EFFICIENT PLANARITY TESTING

John Hopcroft

and

Robert Tarjan

TR 73 - 165

April 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

EFFICIENT PLANARITY TESTING[†]

John Hopcroft

and

Robert Tarjan

Cornell University
Ithaca, New York

Abstract:

This paper describes an efficient algorithm to determine whether an arbitrary graph G can be embedded in the plane. The algorithm may be viewed as an iterative version of a method originally proposed by Auslander and Parter and correctly formulated by Goldstein. The algorithm uses depth-first search and has O(V) time and space bounds, where V is the number of vertices in G. An Algol implementation of the algorithm successfully tested graphs with as many as 900 vertices in less than 12 seconds.

Keywords and phrases: algorithm, complexity, depth-first search, embedding, genus, graph, palm tree, planarity, spanning tree.

CR Categories: 5.32, 5.25, 3.24

[†] This research was supported by the Hertz Foundation, the [illegible] Science Foundation, and the Office of Naval Research [illegible] grants #N-00014-A-0112-0057 and #N-00014-67-A-0077-0021.

EFFICIENT PLANARITY TESTING

John Hopcroft

and

Robert Tarjan

Cornell University
Ithaca, New York

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 *efficient* algorithms for answering graph-theoretic questions. This paper presents an efficient algorithm to determine whether a graph G can be embedded (without any crossing edges) in the plane.

The planarity algorithm may be viewed as an iterative version of a recursive method originally proposed by Auslander and Parter [1] and correctly formulated by Goldstein [2]. The algorithm uses *depth-first search* to order the calculations and thereby achieve efficiency. Depth-first search, or backtracking, has been widely used for finding solutions to problems in combinatorial theory and artifical intelligence [3,4]. Recently this type of search has been used to construct efficient algorithms for solving several problems in graph theory, including finding biconnected components [5,6], finding triconnected components [7,8], finding strongly connected components [6], finding dominators [9], and determining whether a directed graph is reducible [10,11].

In order to analyze the theoretical efficiency of the planarity algorithm, a random access computer model is used. 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 kV for some constant k, where V is the number of vertices in the problem graph. Cook [12] describes an exact computer model along those lines. If f and g are functions of x, we say "f(x) is O(g(x))" if, for some constants k_1 and k_2, $|f(x)| \leq k_1|g(x)| + k_2$ for all x. Within this framework, the planarity algorithm has O(V) time and space bounds and is optimal to within a constant factor.

The practical efficiency of the algorithm was measured by implementing it in Algol W, the Stanford University version of Algol [13]. The algorithm in this paper is much simpler than the one originally programmed, but the program was able to analyze graphs with up to 900 vertices in less than 12 seconds of IBM 360/67 processing time.

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 [7,14-20]. The importance of the problem is suggested by the number of published planarity algorithms. Examples include [1,2,21-32]. 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 [28], which contains an extensive bibliography.

The earliest characterization of planar graphs was given by Kuratwoski [33]. He proved that every nonplanar 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 (Figure 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 construct a representation of a planar embedding of the given graph. If such a representation can be completed then the graph is planar; if not, then the graph is non-planar. The first such algorithm was proposed by Auslander and Parter [1]. 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 [2] correctly formulated the algorithm, using iteration instead of recursion. Shirey [28] 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 [25] have presented an alternate 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 [34] has implemented the algorithm in a way which requires $O(V)$ space and $O(V^2)$ time.

Mondshein [27] 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 [24], using depth-first search in a complicated program, have devised a variant of Goldstein's algorithm with a time bound of $O(V \log V)$. Subsequently they discovered an improved algorithm with $O(V)$ time bound, an early version of which appears in Tarjan's dissertation [29]. The algorithm to be presented here is a considerable simplication of [29].

A few algorithms deserve mention because of their novel approach. Fisher [23] 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. Bruno, Steiglitz, and Weinberg [21] present an algorithm based on some theorems of Tutte relating to tri-connected 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. Preliminaries

This section outlines some of the graph-theoretic concepts needed to understand the planarity algorithm. It also describes how graphs are represented in a computer, and how a depth-first search works. We use definitions similar to those found in any text on graph theory, e.g. [35,36,37,38].

A graph $G = (\mathcal{V},\mathcal{E})$ is an ordered pair consisting of a finite set of vertices $\mathcal{V}$ and a finite set of edges $\mathcal{E}$. V denotes the number of vertices in G; E denotes the number of edges. If each edge is an unordered pair of distinct vertices, then the graph is <u>undirected</u>. If each edge is an ordered pair of distinct vertices, then the graph is <u>directed</u>. If (v,w) is an edge in a directed graph, we say the edge <u>leaves</u> v and <u>enters</u> w. A graph $G = (\mathcal{V}_1, \mathcal{E}_1)$ is a <u>subgraph</u> of a graph $G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ if $\mathcal{V}_1 \subseteq \mathcal{V}_2$ and $\mathcal{E}_1 \subseteq \mathcal{E}_2$. If G is a directed graph, the <u>undirected version</u> of G is the undirected graph formed by converting each edge of G to an undirected edge and removing duplicate edges.

A sequence of vertices v_i, $1 \leq i \leq n$, and edges e_i, $1 \leq i < n$, such that $e_i = (v_i, v_{i+1})$ is called a <u>path of G from v_1 to v_n</u>. There is a path of no edges from any vertex to itself. A vertex w is <u>reachable</u> from a vertex v if there is a path from v to w. A path is <u>simple</u> if all its vertices are distinct. A path from a vertex to itself is a <u>closed path.</u> A closed path from v to v with one or more edges is a <u>cycle</u> if all its edges are distinct and the only vertex to appear twice is v, which appears exactly twice. Two cycles which are cyclic permutations of each other are considered to be the same cycle. We use $p : v \overset{*}{\Rightarrow} w$ to denote that p is a path from v to w.

An undirected graph G is <u>connected</u> if any vertex in G is reachable from any other vertex. The maximal connected subgraphs of G are well-defined and vertex-disjoint [38], and are called the <u>connected components</u> of G. If G contains three distinct vertices x,v,w such that w is reachable from v but every path $p : v \overset{*}{\Rightarrow} w$ contains x, then x is a <u>cutnode</u> or <u>an articulation point</u> of G. If G is connected and contains no separation points, then G is biconnected. The maximal biconnected subgraphs of G are well-defined and edge-disjoint [37], and are called the biconnected components of G. If G is biconnected but contains four distinct vertices x,y,v,w such that every path $p : v \overset{*}{\Rightarrow} w$ contains either x or y, then (x,y) is called a separation pair of G. If G is biconnected and contains no separation pairs, G is triconnected. The triconnected components of a graph may be defined in several ways [8,39]. We may extend these definitions to directed graphs by considering their undirected versions.

A (directed,rooted) tree T is a directed graph with one distinguished vertex, called the root r, such that every vertex in T is reachable from r, no edges enter r, and exactly one edge enters every other vertex in T. The relation "(v,w) is an edge in T" is denoted by $v \to w$. The relation "there is a path from v to w in T" is denoted by $v \overset{*}{\to} w$. If $v \to w$, v is the father of w and w is a son of v. If $v \overset{*}{\to} w$, v is an ancestor of w and w is a descendant of v. Every vertex is an ancestor and a descendant of itself. If $v \overset{*}{\to} w$ and $v \neq w$, v is a proper ancestor of w and w is a proper descendant of v. If T_1 is a tree and T_1 is a subgraph of a tree T_2, then T_1 is a subtree of T_2. If T is a tree which is a subgraph of a directed graph G and T contains all the vertices of G, then T is a spanning tree of G.

A graph G is <u>planar</u> if and only if there exists a mapping of the vertices and edges of the graph into the plane such that

(1) each vertex is mapped into a distinct point,

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

(3) mappings of distinct edges have only the mappings of their common endpoints in common.

A mapping of G which satisfies the conditions above is called a <u>planar embedding</u> of G. (If G is planar, there is a planar embedding of G in which the edges are mapped into straight lines.) We need two lemmas about planar graphs.

<u>Lemma 1</u>: If G is planar, $E \leq 3V-3$.

<u>Proof</u>: This lemma is an immediate consequence of Euler's theorem relating the number of vertices, faces, and edges in a planar graph [35].

<u>Lemma 2</u>: Let G be a planar graph embedded in the plane. For brevity we identify each edge of G with its embedding. Let p_1, p_2, p_3 be three paths leading from x to y such that any two of the paths have only x and y as common vertices. Let (x,v_1), (x,v_2), (x,v_3) be the first edges of p_1, p_2, p_3, respectively, and let (w_1,y),

(w_2,y), (w_3,y) be the last edges of p_1, p_2, p_3. If the orientation of edges clockwise around x in the plane is (x,v_1), (x,v_2), (x,v_3), then the orientation of edges clockwise around y is (w_1,y), (w_3,y), (w_2,y) (Figure 2).

Proof: This lemma is a corollary of the Jordan curve Theorem, which states that a simple closed curve divides the plane into exactly two connected regions. We accept the lemma without proof; the Jordan Curve Theorem is very hard to prove. (See [40,41].)

An arbitrary (undirected) graph with V vertices may have as many as $E = V(V-1)/2$ edges. However, a planar graph has $E \leq 3V-3$ by Lemma 1. Thus it may be possible to devise a planarity algorithm with a time bound which is linear in the number of vertices. One way to represent a graph in a computer is to use a VxV adjacency matrix $M = (m_{ij})$, where $m_{ij} = 1$ if (i,j) is an edge, $m_{ij} = 0$ otherwise. However, the amount of storage space required by an adjacency matrix is $O(V^2)$, and it can be shown rigorously that many graph problems (including the planarity problem) require examination of every bit in the matrix and thus have a computation time proportional to at least V^2[42]. For this reason we use a list structure called an adjacency structure to represent a graph. We construct a set of adjacency lists $A(v)$, one for each vertex v. The list for vertex v contains each vertex w such that

(v,w) is an edge of the graph. If G is an undirected graph, each edge (v,w) is represented twice: w appears in $A(v)$ and v appears in $A(w)$. If G is directed, each edge (v,w) is represented once: w appears in $A(v)$.

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

If G is undirected, a depth-first search of G imposes a direction on each edge of G given by the direction in which the edge is traversed during the search. Thus the search converts G into a directed graph G'. The search also partitions the (now-directed) edges into two classes: a set of tree arcs, defining a spanning tree T of G', and a set of fronds (v,w) which satisfy $w \overset{*}{\rightarrow} v$ in T [6]. A directed graph G' whose edges may be partitioned in this way is called a palm tree. Depth-first search is important because the structure of paths in a palm tree is very simple.

To implement a depth-first search of a connected, undirected graph, we use a simple recursive procedure which keeps a stack of the old vertices with possibly unexplored edges. The procedure uses a set of adjacency lists of the graph to be searched, and the exact search order depends on the order of edges in the adjacency lists. The procedure numbers the vertices from 1 to V in the order they are reached during the search, in addition to identifying tree arcs and fronds.

```
begin comment routine for depth-first search of a graph G
          represented by adjacency lists A(v). Variable
          n denotes the last number assigned to a vertex;
  integer n;
  procedure DFS(v,u); comment vertex u is the father of
          vertex v in the spanning tree being constructed.


  begin
    n: = NUMBER(v): = n+1;
    a: comment dummy statement;
    for w ∈ A(v) do begin
      if NUMBER(w) = 0 then begin
        comment a is a new vertex;
          mark (v,w) as a tree arc;
          DFS(w,v);
        b: comment dummy statement;
      end
      else if NUMBER(w) < NUMBER(v) and w ≠ u then begin
        comment this test is necessary to avoid exploring
        an edge in both directions;
```

```
            mark (v,w) as a frond;
            c: comment dummy statement;
          end;
        end;
      end;
      for i:= 1 until V do NUMBER(i):=0;
      n:=0;
      comment the search starts at vertex s;
      DFS (s,0);
    end;
```

Lemma 3: The procedure above correctly carries out a depth-first search of an undirected graph and requires O(V+E) time if the graph has V vertices and E edges. The vertices are numbered so that if (v,w) is a tree arc, NUMBER(v) < NUMBER(w); and if (v,w) is a frond, NUMBER(w) < NUMBER(v).

Proof: See [6].

Figure 3 shows a connected graph G and a palm tree generated from G using depth-first search.

4. An Outline of the Planarity Algorithm

This section sketches the ideas behind the planarity algorithm. Sections 5 and 6 develop the detailed components and Section 7 presents the algorithm in toto. The first step of the algorithm gets rid of graphs with too many edges. We count the number of edges in the graph and if the count ever exceeds 3V-2, we declare the graph non-planar. Next, we divide the graph into biconnected components. (A graph is planar if and only if all its biconnected components are planar [35].) References [5] and [6] describe how to divide a graph into biconnected components in O(V+E) time. Then we test the planarity of each component.

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

Each recursive call on the algorithm requires that we find a cycle in the piece of the graph to be tested for planarity. This cycle will consist of a simple path of edges not in previously found cycles, plus a simple path of edges in old cycles. We use depth-first search to divide the graph into simple paths which may be assembled into the cycles necessary for planarity testing. We need a second search to find paths, because the search must be carried out in a special order if the planarity test is to be efficient. Section 5 describes the path finding process in detail and proves some important properties of the generated paths.

Now consider the first cycle c. It will consist of a sequence of tree arcs followed by one frond in P. The numbering of vertices is such that the vertices are in order by number along the cycle. Each piece not part of the cycle will consist either of a single frond (v,w), or of a tree arc (v,w) plus a subtree with root w, plus all fronds which lead from the subtree. We process the pieces and add them to a planar representation in decreasing order of v. Each piece can go either "inside" or "outside" c by the Jordan Curve Theorem. When we add a piece, certain other pieces must be moved from the inside to the outside or from the outside to the inside of c. (see Figure 4.) We continue to add new pieces and move old pieces if necessary until either a piece cannot be added or the entire graph is embedded in the plane. Section 6 describes the data structures necessary to keep track of the pieces as they are moved. Below is an outline of the entire algorithm.

```
procedure PLANARITY(G);
begin comment an outline of the planarity algorithm;
   integer E;
   E:=0;
   for each edge of G do begin
      E:=E+1;
      if E > 3V-3 then go to nonplanar;
   end;
```

```
divide G into biconnected components;
for each biconnected component C do begin
   explore C to number vertices and transform C into
      a palm tree P;
   find a cycle c in P;
   construct planar representation for c;
   for each piece formed when c is deleted do begin
      apply algorithm recursively to determine if
         piece plus cycle is planar;
      if piece plus cycle is planar and piece may be
         added to planar representation then add it
         else go to nonplanar;
      end;
   end;
end;
```

5. Pathfinding

Henceforth we assume that G is a biconnected graph which has been explored using DFS to number the vertices and generate a palm tree P. We will identify vertices by their number. If v is a vertex, let

$$S_v = \{w \mid \exists u (v \overset{*}{\rightarrow} u \text{ and } u \dashrightarrow w)\}.$$

S_v is the set of vertices reached by fronds from descendants of v. Let $LOWPT1(v) = \min(\{v\} \cup S_v)$, and $LOWPT2(v) = \min(\{v\} \cup (S_v - \{LOWPT1(v)\}))$. LOWPT1(v) is the lowest vertex below v reachable by a frond from a descendant of v, and LOWPT2(v) is the second lowest vertex below v reachable by a frond from a descendant of v. By convention, these values are equal to v if they are not defined. LOWPT1(v)≠LOWPT2(v) unless LOWPT1(v)=LOWPT2(v)=v. The LOWPT values of a vertex v depend only on the LOWPT values of sons of v and on the fronds leaving v; thus it is easy to calculate LOWPT values using DFS.

Inserting the following statements for the dummy statements a,b, and c in DFS will produce a routine to compute LOWPT values.

```
comment additions to DFS for calculation of LOWPT1, LOWPT2;
a:  LOWPT1(v):= LOWPT2(v):= NUMBER(v);

b:  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)};

c:  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)};
```

It is easy to verify that DFS as modified above will compute LOWPT values correctly in $O(V+E)$ time. (See [6,8,29].) LOWPT1 may be used to test the biconnectivity of G, as described in [5,6]. One related lemma is important:

Lemma 4: If G is biconnected and $v \rightarrow w$, $\mathrm{LOWPT1}(w) < v$ unless $v = 1$, in which case $\mathrm{LOWPT1}(w)=v=1$. Also, $\mathrm{LOWPT1}(1)=1$.

Proof: See [6].

To generate paths, we sort the adjacency lists of P according to LOWPT values and perform another depth-first search. Let ϕ be a function defined on the edges (v,w) of P as follows:

$$\phi((v,w)) = \begin{cases} 2\cdot w & \text{if } v \dashrightarrow w \\ 2\cdot \mathrm{LOWPT1}(w) & \text{if } v \longrightarrow w \text{ and } \\ & \quad \mathrm{LOWPT2}(w) \geq v \\ 2\cdot \mathrm{LOWPT1}(w)+1 & \text{if } v \longrightarrow w \text{ and } \\ & \quad \mathrm{LOWPT2}(w) < v \end{cases}$$

We calculate $\phi((v,w))$ for each edge in P and order the adjacency lists according to increasing value of ϕ, using a radix sort to achieve an $O(V+E)$ time bound. This can be implemented as follows:

```
comment construction of ordered adjacency lists;
for i:=1 until 2*V+1 do BUCKET(i):=the empty list;
for (v,w) an edge of G do begin
   compute φ((v,w));
   add (v,w) to BUCKET (φ((v,w)));
```

```
end;
for i:=1 until V do A(v):= the empty list;
for i:=1 until 2*V+1 do
   for (v,w) ∈ BUCKET(i) do add w to end of A(v);
```

This routine gives a set of properly ordered adjacency lists representing P. Now we generate paths by applying depth-first search to P, using the new adjacency lists. Each time we traverse an edge we add it to the path being built. Each time we traverse a frond, the frond becomes the last edge of the current path. The next edge starts a new path. Thus each path consists of a sequence of tree arcs followed by a single frond. To accomplish this, we use the following steps:

```
begin comment routine to generate paths in a biconnected
   palm tree with specially ordered adjacency lists A(v).
   Vertex s is a global variable, the start vertex of the
   current path, and is initialized to 0;
procedure PATHFINDER(v);
   for w ∈ A(v) do
      if v → w then begin
         if s = 0 then begin
            s:= v;
            start new path;
         end;
         add (v,w) to current path;
         PATHFINDER(w);
```

```
        end
        else begin
           comment v - → w;
              if s = 0 then begin
                 s:=v;
                 start new path
              end;
              add (v,w) to current path;
              output current path;
              s:=0;
        end;
     s:=0;
     comment vertex 1 is the start vertex of the search;
     PATHFINDER(1);
  end;
```

The paths generated in this manner have several important properties, which are summarized in the following lemmas. Figure 5 shows a set of paths generated from the graph in Figure 3.

Lemma 5: Let p: $s \overset{*}{\Rightarrow} f$ be a generated path. If we consider the fronds which have not been used in any path when the first edge in p is traversed, then f is the lowest vertex reachable via such a frond from a descendant of s. If $v \neq s$, $v \neq f$, and v lies on p, then f is the lowest vertex reachable from a descendant of v via any frond.

Proof: This lemma is an immediate consequence of the ordering of the adjacency lists.

Lemma 6: Let $p : s \overset{*}{\Rightarrow} f$ be a generated path. Then $f \overset{*}{\rightarrow} s$ in the spanning tree of P. If p is the first path, p is a cycle; otherwise p is simple. If p is not the initial path, p contains exactly two vertices (f and s) in common with previously generated paths.

Proof: Let $p : s \overset{*}{\Rightarrow} f$ be any generated path. If the path consists of a single frond, the path is simple and $f \overset{*}{\rightarrow} s$. If the path contains a tree arc, let $s \rightarrow v$ be the first such tree arc. Then $f = \text{LOWPT1}(v)$ by Lemma 5. If $s = 1$ the path is a cycle, and if $s > 1$ the path is simple, by Lemma 4. In any case $f \overset{*}{\rightarrow} s$. If f is reached during the pathfinding search, then s has already been reached, so any path except the first has exactly two vertices, f and s, in common with previously generated paths.

Lemma 7: Let $p_1 : s_1 \overset{*}{\Rightarrow} f_1$ and $p_2 : s_2 \overset{*}{\Rightarrow} f_2$ be two generated paths. If p_1 is generated before p_2 and s_1 is an ancestor of s_2, then $f_1 \leq f_2$.

Proof: The frond which ends p_2 leads from a descendant of s_1 and is unused when p_1 is generated. By Lemma 5, $f_1 \leq f_2$.

Lemma 8: Let $p_1 : s \overset{*}{\Rightarrow} f$ and $p_2 : s \overset{*}{\Rightarrow} f$ be two generated paths with the same start and finish vertices. Let v_1 be the second vertex of p_1 and let v_2 be the second vertex of p_2. Suppose p_1 is generated before p_2, $v_1 \neq f$, and $LOWPT2(v_1) < s$. Then $v_2 \neq f$ and $LOWPT2(v_2) < s$.

Proof: Vertex v_1 must appear before vertex v_2 in $A(s)$ because p_1 is generated before p_2. The lemma follows from the ordering imposed on $A(s)$.

Lemma 8 is the reason we need to include LOWPT2 values in the pathfinding algorithm. When we consider the embedding of paths in the plane, we shall see why this lemma is important.

If $p : s \overset{*}{\Rightarrow} f$ is a generated path, we may form a cycle by adding the set of tree arcs $f \overset{*}{\rightarrow} s$ to p. The cycles formed in this way are the cycles generated by recursive calls in the Auslander-Parter-Goldstein planarity algorithm. They have a very simple structure; each corresponds to a frond of P. We need one more definition before we consider the embedding of paths. If $p : s \overset{*}{\Rightarrow} f$ is a simple path generated by the pathfinding algorithm, let $p_0 : s_0 \overset{*}{\Rightarrow} f_0$ be the earliest generated path containing vertex s. If $f_0 < f$, then p is called a normal path. If $f_0 = f$ then p is called a special path. The case $f_0 > f$ cannot occur by Lemma 7.

6. Embedding the Paths

If G is a biconnected graph with a set of paths generated by the pathfinding algorithm, we test the planarity of G by attempting to embed the paths one at a time in the plane. Let c be the first path (a cycle). The cycle consists of a set of tree arcs $1 \to v_1 \to v_2 \to \ldots \to v_n$ followed by a frond $v_n \dashrightarrow 1$. The vertex numbering is such that $1 < v_1 < \ldots < v_n$. When c is removed, G falls into several connected pieces, called segments. Each segment S consists either of a single frond (v_i,w), or of a tree arc (v_i,w) plus a subtree with root w plus all fronds leading from the subtree. The order of path generation is such that all paths in one segment are generated before paths in any other segment, and the segments are explored in decreasing order of v_i.

A segment must be embedded completely on one side of c by the Jordan Curve Theorem. A segment is attached to c by one arc (v_i,w) leading from c and by one or more fronds leading to c. (If the segment is a single frond, both endpoints of the frond are on c.) We say the segment S is embedded on the left (of c) if the orientation of edges (clockwise in the plane) around v_i is $(v_{i-1},v_i),(v_i,w),(v_i,v_{i+1})$. The segment is embedded on the right if the orientation of edges around v_i is $(v_{i-1},v_i),(v_i,v_{i+1}),(v_i,w)$. We say a frond which enters c is embedded on the left (right) if the segment to which it belongs is on the left (right) of c. If (x,v_j) is a frond which enters c on the left, the orientation of edges around v_j is (v_{j-1}, v_j), (x, v_j), (v_j, v_{j+1}) by Lemma 2.

Suppose c and segments explored before S have been somehow embedded in the plane. Let $p : v_i \stackrel{*}{\Rightarrow} v_j$ be the first path found in S. The next lemma gives a necessary and sufficient condition for adding p to the embedding.

<u>Lemma 9</u>: Path $p : v_i \stackrel{*}{\Rightarrow} v_j$ may be added to the planar embedding by placing it on the left (right) of c if and only if no frond (x, v_k) previously embedded on the left (right) satisfies $v_j < v_k < v_i$.

<u>Proof</u>: If no frond satisfies the condition, then no embedded edge of any sort enters or leaves c on the left (right) between v_j and v_i. Path p may be embedded on the left (right) of c if it is placed sufficiently close to c. Conversely, suppose we want to embed p on the left but some embedded frond (x, v_k) with $v_j < v_k < v_i$ enters c on the left. Either x lies on c (say $x = v_\ell$) or (x, v_k) is part of a segment S' with first edge (v_ℓ, w). We know $v_\ell \geq v_i$ by the order of path generation. We must consider two cases.

<u>Case 1</u>: $v_\ell > v_i$ (Figure 6 (a)).

Suppose p is embedded on the left. From p, a path in S' joining v_ℓ and v_k, and the path of tree arcs from v_j to v_ℓ we can construct three paths from v_i to v_k which violate Lemma 2. Thus p cannot be embedded on the left.

Case 2: $v_\ell = v_i$.

Let $p_1 : v_\ell \overset{*}{\Rightarrow} v_m$ be the first path found in segment S'. We have $v_m \leq v_j$ by Lemma 7. There are two subcases.

Subcase A: $v_m < v_j$ (Figure 6 (b)).

Suppose p is embedded on the left. From path p, a path in S' from v_k to v_m, and the path of tree arcs from v_m to v_i we may form three paths from v_k to v_j which violate Lemma 2. Thus p cannot be embedded on the left.

Subcase B: $v_m = v_j$ (Figure 6(c)).

Let y be the second vertex on p (w is already defined as the second vertex on p_1). Since segment S' contains frond (x, v_k), $w \neq v_m$ and LOWPT2(w) $< v_i$. Comparing p and p_1 and applying Lemma 8, we have $y \neq v_j$ and LOWPT2(y) $< v_i$. Furthermore LOWPT2(y) $> v_j$ since LOWPT1(y) $= v_j$, by Lemma 5. Suppose p is embedded on the left. From p , p_1 , a path from a vertex on p to LOWPT2(y), a path from a vertex on p_1 to v_k , and a (possibly empty) path of tree arcs joining v_k and LOWPT2(y) we may form three paths which violate Lemma 2. Thus p cannot be embedded on the left.

We use Lemma 9 to test planarity, in the following way: first we embed the cycle c in the plane. Then we embed the segments one at a time in the order they are explored during pathfinding. To embed a segment S, we find a path in it, say p. We choose a side, say the left, on which to embed p.

We compare p with previously embedded fronds to determine if p can be embedded. If not, we move segments which have fronds blocking p from the left to the right. If p can be embedded after moving segments, we embed it. However, if we move segments from the left to the right we may have to move other segments from the right to the left. Thus it may be impossible to embed p. If so, we declare the graph non-planar. If p can be embedded, we try to embed the rest of S by in essence using the algorithm recursively. Then we try to embed the next segment.

We need some good data structures to efficiently implement this method. If we are about to embed a segment which starts at vertex v_i, we must know which vertices on the tree path from 1 to v_i have fronds entering them from the left and the right We use two stacks (L and R) for this purpose. Stack L will contain (in order) vertices v_k such that $1 \overset{*}{\rightarrow} v_k \overset{*}{\rightarrow} v_i$, $1 < v_k <$ and some embedded frond enters v_k on the left. L need only include a vertex v_k once for each segment which has a frond leading to v_k, but sometimes two fronds from the same segment may lead to the same vertex v_k, and this may cause v_k to appear twice on the stack even though v_k is only representing a single ment. Stack R fullfills the same function as L for embedded fronds entering c on the right.

Stacks L and R must be updated in four ways.

(1) After all segments starting at v_{i+1} are explored and embedded, all occurrences of v_i on L and R must be deleted, since segments yet to be explored start at

vertices no greater than v_i . This updating requires removing a few of the entries on top of L and R.

(2) If $p : s \overset{*}{\Rightarrow} f$ is the first path in a segment S, and p is normal, f must be added to a stack when p is embedded. (Since s lies on c , p is normal if and only if $f > 1$.) By Lemma 9, p can only be embedded on the left (right) when every vertex on L(R) is no greater than f , so f may be added to the top of L(R).

(3) Recursive application of the algorithm must add entries for other paths in the segment S. We shall examine recursive application of the algorithm later.

(4) Entries must be shifted from one stack to another as the corresponding segments are moved. The embedding of a frond, say on the left, forces fronds in the same segment to be embedded on the left by Lemma 2 and may force fronds in other segments to be embedded on the right by Lemma 9. Let a block B be a maximal set of entries on L and R which correspond to fronds such that the placement of any one of the fronds determines the placement of all the others. The blocks change as the content of the stacks change, but the blocks always partition the stack entries. Furthermore, the blocks have a simple structure given by the next Lemma.

Lemma 10: Let B be a block. Then the entries in $B \cap L$ ($B \cap R$) are adjacent on L(R). Also, there are vertices v_j, v_k on c such that for $v_\ell \in L \cup R$:

(1) if $v_j < v_\ell < v_k$ then $v_\ell \in B$

(2) if $v_\ell < v_j$ or $v_\ell > v_k$ then $v_\ell \notin B$.

Proof: The proof is by induction on the number of segments embedded and the number of entries deleted from L and R. The lemma is certainly true before any segments are embedded since both stacks are empty. If the lemma is true before all occurrences of v_i are deleted from the top of L and R, the lemma is certainly true afterwards, since deleting occurrences of v_i from L and R may only cause complete blocks (consisting only of v_i) to be deleted plus causing the top remaining block to lose occurences of its top vertex.

Suppose the lemma is true before segment S is embedded. Let $p : s \overset{*}{\Rightarrow} f$ be the first path in S. Suppose S is to be embedded on the left. When entries corresponding to S are added to L, a new block B is formed containing the entries corresponding to S and also containing all old blocks B with an entry v_ℓ in R satisfying $f < v_\ell < s$, by Lemma 9. All entries v_ℓ in other old blocks satisfy $v_\ell \leq f$. The new block B' consists of old blocks with entries on top of L and R, plus the new entries corresponding to S, which are on top of L. Block B' thus satisfies the lemma with $v_j = f$ and $v_k = s$. Other old blocks are unchanged. The lemma foll by induction on the number of segments embedded and the number of entries deleted from L and R.

Lemma 10 indicates how we can keep track of the blocks. The blocks give us enough information to easily move entries from one stack to another. We use linked lists to store L and R. Then to switch a block of entries between stacks we need only switch list pointers at the beginning and the end of the block. We use a stack B to keep track of the blocks. Each entry on B represents a block and is an ordered pair (x,y), with x pointing to the last block entry on L and y pointing to the last block entry on R. If x = 0 (y = 0), the block has no entries on L(R). The routine which follows implements the embedding algorithm. The necessary list-processing operations are presented in detail in Section 7.

```
procedure EMBED; begin
comment routine to embed a properly ordered graph in the
   plane, if possible;
L:=R:=B:= the empty stack;
find first cycle c;
while some segment is unexplored do begin
   initiate search for path in next segment S;
   when backing down tree arc v → w delete
      entries on L and R and blocks on B containing
      vertices no smaller than v;
   let p:s *=> f be first path found in segment S;
   while position of top block determines position of p do begin
      delete top block from B;
      if block has entries on left then switch block of entries
         from L to R and from R to L by switching list pointers;
```

```
    if block still has an entry on left in conflict with p
       then go to nonplanar;
  end;
    if p is normal add last vertex of p to L;
    add new block to B corresponding to p and blocks just
       removed from B;
  d: apply algorithm recursively to embed other paths in S;
    comment Details of the recursive application are discussed
       later.  After completion of this step, other paths in S
       which lead to ancestors of S will be represented on L.
       One new block corresponding to these paths will appear
       on B;
    combine top two blocks on B;
  end;
end;
```

Lemma 11: Procedure EMBED runs to completion if and only if G is planar. Otherwise the procedure branches to location "nonplanar."

Proof: EMBED is a straightforward implementation of the algorithm previously described. At all times, stacks L and R contain entries for the fronds embedded on the left and right of the cycle c, and stack B contains information about the end of each block of entries. Lemma 9 is used to test planarity, and Lemma is used to modify the blocks as the routine executes. Assuming that Step d (recursive application of the algorithm) is implemented correctly, it is straightforward to prove by induction that

(1) embedding any frond in a block completely determines the embedding of all fronds in a block;

(2) the embedding of a frond from one block does not restrict the embedding of a frond not in the block.

By Lemmas 9 and 10 the routine correctly tests planarity.

Consider the embedding of the second and subsequent paths in a segment. Suppose cycle c, all segments before S, and the first path $p: s \stackrel{*}{\Rightarrow} f$ in S have been embedded. It is easy to see that the rest of S can be added to the embedding if and only if S and c together form a planar graph. (In fact, this follows from Auslander and Parter's results [1].) Figure 7 shows S and c. Path p and the path of tree arcs from f to s form a cycle c' used for recursive application of the embedding algorithm.

After p is embedded on the left by EMBED, the top entry on L is f. All fronds in S lead to vertices no less than f by Lemma 5. Suppose we place an end-of-stack marker on top of R and apply the embedding algorithm recursively to determine if cycle c' plus the segments in S formed when c' is deleted can be embedded in the plane. If the recursion is finished successfully, stacks L and R will contain entries corresponding to fronds ending normal paths in S, and stack B may contain a few new blocks. The rest of cycle c can be added to the embedding of S and c' if and only if no new block has entries on both R and L. If no new block has entries on both R and L, then any new block with an entry on R can be moved to L with the result that no new block will have an entry on R.

Thus to finish testing the planarity of c and S, we must attempt to move the new blocks from R to L. To continue with the top-level application of the algorithm, we must combine all the new blocks into one block corresponding to paths in S minus p and we must delete the end-of-stack marker on R. Then R will be restored, L will have entries for fronds in S on top of its other entries, and B will contain one extra block corresponding to the fronds in S minus p. Step d can be implemented as follows:

```
d: comment apply algorithm recursively to embed
       the rest of S;
   add end-of-stack marker to R;
   call embedding algorithm recursively;
   for each new block (x,y) on B do begin
       if (x ≠ 0) and (y ≠ 0) then go to non planar;
       if (y ≠ 0) then move entries in block to L;
       delete (x,y) from B;
   end;
   delete end-of-stack marker on R;
   add one block to B to represent S minus path p;
```

Lemma 12: If Step d is implemented as above, the embedding algorithm correctly tests planarity.

Proof: This lemma follows from Lemma 11 and Auslander and Parter's result that S minus p can be added to the planar embedding if and only if S plus c is planar. Old entries on L, R, and B cannot interfere with recursive application of the algorithm, since R has an end-of-stack marker and all entries on L are no greater than f. When the recursive application is completed, the information on L, R, and B is exactly what is needed to continue top-level application of the algorithm. Figure 8 illustrates the contents of the stacks L, R, and B as the embedding algorithm is applied to the graph in Figure 5. The next section gives the complete embedding algorithm in detail.

7. The Complete Path Embedding Algorithm

Since paths are embedded as they are found, the emb‹ ing algorithm may be combined with the pathfinding algori‹ A complete implementation appears below. Steps involving and R are implemented in detail to make the running time ‹ the algorithm obvious.

<u>procedure</u> EMBED(G); <u>begin</u>

<u>comment</u> procedure to determine if G is embeddable in th plane. G is represented by a set of properly ordere adjacency lists A(v). Stacks L and R are stored as linked lists using arrays STACK and NEXT. STACK(i) gives a stack entry, and NEXT(i) points to the next entry on the same stack. NEXT(0) points to the firs entry on L. NEXT(-1) points to the first entry on R FREE is the first unused location in STACK. Variabl p denotes the number of the current path. If v is a vertex PATH(v) denotes the number of the first pat containing v. If i is the number of a path, f(i) de the last vertex on the path numbered i. Blocks are represented as ordered pairs on stack B. If (x,y) is on B, x denotes the last entry on L in the block, and y denotes the last entry on R in the block. If x = 0 (y = 0), the block has no entries on L(R). SA is a temporary variable used for switching;

```
integer array STACK(0::E), NEXT(-1::E), f(1::E-V+1),
   PATH(1::V); B(1::E);
procedure PATHFINDER(v); begin
   comment this recursive procedure finds paths and embeds them
      if possible.  It is based on the material in Sections 5
      and 6.  Variable v is the current vertex in the depth-
      first search used to find paths.  s is the start vertex
      of the current path;
   for w ∈ A(v) do
      if v → w then begin
         if s=0 then begin
            s:=v;
            p:=p+1;
         end;
         PATH(w):=p;
         PATHFINDER(w);
         comment delete stack entries and blocks corresponding
            to vertices no smaller than v;
         while (x,y) on B has ((STACK(x)≥v) or (x=0)) and
            ((STACK(y)≥v) or (y=0))
            do delete (x,y) from B;
         if (x,y) on B has STACK(x)≥v then replace (x,y)
            on B by (0,y);
         if (x,y) on B has STACK(y)≥v then replace (x,y)
            on B by (x,0);
         while NEXT(-1)≠0 and STACK(NEXT(-1))≥v
            do NEXT(-1):=NEXT(NEXT(-1));
```

```
while NEXT(0)≠0 and STACK(NEXT(0))≥v
  do NEXT(0):=NEXT(NEXT(0));
if PATH(w) ≠ PATH(v) then begin
   COMMENT all of segment with first edge (v,w) has
      been embedded.  New blocks must be moved from
      right to left;
   L':=0;
   while (x,y) on B has (STACK(x)>f(PATH(w))) or
      (STACK(y)>f(PATH(w))) and (STACK(NEXT(-1)≠0)) do b
      if STACK(x)>f(PATH(w))  then begin
         if STACK(y)>f(PATH(w)) then go to nonplanar;
         L':=x; end
      else begin comment STACK(y)>f(PATH(w));
         SAVE:=NEXT(L');
         NEXT(L'):=NEXT(-1);
         NEXT(-1):=NEXT(y);
         NEXT(y):=SAVE;
         L':=y;
      end;
      delete (x,y) from B;
   end;
   comment block on B must be combined with
      new blocks just deleted;
   delete (x,y) from B;
   if x ≠ 0 then add (x,y) to B
   else if (L'≠0) or (y≠0) then add (L',y) to B;
   comment delete end-of-stack marker on right stack;
   NEXT(-1):= NEXT(NEXT(-1));
end; end
```

```
else begin comment v -→ w.  Current path is complete.
        Path is normal if f(PATH(s)) < w;
   if s=0 then begin
      p:= p+1;
      s:= v;
   end;
   f(p):= w;
   comment switch blocks of entries from left to right
      so that p may be embedded on left;
   L'=0;
   R'=-1;
   while (NEXT(L')≠0) and (STACK(NEXT(L'))>W) or
         (NEXT(R')≠0) and (STACK(NEXT(R'))>W) do begin
      if (x,y) on a link (x≠0) and (y≠0) then begin
         if STACK(NEXT(L'))>w then begin
            if STACK(NEXT(R'))>w then go to nonplanar;
            SAVE:=NEXT(R');
            NEXT(R'):=NEXT(L');
            NEXT(L'):=SAVE;
            SAVE:=NEXT(x);
            NEXT(x):=NEXT(y);
            NEXT(y):=SAVE;
            L':=y;
            R':=x;
         end else begin comment STACK(NEXT(R'))>w;
            L':=x;
            R':=y; end
```

```
end else if (x,y) on B has x≠0 then begin comment
        STACK(NEXT(L'))>w;
        SAVE:=NEXT(x);
        NEXT(x):=NEXT(R');
        NEXT(R'):=NEXT(L');
        NEXT(L'):=SAVE;
        R':=x; end
   else if (x,y) on B has y ≠ 0 then R' = y;
   delete (x,y) from B;
end;
comment add P to left stack if p is normal;
if f(PATH(s)) < w then begin
   if L'=0 then L':=FREE;
   STACK(FREE):=f;
   NEXT(FREE):=NEXT(0);
   NEXT(0):=FREE;
   FREE:=FREE+1;
end;
comment Add new block corresponding to combined old
   blocks.  New block may be empty if segment con-
   taining current path is not a single frond;
if R'!= -1 then R':=0;
if (L'≠0) or (R'≠0) or v≠s then add (L',R') to B;
comment if segment containing current path is not
   a single frond, add an end-of-stack marker to
   right stack;
```

```
if v≠s then begin
    STACK(FREE):=0;
    NEXT(FREE):=NEXT(-1);
    NEXT(-1):=FREE;
    FREE:=FREE+1;
  end;
  s:=0;
end; end;
  comment initialization;
  NEXT(-1):=NEXT(0):=0;
  FREE:=1;
  STACK(0):=0;
  B:= the empty stack;
  p:= s:=0;
  PATH(1):=1;
  comment vertex 1 is the start vertex of the search;
  PATHFINDER(1);
end;
```

Lemma 13: EMBED correctly tests the planarity of a graph G.

Proof: EMBED is a straightforward implementation of the pathfinding and embedding algorithms described in Sections 5 and 6.

Lemma 14: EMBED requires O(V+E) time to test a graph with V vertices and E edges.

<u>Proof</u>: The pathfinding part of the algorithm requires O(V+E) time since it is a depth-first search with a few additional calculations. Only a finite amount of computation is performed by EMBED before some entry is added to L, R, or B, or deleted from L, R, or B. The total number of entries made to L, R, and B, is O(V+E). Thus the stack calculations require O(V+E) time. Initialization requires finite time, so the enitre algorithm requires O(V+E) time.

<u>Lemma 15</u>: The planarity algorithm requires O(V) time to test the planarity of a graph with V vertices.

<u>Proof</u>: The algorithm stops if the number of edges of G exceeds 3V-3, so counting the edges takes O(V) time. If G has O(V) edges, the initial depth-first search requires O(V) time, the sorting of edges using LOWPT values requires O(V) time, and the pathfinding/embedding algorithm requires O(V) time. Thus the total time is O(V). It is also easy to see that the algorithm requires O(V) storage space.

8. <u>Implementation and Experiments</u>

A more complicated version of the planarity algorithm w
programmed in Algol W, the Stanford University version of Alg
and run on an IBM 360/67. A program listing and a more compl
discussion appear in [29]. The program was extensively teste
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.

These test graphs were generated by starting with a complete graph of three vertices. 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. 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, the test graphs seemed to give the planarity program a satisfactory workout.

The test results indicated that for this class of graphs

$$T = .0125V - .07$$

where T is the running time of the program 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 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 also expected that implementing the simpler algorithm presented here would cut down the space and time requirements considerably.

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 [21] required about 30 seconds to process a 28 vertex planar graph using an IBM 360/65. The algorithm presented here required 0.4 seconds to test 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.

9. Applications and Conclusions

The planarity algorithm as described here only tests a graph G for planarity; it does not construct a planar representation of G. However the algorithm collects enough information to make the construction of a planar representation easy, and the algorithm may be modified slightly to carry out this step. One way to accomplish this is to construct a *dependency graph* D. The vertices of the dependency graph will correspond to the paths found in G. Two paths will be joined by an edge if the position of one path determines the position of the other path. Edges are of two types, depending upon whether the paths must be embedded on the same side or on opposite sides of the appropriate cycle. A coloring of the vertices of D with two colors in a way which satisfies the edge constraints then corresponds to an embedding of G. For details see [29]. Figure 9 shows a dependency graph and an embedding for the graph in Figures 3 and 5. It is easy to modify the embedding algorithm so that it constructs and colors a dependency graph. This modified algorithm may be used to accomplish tasks such as laying out electronic circuit boards.

The planarity algorithm may be combined with algorithms for determining connectivity properties of graphs [5,6,8] and with an algorithm of Hopcroft's [15,17] to test isomorphism of triconnected planar graphs, to give an algorithm for determining whether two arbitrary planar graphs are isomorphic [7,16]. This algorithm requires $O(V \log V)$ time and $O(V)$ space. The algorithm promises to be of value to chemists, since most molecules may be represented as planar graphs. A canonical form for molecules which follows from the isomorphism algorithm might be used to speed searches of the chemical literature. The algorithm may also be used for enumeration of various types of planar graphs. (See Grace [43] for instance.)

The planarity algorithm and the other algorithms which use depth-first search illustrate the value of this technique as an efficient, systematic method for exploring graphs. The planarity algorithm also illustrates the value of carefully chosen data structures. These ideas may find application in efficient algorithms for solving many other problems.

REFERENCES

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

[2] 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 Research Logistics Proj., Contract No. NONR 1858-(21), Dept. of Math., Princeton University (May 16-18, 1963), 2 unno. pp.

[3] Golumb, S.W. and Baumert, L.D., "Backtrack Programming," JACM, Vol. 12, No. 4 (Oct., 1965), 516-524.

[4] Nilsson, N., "Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

[5] Hopcroft, J. and Tarjan R., "Efficient Algorithms for Graph Manipulation," STAN-CS-71-207, Computer Science Department, Stanford University (March, 1971).

[6] Tarjan, R., "Depth-First Search and Linear Graph Algorithms," SIAM J. Comput., Vol. 1, No. 2 (June, 1972), pp. 146-159.

[7] Hopcroft, J. and Tarjan, R., "Isomorphism of Planar Graphs," Complexity of Computer Computations, R.E. Miller and J. W. Thatcher (eds.), Plenum Press, New York, 1972, pp. 143-150.

[8] Hopcroft, J. and Tarjan, R., "Dividing a Graph into Triconnected Components", SIAM J. Comput., to appear.

[9] Tarjan, R., "Finding Dominators in Directed Graphs," unpublished, Cornell University (December, 1972).

[10] Hecht, M.S. and Ullman, J.D., "Flow-Graph Reducibility," SIAM J. Comput., Vol. 1, No. 2 (June, 1972), pp. 188-202.

[11] Tarjan, R., "Testing Flow Graph Reducibility," TR 73-159, Dept. of Computer Science, Cornell University (December, 1972).

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

[13] Sites, R.L., "Algol W Reference Manual," STAN-CS-71-230, Computer Science Department, Stanford University (August, 1971).

[14] 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 Nos. G 81-60, NASA CR 68898, STAR No. N-66-14074 (December 15, 1965).

[15] Hopcroft, J., "An $n \log n$ Algorithm for Isomorphism of Planar Triply Connected Graphs," STAN CS-71-192, Computer Science Department, Stanford University (January, 1971).

[16] Hopcroft, J. and Tarjan, R., "A V^2 Algorithm for Determining Isomorphism of Planar Graphs," Information Processing Letters, Vol. 1, No. 1 (1971), pp. 32-34.

[17] Hopcroft, J. and Tarjan, R., "A $V \log V$ Algorithm for Isomorphism of Triconnected Planar Graphs," JCSS, to appear.

[18] 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 (October, 1965), pp. 733-744.

[19] 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 (October, 1965), pp. 913-929.

[20] 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), pp. 142-148.

[21] Bruno, J., Steiglitz, K., and 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), pp. 197-206.

[22] Chung, S.H., and 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, 1970), VII. 4.1 - VII. 4.12.

[23] 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), pp. 154-163.

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

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

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

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

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

[29] Tarjan, R., "An Efficient Planarity Algorithm," STAN-CS-244-71, Computer Science Dept., Stanford University (November, 1971).

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

[31] Wing, O., "On Drawing a Planar Graph," I.E.E.E. Tra[illegible]tions on Circuit Theory, Vol. CT-13, No. 1 (March, 1966), pp. 112-114.

[32] Youngs, J.W.T., "Minimal Imbeddings and the Genus of a Graph," Journal of Mathematics and Mechanics, Vol. 12, No. 2 (1963), pp. 303-315.

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

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

[35] Berge, C., The Theory of Graphs and its Applications, translated by Alision Doig, Methuen and Co., Ltd., London, 1964.

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

[37] Harary, F., Graph Theory, Addison-Wesley, Reading, Massachusetts, 1969.

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

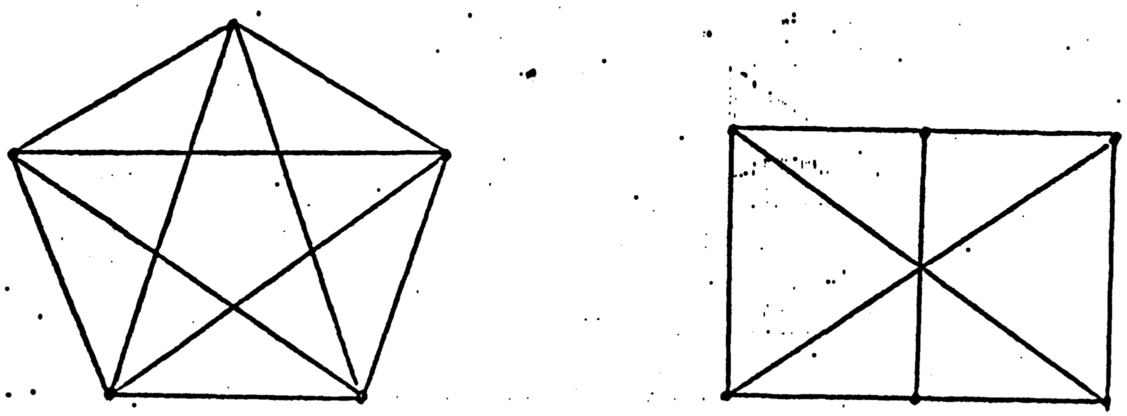
[39] Tutte, W.T., Connectivity in Graphs, Oxford University Press, London, 1966.

[40] Hall, D.W. and Spencer, G., Elementary Topology, Wiley and Sons, Inc., New York, 1955.

[41] Thron, W.J., Introduction to the Theory of Functions of a Complex Variable, Wiley and Sons, Inc., New York, 1953.

[42] Holt, R.C. and Reingold, E.M., "On the Time Required to Detect Cycles and Connectivity in Directed Graphs," TR 70-63, Department of Computer Science, Cornell University (June, 1970).

[43] Grace, D.W., "Computer Search for Non-Isomorphic Convex Polyhedra," Technical Report CS15, Computer Science Dept., Stanford University (January, 1965).



K_5 $K_{3,3}$

Figure 1: The Kuratowski Subgraphs

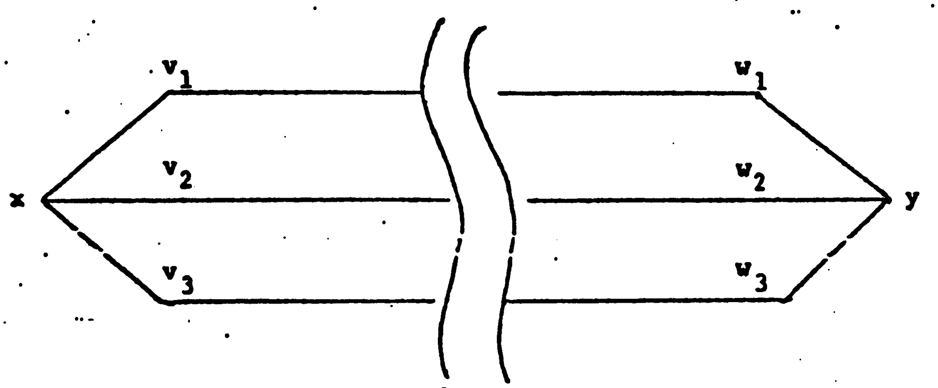


Figure 2: Illustration of Lemma 2. Path p_2 is inside the simple closed curve formed by p_1 and p_3.

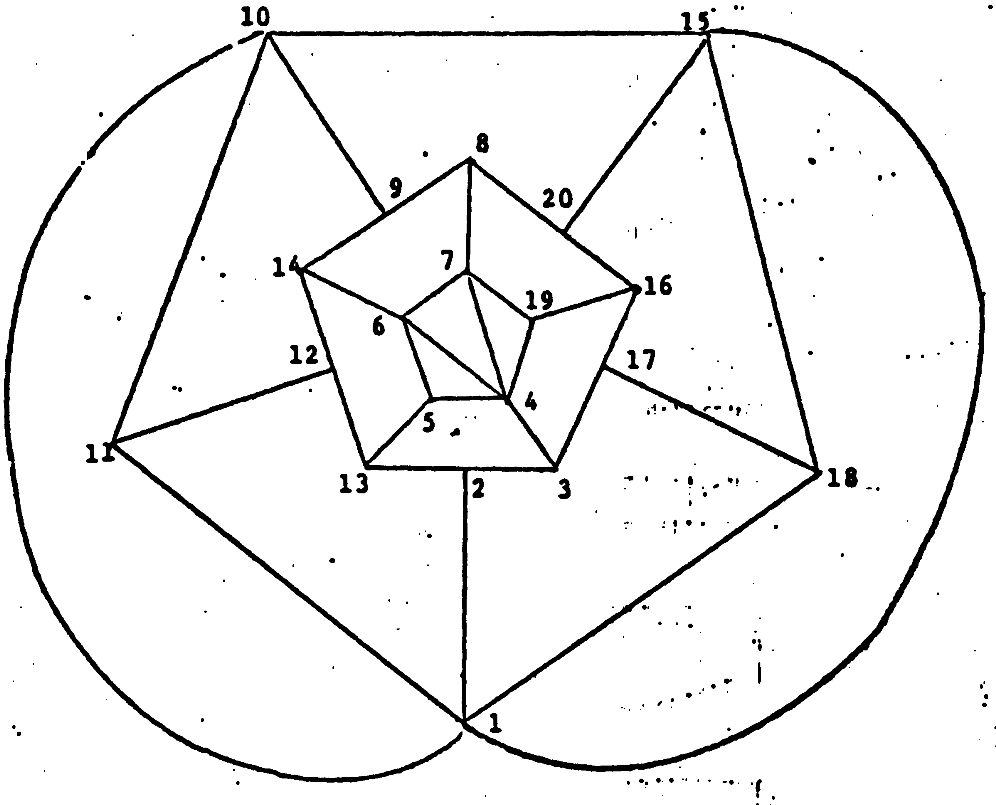
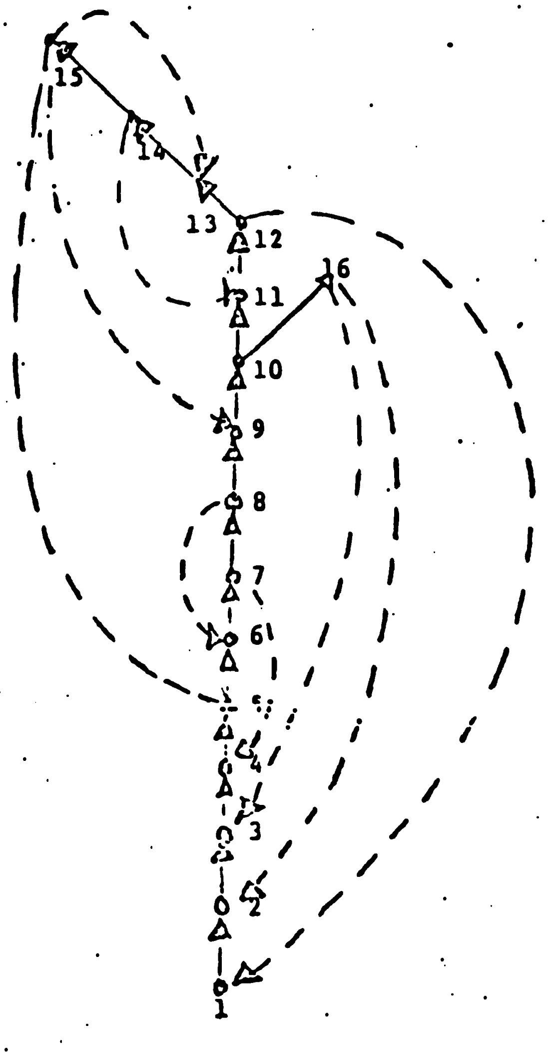


Figure 3: (a) A graph G to be tested for planarity.



<u>Figure 3:</u> (b) A palm tree P generated from G. Upward, solid edges are tree arcs. Downward, dotted edges are fronds. Vertices are numbered in search order.

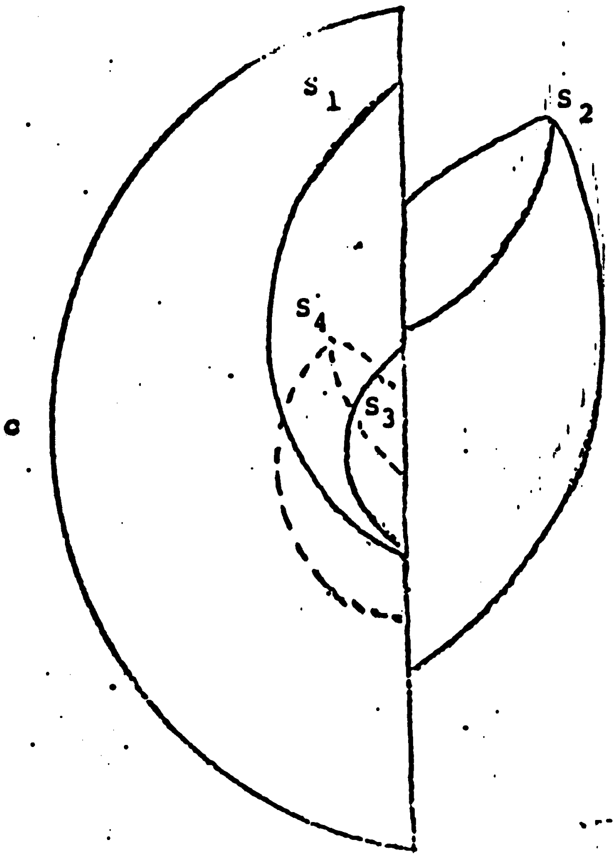


Figure 4: Conflict between pieces. To add dotted piece S_4 on the inside of c and maintain planarity, pieces S_1 and S_3 must be moved from the inside to the outside. Piece S_2 must be moved from the outside to the inside.

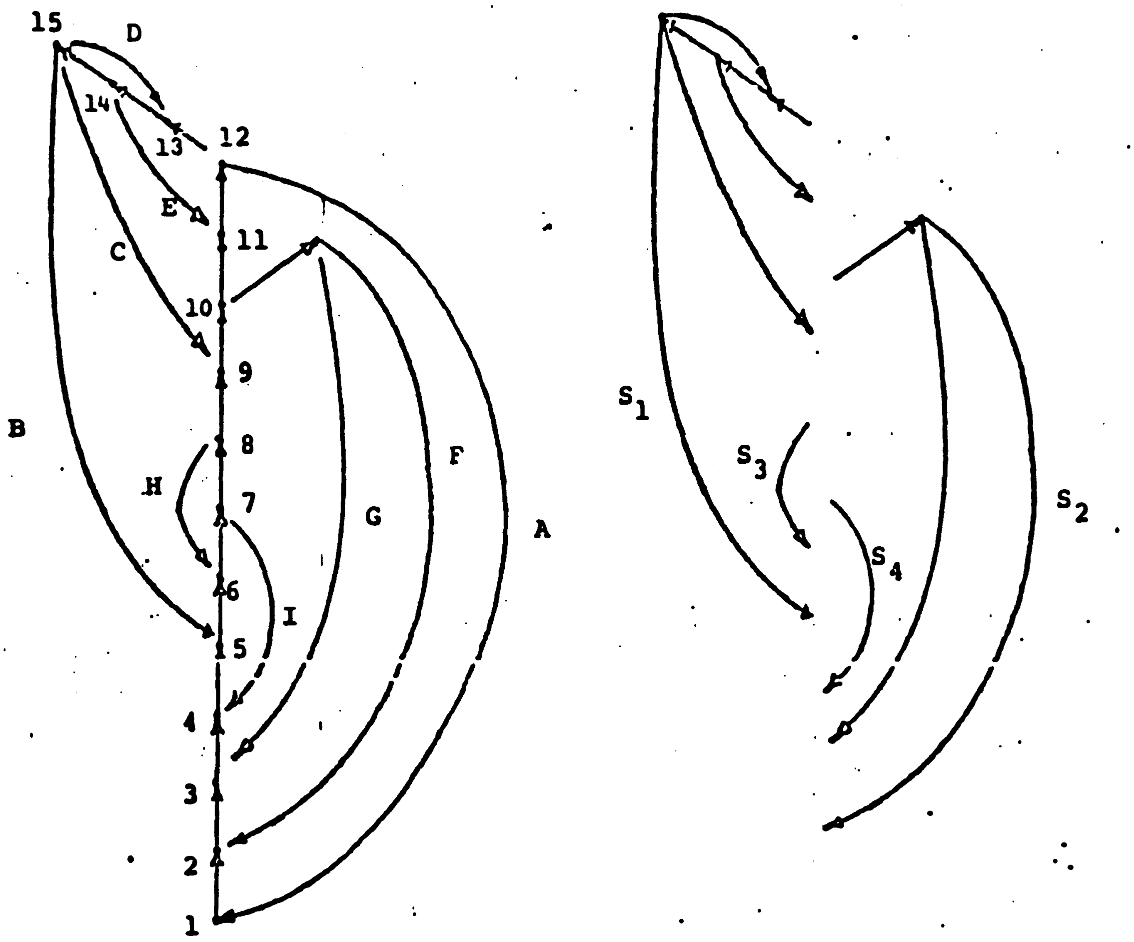


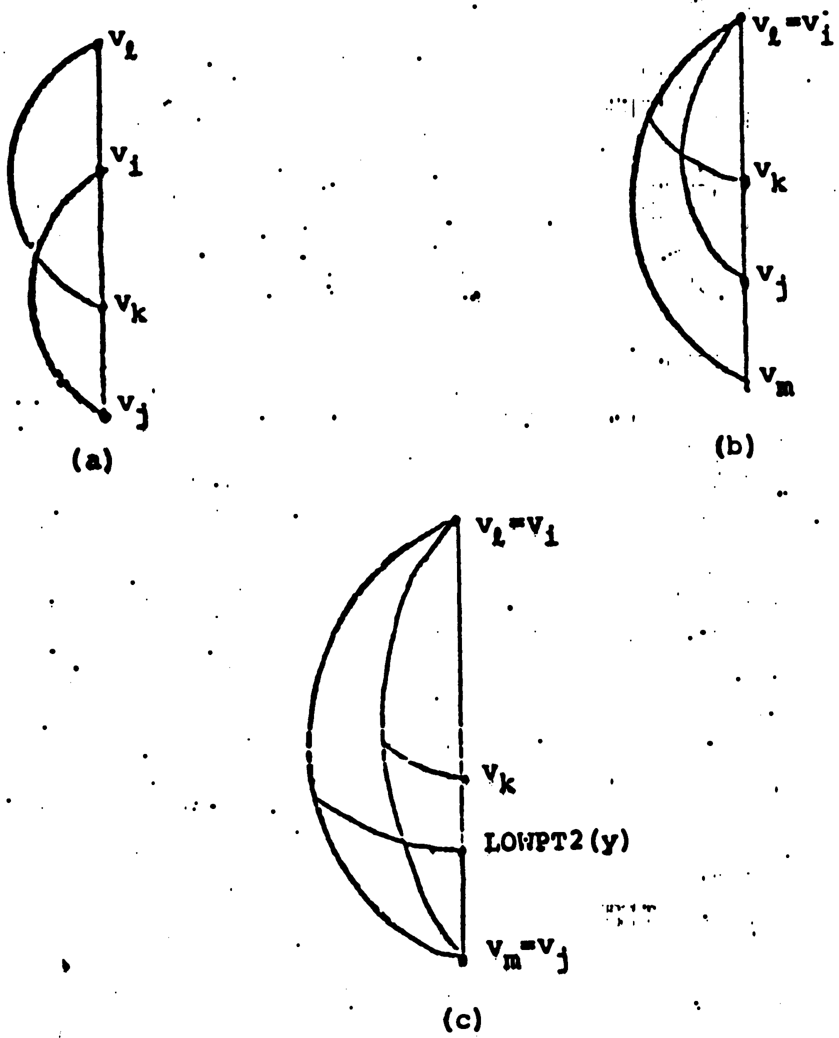
Figure 5: (a) Paths generated by PATHFINDER from the graph in Figure 3.

A: (1,2,3,4,5,6,7,8,9,10,11,12,1) B: (13,14,15,5)

C: (15,9) D: (15,13) E: (14,11) F: (10,16,2)

G: (16,3) A: (8,6) I: (7,4)

(b) Segments with respect to initial cycle.



<u>Figure 6</u>: Illustration of the cases in Lemma 9. Path to be embedded leads from v_i to v_j. Blocking segment leads from v_ℓ to v_k to v_m.
(a) $v_\ell > v_i$.
(b) $v_\ell = v_i$ and $v_m < v_j$.
(c) $v_\ell = v_i$ and $v_m = v_j$. (Note: the order of v_k and LOWPT2(y) is undetermined).

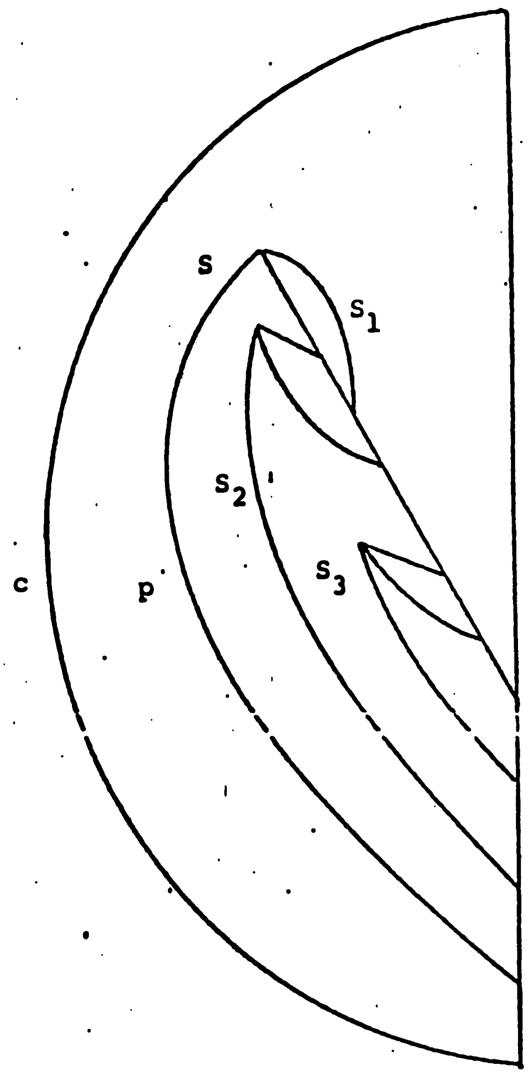


Figure 7: Recursive application of the embedding algorithm. Segment S consists of first path p and new segments S_1, S_2, S_3.