The Hopcroft-Tarjan Planarity Algorithm, Presentations and Improvements*

David Gries
Jinyun Xue**

88-906
April 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported by the NSF under grant DCR-8320274.

** Visiting Cornell from the Computer Science Department, Jiangxi Normal University, Nanchang, People's Republic of China.

The Hopcroft-Tarjan Planarity Algorithm, Presentation and Improvements[0]

David Gries, Jinyun Xue[1]
Computer Science Department
Cornell University
Ithaca, NY 14853
March 1988

Abstract

We give a rigorous, yet, we hope, readable, presentation of the Hopcroft-Tarjan linear algorithm for testing the planarity of a graph, using more modern principles and techniques for developing and presenting algorithms that have been developed in the past 10-12 years (their algorithm appeared in the early 1970s). Our algorithm not only tests planarity but also constructs a planar embedding, and in a fairly straightforward manner. The paper concludes with a short discussion of the advantages of our approach.

Table of Contents

1. Introduction

This paper attempts to provide a rigorous and readable presentation of the Hopcroft-Tarjan linear algorithm for testing the planarity of a graph and building its planar embedding [3]. Their presentation is quite opaque, due in large part to our inabilities in presenting algorithms in the early 1970's and to the newness of the algorithm. There have been later presentations, see for example [4], but we felt the need for a more careful presentation that defined clearly the problems involved in the algorithm, that attacked each in a more isolated manner, and that made use of more modern ideas in presenting algorithms and their proofs.

The paper is organized as follows. The rest of this section presents our notation. Sect. 2 develops the basic planarity algorithm, assuming no particular representation of the graph and data. This a high-level algorithm can be refined in a number of ways. Section 3 presents some desired properties of the graph representation and refines the algorithm to take them into account, while Sect. 4 shows how to change the undirected graph into one satisfying these properties. Sect. 5 discusses a suitable representation of the planar embedding of a graph and refines the algorithm accordingly, while Sect. 6 makes a final data refinement and analyzes the order of execution time of the algorithm. Sect. 7 concludes with a discussion.

[0] This research was supported by the NSF under grant DCR-8320274.

[1] Visiting Cornell from the Computer Science Department, Jiangxi Normal University, Nanchang, People's Republic of China.

Our algorithms manipulate arrays and sequences. A sequence $s = [s_0, s_1, \ldots, s_{\#s-1}]$ has $\#s$ elements. Element k of an array or sequence s is referenced using $s.k$ instead of the more usual $s[k]$, in an attempt to simplify notation. Operation '.' binds tightest and left to right, so that $s.k.j = (s.k).j$. The notation $s.(h..k)$ denotes the subsequence of s consisting of $s.h$ through $s.k$; $s.(i..) = s.(i..\#s-1)$; and $s.-i = s.(\#s-i)$, for positive i, so that $s.-1$ is the last element of s. Finally, $s \mathbin{\hat{}} t$ denotes the catenation of sequences s and t.

We deal with a finite undirected graph $G(V,E)$, where $V = \{i \mid 0 \le i < \#V\}$ is the set of vertices and E is the set of edges. An edge is represented by (u,v) where u is its head vertex and v its tail vertex. The graph has at least one edge and has no self-loop —i.e. no edge of the form (u,u).

By a (simple) path of G we mean a sequence of vertices $p = [v_0, v_1, \ldots, v_{\#p-1}]$ such that each pair (v_i, v_{i+1}) is an edge of G and the first $\#p-1$ vertices are distinct. A path of length at least 2 is a cycle if its first and last vertices are the same. A path is itself a graph with vertices the v_i and edges the (v_i, v_{i+1}). We use $\frown$ to denote extension of a path: for paths $p = [a, \ldots, c]$ and $q = [c, d, \ldots, e]$, $p \frown q$ is the path $[a, \ldots, c, d, \ldots, e]$.

$G0 \cup G1$ denotes the union of graphs $G0$ and $G1$, i.e. the graph consisting of all edges and vertices that are in at least one of $G0$ and $G1$. Graph $G0-G1$ contains as its edges all the edges of $G0$ that are not in $G1$. An undirected (directed) graph is *biconnected* (*strongly connected*) if any two distinct vertices lie on a cycle of the graph.

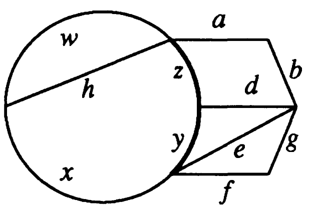
2. The basic planarity-testing algorithm

G is planar iff there exists a mapping of its vertices and edges into the plane such that each vertex is mapped into a distinct point, each edge is mapped onto a simple curve connecting the images of its end points, and no two curves that are the images of two edges share points except possibly for their ends. The mapping is called a planar embedding of G. The following lemmas by Euler and Berge (see e.g. [1]) allow us to restrict attention to graphs that are biconnected and satisfy $\#E \le 3*\#V-6$ for $\#V \ge 3$, which we do from now on.

(2.0) **Lemma**. If G is planar, then $\#E \le 3*\#V-6$ for $\#V \ge 3$.

(2.1) **Lemma**. G is planar iff all its biconnected components are.

Consider testing whether G is planar. Since G is biconnected, it contains a cycle c (say). Partition the edges of $G-c$ into a set S of *segments*: two edges are in the same segment iff there is a path from an end point of one edge to an end points of the other and no vertex of the path is in c. We say that c *defines* S. The *Jordan Curve Theorem* (see e.g. [2]) tells us that a segment is completely on one side of c in any planar embedding of G.



cycle c has the edges w, x, y, z
segment $s0$ has the edges a, b, d, e, f, g
seam$(c, s0)$ has the edges y, z
spine$(c, s0)$ has the edges a, b, e (or a, b, g, f)
segment $s1$ has the edge h
seam$(c, s1)$ has the edge w
spine$(c, s1)$ has the edge h

Figure 0. Illustration of a cycle and two segments

We introduce several terms concerning cycle c and a segment s. (See Fig. 0.) The vertices of s that are on c are called *attachments* of s. A segment has at least two distinct attachments, since the graph is biconnected. Since the attachments are on c, there is a path in c that contains all the attachments and whose first and last vertices are different attachments —in fact there are as many such paths as there are attachments, since any attachment can be the head of the path. Arbitrarily call *one* of these paths the *seam* of c and s, denoted by $seam(c,s)$. By the definition of s, there is a path in s from one end vertex of $seam(c,s)$ to the other. Arbitrarily call one of these paths the *spine path*, $spine(c,s)$. Note that $seam(c,s) - spine(c,s)$ is a cycle.

In embedding the segments in the plane around c, some of the segments will be inside c and some outside. We now describe conditions under which two segments must be on different sides.

(2.2) Two segments *interlace* (see Fig. 1) iff either of the following holds:

(a) There are vertices w, x, y, z on c, in that order, such that w and y are attachments of one segment and x and z are attachments of the other, or

(b) The two segments have at least three attachments in common. □

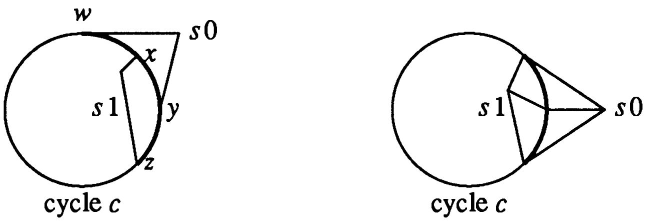


Figure 1. Illustration of interlacing of segments s0 and s1

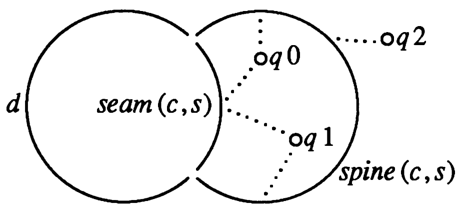
Interlacing segments cannot be embedded on the same side of cycle c. We call the set of segments S defined by c *bipartite* iff it has a partition (SI, SO) (standing for the Inside and Outside sets of segments) such that no two segments in SI interlace and no two segments in SO interlace. If S is bipartite, (SI, SO) is called a *bipartite partition.* It has been proved [0] that

(2.3) **Theorem**. Biconnected graph G with cycle c is planar iff

(a) The set S of segments generated by c has a bipartite partition;

(b) For each segment s in S, $c \cup s$ is planar. □

Theorem (2.3) is the germ of a recursive algorithm for testing planarity. However, it requires testing whether $c \cup s$ is planar, and it will be useful to analyze this test further. The subgraph $seam(c,s) \cup s$ (see Fig. 2) has the cycle $seam(c,s) - spine(c,s)$, and this cycle defines a set Q (say) of segments, just as c defined a set of segments of G. We can now prove Theorem (2.4).



cycle $c = seam(c,s) - d$.
$s = spine(c,s) \cup Q$ where
$Q = q0 \cup q1 \cup q2 \cup$ ... and
each qi has attachments only to $seam(c,s)$ $- spine(c,s)$, as shown by dotted lines.

Figure 2. The composition of c and s

(2.4) **Theorem.** $c \cup s$ is planar iff

(a) $seam(c,s) \cup s$ is planar;

(b) Let Q be the set of segments of $seam(c,s) \cup s$ defined by cycle $seam(c,s)$ - $spine(c,s)$. Q has a bipartite partition $QP = (QI, QO)$ such that no element of QO has an attachment to an inner vertex of $seam(c,s)$.

Proof. Let $d = c - seam(c,s)$ (see Fig. 2). The graph $c \cup s$ has the cycle $seam(c,s)$ - $spine(c,s)$, and the set of segments generated by this cycle is $\{d\} \cup Q$. Therefore, by Theorem (2.3), $c \cup s$ is planar iff

(a) $\{d\} \cup Q$ is bipartite and
(b) For each segment t of $\{d\} \cup Q$, $seam(c,s) \cup spine(c,s) \cup t$ is planar.

This we can write as

(a1) Q has a bipartite partition (QI, QO), for which
(a2) d interlaces with no segment of QO (so that d can be placed in QO);
(b1) for each segment t of Q, $seam(c,s) \cup spine(c,s) \cup t$ is planar;
(b2) $seam(c,s) \cup spine(c,s) \cup d$ is planar.

By Theorem (2.3), (a1) and (b1) are together equivalent to '$seam(c,s) \cup spine(c,s) \cup s$ is planar', which is part (a) of the theorem we are proving. Further, (b2) obviously holds. It remains only to show that (a2) is equivalent to (b) of Theorem (2.4), which we now do by showing that $\neg$ (a2) $\equiv$ $\neg$ (b):

d interlaces with a segment t of QO
= {Since d has only two attachments to $seam(c,s)$ - $spine(c,s)$}
d and t satisfy part (a) of interlacing definition (2.2)
= {Since d attaches exactly to the end vertices of $seam(c,s)$}
t attaches to inner vertices of both $seam(c,s)$ and $spine(c,s)$}
= {Since, by construction, t attaches to an inner vertex of $spine(c,s)$}
t attaches to at least one inner vertex of $seam(c,s)$ □

We give in (2.6) a recursive procedure for testing planarity, the body of which is little more than a rendering of Theorems (2.3) and (2.4) into procedural form. In studying (2.6), it may help to refer to Fig. 2.

A call on *Planarity* is given a partially constructed bipartite partition SP of some of the segments defined by a cycle c, together with the seam and spine of another segment. The purpose of the call is to test the planarity of the new segment and to add it to SP. At this stage, we make no statement about the representation of SP.

The first statement of the procedure body helps test part (a) of Theorem 2.3. Next, the loop determines whether $seam(c,s) \cup s$ is planar (Part (a) of Theorem (2.4)), where $p = seam(c,s)$, and sets x to *false* if it is not. At each iteration of the loop, QP contains a bipartite partition of the segments of $p \cup s$ that have been processed thus far, and each iteration processes one of the segments. The invariant of this loop is given in (2.5). The last statement of the body checks part (b) of Theorem (2.4).

(2.5) *Invariant*: QP is a partition of ((the set of segments defined by p - sp) $- Q$),
x = 'QP is bipartite'

(2.6) {SP is a bipartite partition of some of the segments defined by a cycle c (say).
Path p is the seam and sp the spine of a segment s (say) defined by c, and s is not in SP.
Variable x is true.
If $c \cup s$ is planar and s can be added to SP, then add it; otherwise, set x to false.}

proc *Planarity*(value p, sp: *path*; **var** x: *Boolean*; **var** SP: *Partition*);
begin Extend bipartite partition SP with s (if not possible, set x to false);
  var Q := The set of segments of s defined by cycle p - sp;
  var QP : *Partition* := (Φ, Φ);
  do $x \wedge Q \neq \Phi \rightarrow$ **var** q: *segment*;
    Choose (q, Q); $Q := Q - \{q\}$;
    Planarity (*seam* $(p - sp,\ q)$, *spine* $(p - sp,\ q)$, x, QP)
  od;
  Rearrange QP to satisfy part b of Theorem (2.4) (if not possible, set x to *false*)
end .

It remains to show how to call *Planarity* the first time to test for the planarity of G. Consider an arbitrary vertex u of G. Since G is biconnected, some cycle begins and ends in u. Further, each segment defined by the cycle has at least two attachments and thus has an attachment that is not u. Let path p be $[u]$ (i.e. p has no edges) and spine sp be the cycle beginning with an edge leaving u. Assuming the existence of a virtual cycle $[u, u]$ consisting of a self-loop, the following builds a planar embedding of G:

$\{u$ is a vertex of $G\}$
var x := *true*;
var SP: *Partition* := (Φ, Φ);
Planarity $([u]$, some cycle beginning with u, x, $SP)$

3. Defining a suitable representation of the graph

Algorithm (2.6) contains operations for which we have a choice. There may be several possible spines to complete a cycle, and the order in which the segments in QP are processed is arbitrary. We now define a representation of the graph in which these choices are already made. The edges of the graph will be directed and its vertices renumbered so that at each invocation of procedure *Planarity* the spine and the order in which to process the segments are predetermined. Further, we give a theorem that simplifies the test for the interlacing of a new segment with those in an already-formed bipartite partition. We rewrite *Planarity* to make use of this representation.

The original graph G, with vertices $V = 0..\#V-1$, is undirected. Instead of G, our algorithm processes a corresponding directed graph that is constructed by giving a direction to each edge of G. We use a conventional adjacency-list representation $A.(0..\#V-1)$ in which $A.i$ is a sequence of all neighbors of vertex i in the directed-graph representation. Our purpose here is to state properties of A so that several parts of procedure (2.6) can be refined. It will be advantageous to state the properties entirely in terms of the seam p and the spine sp that define a segment s at the beginning of execution of the procedure body of (2.6). Only enough properties are stated to allow us to refine the procedure body, but remember that the properties hold each time the procedure is called.

The directed edges and renumbered vertices are to satisfy the following:

(3.0) (a) In the directed graph, $s \cup p$ is strongly connected.
(b) The edges of path p - sp are directed in that order. The vertex numbers of the path are in increasing (but not necessarily consecutive) order, except for the last (remember, p - sp is a cycle).
(c) The only directed edge from p to s is the first edge of the spine path; it is called the root edge of the segment. It leaves the highest attachment of p and s. All other edges joining p and s are directed from s to p.

One of the tasks of *Planarity* is to construct a spine. Consider an inner vertex v of a spine. Its neighbors are given by the sequence $A.v$. Imposing the following constraint simplifies construction of a spine:

For v an inner vertex of a spine, the *first* element of $A.v$ is also in the spine.

This constraint and (3.0) have two consequences. First, path p and spine sp need not be parameters of *Planarity*. Instead, only the first two vertices u and v (say) of the spine are needed. For example, the following algorithm stores the sequence of inner vertices of the spine in variable spi and the tail of the spine in w. The loop guard is correct because, by (3.0b), inner vertices of sp are $> u$ and, by (3.0c), u is the highest attachment of p and s.

(3.1) $spi,\ w := [\,],\ v;$
$\{Inv\colon\ u\ \hat{}\ spi\ \hat{}\ w$ is a prefix of spine sp$\}$
do $w > u \rightarrow spi := spi\ \hat{}\ w;\ w := A.w.0$ **od** .

Second, by construction, for each inner vertex y of the spine the following holds. The first element of $A.y$ is a vertex of the spine. For each other vertex z in $A.y$, (y,z) is the first edge of a spine path of a segment q defined by cycle p - sp. Hence, the loop over elements of the set of segments Q of algorithm (2.6) can be written as

(3.2) **for** $y \in spi$ **do var** $k := 1;$
do $x\ \wedge\ k < \#A.y \rightarrow Planarity(y,\ A.y.k,\ x,\ QP);\ k := k+1$ **od**
od .

It remains to choose an ordering in which to process the segments of Q. The segments with the higher-numbered attachments will be processed first. For two segments with the same highest attachment, the one with the lowest attachment is processed first. If two segments have the same highest and lowest attachments, we have a third way of ordering them. Definition (3.3) introduces the term *may precede* to help describe the ordering; Fig. 3 illustrates the four cases of (3.3) in order to provide a gentle introduction, but remember that (3.3) is the definition.

(3.3) **Definition.** Let $at.s$ denote the set of attachments of a segment s to p - sp. Define

$h.s = max\,(at.s)$ (= the highest attachment of s)
$l.s = min\,(at.s)$ (= the lowest attachment of s)
$ll.s = min\,(at.s - \{l.s\})$ (= the second lowest attachment of s) .

Segment s *may precede* segment t if

(a) $h.s > h.t$, or
(b) $h.s = h.t$ and $l.t > l.s$, or
(c) $h.s = h.t$, $l.s = l.t$, and $h.s = ll.s > ll.t$, or
(d) $h.s = h.t$, $l.s = l.t$, $\neg(h.s = ll.s > ll.t)$, and $\neg(h.t = ll.t > ll.s)$. □

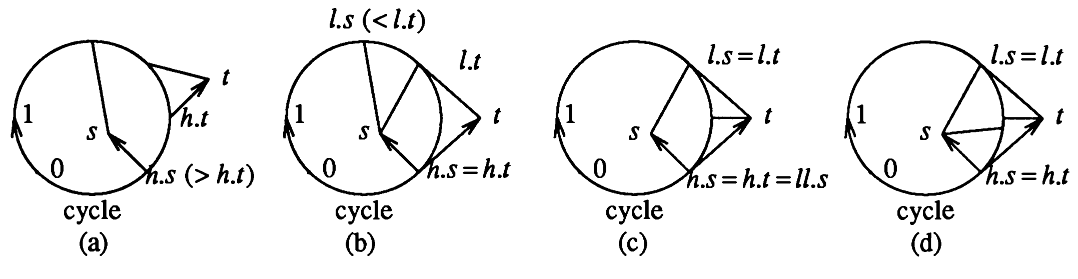


Figure 3. Illustration of four cases of Definition (3.3)

'May precede' is defined so that when s may precede t a test for interlacing of s and t can be done in terms of the spine of t only —the other attachments of t need not be considered, as Theorem (3.4) shows.

Note that at least one of 's may precede t' and 't may precede s' holds. However, if part d of the definition holds, then s may precede t and t may precede s. This is mandated by the need to sort segments in linear time according to this ordering when constructing the directed graph (see Sect. 4). If we require that only one hold, then a linear-time sorting algorithm may not be possible.

(3.4) **Theorem**. Suppose segment s may precede segment t. Then s and t interlace iff there is an attachment v of s satisfying $h.t > v > l.t$.

Proof. If there is no such v, then, since all attachments of t lie in the range $l.t..h.t$, s and t do not interlace. Suppose there is such an attachment v. A four-case analysis based on definition (3.3) shows that s and t interlace:

Case 0. $h.s > h.t$. The four points $h.s > h.t > v > l.t$ form the sequence w,x,y,z of definition (2.2a) of interlacing.

Case 1. $h.s = h.t \wedge l.t > l.s$. The four points $h.t > v > l.t > l.s$ form the sequence w,x,y,z of definition (2.2a) of interlacing.

Case 2. $h.s = h.t \wedge l.s = l.t \wedge h.s = ll.s > ll.t$. Since the second lowest attachment of s is also the highest, v does not exist and the case does not arise.

Case 3. $h.s = h.t \wedge l.s = l.t \wedge \neg(h.s = ll.s > ll.t) \wedge \neg(h.t = ll.t > ll.s)$. There are four subcases to consider. If $h.s = ll.s = ll.t$, then v does not exist and the subcase does not arise. If $h.s > ll.s > ll.t$, then the four points $h.t > ll.s > ll.t > l.s$ form the sequence w,x,y,z of definition (2.2a) of interlacing. If $h.s > ll.s = ll.t$, then s and t have the three distinct attachments $h.s$, $ll.s$, and $l.s$ in common and hence interlace. If $h.s > ll.t > ll.s$, then the four points $h.s > ll.t > ll.s > l.t$ form the sequence w,x,y,z of definition (2.2a) of interlacing. □

As mentioned earlier, this theorem allows us to test for interlacing of two segments by considering all attachments of one segment and only the two attachments of the spine of the other. We order the neighbors of an inner vertex of the spine to take advantage of this property:

(3.5) For y an inner vertex of spine sp, $A.y.0$ is the next vertex on the spine. For each other element $A.y.i$, $(y, A.y.i)$ is the first edge of the spine of a segment. Further, for each i, $1 < i < \#A.y$, the segment given by edge $(y, A.y.(i-1))$ may precede the segment given by edge $(y, A.y.i)$.

We give in (3.6) procedure *Planarity*, modified to take into account the above analysis. At the same time, we add a parameter *cstart*, whose purpose will become clear in a later modification; it is introduced now just to reduce the number of different levels of modification to be presented. Note that the loop that builds *QP* is simply a refinement of the corresponding loop of version (3.2), processing the segments in an order given by Definition (3.4). In addition, it sets *spi*, the sequence of inner vertices of the spine, to the empty sequence.

(3.6) {Global directed graph A satisfies (3.0) and (3.5).
Parameter *cstart* is the lowest vertex in a cycle c (say) and *SP* is a bipartite partition of some segments defined by c.
Edge (u, v) is the first edge of the spine *sp* (say) of a segment s (say) defined by c, s is not in *SP*, and all segments in *SP* may precede s.
Parameter x is true.
If $c \cup s$ is planar and s can be added to *SP*, then add it; otherwise, set x to false.}

```
proc Planarity(value cstart, u, v: vertex;  var x: Boolean;  var SP: Partition);
begin var spi: seq (vertex);
      var w:  vertex;
      Store the inner vertices of the spine in spi and its end vertex in w (see (3.1));
      Add s to SP (if not possible, set x to false);
      var QP: Partition;
      {Build bipartite partition QP for the segments (see (3.2))}
          QP := (Φ,Φ);
          do x ∧ spi ≠ [] → var y, spi := spi.−1, spi.(0..#spi−2);
                            var k := 1;
                            do x ∧ k < #A.y → Planarity(w, y, A.y.k, x, QP);  k := k+1 od
          od;
      Rearrange QP to satisfy part (b) of Theorem (2.4) (if not possible, set x to false)
end
```

The sequence of statements to initiate the planar embedding of a graph G is transformed into

```
var x := true;
var SP:Partition := (Φ, Φ);
Planarity(0, 0, A.0.0, x, SP)
```

4. Constructing the directed-graph representation A

We assume that undirected graph G is given as an adjacency list A in which, for each vertex v, $A.v$ is a sequence of all vertices adjacent to v. That G is undirected means that an edge (u, v) is represented twice: u is in $A.v$ and v is in $A.u$. This section presents an algorithm that directs the edges of A and renumbers its vertices so that A satisfies (3.0) and (3.5) for all cycles and segments identified in recursive procedure *Planarity*. This directed graph depends on the spanning tree generated during a depth-first traversal of G. (Since the original and resulting graphs differ only in that edges have been directed and vertices renumbered, one is planar iff the other is.) Written under the assumption that the reader has knowledge of depth-first search and its properties, this section is quite terse.

We begin by defining a *palm graph* derived from G and prove in Theorem (4.1) that it satisfies (3.0). An example of a graph and an associated palm graph is given in Fig. 4.

(4.0) **Definition.** A *spanning tree* of undirected biconnected graph G is a tree whose edges are in G and whose nodes are all the vertices of G. The edges of G that are not in the spanning tree are called *fronds*. A path consisting of a sequence of zero or more spanning-tree edges followed by a frond is called a *span-frond path*. A *palm graph* P for G (based on a given spanning tree of G) is a version of G with all edges directed and vertices renumbered to satisfy:

(a) Each spanning-tree edge of P is directed from parent to child; for such a directed edge (u, v), $u < v$.

(b) Each frond (v, w) is directed so that $v > w$.

(c) In each span-frond path $p = (u, ..., w)$ of P where w is not an inner vertex of p, $w \le u$. □

One can prove that a palm graph is strongly connected if the graph from which it is formed is biconnected. We leave the proof to the reader.

graph G Palm graph P, with fronds dashed

Figure 4. A graph and an associated palm graph

(4.1) **Theorem**. Palm graph P can be iteratively decomposed so that each subcomponent satisfies (3.0).

Proof. The proof is by induction on the structure of A. The following first decomposition satisfies (3.0):

(a) Virtual cycle c is $[0,0]$,

(b) Segment s consists of the whole graph P.

(c) Seam p is $[0]$ and spine sp is any cycle beginning at vertex 0 (remember, P is strongly connected).

Now consider any subgraph g of P that satisfies

(4.2) g is strongly connected and consists of a cycle c together with all span-frond paths leaving c.

Note that the first decomposition of P satisfies (4.2). We show how to decompose g into c and a set of segments s, each of which has a seam $seam(c,s)$ and a spine that satisfy (3.0). Further, $s \cup seam(c,s)$ will also satisfy (4.2) (so that it can be decomposed as well, and by induction the whole graph can be thus decomposed.)

Consider an edge (u,v) not in c but with u in c. Define the segment s corresponding to (u,v) to consist of all span-frond paths with (u,v) as the first edge. Note that each edge of g that is not in c belongs to exactly one such segment, because the spanning-tree edges of s form a subtree of the whole spanning tree. Since g is strongly connected, s has attachments to c. By property (c) of Definition (4.0), u is the highest attachment of s and c. Let the lowest attachment be w, and let $p = seam(c,s)$ be the path in c from w to u. Let $sp = spine(c,s)$ be any span-frond path in s from u to w.

Now, (3.0a) holds because $s \cup p$ is strongly connected. Second, (3.0b) holds because $p \frown sp$ is a span-frond path and each spanning-tree edge (x,y) satisfies $x<y$. Third, (3.0c) holds because any attachment w of s and p is reached from u only by span-frond paths whose first edge is (u,v), and these all satisfy (4.0c). Finally, by the definition of s, (4.2) holds for the subgraph $s \cup p$. □

Hence, a palm graph as defined in (4.0) satisfies (3.0). We now discuss the construction of a palm graph. A depth-first search of undirected graph A identifies a spanning tree. During the search, an array $N.(0..\#V-1)$ can be constructed that contains in each element $N.v$ the number of vertices processed before vertex v. Thus, $N.r=0$ for the root r of the spanning tree, $N.v=1$ for the next vertex v processed, and so forth. Finally, A can be changed during the search so that each spanning-tree edge is directed from parent to child and each frond (v,w) is directed so that $N.v > N.w$. (Directing an edge between vertices x and y means deleting either y from $A.x$ or x from $A.y$.)

Algorithm (4.3) performs a depth-first search, directing the edges of A and producing new vertex numbers in N. (It also contains three statements B, C, and D, to be described later, which deal with arrays L and LL.) We claim, without further proof, that assigning the number $N.v$ to each vertex v of the result of such a depth-first search yields a graph that satisfies (4.0).

(4.3) **var** L, LL, N: **array** $[0..\#V-1]$ **of** $0..\#V-1$;
$L.0, LL.0 := 0, 0$;
for $i \in 0..\#V-1$ **do** $N.i := -1$;
$N.0 := 0$;
var $n := 1$;
$DFS(0,\ A.0.0)$;

{Node u is numbered and v is not: $0 \le N.u < n$ and $N.v = -1$.
There is no path $(u, ..., w)$ (for any w) satisfying $N.u < N.w$ whose inner vertices are unnumbered.
Thus far, n nodes have been assigned distinct numbers from $0..n-1$.

Assign v the number n. Direct edge (u, v) from u to v (i.e. delete u from $A.v$). Traverse in depth-first order all edges reachable from v along paths of unnumbered vertices, assigning numbers to vertices and directing edges as required in the discussion above.}

```
proc DFS (value u, v: vertex);
begin N.v := n; n := n+1;
      B {see (4.4)};
      var k := 0;
      do k < #A.v → var w := A.v.k; {w is the next neighbor of v to process}
                    if w ≠ u ∧ 0 ≤ N.w < N.v → {(v,w) is a frond}
                                                C {see (4.5)} ; k := k+1
                    [] w = u ∨ N.v < N.w     → {(v,w) is to be deleted}
                                                Delete A.v.k from A.v
                    [] N.w = −1              → {(v,w) is a new spanning-tree edge}
                                                DFS (v,w); D {see (4.6)} ; k := k+1
                    fi
      od
end
```

After execution of (4.3), the vertices of A have to be renumbered as given by array N and A has to be modified to satisfy (3.5). Thus, for each vertex v, $A.v.0$ should be a vertex of a spine path and the elements of $A.v.(1..)$, which define segments defined by the spine path, should be in an order given by (3.3). $A.v.0$ can be a vertex of a spine path iff it is possible to reach the lowest vertex of the seam from $A.v.0$ along a span-frond path.

To help in modifying A to satisfy (3.5), algorithm (4.0) is augmented to calculate two arrays $L.(0..\#V-1)$ and $LL.(0..\#V-1)$, whose values upon termination of (4.0) will be as follows:

$L.v$ is the minimum value $N.w$ for vertices w reachable from v along span-frond paths;

$LL.v$ is the second minimum value $N.w$ of vertices w reachable from v along span-frond paths (or v if there is no such second value).

The following statements for B, C, and D in algorithm (4.0) take care of calculating L and LL. We leave to the reader the correctness arguments.

(4.4) B: $L.v, LL.v := v, v$

(4.5) C: **if** $N.w < L.v \rightarrow L.v, LL.v := N.w, L.v$
$[]\ N.w = L.v \rightarrow skip$
$[]\ N.w > L.v \rightarrow LL.v := min(LL.v, N.w)$
fi

(4.6) D: **if** $L.w < L.v \wedge L.w < LL.w \rightarrow LL.v := min(L.v, LL.w);\ L.v := L.w$
$[]\ L.w < L.v \wedge L.w = LL.w \rightarrow LL.v := L.v;\ L.v := L.w$
$[]\ L.w = L.v \wedge L.w < LL.w \rightarrow LL.v := min(LL.v, LL.w)$
$[]\ L.w = L.v \wedge L.w = LL.w \rightarrow skip$
$[]\ L.w > L.v \rightarrow LL.v := min(LL.v, L.w)$
fi

After execution of algorithm (4.0), we have directed graph A and arrays N, L, and LL. The neighbors $A.v$ of each node v have to be sorted into the order defined by (3.5). This means that they should be in the order given by array L. But if two neighbors have the same L value, then, by (3.5), priority is given to those neighbors w satisfying $LL.w = w$; otherwise, their order is immaterial. We can use a radix sort to sort all the edges (v, w) according to this ordering in time proportional to the number of edges. Consider the function

$$\Theta((v,w)) = \begin{cases} 2*N.w & \text{if } (v,w) \text{ is a frond} \\ 2*L.w & \text{if } (v,w) \text{ is a spanning-tree edge and } LL.w = w \\ 2*L.w+1 & \text{if } (v,w) \text{ is a spanning-tree edge and } LL.w < w \end{cases}$$

Then $\Theta((v,w)) < \Theta((v,u))$ means that w may precede u in $A.v$ but if $\Theta((v,w)) = \Theta((v,u))$ their ordering does not matter. We build an array $Bucket.(0..2*\#V-1)$ of sets of edges, where all edges with the same Θ-value j (say) are placed in set $Bucket.j$. Having done this, we can then move the edges back into A in the order given by array $Bucket$, renumbering them as we do in (4.7). The complete algorithm to change undirected graph A to satisfy (3.0) and (3.5) is then (4.3); (4.7). Note that it takes time $O(\#E)$.

(4.7) {Construct directed graph A satisfying (3.0) and (3.5)}
var $Bucket$: **array** $[0..2*\#V-1]$ **of** $set(edge)$;
for $j \in 0..2*\#V-1$ **do** $Bucket.j := \Phi$;
for (v,w) an edge of A **do** $Bucket.(\Theta(v,w)) := Bucket.(\Theta(v,w)) \cup \{(v,w)\}$;
for $i \in 0..\#V-1$ **do** $A.i := \Phi$;
for $j := 0$ **to** $2*\#V-1$ **do** **for** $(v,w) \in Bucket.j$ **do** $A.(N.v) := A.(N.v)\ \hat{}\ N.w$

5. Embedding segments and testing for interlacing

We now transform *Planarity* so that it produces as well as tests for a planar embedding. We define a representation of the planar embedding, introduce a representation for a partition SP, and refine *Planarity* accordingly.

Before we begin, it will help us to determine exactly how many different spines, segments, and attachments are created during execution of *Planarity*. This information will be useful in defining arrays and in discussing execution time. These numbers are actually quite easy to determine, using the fact that the directed graph is a spanning tree together with a set of fronds. The spanning tree connects all the vertices, using $\#V-1$ edges, so the rest of the edges, $\#E-\#V+1$ of them, are fronds. Since each frond is the tail of the spine of a segment, we have the

(5.0) **Lemma**. Graph G has $\#E-\#V+1$ different segments and spines of segments.

A planar embedding describes, for each cycle c determined during execution of *Planarity*, on which side (inside or outside) of c each segment defined by c is to be placed. Let us number segments in the order in which they are processed during execution of the first call *Planarity* $(0,0,A.0.0,x,SP)$: segment 0 is the one defined by the edge $(u,v) = (0,A.0.0)$, segment 1 is the one defined by the second and third arguments in the first recursive call, etc. We use a global array *Side* and a global variable *N_Side* as defined in (5.1). Upon termination, all segments will have been embedded on one side or the other of their respective cycles.

(5.1) {*N_Side* segments have been numbered 0, ..., *N_Side*−1, and, for $0 \le i < N\_Side$,
if segment i has been embedded, *Side.i* = 'segment i is on the inside'.}
var *Side*: **array** [0..#E−#V] **of** *Boolean*;
N_Side: *integer*

Remark. From *Side* itself it is impossible to draw the graph in the plane. It is necessary to execute a procedure like *Planarity*, but without all the processing of *SP*, in order to determine the segment numbers again. Alternatively, one could maintain more information with each segment number, for example, the first vertex of the cycle defining it along with its root edge. □

Now consider a call *Planarity* (u,v,x,SP), where *Planarity* is defined in (3.6), whose execution adds s to *SP*.

It is useful to decompose *SP* into blocks such that segments in different blocks do not interlace, while segments in the same block interlace in such a way that the placement of one of them (inside or outside c) determines the placement of all of them. (At this point, we deal with relations between blocks only, leaving until later the investigation of the contents of the individual blocks.) Note that, by Theorem (3.4), s can interlace with a segment in *SP* only if the segment has an attachment less than u, so testing interlacing requires considering only blocks with such attachments. Also, ordering the blocks in a special way will simplify testing for interlacing and merging blocks (when necessary). We gather the properties of this part of the representation of *SP* in the following definition.

(5.2) **Representation of certain segments of SP.**

(a) The segments in *SP* are partitioned into blocks: no segment of one block interlaces with a segment of another block and, within each block, the interlacing of its segments is such that the placement of any one segment inside (or outside) c determines the placement of all them.

(b) Variable b is a sequence of representations of the blocks of *SP* that contain a segment with an attachment less than u. Each segment i in *SP* that is not in a block of b has been embedded, so that *Side.i* has its final value.

(c) For each i, $0 < i < \#b$, each segment in blocks $b.[0..i-1]$ may precede each segment in block $b.i$.

We now investigate placing the new segment s, with highest attachment $h.s = u$, in *SP*. In the investigation, we say that a block interlaces with a segment s if at least one of its segments interlaces with s.

(5.3) **Theorem.** Segment s interlaces with a segment of *SP* iff it interlaces with the last block B (say) of b.

Proof. By (5.2b) and Theorem (3.4), s does not interlace with segments of *SP* that are not in a block of b. Suppose B does not interlace with s, and consider any other segment t in another block of b. Using $h.b$ and $l.B$ to denote the highest and lowest attachments of any segments in block B, we have

(a) No segment of B has an attachment w satisfying $h.s > w > l.s$ (by Theorem (3.4)).

(b) $h.B \ge h.s > l.s \ge l.B$. (The first relation holds because the segments of B may precede s. The last holds because of (a) and because, by definition (5.2b), B has an attachment that is less than $u = h.s$.)

(c) t has no attachment w satisfying $h.s > w > l.s$. (Since t does not interlace with B, by Theorem (3.4) t has no attachment w satisfying $h.B > w > l.B$; the result follows from (b).)

(d) t does not interlace with s (because of (c) and Theorem (3.4)). □

If B does interlace with s, then s should be placed in B. However, then the previous block $b.-2$ may interlace with s as well. Therefore, the following, more general, theorem is needed; its proof is similar to that of (5.3) and is left to the reader.

(5.4) **Theorem**. Let the representation of SP satisfy all of definition (5.2) except that segment s, which is in the last block of b, may interlace with some segment of $b.(0..\#b-2)$. Then s interlaces with a segment of $b.(0..\#b-2)$ iff it interlaces with $b.-2$.

We now investigate the contents of a block B of b. First, let us give an example declaration of sequence b:

(5.5) **type** *BlockT* = **record** A:*seq*(*vertex*); S:*seq*$(0..\#E-\#V)$ **end**;
var b: *seq*(**record** I,O: *BlockT* **end**);

Field $B.I$ ($B.O$) of a block B (say) of b contains information concerning segments of B that are (currently) on the inside (outside) of cycle c. In the field, we place two kinds of information. First, $B.I.S$ ($B.O.S$) is a sequence of numbers of segments that are inside (outside) c. These sequences will be used to update array *Side* when the segments in the block can be embedded. Second, in order to test for interlacing with s, a block contains certain attachments of the segments to cycle c. Just how these segment numbers and attachments are stored and the properties of this representation are given in (5.6):

(5.6) **Representation of the blocks in b.**

(a) For each block $b.i$, $b.i.I.S$ ($b.i.O.S$) is a sequence of numbers of its segments that are inside (outside) cycle c.

(b) For each block $b.i$, $b.i.I.A$ and $b.i.O.A$ are sequences of attachments x of the segments of the block that satisfy $u > x > cstart$. The attachments in $b.i.I.A$ ($b.i.O.A$) are tails of edges that are embedded Inside (Outside) cycle c.

(c) Sequences $b.i.I.A$ and $b.i.O.A$ are in nondecreasing order.

(d) For each i, $0 < i < \#b$, the attachments in $b.(i-1)$ are at most those in $b.i$.

Let us discuss the fact that attachments $= cstart$ are not recorded in b (part (b). Since *cstart* is the lowest point of the cycle c, each attachment x of the segments satisfies $x \geq cstart$. Consider a segment s, not in b, with lowest attachment $w \geq cstart$, such that all segments described in b may precede s. By Theorem (3.4), s interlaces with a segment of b iff the segment has an attachment x satisfying $u > x > w \geq cstart$, which means that $x > cstart$. Therefore, attachments $= cstart$ are not needed to test for interlacing. One could record such attachments, but *not* recording them is essential for the efficiency of the algorithm, as pointed out later.

Parts (c) and (d) are important for efficiency of the algorithm, in that they allow a constant-time test for interlacing.

In *Planarity*, (5.7) below, and the main program that first invokes it, the representation b of SP given by (5.2) and (5.6) is established by the initialization $SP0$ of the main program and is maintained by execution of the (only) two statements $SP1$ and $SP2$ within (5.7) that change b.

Local variable q of *Planarity* represents QP in the same way that b represents SP, with variable w, the tail of the spine, playing the role of *cstart*: only attachments $> w$ should be placed in q. Statement $QP0$ initializes q, and each inner call of *Planarity* adds a segment to QP (which by recursion maintains the representation). Execution of $Purge(q,y)$ just before a call of an inner *Planarity* deletes attachments from q that are $\geq y$, thus ensuring that representation part (5.6b) holds as required of the call of *Planarity*.

(5.7) {Global directed graph A satisfies (3.0) and (3.5).
N_Side segments have been assigned distinct numbers from $0..Nside-1$.
Parameter *cstart* is the lowest vertex in a cycle c (say), and b represents a bipartite partition SP of some of the segments defined by c, as given by (5.2) and (5.6).
Edge (u, v) is the root edge of the spine of a segment s (say) defined by c, s is not in SP, s precedes none of the segments in SP.
Parameter x is true.

If $c \cup s$ is planar and s can be added to SP, then add it; otherwise, set x to false.}

```
proc Planarity(value cstart, u, v: vertex; var x: Boolean; var b: seq(record I, O: BlockT end));
begin var spi: seq (vertex);
      var w: vertex;
      Store the inner vertices of the spine in spi and its end vertex in w (see (3.1));
      SP1: Add s to SP (see (5.8)) ;
      var q: seq(record I, O: BlockT end);
      {Build bipartite partition QP for the segments (see (3.2))}
       QP0 q := [];  {Implement QP := (Φ, Φ)}
         do x ∧ spi ≠ [] → var y := spi.-1; spi := spi.(0..#spi-2);
                           Purge(q, y) (see (5.10)) ;
                           var k := 1;
                           do x ∧ k < #A.y → Planarity(w, y, A.y.k, x, q); k := k+1 od
         od;
      SP2: Check Theorem (2.4) part b and add attachments to b (see (5.11))
end
```

The sequence of statements to initiate the planar embedding of a graph G is transformed into

```
var Side: array [0..#E-#V] of Boolean;
var N_Side: integer := 0;
var x := true;
var b: seq(record I, O: BlockT end);
SP0: b := [];  {Implements SP := (Φ, Φ)}
Planarity(0, 0, A.0.0, x, b)
```

Let us now investigate the four operations that deal with SP or QP.

Operation SP0: $SP := (\Phi, \Phi)$ (also $QP := (\Phi, \Phi)$). This operation initializes SP (QP) to contain no segments. In terms of the representation b of SP, this is performed by $b := [\,]$ ($q := [\,]$). Trivially, these maintain definitions (5.2) and (5.6).

Operation SP1: Add s to SP (see (5.8)). This operation is performed in *Planarity* just after the spine $(u, v, ...w)$ of s has been constructed. The operation is performed in terms of b by the following algorithm, which we explain subsequently.

(5.8) $SP\,1$: 'Add s to SP'

'Build a new block v to contain s'

$\quad$ **var** v: **record** I, O: $BlockT$ **end**;

$\quad v.I.A := [\,];\ \ v.O.A := [\,];\ \ v.I.S := N\_Side;\ \ v.O.S := [\,];$

$\quad$ **if** $w > cstart \rightarrow v.I.A := v.I.A \ \hat{}\ w$

$\quad$ $[]$ $w = cstart \rightarrow skip$

$\quad$ **fi**;

$N\_Side := N\_Side + 1;$

$b := b\ \hat{}\ v;$

do $x \wedge \#b > 1$ **cand** $Lace(b.-2.I.A)$ **cand** $Lace(b.-2.O.A) \rightarrow x := false$

$[]$ $x \wedge \#b > 1$ **cand** $\neg Lace(b.-2.I.A)$ **cand** $Lace(b.-2.O.A) \rightarrow Merge$

$[]$ $x \wedge \#b > 1$ **cand** $Lace(b.-2.I.A)$ **cand** $\neg Lace(b.-2.O.A) \rightarrow$

$\qquad b.-2.I, b.-2.O := b.-2.O, b.-2.I;\ Merge$

od

where

$Lace(t) \equiv$ {= 'a segment whose attachments are in t interlaces with s'}

$\qquad t \neq [\,]$ **cand** $t.-1 > w$

and

$Merge \equiv$ {s is in $b.-1.I$, and s interlaces with $b.-2.O$ but not with $b.-2.I$.

Merge $b.-1$ into $b.-2$) and reestablish invariant (5.9)}.

$\quad b.-2.I.A := b.-2.I.A\ \hat{}\ b.-1.I.A;$

$\quad b.-2.O.A := b.-2.O.A\ \hat{}\ b.-1.O.A;$

$\quad b.-2.I.S := b.-2.I.S\ \hat{}\ b.-1.I.S;$

$\quad b.-2.O.S := b.-2.O.S\ \hat{}\ b.-1.O.S;$

$\quad b := b.(0..\#b-2)$

The invariant of the loop of algorithm (5.8) is

(5.9) P: segment s and its attachment w (if it is $\neq cstart$) is in $b.-1.I$,

$\neg x \Rightarrow SP$ has no bipartite partition, and

$x \Rightarrow b$ satisfies (5.2) and (5.6) except that s may interlace with segments of $b.(0..\#b-2)$, so (5.6.d) may be violated.

The purpose of (5.8) is to assign s a segment number and to place it, along with its lowest attachment w, into the last block of b, which is the representation of SP, and to reestablish invariants (5.2) and (5.6). Later, all the other attachments of s to c will be placed in the same block.

The first statement of (5.8) places a new block with one segment (in the field $I.S$) in b; the new attachment w, the attachment of the spine of segment s, is placed in the block only if it is greater than the lowest vertex of cycle c, as required by representation (5.6b). The statement trivially establishes invariant P.

Suppose all the guards of the loop are false. Then either x is false, which from the invariant means that SP has no bipartite partition, or s does not interlace with $b.-2$. In the latter case, by Theorem (5.4), s interlaces with no segment of $b.(0..\#b-2)$, and we conclude that (5.2) and (5.6) hold —note that the new attachment w is $\geq$ the largest attachment in $b.-2$.

Now consider an iteration of the loop. If the first guard is true, then s interlaces with a segment of $b.-2$ on the inside of c and with a segment on the outside, so SP has no bipartite partition. Thus, setting x to false maintains the invariant. In each of the other cases, exactly one of $b.-2.I$ and $b.(\#b-2).O$ contains a segment that interlaces with

s. The inside and outside are swapped, if necessary, so that segments in $b.-2.I$ do not interlace with s, and blocks $b.-2$ and $b.-1$ are merged.

Operation Purge(q, y). Variable q is the sequence of blocks representing QP within *Planarity*. Just before a call $Planarity(w, y, ...)$, q may only contain attachments that are less than y. Thus, before the call, it is necessary to purge attachments $\geq y$ from q. The following operation performs this service. If purging creates a block with no attachments, then the block interlaces with no other segments still to be processed, so the segments of the block are embedded (by assigning to *Side*). Note that if the last block of q contains a value $< y$, then, by (5.6d), no previous block contains an attachment $> y$.

```
(5.10) Purge(q,y) ≡ {Delete attachments ≥y from q}
                    var h := (q ≠ []);
                    {invariant: h = 'The last block of q may contain an attachment ≥y'}
                    do h → Deletefrom(q.-1.I.A);
                           Deletefrom(q.-1.O.A);
                           if q.-1.I.A = [] ∧ q.-1.O.A = [] → FixSide(q.-1.I.S, true);
                                                              FixSide(q.-1.O.S, false);
                                                              q := q.(0..#q-2); h := (q ≠ [])
                           [] q.-1.I.A ≠ [] ∨ q.-1.O.A ≠ [] → h := false
                           fi
                    od
```

where

```
Deletefrom(t) ≡ {Delete from ordered sequence t all elements that are ≥y}
                  do t ≠ [] cand t.-1 ≥ y → t := t.(0..#t-2) od
```

and

```
FixSide(t,X) ≡ {Side.i := X for i ∈ t;  t := []}
                 do t ≠ [] → Side.(t.-1) := X;  t := t.(0..#t-2) od
```

Operation SP2: Check Theorem (2.4) part b and add attachments to b (see (5.11)). This is the last statement of procedure *Planarity*. Part b of Theorem (2.4) requires that Q have a bipartite partition in which all attachments to inner vertices of $seam(c,s)$ —i.e. attachments x satisfying $u > x > w$— are in the inside of cycle $seam(c,s)$ - $spine(c,s)$. Sequence q describes a bipartite partition of QP. Further, it contains exactly the attachments x to $seam(c,s)$ that satisfy $u \geq x > w$. Hence, the attachments $= u$ should be deleted and each block $q.i$ checked to ensure that $b.k.O$ can be made empty. If this is not possible, if both $b.k.I.A$ and $b.k.O.A$ contain an attachment, then part b of Theorem (2.4) does not hold and the graph is not planar.

At the same time, the attachments in q that are $< u$ should be placed into $b.-1.I.A$, since they are attachments of s. The algorithm to do this is presented in (5.11).

We can now explain the reason for not recording attachments in q that are $= cstart$ (see the paragraph following (5.6). If they were recorded, some of the sequences of attachments $O.A$ for blocks in q might not be empty but would contain the value w. This would seriously jeapordize the efficiency of this part of the algorithm.

(5.11) $SP2$: {Check Theorem (2.4) part b and add attachments to b}

```
Purge (q, u);
var t: seq (vertex) := [];
do x ∧ q ≠ [] → Make_O_empty;
                FixSide (q.−1.I.S, true);
                FixSide (q.−1.O.S, false);
                t := q.−1.I.A ^ t;
                q := q (0..#q−2)
od;
b.−1.I.A := b.−1.I.A ^ t
```

where

```
Make_O_Empty ≡ {Swap q.−1.I and q.−1.O, if necessary, so that q.−1.O.A is empty;
                if not possible, set x to false}
                 if q.−1.I.A ≠ [] ∧ q.−1.O.A ≠ []   → x := false
                 [] q.−1.I.A = [] ∧ q.−1.O.A ≠ []   → q.−1.I, q.−1.O := q.−1.O, q.−1.I
                 [] q.−1.O.A = []                   → skip
                 fi
```

6. Representing sequences of blocks, of attachments, and of segments

It remains to show how to implement various sequences so that we can claim linear running time of the algorithm. The sequences we have to implement are:

Sequence variable b in the main program,
Local sequence variables *spi* and q of *Planarity*,
The sequences $I.A$, $I.S$, $O.A$, $O.S$ within the blocks of b and local variable q,
The sequences $v.I.A$, $v.I.S$, $v.O.A$, and $v.O.S$ of variable v in (5.8),
Local sequence variable t within algorithm (5.11).

We have written the operations on these variables in a style that allows each one to be implemented by a reversed linked list (using Pascal's *new* and *free* operations, say) with head and tail pointers (see Fig. 5). Each sequence variable s: *seq* (T) (for some type T) is replaced by a variable s' declared by

var s': **record** *head*, *tail*: *ptr* **end**

where type *ptr* is defined by

type *ptr* = ↑ **record** v: T; *prev*: *ptr* **end**;

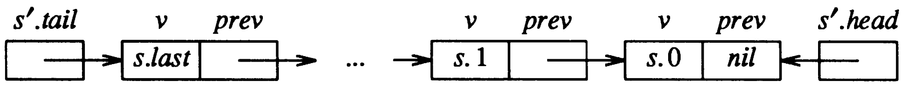


Figure 5. Reversed linked-list representation of sequence s

The relation between abstract variable s and its implementation s' is as follows:

(a) If $s = [\,]$, then $s'.tail = nil$

(b) If $s \neq [\,]$, then $s'.head\!\uparrow\!.v = s.0$, $s'.head\!\uparrow\!.prev = nil$, $s'.tail\!\uparrow\!.v = s.-1$, and for each i, $0 < i < \#s$, if $p\!\uparrow\!.v = s.i$ then $p\!\uparrow\!.prev\!\uparrow\!.v = s.(i-1)$.

Let us show the implementation of each operation on a sequence in terms of the concrete variables. Below, s and t have type $seq(T)$ and e has type T, for some type T.

Operation on s	**Implementation**
$s=[\,],\ \ s \neq [\,]$	$s'.tail = nil,\ \ s'.tail \neq nil$
$s := [\,]$	$s'.tail := nil$
$\{s \neq [\,]\}\ \ s.-1$	$s'.tail\!\uparrow\!.v$
$\{\#s \geq 1\}\ \ \#s > 1$	$s'.tail\!\uparrow\!.prev \neq nil$
$\{\#s > 1\}\ \ s.-2$	$s'.tail\!\uparrow\!.prev\!\uparrow\!.v$
$s := s\ \hat{}\ e$	**var** pt; $new(pt)$; $pt\!\uparrow\!.v := e$; $pt\!\uparrow\!.prev := s'.tail$; **if** $s'.tail = nil \rightarrow s'.head := pt$ $[\,]\ s'.tail \neq nil \rightarrow skip$ **fi**; $s'.tail := pt$
$\{s \neq [\,]\}\ \ s := s.(0..\#s-2)$	**var** $pt := s'.tail$; $s'.tail := pt\!\uparrow\!.prev$; $free(pt)$
$s := s\ \hat{}\ t$	**if** $s'.tail = nil \rightarrow s'.head, s'.tail := t'.head, t'.tail$ $[\,]\ s'.tail \neq nil \wedge t'.tail = nil \rightarrow skip$ $[\,]\ s'.tail \neq nil \wedge t'.tail \neq nil \rightarrow t'.head\!\uparrow\!.prev := s'.tail$; $s'.tail := t'.tail$ **fi**
$s := t\ \hat{}\ s$	**if** $t'.tail = nil \rightarrow skip$ $[\,]\ t'.tail \neq nil \wedge s'.tail = nil \rightarrow s'.head, s'.tail := t'.head, t'.tail$ $[\,]\ t'.tail \neq nil \wedge s'.tail \neq nil \rightarrow s'.head\!\uparrow\!.prev := t'.tail$; $s'.head := t'.head$ **fi**

We now analyze the order of execution of *Planarity* (5.7). We examine the *total* time required by the various operations of (5.7), in the order in which they occur textually, over all recursive calls needed to embed a planar graph G. For a loop, this requires bounding the number of iterations it makes over all recursive calls and bounding the total work required by all iterations. Our result will be that the time is $O(\#E)$, where $\#E$ is the number of edges of the graph. Since $\#E$ is linear in $\#V$ (see (2.0)), the algorithm is $O(\#V)$. Remember that the graph contains $\#E-\#V+1$ spines, segments, which also means that there are at most $\#E-\#V+1$ distinct attachments. (see (5.0)).

Operation 'Store the spine in sp and its end vertex in w (see (3.1))'.

One execution of this operation takes time proportional to the number of edges in the spine. Over all recursive calls, the operation processes each edge exactly once, so the time required is $O(\#E)$.

Operation 'Add s to SP (see (5.8))'.

Adding the new block v to b takes constant time. This is done once for each segment (in all recursive calls), so the total time is proportional to the total number of segments, which is $\#E-\#V+1$.

Consider the loop of (5.8). Each iteration takes constant time, because of the implementation of the sequences as linked lists —note that swapping $b.k.I$ and $b.k.O$ takes constant time because each part consists of four pointers.

Each iteration either causes termination of the algorithm (by setting x to false) or merges two blocks. Merging blocks reduces the number of blocks by one, so merging can not occur more than the total number of blocks created during all recursive calls, which equals the total number of segments, i.e. $\#E-\#V+1$. Thus, in total, (5.8) contributes time $O(\#E-\#V)$.

Operation 'Build bipartite partition ***QP*** **...'.**

Over all calls, the inner-loop body calls *Planarity* once for each segment except the first. Hence, the total number of times the inner loop body is executed is $\#E-\#V$. We do have to determine the total amount of time contributed by operation *Purge*; that is next.

Operation 'Purge(q , y)'.

The purpose of *Purge* is to delete attachments from q that are at least y. Secondarily, if a deletion means that block $q.-1$ contains no more attachments, then the block is also deleted.

Each iteration of the loop of *Purge* either removes a block of q or causes termination of the loop (by assigning to h). Hence, in total, the number of iterations is at most the number of calls on *Purge* plus the total number of blocks. For each segment, *Purge* is called once in the nested loop that builds QP and once in (5.11), in checking that Theorem (2.4) holds. Therefore, it is called $2*(\#E-\#V+1)$ times, so the total number of iterations of the loop is $O(\#E)$.

Deleting an attachment takes constant time. The total number of attachments is $\#E-\#V+1$, and each is deleted once, so deleting attachments takes time $O(\#E-\#V+1)$.

We have to analyze the time required by the calls $FixSide(t, ...)$ within the loop of *Purge*. Such a call embeds each segment i of t (by assigning to $Side.i$) and deletes i from t; this takes constant time. In total, each segment is embedded in this fashion exactly once (and then removed from t), so that, over all executions, *FixSide* takes time proportional to the number of segments, i.e. to $\#E-\#V+1$.

Finally, as mentioned earlier when analyzing (5.8), the total number of blocks created is also $\#E-\#V+1$, so that deleting a block can be done at most $\#E-\#V+1$ times. Hence, in total, *Purge* is $O(\#E)$.

Operation 'Check Theorem (2.4) part b ... (see (5.11))'.

This operation is invoked once for each call of *Planarity*, which is bounded by $\#E$. Eliminating from consideration the calls on *FixSide*, which were analyzed previously, each iteration of the loop takes constant time. Further, each iteration removes a block from q. Since the total number of blocks created is $\#E-\#V+1$, the number of iterations is bounded by $\#E-\#V+1$. Hence, this operation is $O(\#E)$.

7. Conclusions

This is not an easy paper to read, for The Hopcroft-Tarjan planarity algorithm is difficult and subtle. Yet, we believe this presentation is an advance over others, for several reasons. First, it isolates the various concerns of the algorithm and presents them in a 'top-down' fashion. For example, a general algorithm is first given in Sect. 2 and later refined in Sect. 3.

Further, throughout the paper, the goal was always to present the properties upon which the next refinement was built before the refinement. For example, in Sect. 3, properties (3.0) and (3.5) were developed and analyzed first, before giving refinement (3.6). This follows our principle that proof of correctness and program should go hand in hand, with the latter leading the way —although we haven't given a formal proof, we believe that we have given a balanced, understandable presentation from which a more formal proof can be developed. The principles behind program correctness were used, if not the formal details. This is in direct constrast to other developments, where the algorithm is given and thereafter one tries to glean various properties of it, often with too many operational arguments, leading to real confusion.

In Sect. 3, once properties (3.0) and (3.5) are understood, the task of refining the algorithm to make use of them is almost trivial; it is these properties that one should analyze and remember, and not the algorithm. The same holds for the representations (5.1) and (5.6) of a bipartite partition.

A prime principle is to structure an algorithm to reflect the structure of the theory upon which it is based. In this case, Theorems (2.3) and (2.4) give the theory. Correspondingly, *Planarity* tests planarity of $seam(c,s) \cup s$ using the theorems, and, within *Planarity*, *Planarity* is called once for each subsegment of $seam(c,s) \cup s$ in order to test its planarity. Within the body, a mixture of iteration and recursive calls lead to a simple algorithmic description. In the original Hopcroft-Tarjan planarity algorithm, this correspondence between theory and algorithm was not as evident. In fact, the purpose of their recursive procedure was never precisely stated.

Another key point was to write the algorithm in Sect. 5 completely in terms of sequence notation. A separation of correctness concerns (in terms of sequences) from efficiency concerns (in terms of linked lists) was extremely important —although the astute reader will notice that the algorithms in Sect. 5 were written in a particular style that was conducive to refinement of sequences by linked lists. At some point, it is hoped that our languages will allow the algorithm of Sect. 5 to be essentially the final one; through a simple command like '**implement** *b* **by** *ReversedLinkedList*', it should be possible to direct the system to implement automatically sequence *b* using a previously written reverse-linked-list module.

The original Hopcroft-Tarjan planarity algorithm used global arrays to implement the sequences, instead of Pascal-like pointers. This was a much more confusing data structure to work with, as a reading of the original paper [3] will show. We are currently implementing both a pointer version and a global-array version in Pascal to compare the two experimentally.

8. References

[0] Auslander, L., and S.V. Parter. On imbedding graphs in the plane. *J. Math. and Mech. 10*, 3 (May 1961), 517-523.

[1] Berge, C. *The Theory of Graphs and its Applications.* (Translated by Alison Doig), Methuen, London, 1964.

[2] Hall, D.W., and G. Spencer. *Elementary Topology.* Wiley, New York, 1955.

[3] Hopcroft, J.E., and R. Tarjan. Efficient planarity testing. *J. ACM 21*, 4 (October 1974), 549-568.

[4] Melhorn, K. *Graph Algorithms and NP-Completeness.* Springer Verlag, Heidelberg, 1984.