PLANARITY TESTING IN V LOG V STEPS: EXTENDED ABSTRACT

BY

JOHN HOPCROFT

ROBERT TARJAN

STAN-CS-71-201

FEBRUARY, 1971

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

PLANARITY TESTING IN V log V STEPS:

EXTENDED ABSTRACT

John Hopcroft

Robert Tarjan

Stanford University, Stanford, California

Abstract

An efficient algorithm is presented for determining whether or not a given graph is planar. If V is the number of vertices in the graph, the algorithm requires time proportional to V log V and space proportional to V when run on a random-access computer. The algorithm constructs the facial boundaries of a planar representation without backup, using extensive list-processing features to speed computation. The theoretical time bound improves on that of previously published algorithms. Experimental evidence indicates that graphs with a few thousand edges can be tested within seconds.

This research was supported by the Advanced Research Projects Agency of the Office of the Department of Defense, the Atomic Energy Commission, the Hertz Foundation, the National Science Foundation, and the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR 044-402.

PLANARITY TESTING IN V log V STEPS:

EXTENDED ABSTRACT

John Hopcroft

Robert Tarjan

Stanford University, Stanford, California

Introduction

The problem of embedding a graph in a plane arises in several fields. In engineering, discovering whether a given circuit may be laid out in a plane is of interest in integrated circuit design; in chemistry, determining isomorphism of chemical structures may be made much easier if the structures are planar. The earliest characterization of planar graphs was given by Kuratowski [5], who showed that every non-planar graph contains a subgraph which upon removal of degree two vertices is isomorphic to one of the graphs in Figure 1. However, searching for such subgraphs directly may require an amount of time at least proportional to V^6, if not much worse, where V is the number of vertices in the graph. It is clear that more efficient procedures are needed to analyze large graphs.

The planarity problem has attracted numerous researchers and many algorithms have been described in the literature [1, 2, 3, 6, 7]. Surprisingly little work has been directed toward a rigorous analysis of their running times, however, and algorithms which are obviously inferior to previously published algorithms continue to be published. Shirey [7] has grouped a number of the better methods into what he calls the Goldstein approach. Using list processing and programming tricks, he has proved an asymptotic time bound of V^3 for his variation of the algorithm. The only other competitive algorithm is due to Lempel, Even, and Cederbaum [6].

Shirey indicates a probable time bound of V^3 for their algorithm, although neither he nor the originators discuss implementation. Tarjan has programmed the L.E.C. algorithm, giving a time bound of V^2, although without proof [8]. We now have a proof of this time bound, and we also believe that Goldstein's algorithm, if implemented optimally, will run in time V^2. However, our proposed algorithm is even faster, and it includes several procedures for graph manipulation which are interesting in their own right.

K_5 $K_{3,3}$

Figure 1: Kuratowski subgraphs

General Description

Let $G = (V, \mathcal{E})$ be an undirected graph, where V is a set, called the vertices, and $\mathcal{E}$ is a set of unordered pairs of vertices, called the edges. Let V be the number of vertices, and E the number of edges. We assume that G contains no loops and no multiple edges. Let G be planar, and suppose it is drawn in the plane. The connected sets of points in the plane formed when the edges and vertices of G are deleted from this representation are called the faces $\mathcal{F}$ of the graph. If there are F faces, then V+F = E+2 [4]. It follows from this familiar result that any planar

graph satisfies the inequality $E \leq 3V - 6$. Thus we may restrict our consideration to graphs having $3V - 6$ or less edges; other graphs may be immediately classified as non-planar.

The planarity algorithm operates by dividing G into biconnected components, and attempting to construct a planar representation for each biconnected component (G is planar if and only if all of its biconnected components are planar [4]). Given a component, the facial boundaries of a planar representation are constructed in the following manner: first a cycle (a simple closed path) is found. All the points on this cycle are marked as old, and a planar representation of the cycle is constructed. Next a simple path joining two old points is found and added to the representation. The new path either divides a face of the current representation into two new faces, or else makes the graph nonplanar. New paths are added until the entire component has been represented or until a path which may not be added is found.

We encounter one difficulty in adding paths to a partially constructed representation; namely, we may find it possible to put a path into the interior of several different faces. If a choice is made arbitrarily for such a path, then at some later time we may find a path which cannot be added to the current representation, but may be added if a different choice had been made at a previous step. If the algorithm is to be efficient, this possibility of alternate choices must be eliminated. A special part of the algorithm is designed to look ahead to discover which selection is necessary in order to represent the graph in the plane, if such a representation is possible. Thus a path is not added to the representation until either the face it must divide is known exactly, or the decision is known to be arbitrary.

One more step is crucial in decreasing the running time of the algorithm. Each path is selected to start from a vertex of small degree in the current representation; in particular, from a vertex of degree five or less. If no such new path exists (because all edges from vertices of degree five or less are already used) then a vertex of degree five or less and all adjacent edges are marked as deleted. This device decreases the running time of the algorithm from kV^2 to $k\ V \log V$.

List processing techniques are used extensively to speed the algorithm. The graph is represented as a set of lists of edges. A list of adjacent edges is given for each point; in addition, since each edge appears twice, both occurrences of an edge have pointers to each other. This facilitates deletion of edges from the graph. We will consider separately and in detail the various parts of the algorithm, but due to space limitation we omit complete proofs of correctness and of the time bounds. These will appear in a Stanford Computer Science Technical Report. The parts are the routine for finding biconnected components, the routine for finding paths, the routine for deciding where to add a path, and the routine for building a planar representation. The total time required by each of these algorithms except the last is proportional to the number of edges in the graph. The representation-building algorithm requires time proportional to $V \log V$. Thus the total algorithm has a theoretical time bound of $k\ V \log V$ for some k . Further, the storage space required by the algorithm is proportionalto V .

Finding Biconnected Components

We break a graph into its biconnected components by performing a depth-first search along the edges of the graph. Each new vertex reached is placed on a stack, and for each vertex a record is kept of the lowest vertex on the stack to which it is connected by a path of unstacked vertices. When no new vertex can be reached from the top of the stack, the top vertex is deleted, and the search is continued from the next vertex on the stack. If the top vertex does not connect to a vertex lower than the second vertex on the stack, then this second vertex is an articulation point of the graph. All edges examined during the search are placed on another stack, so that when an articulation point is found the edges of the corresponding biconnected component may be retrieved and placed in an output array.

When the search is exhausted, a complete search of a connected component has been performed. An unreached vertex is selected as a new starting point and the process is repeated until the entire graph has been examined. Isolated vertices are not listed as biconnected components, since they'have no adjacent edges. They are merely skipped. The details of the algorithm are given in Figure 2. Note that the flowchart gives a non-deterministic algorithm, since any new edge may be selected in block A. The actual program is deterministic; the choice of an edge depends on the particular representation of the graph. The algorithm requires less than $k \max(V,E)$ steps, for some suitable k, since only a finite amount of manipulation is performed with each vertex and each edge. Since $E \leq 3V-6$ for planar graphs, the time bound is $k_1 V$ for a suitable k_1. The amount of space required is also proportionalto V.

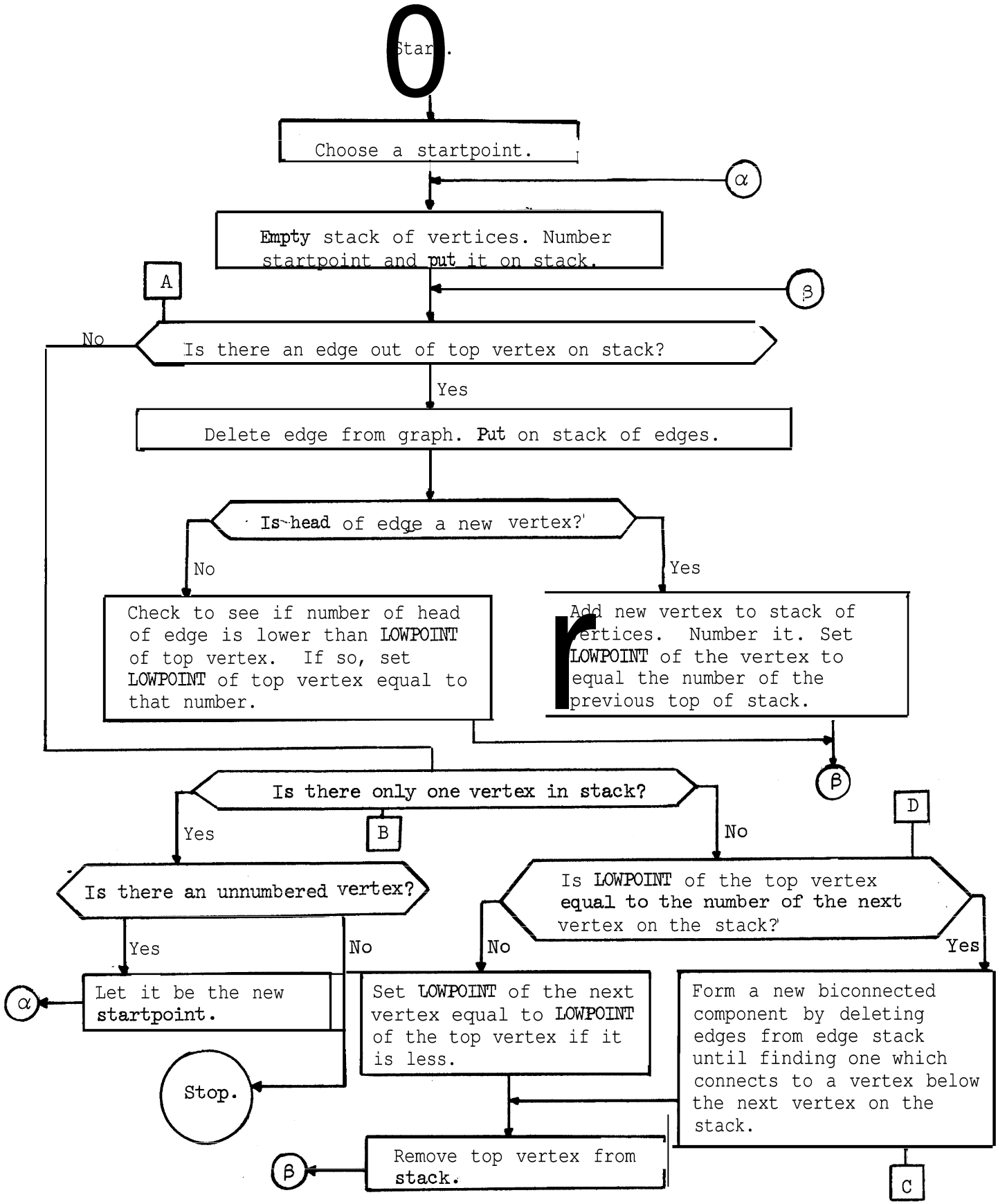

Figure 2: Flowchart for Biconnected Components Algorithm.

Finding Paths

The path finding algorithm finds the paths used to build the graph in the plane. The starting vertex for each successive path may be chosen arbitrarily; in fact, the initial edge of each successive path may be selected arbitrarily from the set of unused edges. The algorithm is highly dependent on the graphs being biconnected. In order to find a new path, the initial edge is selected and the head of the edge is checked. If this vertex is old, the path consists of a simple edge. If the vertex has never been reached before, a depth-first search is begun which must end in a path since the graph is biconnected. The search generates a tree-like structure; specifically, it is a tree with extra edges connecting some nodes with their (not necessarily immediate) ancestors. (We will visualize the tree drawn so that the root, which is an ancestor of all vertices, is at the bottom of the tree.) Enough information is saved from this tree so that if a vertex in *it* is reached when building another path, the path *may* be completed without any further search.

The flowchart (Figures 3 and 4) gives the details of the algorithm. It is divided *into* two parts: one for the depth-first search process and one for path construction using previously gathered information. Let us consider path generation using depth-first search; that is, suppose the algorithm is applied and that the head of the first edge selected is previously unreached. Referring to the flowchart, we see that the search process is very similar to that used in the biconnectivity algorithm. A search tree is generated, and each edge examined is either part of the tree or connects a vertex to one of its predecessors in the tree. LOWPOINT is a variable which gives the number of the lowest vertex in the tree reachable from a given vertex by continuing out along the tree and taking

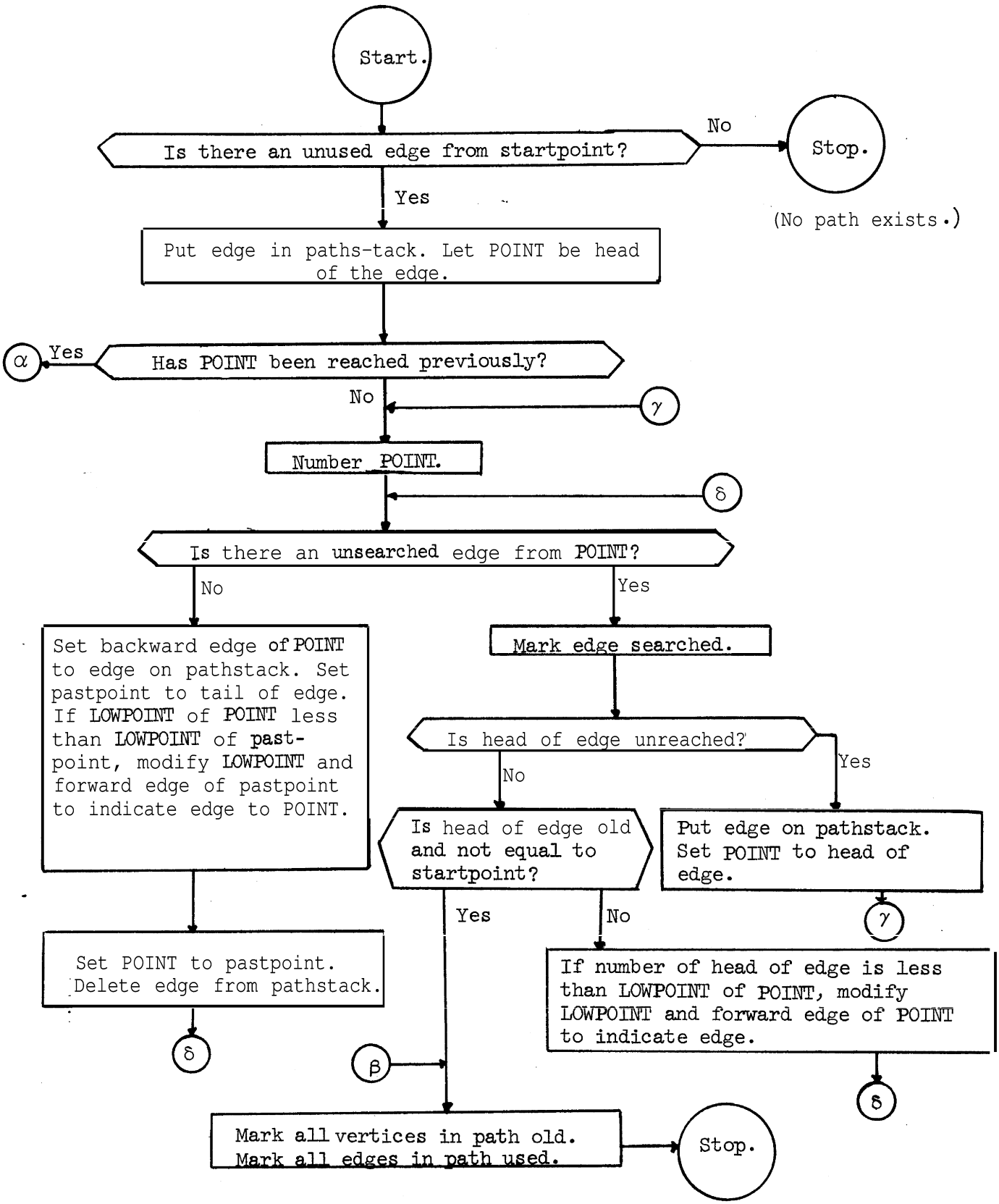

Figure 3: Flowchart for Pathfinding Algorithm (I)

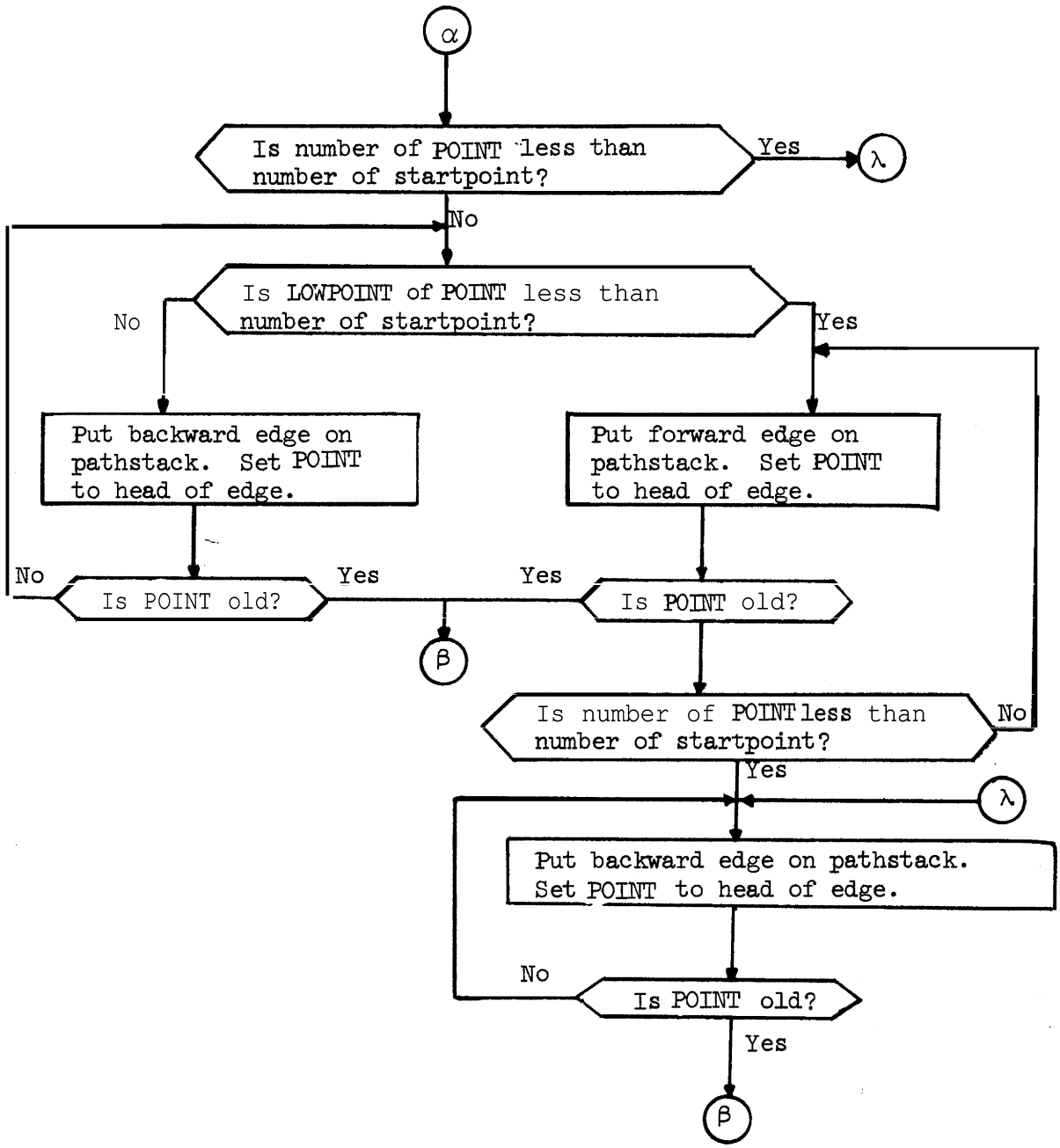

Figure 4: Flowchart for Pathfinding Algorithm (II)

one edge back toward the root. The forward edges point along this path, while the backward edges point back along the tree branches. The depth-first search used here is exactly the same as in the biconnectivity algorithm and because the graph is biconnected, LOWPOINT of a given vertex must point to a node which is an ancestor of the immediate predecessor of the given vertex. In particular, LOWPOINT of the second vertex in the search tree must indicate an old vertex which is not the startpoint. Therefore the algorithm will find a path containing the initial edge. Note that all vertices encountered during the search process must either be old or unreached, since every vertex reached in a previous search either has had all its edges examined or has been included in a path.

Let us now suppose that the head of the first edge has been reached previously but is not marked old. Then the forward and backward pointers, along with the LOWPOINT values, allow the algorithm to construct a path without further search. There are several possible cases. First, if the number of the head of the initial edge is less than the number of the startpoint, then following backward edges will certainly produce a simple path, since the numbers of vertices along such a path decrease and the root of every search tree is an old vertex. If the initial edge is part of the search tree and the startpoint is the predecessor of the second vertex, then LOWPOINT of the second vertex must be less than the number of startpoint. Following forward edges until reaching a vertex lower than the startpoint and then following backward edges will produce a simple path. The last case to consider occurs when the initial edge is not part of the search tree but points from a vertex to one of its descendents in the tree. In this case some vertex in the tree between the startpoint and the second vertex of the path must have a LOWPOINT value less than the number of the

startpoint. If we follow backward edges until reaching the first such vertex, then follow forward edges until a vertex numbered less than the startpoint is reached, and finally follow backward edges until an old vertex is reached, we will generate a simple path.

To derive a time bound for the algorithm, we assume that one vertex is marked old initially, and a different vertex is selected as the initial startpoint. The algorithm is then run repeatedly with arbitrary startpoints until all edges are used to form paths. Each execution of the algorithm produces a simple path, if the graph is biconnected. Since each edge is examined at most once in the search section of the algorithm, and since each edge is put into a path once, the time required to execute the algorithm until no edges are unused is proportionalto E . Since $E \leq 3V - 6$, for some k , the time required is less than kV . The space used is also proportionalto V .

Resolving Ambiguity in Adding Paths

In this section we describe the algorithm which determines the vertices from which new paths should be started and which supplies the representation building algorithm with a sequence of paths along with sufficient information to resolve ambiguities as to which face should be divided. First a cycle is found and then a path p_1 is found connecting two vertices on the cycle. The cycle is then considered to consist of two paths between the two vertices.

The following sequence of steps is repeated until every edge of the graph lies in some path or the graph is found to be nonplanar. (In the process a pushdown store is maintained of all paths which may divide more than one face along with a list of paths found but not yet given to the

representation building algorithm. Along with each path on the pushdown store is a pointer to a copy of the path in the list of paths not yet given to the representation building algorithm.)

The path finding algorithm is asked for a path p_i from a vertex v_1. Let $\bar{v}_i$ denote the last vertex in path p_i. A check is made to see if there exists a path already found containing both v_i and $\bar{v}_i$. (Vertices on paths, with the exception of the first and last vertices, are numbered sequentially so that this test can be performed in a finite number of steps per path independent of path length.) If there is no path containing both v_i and $\bar{v}_i$, then the path p_i divides a unique face. If the pushdown store is empty, the path is given to the representation building algorithm. Otherwise the path is placed on the list of paths since even though it divides a unique face we must first put in all previously found paths and one of these is waiting on the pushdown store since it divides more than one face. In either case, the paths containing v_i and $\bar{v}_i$ are divided in two at the vertices v_1 and $\bar{v}_i$.

If on the other hand the path starts and ends at vertices on the same path, then an ambiguity as to which face should be divided exists. Whenever such a situation arises, the algorithm immediately tries to resolve the ambiguity. Assume path p_i starts at vertex v_1 and ends at vertex 6_i where v_i and $\bar{v}_i$ are both on path p_j. Then the path p_i is stored on the pushdown list and the path finding algorithm is directed to start paths from vertices on path p_i or vertices between v_i and $\bar{v}_i$ on path p_j. Let the next path found be p_k. One of four situations arises.

i) The last vertex of path p_k is on the same path as the first vertex of p_k . In this case, a new ambiguity exists and p_k is added to the pushdown list. The process of resolving the ambiguity of p_i is interrupted and the algorithm starts to resolve the ambiguity of p_k .

ii) The last vertex of path p_k is on a path other than p_i or p_j . In this case the endpoint of p_k uniquely determines the face to be divided by p_i and is called the resolving point of path p_i . The path p_i is removed from the pushdown store. If the pushdown store is empty, then all paths on the list of paths can be given to the representation building algorithm. If the pushdown store is not empty, then the top path is a path which we were trying to resolve but were interrupted in the process. We return again to this process.

iii) If p_k connects p_j to a vertex on p_i between v_i and $\bar{v}_i$, then p_k divides a unique face but does not resolve the ambiguity of p_i . We simply place p_k on the list of paths and continue trying to resolve p_i .

iv) If p_k ends at a vertex on p_i but outside the portion of the path from v_i to $\bar{v}_i$, then the situation is similar to case iii but the set of vertices from which we can start paths to resolve p_i must be extended.

Space does not permit a more detailed analysis and many details have been omitted. For example, the test whether there exists a path 'containing both v_i and $\bar{v}_i$ takes time proportionalto the degree of the vertex v_i . This would add an additional factor of V to the running time were it not for the fact that the computation is arranged so that all paths are started from vertices which appear to be of degree 5 or less. The interested reader

is referred to the more detailed treatise of the algorithm along with its ALGOL implementation and an analysis of the running time which will appear in a Stanford Computer Science Technical Report.

Building a Planar Representation

We store the part of the graph which has been examined as a set of cycles, one corresponding to each face of a planar representation of the graph. Each cycle is a doubly linked circular list of the vertices on the boundary of the face. Corresponding to each vertex is a list of cycles in which the vertex appears. The cycle-building process, when given a path starting from a vertex of degree five or less, and when also (possibly) given a resolving point for the path, searches the five (or less) cycles containing the startpoint of the path, in both directions, looking for the endpoint of the path. The resolving point is also noticed if found. If the endpoint is not found in any of these cycles, the graph is nonplanar. If the endpoint is found, the cycles in which it appears are searched for the resolving point, if such a search is necessary to resolve ambiguity concerning which cycle should be divided. The selected cycle is then split at the start and end points of the path, the path is added to each piece, and the construction of the new cycle is complete. If a resolving point is used, its location is saved until it is an endpoint of a path, so that unnecessary searches are not made. In addition, cycles which are known to need no further dividing are removed from the vertex lists. Such cycles occur when points are deleted because they have five or fewer incident edges, all of which have been added to the planar representation.

The theoretical execution time of the planarity algorithm is dominated by the time necessary to search the cycles containing the startpoint of a path for the endpoint and resolving point. We consider the time necessary to search the particular cycle containing the endpoint and the resolving point. Since the same amount of time is spent searching each cycle containing the startpoint, the total time is at worst five times this quantity. To bound the time we will consider searching for endpoints and resolution points separately. The time spent searching for both simultaneously must be no greater than the time spent in separate searches.

We need a lemma, the proof of which is omitted.

Lemma: (a) $x \log x + y \log y + \min(x,y) \leq x + y \log(x+y) \qquad 1 \leq a,x,y$

(b) $(x+a) \log(x+a) + (y+a) \log(y+a) + \min(x,y) \leq (x+y+2a) \log(x+y+2a)$

(c) $(x+a) \log(x+a) + y \log y \leq x \log x + (y+a) \log(y+a) \quad$ if $x \leq y$.

Consider the time spent looking for endpoints. Assume that at some step we have cycles of length $\ell_1, \ldots, \ell_m$ and that k vertices do not yet appear in cycles. Suppose $\ell_m \geq \ell_i$ for all $1 \leq i \leq m-1$. We claim that the search time from this step on is bounded by

$$T(\ell_1, \ell_2, \text{[illegible]} \sum_{i=1}^{m-1} \ell_i \log \ell_i + (\ell_m + 2k) \log(\ell_m + 2k) .$$

Since m = 0 and k = V initially, this will give a total time bound of v log v . (To simplify matters, we neglect constants of proportionality throughout this discussion.) We can prove the bound by induction on k . For fixed k , we use induction on m .

Assume a path of length a is to be added to the representation. Without loss of generality, we may assume that the endpoint being considered is either in cycle 1 or in cycle m. Suppose it appears in cycle 1 and the

cycle is to be broken into two pieces of lengths ℓ_1-b and b and the path added to each piece to form two new cycles. Then the time from this step is

$$T' = \min(\ell_1\text{-}b, b) + T(\ell_1\text{-}b+a, a+b, \ell_2, \ell_3 \ldots \ell_m k-a)$$

$$\leq T(\ell_1, \ell_2, \ldots, \ell_m, k)$$

since

$$\min(\ell_1\text{-}b, b) + (\ell_1\text{-}b+a)\log(\ell_1\text{-}b+a) + (a+b)\log(a+b) + (\ell_m+2k-2a)\log(\ell_m+2k-2a)$$

$$\leq (\ell_1+2a)\log(\ell_1+2a) + (\ell_m+2k-2a)\log(\ell_m+2k-2a)$$

$$\leq \ell_1 \log \ell_1 + (\ell_m+2k)\log(\ell_m+2k) \qquad \text{by Lemma 1.}$$

If the endpoint appears in cycle m, then either ℓ_m-b+a $\geq \ell_i$ for $1 \leq i < m$ or there exists j such that $\ell_j > \ell_m$-b+a and $\ell_j \geq \ell_i$ for $1 \leq i < m$. In these cases we may prove $T' \leq T$ as above, using Lemma 1. Thus we have the desired time bound, since T after the last stop is 0.

Now consider the time necessary to find resolving points. After finding a resolving point, we do not search for another until a path which has this resolving point as one of its endpoints is added to the representation. Then a path not in the original cycle exists between the startpoint and the resolving point. Thus the original cycle is by this time divided at the startpoint and resolving point. The analysis above thus also gives a time bound on searching for resolving points. Thus the total time required to construct a planar representation is k V log V, for some constant k, The representation-building algorithm also uses space proportional to V.

Experimental Results

The planarity algorithm has been programmed in ALGOL W for the Stanford 360/67 and tested on a number of graphs. Experimental evidence indicates that memory space is the limiting factor and the algorithm must be reprogrammed if it is to handle graphs with more than 2000 edges. The current version of the algorithm consists of 985 lines of ALGOL. Graphs with 100 edges are handled in less than one second. A planar graph with 1000 vertices and 2000 edges was run in 12.7 seconds. Nonplanar graphs may have somewhat smaller running times since the algorithm stops as soon as a nonplanar subgraph is discovered.

References

[1] Auslander, L., Parter, S. V., "On embedding graphs in the sphere", Journal of Mathematics and Mechanics, Vol. 10, No. 3 (1961), pp. 517-523.

[2] Bruno, J., Steiglitz, K., Weinberg, L., "A new planarity test based on 3-connectivity", IEEE Transactions on Circuitry Theory CT 17: 2, May 1970, pp. 197-206.

[3] Goldstein, A. J., "An efficient and constructive algorithm for testing whether a graph can be imbedded in a plane", Graph and Combinatorics Conference, Princeton University, May 1963.

[4] Harary, Frank, Graph Theory, Addison-Wesley Publishing Co., Reading, Massachusetts, 1969.

[5] Kuratowski, Casimir, "Sur le probleme des courbes gauches en topologie", Fundamenta Mathematicae, Vol. 15 (1930), pp. 271-283.

[6] Lempel, A., Even, S., and Cederbaum, I., "An algorithm for planarity testing of graphs", in Theory of Graphs: International Symposium: Rome, July 1966. P. Rosenstiehl, Ed., New York: Gordon and Breach, 1967, pp. 215-232.

[7] Shirey, R. W., "Implementation and analysis of efficient graph planarity testing algorithms", Ph.D. dissertation, Computer Science Department, University of Wisconsin, June 1969.

[8] Tarjan, R., "Implementation of an efficient algorithm for planarity testing of graphs", unpublished implementation, Dec. 1969.