

On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm

Kurt Mehlhorn * Petra Mutzel †

Abstract

We give a detailed description of the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. The embedding phase runs in linear time. An implementation based on this paper can be found in [MMN93].

An undirected graph $G = (V, E)$ is called *planar* if it can be embedded into the plane without edge crossings, i.e., the vertices of G are mapped into distinct points in the plane and the edges of G are mapped into disjoint Jordan curves connecting their endpoints. An embedding of a planar graph induces a cyclic ordering on the edges incident to any fixed vertex, namely the clockwise ordering of the edges around their common endpoint. A graph G together with a cyclic ordering on the edges incident to any vertex is called a *map* or *combinatorial embedding*, it is called a *planar map* or *combinatorial planar embedding* if it is induced by some planar embedding.

Hopcroft and Tarjan [HT74] gave an algorithm that tests the planarity of an undirected graph in linear time. Alternative linear time algorithms were developed by Lempel, Even, and Cederbaum [LEC67, ET76], Booth and Lueker [BL76, CNAO85], and Fraysseix and Rosenstiehl [FR82]. Hopcroft and Tarjan also stated but gave no details that their planarity testing algorithm can be extended to also construct a combinatorial planar embedding. More details of the embedding phase are given in the textbook of the first author [Meh84, vol. 2] but the presentation is incorrect. Figure 1 shows a counterexample.

In this note we give a complete description of the embedding phase. An alternative presentation can be found in [Mut92]. Our embedding algorithm has the same recursive structure as the testing algorithm of Hopcroft and Tarjan and also runs in linear time. An implementation based on this note is described in [MMN93] and is distributed with the LEDA platform of combinatorial and geometric computing [Näh93, MN89] (anonymous ftp cs.uni-sb.de (134.96.325.31)).

The testing phase of the Hopcroft and Tarjan algorithm is discussed in detail in [Meh84, vol. 2, pages 96 – 111]. We summarize that discussion. The graph G is assumed to be biconnected. We also fix a particular DFS-tree of G and identify the vertices of G with their DFS-numbers. We direct all tree edges from lower to higher DFS-number and all non-tree edges from higher to lower DFS-number. Non-tree edges are called back edges. Figure 1 shows an example. We use T and B to denote the set of tree edges and back edges respectively.

We associate a *segment* $S(e)$ and a *cycle* $C(e)$ with every edge $e = (x, y)$ of G . If e is a back edge then $C(e)$ and $S(e)$ consist of the tree path from y to x and the edge e . If e is a

*Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

†Institut für Informatik, Universität zu Köln, Pohligstraße 1, 50969 Köln, Germany

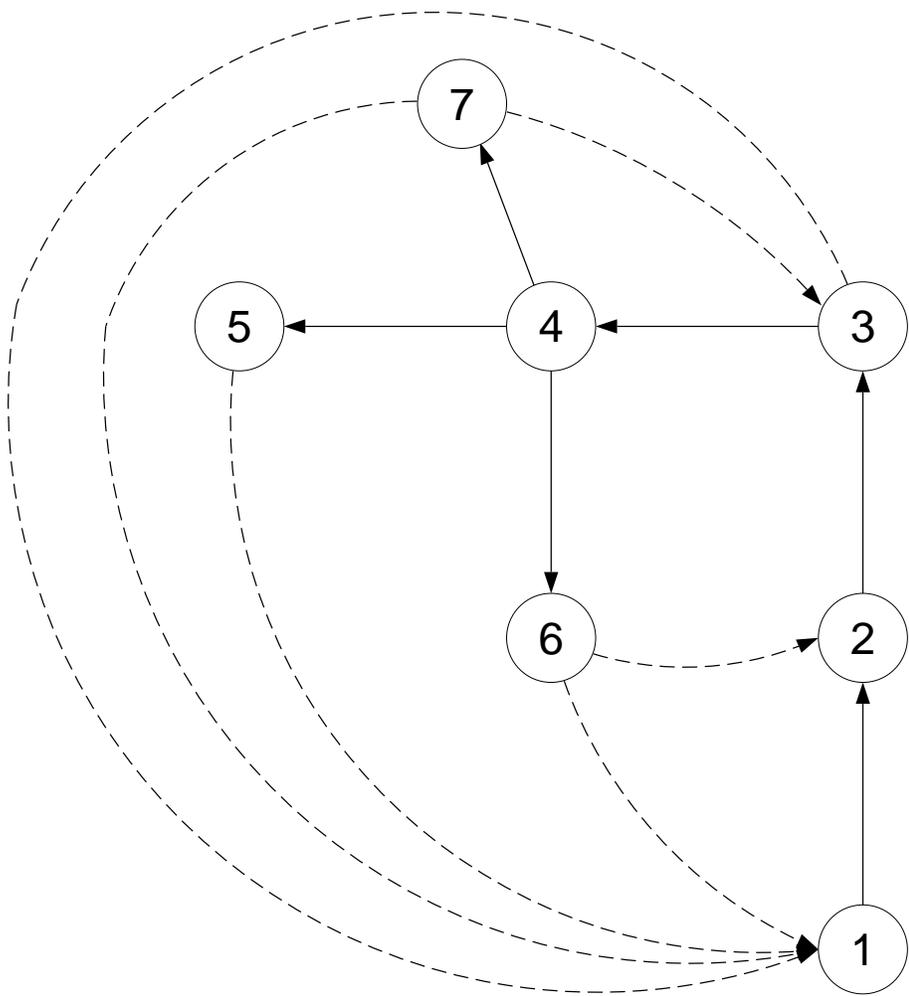


Figure 1: A DFS-tree of a planar graph. Tree edges are shown solid and back edges are shown dashed. The algorithm given in [Meh84, vol.2, page 113] fails on this example because it will incorrectly insert the edge $(1,7)$ between the edges $(1,6)$ and $(1,5)$ into the cyclic adjacency list of vertex 1.

tree edge then let $V(e)$ be the set of tree successors of y and let w_0 be the lowest numbered endpoint of any back edge starting in $V(e)$. The cycle $C(e)$ consists of a tree path from the vertex w_0 to a vertex $w \in V(e)$ with $(w, w_0) \in B$ and the back edge (w, w_0) and the segment $S(e)$ consists of $C(e)$, the subgraph induced by $V(e)$, and all back edges starting in a node in $V(e)$. Note that the segment $S(e)$ is uniquely defined but that there may be several choices for the cycle $C(e)$. We divide the tree path underlying the cycle $C(e)$ into two parts, its stem and its spine. The *stem* consists of the part ending in x . The *spine* is empty if e is a back edge and it is the part starting in y if e is a tree edge.

In our example, the cycle $C((3, 4))$ consists of the tree path from 1 to 5 followed by the back edge $(5, 1)$. The stem is the tree path from 1 to 3 and the spine is the tree path from 4 to 5. The cycle $C((1, 2))$ consists of the tree path from 1 to 3 and the back edge $(3, 1)$. Its stem is the node 1.

A segment $S(e)$ is called *strongly planar* if there is an embedding of $S(e)$ such that the stem of the cycle $C(e)$ borders the outer face. An embedding with this property is called a strongly planar embedding of $S(e)$. Let w_0, w_1, \dots, w_r with $e = (w_r, y)$ be the stem of $C(e)$. A strongly planar embedding of $S(e)$ is called *canonical (reversed canonical)* if for all i , $0 < i < r$, the edge $\{w_i, w_{i+1}\}$ immediately follows (precedes) the edge $\{w_i, w_{i-1}\}$ in the counterclockwise ordering of edges incident to w_i .

In Figure 1 the embeddings of segments $S((4, 6))$ and $S((4, 7))$ are both strongly planar, the embedding of $S((4, 6))$ is canonical and the embedding of $S((4, 7))$ is reversed canonical.

Since G is assumed to be biconnected there is exactly one tree edge out of vertex 1, namely the edge $(1, 2)$. Moreover $G = S((1, 2))$ and G is planar if $S((1, 2))$ is strongly planar.

Let e_0 be any edge and let $C = C(e_0)$ be the cycle associated with e_0 . An edge $e = (x, y)$ is said to *emanate* from C if x lies on the spine of C but e does not belong to C . If e_1, \dots, e_m are the edges emanating from C then $S(e_0) = C + S(e_1) + \dots + S(e_m)$, i.e., $S(e_0)$ is the union of the cycle C and the segments $S(e_1), \dots, S(e_m)$.

We need some more concepts. As above, let $C = C(e_0)$ and let $e = (x, y)$ emanate from C . The set $A(e)$ of *attachments* of segment $S(e)$ to cycle C is defined to be the set $\{x, y\}$ if e is a back edge and the set $\{x\} \cup \{z; (w, z) \text{ is a back edge, } w \in V(e) \text{ and } z \notin V(e)\}$ if e is a tree edge. Two segments $S(e)$ and $S(e')$ where e and e' emanate from C are said to *interlace* if either there are nodes $x < y < z < u$ on cycle C such that $x, z \in A(e)$ and $y, u \in A(e')$ or $A(e)$ and $A(e')$ have three points in common.

The *interlacing graph* $IG(C)$ with respect to cycle $C = C(e_0)$ is defined as follows. The nodes of $IG(C)$ are the segments $S(e)$ where e emanates from C . Also, $S(e)$ and $S(e')$ are connected by an edge iff $S(e)$ and $S(e')$ interlace. It is shown in [Meh84, vol.2, section IV.10, Lemma 4] that $S(e_0)$ is strongly planar if $S(e)$ is strongly planar for every e emanating from C , and there is a partition $\{L, R\}$ of the vertex set of $IG(C)$ such that no segment in L interlaces with a segment in R and such that $A(e) \cap \{w_1, \dots, w_{r-1}\} = \emptyset$ for any segment $S(e) \in R$ where $w_0, w_1, \dots, w_{r-1}, w_r$ is the stem of cycle C .

For reasons of efficiency, it is useful to order the adjacency list of any vertex v as follows: edge (v, w) is before edge (v, w') if $\min A((v, w)) < \min A((v, w'))$ or if $\min A((v, w)) = \min A((v, w'))$, $A((v, w))$ has cardinality two, and $A((v, w'))$ has cardinality three or more. In all other cases the order is irrelevant. We assume from now on that the cycle $C(e)$ for a tree edge $e = (x, y)$ is defined in the following way. Starting in y we construct a path by always taking the first edge out of each node until a back edge is encountered. The path *constructed* this way is the spine of the cycle $C(e)$.

The discussion above suggests a procedure *stronglyplanar* (e_0), cf. [Meh84, vol. 2, page 109] that given an edge e_0 decides the strong planarity of the segment $S(e_0)$. It first constructs

the cycle $C = C(e_0)$, then recursively tests the strong planarity of all segments $S(e)$, where e emanates from C , and finally tests, whether there is an appropriate bipartition of the vertex set of the interlacing graph. The recursive calls are made in the following order. If w_{r+1}, \dots, w_k is the spine of the cycle C then the segments $S((w_k,))$ are tested first, the segments $S((w_{k-1},))$ are tested next, For each fixed i the segments $S((w_i,))$ are tested in the order in which the edges $(w_i,)$ appear on the adjacency list of w_i . The call *stronglyplanar*((1, 2)) tests the strong planarity of segment $S((1, 2))$ and hence the planarity of G .

It is shown in [Meh84, vol. 2, page 112] that procedure *stronglyplanar* can also be used to compute a labelling α of the edges of G by L and R such that:

- an edge e is labelled iff *stronglyplanar*(e) is called
- edge (1, 2) is labelled L
- if e_0 is a labelled edge and e_1, \dots, e_m are the edges emanating from $C = C(e_0)$ then α induces the appropriate bipartition of the interlacing graph, i.e., if $\alpha(e_i) = \alpha(e_j)$ then $S(e_i)$ and $S(e_j)$ do not interlace and if $\alpha(e_j) = R$ then $A(e_j) \cap \{w_1, \dots, w_{r-1}\} = \emptyset$ where w_0, \dots, w_r is the stem of C .

The proof of [Meh84, vol. 2, section IV.10, Lemma 5] shows how a strongly planar embedding of $S(e_0)$ can be obtained from strongly planar embeddings of the $S(e_i)$'s:

To construct a canonical embedding of $S(e_0)$ draw the path w_0, \dots, w_k (consisting of stem w_0, \dots, w_r , edge $e_0 = (w_r, w_{r+1})$ and spine w_{r+1}, \dots, w_k) as a vertical upwards directed path, add edge (w_k, w_0) , and then for i , $1 \leq i \leq m$, and $\alpha(e_i) = L$ extend the embedding of $C + S(e_1) + \dots + S(e_{i-1})$ by glueing a canonical embedding of $S(e_i)$ onto the left side of the vertical path, and for i , $1 \leq i \leq m$, and $\alpha(e_i) = R$ extend the embedding of $C + S(e_1) + \dots + S(e_{i-1})$ by glueing a reversed canonical embedding of $S(e_i)$ onto the right side of the vertical path. Similarly, if the goal is to construct a reversed canonical embedding of $S(e_0)$ then, if $\alpha(e_i) = L$, a reversed canonical embedding of $S(e_i)$ is glued onto the right side of the vertical path, and if $\alpha(e_i) = R$, then a canonical embedding of $S(e_i)$ is glued onto the left side of the vertical path.

We can now give the algorithmic details. We first use procedure *stronglyplanar* to compute the mapping α . We then use a procedure *embedding* to actually compute an embedding. The procedure *embedding* takes two parameters: an edge e_0 and a flag $t \in \{L, R\}$. A call *embedding*(e_0, L) computes a canonical embedding of $S(e_0)$ and a call *embedding*(e_0, R) computes a reversed canonical embedding of $S(e_0)$. The call *embedding*((1, 2), L) embeds the entire graph.

The embedding of $S(e_0)$ computed by *embedding*(e_0, t) is represented in the following non-standard way:

1. For the vertices $v \in V(e_0)$ we use the standard representation, i.e., the cyclic list of the incident edges corresponding to the clockwise ordering of the edges in the embedding.
2. For the vertices in the stem we use a non-standard representation. For each vertex $w_i \in \{w_0, \dots, w_r\}$ let the lists $AL(w_i)$ and $AR(w_i)$ be such that the catenation of (w_i, w_{i+1}) , $AR(w_i)$, (w_i, w_{i-1}) , and $AL(w_i)$ corresponds to the clockwise ordering of the edges incident to w_i in the embedding. Here, $w_{-1} = w_k$. Note that $AR(w_i) = \emptyset$ for $1 \leq i < r$ if $t = L$, and $AL(w_i) = \emptyset$ for $1 \leq i < r$, if $t = R$. The lists $AL(w_i)$, $AR(w_i)$, $0 \leq i \leq r$, are returned in an implicit way: $AL(w_r)$ and $AR(w_r)$ are returned as the list $T = AL(w_r), (w_r, w_{r+1}), AR(w_r)$ and the other lists are returned as the list $A = AR(w_{r-1}), \dots, AR(w_0), (w_0, w_k), AL(w_0), \dots, AL(w_{r-1})$, cf. Figure 2.

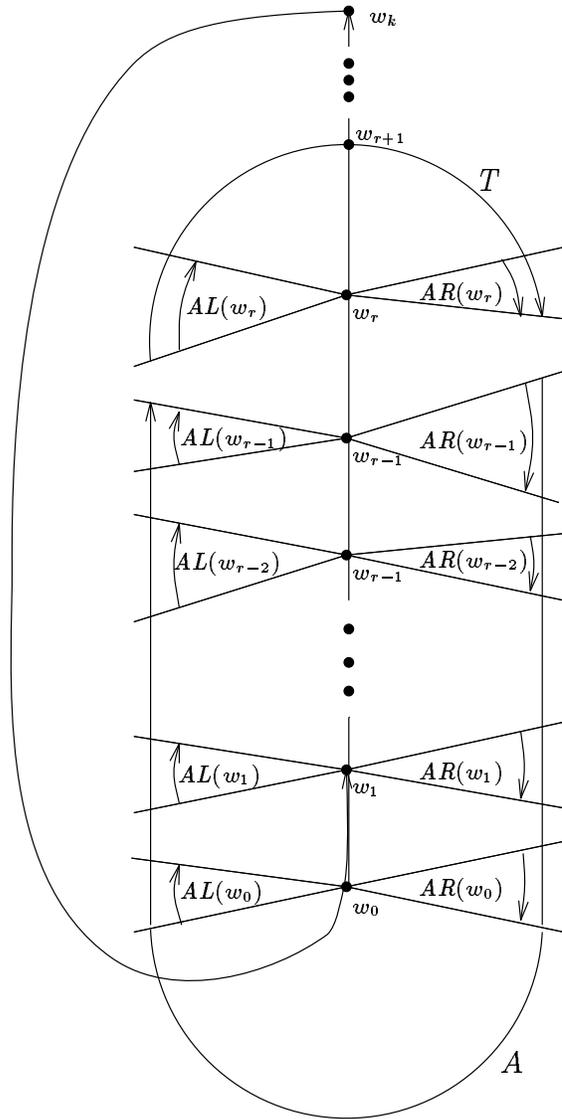


Figure 2: A call *embedding* (e_0, t) returns lists T and A . Lists are drawn as arrows. The arrowhead corresponds to the end of the list.

```

(0) procedure embedding( $e_0$ : edge,  $t$ :  $\{L, R\}$ )
    (* computes an embedding of  $S(e_0)$ ,  $e_0 = (x, y)$ , as described in the text;
    it returns the lists  $T$  and  $A$  defined in the text *)
(1) find the spine of segment  $S(e_0)$  by starting in node  $y$  and always
    take the first edge of every adjacency list until a back edge is
    encountered. This back edge leads to node  $w_0$ .
    Let  $w_0, \dots, w_r$  be the tree path from  $w_0$  to  $x = w_r$  and
    let  $w_{r+1} = y, \dots, w_k$  be the spine constructed above.
(2)  $AL \leftarrow AR \leftarrow$  empty list of darts;
     $T \leftarrow (w_k, w_0)$ ; (* a list of darts *)
(3) for  $j$  from  $k$  downto  $r + 1$ 
(4) do for all edges  $e'$  (except the first) emanating from  $w_j$ 
(5) do  $(T', A') \leftarrow \textit{embedding}(e', t \oplus \alpha(e'))$ 
(6) if  $t = \alpha(e')$ 
(7) then  $T \leftarrow T' \textit{ conc } T$ ;  $AL \leftarrow AL \textit{ conc } A'$ 
(8) else  $T \leftarrow T \textit{ conc } T'$ ;  $AR \leftarrow A' \textit{ conc } AR$ 
(9) fi
(10) od
(11) output  $(w_j, w_{j-1}) \textit{ conc } T$ ; (* the cyclic adjacency list of vertex  $w_j$  *)
(12) let  $AL = AL' \textit{ conc } T'$  and  $AR = T'' \textit{ conc } AR'$ 
    where  $T'$  and  $T''$  contain all darts incident to  $w_{j-1}$ ;
(13)  $AL \leftarrow AL'$ ;  $AR \leftarrow AR'$ ;  $T \leftarrow T' \textit{ conc } (w_{j-1}, w_j) \textit{ conc } T''$ 
(14) od
(15)  $A \leftarrow AR \textit{ conc } (w_0, w_k) \textit{ conc } AL$ ;
(16) return  $T$  and  $A$ 
(17) end

```

Table 1: The procedure *embedding*

The procedure *embedding* has the same structure as the procedure *stronglyplanar* and is given in Table 1. It first constructs the stem and the spine (line (1)) of cycle $C(e_0)$, then walks down the spine (lines (3) to (14)), and finally computes the lists T and A it wants to return (lines (15) and (16)).

We first discuss the walk down the spine. Suppose that the walk has reached vertex w_j . We first recursively process the edges emanating from w_j (lines (4) to (10)), and then compute the cyclic adjacency list of vertex w_j and prepare for the next iteration (lines (11) to (13)).

We discuss lines (4) to (10) first. In general, some number of edges emanating from w_j and all edges incident to vertices w_l with $l > j$ will have been processed already. Call the processed edges e_1, \dots, e_{i-1} . We claim that the following statement is an invariant of the loop (4) to (10): T concatenated with (w_j, w_{j-1}) is the cyclic adjacency list of vertex w_j in the embedding of $C + S(e_1) + \dots + S(e_{i-1})$, and AL and AR are the catenation of lists $AL(w_0), \dots, AL(w_{j-1})$ and $AR(w_{j-1}), \dots, AR(w_0)$ respectively where $(w_l, w_{l+1}), AR(w_l), (w_l, w_{l-1}), AL(w_l)$ is the cyclic adjacency list of vertex w_l , $0 \leq l \leq j - 1$, in the embedding of $C + S(e_0) + \dots + S(e_{i-1})$. The lists T , AL , and AR are certainly initialized correctly in line (2). Assume now that we process edge $e' = e_i$ emanating from w_j . The flag $\alpha(e')$ indicates what kind of embedding of $S(e_i)$ is needed to build a canonical embedding of $S(e_0)$; the opposite kind of embedding of $S(e_i)$ is needed to build a reversed canonical

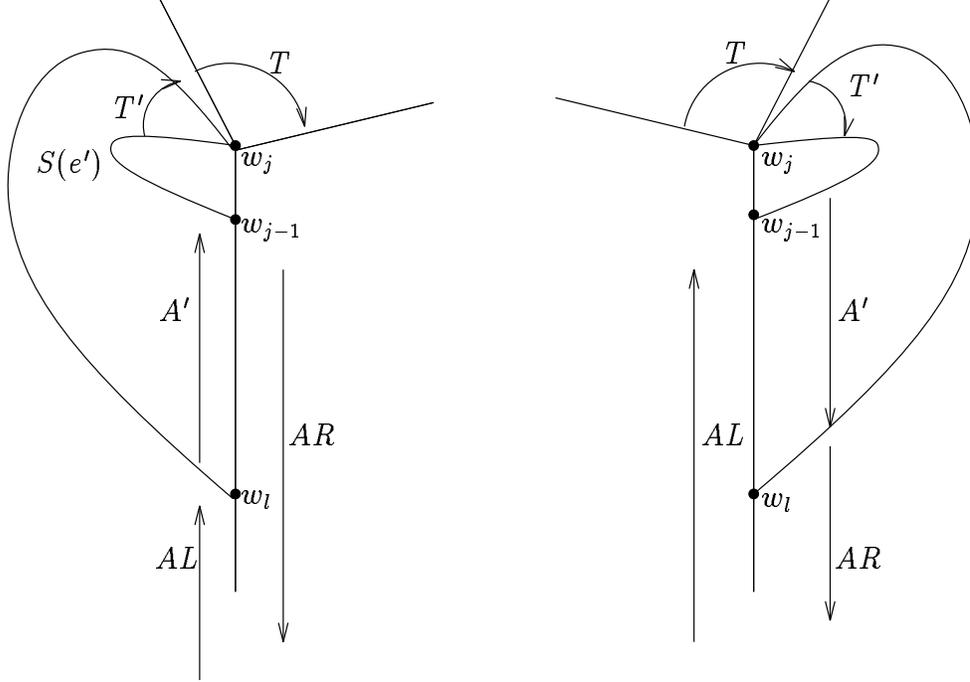


Figure 3: Glueing $S(e')$ to the left or right side of the path w_0, \dots, w_k respectively.

embedding of $S(e_0)$. So the required kind is given by $t \oplus \alpha(e')$, where $L \oplus L = R \oplus R = L$ and $L \oplus R = R \oplus L = R$. The call $embedding(e', t \oplus \alpha(e'))$ computes the cyclic adjacency lists of the vertices in $V(e')$ and returns lists T' and A' as defined above. If $S(e_i)$ has to be glued to the left side of the vertical path w_0, \dots, w_k , i.e., if $t = \alpha(e')$ then we append T' to the front of T and A' to the end of AL , cf. Figure 3. Analogously, if $S(e_i)$ has to be glued to the right side of the path w_0, \dots, w_k , i.e., if $t \neq \alpha(e')$, then we append T' to the end of T and A' to the front of AR . This clearly maintains the invariant.

Suppose now that we have processed all edges emanating from w_j . Then (w_j, w_{j-1}) concatenated with T is the cyclic adjacency list of vertex w_j (line (11)).

We next prepare for the treatment of vertex w_{j-1} . Let T' and T'' be the list of darts incident to w_{j-1} from the left and from the right respectively and having their other endpoint in an already embedded segment. List T' is a suffix of AL and list T'' is a prefix of AR . The catenation of T' , (w_{j-1}, w_j) , T'' , and (w_{j-1}, w_{j-2}) is the current clockwise adjacency list of vertex w_{j-1} . Thus lines (12) and (13) correctly initialize AL , AR , and T for the next iteration.

Suppose now that all edges emanating from the spine of $C(e_0)$ have been processed, i.e., control reaches line (15). At this point, list T is the ordered list of darts incident to w_r (except (w_r, w_{r-1})) and the two lists AL and AR are the ordered list of darts incident to the two sides of the stem of $C(e_0)$. Thus T and the catenation of AR , (w_0, w_k) , and AL are the two components of the output of $embedding(e_0, t)$. We summarize in

Theorem 1 *Let $G = (V, E)$ be a planar graph. Then G can be turned into a planar map (G, σ) in linear time.*

For an example let us consider the DFS-tree of G given in Figure 1. Consider the situation in the call of $embedding((3, 4), L)$. The call $embedding((4, 6), L)$ computes the cyclic adjacency lists of the vertices in $V((4, 6))$ and returns lists $T' = (4, 6)$ and $A' = (1, 6)(2, 6)$. In line (7), $T = (4, 6)(4, 5)$ and $AL = (1, 6)(2, 6)$. The call of $embedding((4, 7), R)$ gives $T' = (4, 7)$ and $A' = (3, 7), (1, 7)$. Thus in line (7) we have $T = (4, 6)(4, 5)(4, 7)$ and $AR = (3, 7)(1, 7)$. The adjacency list of node $w_j = 4$ is completed in line (11). It is $(4, 3), (4, 6), (4, 5), (4, 7)$. In line (13) we get $AL = (1, 6)(2, 6)$, $AR = (1, 7)$ and $T = (3, 4)(3, 7)$. At the end of $embedding((3, 4), L)$ we have $A = (1, 7)(1, 5)(1, 6)(2, 6)$.

An implementation based on this note is described in [MMN93] and is distributed with the LEDA platform of combinatorial and geometric computing [Näh93, MN89] (anonymous ftp cs.uni-sb.de (134.96.325.31)). Its running time is about 50 times the running time of the LEDA strongly connected components algorithm. More concretely, on a SUN ELC the program takes about 19 seconds to construct the planar map for a graph with 8000 nodes and 16000 edges. About 11 seconds are needed to make a copy of the input graph and to make the input graph biconnected and bidirected, about 4 seconds are needed for the planarity test, and about 4 seconds are needed to actually construct the embedding. The example graphs were generated by choosing an appropriate number of random line segments, computing their intersections, and putting a vertex on every endpoint and intersection. For a non-planar graph the algorithm can also be asked to identify a subdivision of a K_5 or $K_{3,3}$. This will however take quadratic time.

References

- [BL76] K. Booth and L. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *J. of Computer and System Sciences*, 13:335–379, 1976.
- [CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. of Computer and System Sciences*, 30(1):54–76, 1985.
- [ET76] S. Even and R.E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.
- [FR82] H.de Fraysseix and P. Rosenstiehl. A depth-first-search characterization of planarity. *Annals of Discrete Mathematics*, 13:75–80, 1982.
- [HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.
- [LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. *Theory of Graphs, Int. Symp.(Rome 1966)*, pages 215–232, 1967.
- [Meh84] K. Mehlhorn. *Data Structures and Efficient Algorithms*, volume I, II, III. Springer Verlag, Berlin, 1984.
- [MMN93] K. Mehlhorn, P. Mutzel, and St. Näher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Technical Report MPI-I-93-151, Max-Planck-Institut für Informatik, Saarbrücken, 1993.
- [MN89] K. Mehlhorn and St. Näher. LEDA: A library of efficient data types and algorithms. *LNCS*, 379:88–106, 1989. full version to appear in CACM.

- [Mut92] P. Mutzel. A fast linear time embedding algorithm based on the Hopcroft-Tarjan planarity test. Technical report, Universität zu Köln, 1992.
- [Näh93] St. Näher. LEDA manual. Technical Report MPI-I-93-109, Max-Planck-Institut für Informatik, 1993.