
*      IS CHANGED, ITS VALUE IS RESTORED UPON
*      EXIT FROM THIS PROCEDURE.  RECALCULATE
*      LOWPOINT.
**********************************************;
LOWPOINT:=MIN(LOWPOINT,NEWLOWPT);
COMMENT ****************************************
*      IS POINT AN ARTICULATION POINT OF THE
*      GRAPH?
**********************************************;
IF NEWLOWPT>=NUMBER(POINT) THEN
  BEGIN
    COMMENT ********************************
    *      POINT IS AN ARTICULATION POINT.
    *      OUTPUT EDGES OF COMPONENT FROM
    *      EDGESTACK.
    ****************************************;
    OLDPTR:=BPTR:=BPTR+1;
    WHILE NUMBER(EDGESTACK(EPTR-1))>NUMBER
        (POINT) DO
      BEGIN
        ADD2(EDGESTACK(EPTR-1),EDGESTACK
            (EPTR),BICOMPONENTS,BPTR);
        EPTR:=EPTR-2
      END;
    COMMENT ********************************
    *      ADD LAST EDGE.
    ****************************************;
    ADD2(POINT,V2,BICOMPONENTS,BPTR);

    EPTR:=EPTR-2;
    COMMENT ********************************
    *      COMPUTE NUMBER OF EDGES OF
    *      COMPONENT.
    ****************************************;
    BICOMPONENTS(OLDPTR):=(BPTR-OLDPTR) DIV 2;
  END
END
ELSE
  COMMENT ****************************************
  *   NEW POINT NOT FOUND.  RECALCULATE LOWPOINT.
  ***********************************************;
  LOWPOINT:=MIN(LOWPOINT,NUMBER(V2))
END
END;
```

```
COMMENT ********************************************************
*     CONSTRUCT THE STRUCTURAL REPRESENTATION OF THE GRAPH.
**************************************************************;
FREENEXT:=V;
FOR I:=1 UNTIL V DO NEXT(I):=0;
FOR I:=1 UNTIL E DO
  BEGIN
    COMMENT ****************************************************
    *     EACH EDGE OCCURS TWICE, ONCE FOR EACH ENDPOINT.
    **********************************************************;
    NEXTLINK(EDGELIST(2*I-1),EDGELIST(2*I));
    NEXTLINK(EDGELIST(2*I),EDGELIST(2*I-1))
  END;
COMMENT ********************************************************
*     INITIALIZE VARIABLES FOR SEARCH.
**************************************************************;
EPTR:=0;
BPTR:=0;
POINT:=1;
V2:=0;
FOR I:=1 UNTIL V+1 DO NUMBER(I):=0;
WHILE POINT<=V DO
  BEGIN
    COMMENT ****************************************************
    *     EACH EXECUTION OF BICONNECTOR SEARCHES A
    *     CONNECTED COMPONENT OF THE GRAPH.  AFTER EACH
    *     SEARCH, FIND AN UNNUMBERED VERTEX AND SEARCH
    *     AGAIN.  REPEAT UNTIL ALL VERICES ARE EXAMINED.
    **********************************************************;
    NUMBER(POINT):=CODE:=1;
    NEWLOWPT:=V+1;
    BICONNECTOR(POINT,V2,NEWLOWPT);
    WHILE NUMBER(POINT)¬=0 DO POINT:=POINT+1
  END;
END;
```

Complete program implementing the planarity algorithm

```
BEGIN
   INTEGER V,E;
   STRING(80) NAME;
       NODEXT:
      READ(NAME);
   READ(V,E);
      WRITE(NAME);
WRITE("TIME",TIME(1));
      WRITE("V=",V,"E=",E);
      BEGIN
         INTEGER FREENEXT;
         INTEGER ARRAY HEAD(V+1::V+2*E);
         INTEGER ARRAY NEXT(1::V+2*E);
         PROCEDURE NEXTLINK(INTEGER VALUE POINT,VAL);
           BEGIN
             COMMENT ***********************************************************
             *      PROCEDURE TO ADD DIRECTED EDGE (POINT,VAL) TO
             *      STRUCTURAL REPRESENTATION OF A GRAPH.
             *
             *      GLOBAL VARIABLES:
             *        HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
             *          REPRESENTATION OF THE GRAPH.
             *        FREENEXT: CURRENT LAST ENTRY IN NEXT ARRAY.
             ***********************************************************;
             FREENEXT:=FREENEXT+1;
             NEXT(FREENEXT):=NEXT(POINT);
             NEXT(POINT):=FREENEXT;
             HEAD(FREENEXT):=VAL
           END;
               COMMENT ******************************************************
               *      CONSTRUCT STRUCTURAL REPRESENTATION FOR FIRST
               *      SEARCH.
               ******************************************************;
     BEGIN
       INTEGER V1,V2;
               FREENEXT:=V;
               FOR I:=1 UNTIL V DO NEXT(I):=0;
   FOR I:= 1 UNTIL E DO
    BEGIN
       READON(V1,V2);
       NEXTLINK(V1,V2);
       NEXTLINK(V2,V1);
    END;
WRITE("TIME AFTER SET UP",TIME(1));
END;
```

```
   BEGIN
  INTEGER CUR,EDGE;
  INTEGER ARRAY PATH,NUMBER(1::V);
  INTEGER ARRAY LOWPT1,LOWPT2,RANGEP(1::V);
  INTEGER ARRAY COLOR(1::E-V+1);
  INTEGER ARRAY S,F(0::E-V+1);
  INTEGER ARRAY EDGESTACK(1::2*E);
  BOOLEAN FLAG;
  INTEGER V2,CODE,POINT,STARTPOINT,PATHNUMBER;
     INTEGER APTR,YPTR,XNPTR;
  INTEGER EPTR,STARTPATH,XSPTR;
  INTEGER ARPTR,ALPTR,XLPTR,XRPTR;
  INTEGER EDGEFREE;
  INTEGER ARRAY NEXTEDGE(1::7*E-5*V+2);
  INTEGER ARRAY HEADEDGE(E-V+1::7*E-5*V+2);
  BOOLEAN ARRAY LINKTYPE(E-V+1::7*E-5*V+2);
  BOOLEAN ARRAY NEWNODE(1::E-V+2);
  PROCEDURE ADD2(INTEGER VALUE A,B;INTEGER ARRAY STACK(*);
      INTEGER VALUE RESULT PTR);
    BEGIN
      COMMENT ***************************************************
      *     PROCEDURE TO ADD VALUES A, B TO STACK "STACK" AND
      *     INCREASE STACK POINTER "PTR" BY 2.
      *******************************************************;
      PTR:=PTR+2;
      STACK(PTR-1):=A;
      STACK(PTR):=B
    END;
   PROCEDURE EDGELINK(INTEGER VALUE A,B);
    BEGIN
      EDGEFREE:=EDGEFREE+1;
      NEXTEDGE(EDGEFREE):=NEXTEDGE(A);
      NEXTEDGE(A):=EDGEFREE;
      HEADEDGE(EDGEFREE):=B;
    END;
   INTEGER PROCEDURE MIN(INTEGER VALUE A,B);
    COMMENT ****************************************************
    *     PROCEDURE TO COMPUTE THE MINIMUM OF TWO INTEGERS.
    ********************************************************;
    IF A<B THEN A ELSE B;
   INTEGER PROCEDURE MAX(INTEGER VALUE A,B);
    IF A>B THEN A ELSE B;
   PROCEDURE ADD3(INTEGER VALUE A,B,C;INTEGER ARRAY STACK(*);
      INTEGER VALUE RESULT PTR);
    BEGIN
      PTR:=PTR+3;
      STACK(PTR-2):=A;
      STACK(PTR-1):=B;
      STACK(PTR):=C;
    END;
      PROCEDURE XLINK(INTEGER VALUE X,Y);
        BEGIN
          WRITE("XLINK");
GO TO NONPLANAREXIT;
        END;
      PROCEDURE YLINK(INTEGER VALUE X,Y);
```

```
                BEGIN
                  WRITE("YLINK");
GO TC NONPLANAREXIT;
                END;
              PROCEDURE PRESEARCH(INTEGER VALUE RESULT POINT,OLDPT);
                COMMENT ************************************************
                *     PROCEDURE TO SEARCH CONNECTED COMPONENT AND COMPUTE
                *     LOWPOINT VALUES.
                *     PARAMETERS:
                *       POINT: CURRENT POINT.
                *       OLDPT: PREVIOUSLY SEARCHED POINT.
                *     GLOBAL VARIABLES:
                *       HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
                *       REPRESENTATION OF GRAPH (UNDIRECTED, NO CROSS-
                *       LINKS).
                *     V2: NEXT POINT SEARCHED.
                *       NUMBER(1::V): CONSECUTIVE SEARCH NUMBER OF A
                *       VERTEX.
                *     CODE: HIGHEST CONSECUTIVE SEARCH NUMBER.
                *       EXTREMUM(1::V): LOWEST POINT REACHABLE THROUGH
                *       NEW EDGES FROM A GIVEN POINT.
                *     GLOBAL PROCEDURES:
                *       MIN, ADD2.
                *     THIS PROCEDURE OPERATES AS ANY OTHER DEPTH-FIRST
                *     SEARCH.
                ***********************************************************;
                WHILE NEXT(POINT)>0 DO
                  BEGIN
                    V2:=HEAD(NEXT(POINT));
                    NEXT(POINT):=NEXT(NEXT(POINT));
                    IF (NUMBER(V2)<NUMBER(POINT))AND(V2¬=OLDPT) THEN
                      BEGIN
                        ADD2(POINT,V2,EDGESTACK,EPTR);
                        IF NUMBER(V2)=0 THEN
                          BEGIN
                            NUMBER(V2):=CODE:=CODE+1;
                            PRESEARCH(V2,POINT);
                            IF LOWPT1(V2)<LOWPT1(POINT) THEN
                              BEGIN
                                LOWPT2(POINT):=MIN(LOWPT2(V2),
                                    LOWPT1(POINT));
                                LOWPT1(POINT):=LOWPT1(V2);
                              END
                            ELSE IF LOWPT1(V2)=LOWPT1(POINT) THEN
                              LOWPT2(POINT):=MIN(LOWPT2(V2),
                                LOWPT2(POINT))
                              ELSE LOWPT2(POINT):=MIN(LOWPT1(V2),
                                  LOWPT2(POINT));
                          END
                        ELSE IF NUMBER(V2)<LOWPT1(POINT) THEN
                          BEGIN
                            LOWPT2(POINT):=LOWPT1(POINT);
                            LOWPT1(POINT):=NUMBER(V2);
                          END
                        ELSE IF NUMBER(V2)>LOWPT1(POINT) THEN
                          LOWPT2(POINT):=MIN(NUMBER(V2),LOWPT2(POINT));
                      END
                  END;
```

```
PROCEDURE SECONDSEARCHER(INTEGER VALUE RESULT POINT);
  COMMENT ****************************************************
  *     PROCEDURE TO SEARCH GRAPH IN DESIRED ORDER AND
  *     RENUMBER VERTICES FOR TRICONNECTOR.  STRUCTURAL
  *     REPRESENTATION OF GRAPH IS IN DIRECTED FORM.
  *
  *     PARAMETERS:
  *       PCINT: CURRENT POINT BEING EXAMINED.
  *******************************************************;
  WHILE NEXT(POINT)>0 DO
    BEGIN
      V2:=HEAD(NEXT(POINT));
      NEXT(POINT):=NEXT(NEXT(POINT));
      IF NUMBER(V2)=0 THEN
        BEGIN
          NUMBER(V2):=CODE:=CODE+1;
          SECONDSEARCHER(V2);
    END;
      ADD2(NUMBER(POINT),NUMBER(V2),EDGESTACK,EPTR);
    END;
PROCEDURE PATHMARKER(INTEGER VALUE POINT);
  WHILE NEXTEDGE(POINT)¬=0 DO
    BEGIN
      EDGE:=NEXTEDGE(POINT);
      V2:=HEADEDGE(EDGE);
      NEXTEDGE(POINT):=NEXTEDGE(EDGE);
      IF COLOR(V2)=0 THEN
        BEGIN
          IF LINKTYPE(EDGE) THEN COLOR(V2):=
            COLOR(POINT) ELSE COLOR(V2):=3-COLOR(POINT);
WRITE("COLOR(",V2,")=",COLOR(V2));

        END
      ELSE IF (COLOR(V2)=COLOR(POINT))=¬LINKTYPE(EDGE) THEN
 BEGIN

   WRITE("CONFLICT IN PATHMARKER");

          GO TO NONPLANAR2;
END;
      IF NEWNODE(V2) THEN
        BEGIN
          NEWNODE(V2):=FALSE;
          PATHMARKER(V2);
        END;
    END;
PROCEDURE SORT;
```

```
BEGIN
  COMMENT **********************************************
   *      PROCEDURE TO SORT EDGES TO GIVE ADJACENCY
   *      STRUCTURE USED BY PATHFINDING SEARCH.
   *      LOCAL VARIABLES:
   *        NEXTSORT(1::2*V+E): LINKS FOR BUCKET SORT.
   *        SORTPT1(2*V+1::2*V+E): TAIL OF EDGE IN
   *        BUCKET.
   *        SORTPT2(2*V+1::2*V+E): HEAD OF EDGE IN
   *        BUCKET.
   *        FREESORT: LAST ENTRY IN NEXTSORT.
   *        SORTPTR: POINTER USED TO EMPTY BUCKETS AFTER
   *        SORT.
   **********************************************;
   INTEGER FREESORT,SORTPTR;
   INTEGER ARRAY NEXTSORT(1::2*V+E);
   INTEGER ARRAY SORTPT1,SORTPT2(2*V+1::2*V+E);
   COMMENT **********************************************
   *      INITIALIZE FOR SORTING EDGES ACCORDING TO
   *      LOWEST POINT REACHABLE FROM HEAD AND FOR
   *      CONSTRUCTING NEW ADJACENCY STRUCTURE.
   **********************************************;
   FREESORT:=2*V;
   FREENEXT:=2*V;
   FOR I:=1 UNTIL 2*V DO NEXTSORT(I):=0;
   COMMENT **********************************************
   *      INSERT EACH EDGE INTO A BUCKET.  EACH BUCKET
   *      IS A LIST OF EDGES.  CHOICE OF BUCKET DEPENDS
   *      FIRST ON EXTREMUM VALUE AND SECOND ON WHETHER
   *      LOWPT2 IS NONTRIVIAL.
   **********************************************;
   FOR I:=2 STEP 2 UNTIL 2*E DO
     BEGIN
       FREESORT:=FREESORT+1;
       COMMENT **********************************************
       *      PLACE ENDPOINTS OF EDGE IN BUCKET.
       **********************************************;
       SORTPT1(FREESORT):=EDGESTACK(I-1);
       V2:=SORTPT2(FREESORT):=EDGESTACK(I);
       IF NUMBER(V2)<NUMBER(SORTPT1(FREESORT)) THEN
         BEGIN
           COMMENT **********************************************
           *      PATH TO LOWEST POINT IS SINGLE EDGE.
           **********************************************;
           NEXTSORT(FREESORT):=NEXTSORT(2*NUMBER(V2)-1);
           NEXTSORT(2*NUMBER(V2)-1):=FREESORT;
         END
       ELSE
```

```
                BEGIN
                  COMMENT ****************************************
                  *      PATH TO LOWEST POINT IS INDIRECT.
                  ******************************************;
                  IF LOWPT2(V2)>=NUMBER(SORTPT1(FREESORT)) THEN
                    BEGIN
                      NEXTSORT(FREESORT):=NEXTSORT(2*LOWPT1(V2)
                        -1);
                      NEXTSORT(2*LOWPT1(V2)-1):=FREESORT
                    END
                  ELSE
                    BEGIN
                      NEXTSORT(FREESORT):=NEXTSORT(2*LOWPT1(V2)
                        );
                      NEXTSORT(2*LOWPT1(V2)):=FREESORT
                    END
                END
            END;
          COMMENT ******************************************
          *      EMPTY BUCKETS AND CONSTRUCT STRUCTURAL
          *      REPRESENTATION.  EDGES WILL BE IN REVERSE OF
          *      DESIRED ORDER.  THIS IS CORRECTED BY NEXT
          *      SEARCH.
          ************************************************;
          FOR I:=1 UNTIL 2*V DO
            BEGIN
              SORTPTR:=NEXTSORT(I);
              WHILE SORTPTR¬=0 DO
                BEGIN
                  NEXTLINK(SORTPT1(SORTPTR),SORTPT2(SORTPTR));
                  SORTPTR:=NEXTSORT(SORTPTR)
                END
            END
        END;




IF E>3*V-6 THEN GO TO NONPLANAREXIT;
      COMMENT ************************************************
      *      INITIALIZE AND RUN FIRST SEARCH TO COMPUTE
      *      LOWPOINTS.
      ******************************************************;
      FOR I:=1 UNTIL V DO
        BEGIN
          NUMBER(I):=0;
          LOWPT1(I):=LOWPT2(I):=V+1;
        END;
```

```
          POINT:=EPTR:=0;
          V2:=NUMBER(1):=CODE:=1;
          PRESEARCH(V2,POINT);
          FOR I:=1 UNTIL V DO IF LOWPT2(I)>=NUMBER(I) THEN
               LOWPT2(I):=LOWPT1(I);
      SORT;
      FOR I:=2 UNTIL V DO NUMBER(I):=0;
      EPTR:=0;
      POINT:=CODE:=1;
      PATH(1):=1;
      S(0):=F(0):=0;
      SECONDSEARCHER(POINT);
      FREENEXT:=V;
      FOR I:=1 UNTIL E DO NEXTLINK(EDGESTACK(2*I-1),EDGESTACK(2*I));
      APTP:=YPIR:=XNPTR:=0;
      XSPTR:=0;

      STARTPOINT:=0;
      PATHNUMBER:=1;
        BEGIN
          PROCEDURE PATHFINDER(INTEGER VALUE RESULT POINT);
            WHILE NEXT(POINT)¬=0 DO
              BEGIN
                V2:=HEAD(NEXT(POINT));
                NEXT(POINT):=NEXT(NEXT(POINT));
WRITE("POINT IS",POINT,"V2 IS",V2);

                IF STARTPOINT=0 THEN
                  BEGIN
                    STARTPOINT:=POINT;
                    END;
                IF V2>POINT THEN
                  BEGIN
                    RANGEP(V2):=CUR;
                    PATH(V2):=PATHNUMBER;
                    PATHFINDER(V2);
                    CUR:=V2-1;
                    STARTPOINT:=0;
                    WHILE POINT<=Y(YPTR) DO YPTR:=YPTR-2;
                    WHILE POINT <=A(APTR) DO APTR:=APTR-2;
                    WHILE POINT<=XN(XNPTR) DO XNPTR:=XNPTR-3;
                    WHILE POINT <=XS(XSPTR) DO XSPTR:=XSPTR-3;
                    FLAG:=FALSE;
                    WHILE (HIGHPATH(2*POINT-1)>XN(XNPTR-1)) AND
                         (POINT<XN(XNPTR-1)) AND
                         (HIGHPATH(2*POINT)<XN(XNPTR-2)) DO
                      BEGIN
                        FLAG:=TRUE;
                             EDGELINK(HIGHPATH(2*POINT),XN(XNPTR-2));
WRITE("HIGHPATH XLINK",XN(XNPTR-2),HIGHPATH(2*POINT));

                             EDGELINK(XN(XNPTR-2),HIGHPATH(2*POINT));
                             LINKTYPE(EDGEFREE-1):=LINKTYPE(EDGEFREE)
                                   :=FALSE;
                        XNPTR:=XNPTR-3;
                      END;
                    IF FLAG THEN XNPTR:=XNPTR+3;
                    HIGHPATH(2*POINT):=HIGHPATH(2*POINT-1):=0;
                  END
                ELSE
```

```
                    BEGIN
WRITE("PATHNUMBER IS",PATHNUMBER,"STARTPOINT IS",STARTPOINT,"V2 IS",V2);
                      S(PATHNUMBER):=STARTPOINT;
                      F(PATHNUMBER):=V2;
                      FLAG:=FALSE;
                      IF A(APTR)¬=0 THEN ADD2(A(APTR-1),A(APTR),Y,YPTR)
                        ;
                      IF F(A(APTR-1))¬=V2 THEN
                        BEGIN COMMENT PATH IS NORMAL;
                      WHILE V2 < Y(YPTR) DO
                        BEGIN
                          EDGELINK(PATHNUMBER,Y(YPTR-1));
WRITE("YLINK",Y(YPTR-1),PATHNUMBER);
                          EDGELINK(Y(YPTR-1),PATHNUMBER);
                          LINKTYPE(EDGEFREE-1):=LINKTYPE(EDGEFREE):=
                              TRUE;
                          FLAG:=TRUE;
                          YPTR:=YPTR-2;
                        END;
                      IF FLAG THEN YPTR:=YPTR+2;
                      FLAG:=FALSE;
                          WHILE(V2<XN(XNPTR))AND(STARTPOINT<XN
                            (XNPTR-1)) DO
                            BEGIN
WRITE("XLINK",PATHNUMBER,XN(XNPTR-2));
                          EDGELINK(PATHNUMBER,XN(XNPTR-2));
                          EDGELINK(XN(XNPTR-2),PATHNUMBER);
                          LINKTYPE(EDGEFREE-1):=LINKTYPE(EDGEFREE):=
                              FALSE;
                          XNPTR:=XNPTR-3;
                            END;
                       WHILE(V2<XS(XSPTR))AND(STARTPOINT<XS(XSPTR-1))
                          DO XSPTR:=XSPTR-3;
                       IF STARTPOINT>HIGHPATH(2*V2-1) THEN
                         BEGIN
                           HIGHPATH(2*V2-1):=STARTPOINT;
                           HIGHPATH(2*V2):=PATHNUMBER;
                         END;
                       ADD3(PATHNUMBER,STARTPOINT,V2,XN,XNPTR);
                       ADD3(PATHNUMBER,STARTPOINT,V2,XS,XSPTR);
                         END
                       ELSE
                         BEGIN COMMENT PATH IS SPECIAL;
                           WHILE(V2<XS(XSPTR))AND(STARTPOINT<XS
                             (XSPTR-1))AND(XS(XSPTR-1)<=RANGEP
                             (STARTPOINT)) DO
                               BEGIN
                                 FLAG:=TRUE;
WRITE("SPECIAL XLINK",PATHNUMBER,XS(XSPTR-2));
                           EDGELINK(PATHNUMBER,XS(XSPTR-2));
                           EDGELINK(XS(XSPTR-2),PATHNUMBER);
                           LINKTYPE(EDGEFREE-1):=LINKTYPE(EDGEFREE):=
                               FALSE;
                           XSPTR:=XSPTR-3;
                               END;
                             IF FLAG THEN XSPTR:=XSPTR+3;
                           END;
                        IF POINT¬=STARTPOINT THEN
```

```
                        ADD2(PATHNUMBER,STARTPOINT,A,APTR);
                    PATHNUMBER:=PATHNUMBER+1;
                    STARTPOINT:=0;
                  END
              END;
          INTEGER ARRAY A,Y(-1::2*E);
      INTEGER ARRAY XN   ,XS(-2::3*E);
          INTEGER ARRAY HIGHPATH(1::2*V);
      Y(-1):=Y(0):=A(-1):=A(0):=XN(-2):=XN(0):=0;
      XS(-2):=XS(0):=0;
      XN(-1):=XS(-1):=V+1;
      FOR I:=1 UNTIL 2*V DO HIGHPATH(I):=0;
      EDGEFREE:=E-V+1;
      FOR I:=1 UNTIL 7*E-5*V+2 DO NEXTEDGE(I):=0;
      V2:=1;
      CUR :=V; RANGEP(1):=V;
      PATHFINDER(V2);
        END;

      PATHNUMBER:=PATHNUMBER-1;
      FOR I:=1 UNTIL E-V+1 DO COLOR(I):=0;
      FOR I:=2 UNTIL PATHNUMBER+1 DO NEWNODE(I):=TRUE;
      STARTPATH:=1;
      WHILE STARTPATH<=PATHNUMBER DO
        BEGIN
          COLOR(STARTPATH):=1;
          NEWNODE(STARTPATH):=FALSE;
          PATHMARKER(STARTPATH);
          WHILE¬NEWNODE(STARTPATH) DO STARTPATH:=STARTPATH+1;
        END;

        BEGIN
          PROCEDURE COLORCHECK;
            FOR I:=1 UNTIL PATHNUMBER DO
              BEGIN
                POINT:=S(I);
                V2:=F(I);
                    WHILE POINT<=ALEFT(ALPTR) DO ALPTR:=ALPTR-2;
                    WHILE POINT<=ARIGHT(ARPTR) DO ARPTR:=ARPTR-2;
                    WHILE POINT<=XLEFT(XLPTR) DO XLPTR:=XLPTR-2;
                    WHILE POINT<=XRIGHT(XRPTR) DO XRPTR:=XRPTR-2;

                    IF COLOR(I)=1 THEN
                      BEGIN
                        IF (        F(PATH(POINT))¬=V2) THEN
                      BEGIN
                        IF V2<ARIGHT(ARPTR) THEN
   BEGIN
     WRITE("CONFLICT IN ARIGHT",I,ARIGHT(ARPTR-1));
                            GO TO NONPLANAREXIT;
END;
                        IF V2<XLEFT(XLPTR) THEN
 BEGIN
     WRITE("CONFLICT IN XLEFT",I,XLEFT(XLPTR-1));
                            GO TO NONPLANAREXIT;
   END;
                            ADD2(I,V2,XLEFT,XLPTR);
                      END
                    ELSE IF(V2<XLEFT(XLPTR))AND(        S(XLEFT(XLPTR
                          -1))<=RANGEP(POINT))THEN
```

```
 BEGIN
  WRITE("SPECIAL CONFLICT",I,XLEFT(XLPTR-1));
 GO TO NONPLANAREXIT;
  END;
                            ADD2(I,POINT,ALEFT,ALPTR);
                          END
                        ELSE
                          BEGIN
                            IF (        F(PATH(POINT))¬=V2)  THEN
                          BEGIN
                            IF V2<ALEFT(ALPTR)  THEN
 BEGIN
   WRITE("CONFLICT IN ALEFT",I,ALEFT(ALPTR-1));
                                GO TO NONPLANAREXIT;
  END;
                            IF V2<XRIGHT(XRPTR)  THEN
BEGIN
   WRITE("CONFLICT IN XRIGHT",I,XRIGHT(XRPTR-1));
                                GO TO NONPLANAREXIT;
END;
                                ADD2(I,V2,XRIGHT,XRPTR);
                          END
                         ELSE IF(V2<XRIGHT(XRPTR))AND(        S(XRIGHT
                           (XRPTR-1))<=RANGEP(POINT))  THEN
   BEGIN
   WRITE("SPECIAL CONFLICT",I,XRIGHT(XRPTR-1));
                             GO TO NONPLANAREXIT;
 END;
                                ADD2(I,POINT,ARIGHT,ARPTR);
                           END;
                     END;
                  INTEGER ARRAY ALEFT,ARIGHT,XLEFT,XRIGHT(-1::2*E);
               ARPTR:=ALPTR:=XRPTR:=XLPTR:=0;
               ALEFT(0):=ARIGHT(0):=XLEFT(0):=XRIGHT(0):=0;
               ALEFT(-1):=ARIGHT(-1):=XLEFT(-1):=XRIGHT(-1):=0;

               COLORCHECK;
                 END;
WRITE("PLANAR");
WRITE("TIME",TIME(1));
GO TO DONE;
NONPLANAREXIT:     ; NONPLANAR2:WRITE("NONPLANAR");
DONE: GO TO NODEXT;
           END;
      END;
END.
```