

# An $O(m \log n)$ -Time Algorithm for the Maximal Planar Subgraph Problem

*Jiazhen Cai*<sup>1</sup>

Courant Institute, NYU  
New York, NY 10012

*Xiaofeng Han*

Department of Computer Science  
Princeton University  
Princeton, NJ 08544

*Robert E. Tarjan*<sup>2</sup>

Department of Computer Science and NEC Research Institute  
Princeton University                      4 Independence Way  
Princeton, NJ 08544                      Princeton, NJ 08540

## *ABSTRACT*

Based on a new version of Hopcroft and Tarjan's planarity testing algorithm, we develop an  $O(m \log n)$ -time algorithm to find a maximal planar subgraph.

**Key words.** algorithm, complexity, depth-first-search, embedding, planar graph, selection tree

**AMS(MOS) subject classifications.** 68R10, 68Q35, 94C15

## 1. Introduction

In [15], Wu defined the problem of planar graphs in terms of the following four subproblems:

---

<sup>1</sup> This work was partly supported by Thomson-CSF/DSE and by the National Science Foundation under grant CCR90-02428.

<sup>2</sup> Research at Princeton University partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center, grant NSF-STC88-09648, and the Office of Naval Research, contract N00014-87-K-0467.

P1. Decide whether a connected graph  $G$  is planar.

P2. Find a minimal set of edges the removal of which will render the remaining part of  $G$  planar.

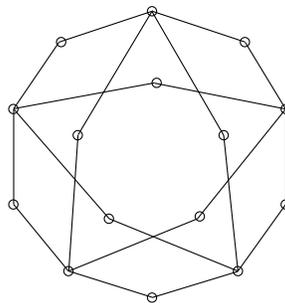
P3. Give a method of embedding  $G$  in the plane in case  $G$  is planar.

P4. Give a description of the totality of possible planar embeddings of  $G$  in the plane in case  $G$  is planar.

Linear-time algorithms for P1, P3, and P4 have been known for a long time. The first linear-time solution (which we call the H-T algorithm) for problem P1 ( the planarity-testing problem) was given by Hopcroft and Tarjan [7] in 1974 using depth-first search (DFS) trees. A P-Q tree solution for P1 based on an earlier algorithm given by Lempel, Even, and Cederbaum [11] was proved to have a linear-time implementation in 1976 partly by Even and Tarjan [4] and partly by Booth and Lueker [1]. The P-Q tree approach is conceptually simpler, but its implementation is more complicated than that of the H-T algorithm. Linear-time solutions for P3 and P4, also based on P-Q trees, were given by Chiba *et al.* [2] in 1985.

Wu [15] gave an algebraic solution for all four problems. He proved that a graph is planar if and only if a certain system of linear equations is solvable. In case the graph is planar, an actual embedding can be obtained by considering another system of quadratic equations. His solution is elegant, but his algorithm takes  $O(m^2)$  time on an  $m$ -edge graph.

Recently, Jayakumar *et al.* [9] studied problem P2 ( the maximal planar subgraph problem). For the special case in which a biconnected spanning planar subgraph is given, their algorithm runs in  $O(n^2)$  time and  $O(mn)$  space on a graph with  $n$  vertices and  $m$  edges. For more general situations, their algorithm runs in  $O(mn)$  time. Their algorithm is also based on P-Q trees. Note that not every biconnected graph has a biconnected spanning planar subgraph (See Fig. 1.)



A biconnected graph that does not have a biconnected spanning planar subgraph

Fig. 1

In this paper we give an  $O(m \log n)$ -time and  $O(m)$ -space solution to P2. For sparse graphs (*i.e.*, graphs with  $m = O(n^{1+\epsilon})$ , where  $\epsilon < 1$ ), it beats the algorithm of Jayakumar *et al.* even in the special case when a biconnected spanning planar subgraph is given. Independent of our work, Di Battista and Tamassia [3] have claimed an  $O(\log n)$ -time-per-operation solution to the problem of maintaining a planar graph under edge additions. Their algorithm also solves the minimal planar subgraph problem in  $O(m \log n)$  time. Their method is much more complicated than ours, however, as it is designed to solve a more general problem. Recently, Kobayashi *et al.* [10] have shown that if a Hamiltonian tour of the graph is given, then P2 can be solved in linear time. We show that this result can be easily derived from our algorithm as a special case.

The maximal planar subgraph problem is closely related to the planarity-testing problem. In fact, a graph is planar iff it is the maximal planar subgraph of itself. Our solution to the maximal planar subgraph problem is based on the H-T algorithm. But for our purpose, we need to modify the algorithm. The main difference is that our version of the algorithm admits a more general ordering than the original H-T algorithm does in processing the successors of each tree edge. Also, the H-T algorithm processes one path at a time, while our algorithm processes one edge at a time. In this sense, our algorithm is a more recursive version of the H-T algorithm.

For the above reason, many of our lemmas and theorems are similar, but not identical, to those in [7]. Instead of referring the readers to [7] for the proofs, we find it more convenient and accurate to supply all main proofs in this paper.

The rest of this paper is organized as follows. Section 2 gives preliminary definitions. Section 3 is a new version of the H-T planarity testing algorithm, which leads to our maximal planar subgraph algorithm in Section 4. Section 5 is a summary.

## 2. Preliminaries

Consider an undirected graph  $G_0 = (V_0, E_0)$  with edge set  $E_0$  and vertex set  $V_0$ . Let  $n = |V_0|$  and  $m = |E_0|$ . We can draw a picture  $G_0'$  of  $G_0$  in the plane as follows: for each vertex  $v \in V_0$ , we draw a distinct point  $v'$ ; for each edge  $(u, v) \in E_0$ , we draw a simple arc connecting the two points  $u'$  and  $v'$ . We call this arc an *embedding* of the edge  $(u, v)$ . For brevity, we will sometimes identify graphs with their pictures thus drawn on the plane. If no arcs of  $G_0'$  cross each other, then we call  $G_0'$  a *planar embedding*, or simply an *embedding*, of  $G_0$ . If  $G_0$  has an embedding, then we say that  $G_0$  is *planar*.

The following facts are important to our discussion:

**Observation 1.** Let  $C$  be a simple closed curve in the plane as in Fig. 2; let  $a$  be a point inside  $C$  and  $b$  be a point outside  $C$ . Then any curve that joins  $a$  and  $b$  crosses  $C$ .

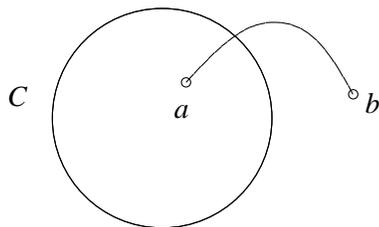


Fig. 2

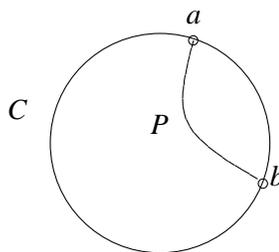


Fig. 3

**Observation 2.** Let  $G_1$  be the undirected graph represented by Fig. 3, in which  $P$  is a path joining the two vertices  $a$  and  $b$  on cycle  $C$ . Then in any embedding of  $G_1$ , all the edges of path  $P$  are on the same side of the cycle  $C$  (either inside or outside).

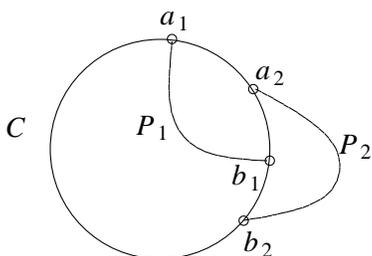


Fig. 4

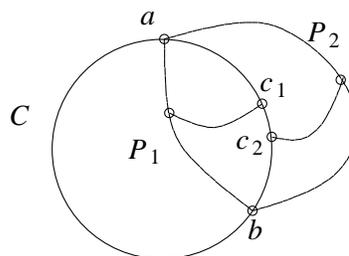


Fig. 5

**Observation 3.** Let  $G_2$  be the undirected graph represented by Fig. 4, in which  $a_1, a_2, b_1$  and  $b_2$  are four distinct vertices that appear in order on  $C$ . Then in any embedding of  $G_2$ , the two paths  $P_1$  and  $P_2$  are on opposite sides of the cycle  $C$ .

**Observation 4.** Let  $G_3$  be the undirected graph represented by Fig. 5, in which  $a, c_1, c_2$  and  $b$  are vertices that appear in order on  $C$ , and  $c_1$  and  $c_2$  may be the same. Then in any embedding of  $G_3$ , the two subgraphs  $P_1$  and  $P_2$  are on opposite sides of the cycle  $C$ .

All four observations above are intuitively obvious and follow from the Jordan Curve Theorem [6, 14].

A depth-first-search (*abbr.* DFS) [7] will convert the undirected graph  $G_0 = (V_0, E_0)$  into a directed graph  $G = (V, T, B)$ , where  $V$  is the set of DFS numbers of vertices in  $V_0$ ,  $T$  is the set of tree edges, and  $B$  is the set of back edges. Each edge of  $G_0$  is converted into either a tree edge or a back edge. All the tree edges form a DFS forest. If  $[a, b]$  is a tree edge, then  $a < b$ . If  $[a, b]$  is a back edge, then  $b < a$ , and there is a tree path in  $T$  from  $b$  to  $a$ . In either case,  $a$  is called the *tail* of  $[a, b]$ , and  $b$  is called the *head* of  $[a, b]$ . The union of  $T$  and  $B$  will be denoted by  $E$ .

For notational convenience, we will frequently identify undirected graphs with their DFS representations. Since we are interested only in graphs with no isolated vertices, we will represent

graphs with their edge sets.

We define *successors* for both vertices and edges. If  $[a, b]$  is a tree edge, then  $b$  is a *successor* of  $a$ . If  $[a, b]$  is a tree edge and  $[b, c]$  is any edge, then  $[b, c]$  is a *successor* of  $[a, b]$ . Back edges have no successors. We also define descendants and ancestors for both vertices and edges. A *descendant* of vertex (resp. edge)  $x$  is defined recursively as either  $x$  itself or a successor of a descendant of  $x$ . If  $y$  is a descendant of  $x$ , then  $x$  is an *ancestor* of  $y$ .

Let  $e = [a, b] \in E$ . Let  $Y$  be the set of vertices  $y$  such that for some  $x$ ,  $[x, y]$  is a back edge and also a descendant of  $e$ . If  $Y$  is not empty, we define  $low_1(e)$  to be the smallest integer in  $Y$ , and  $low_2(e)$  to be the second smallest integer in  $Y \cup \{n+1\}$ . Otherwise, we define  $low_1(e) = low_2(e) = n+1$ . The two mappings  $low_1$  and  $low_2$  can be computed in  $O(m)$  time during the depth-first-search on  $G_0$  [7]. If  $a$  is not the root of a DFS tree, and  $low_1(e) \geq a$ , then  $a$  is an articulation point of  $G$  [12].

If  $e = [a, b]$  is any edge in  $E$ , then we define the function  $\phi$  on  $E$  as follows.

$$\phi(e) = \begin{cases} 2 low_1(e) & \text{if } low_2(e) \geq a \\ 2 low_1(e) + 1 & \text{otherwise} \end{cases}$$

We arrange the successors of each tree edge in increasing order on their  $\phi$  values. This ordering can be computed in  $O(m)$  time using a bucket sort [7]. If  $e_1, \dots, e_k$  are the successors of  $e$  ordered this way, we will call  $e_i$  the  $i$ th successor of  $e$  for  $i = 1 \dots k$ .

As in [7], for  $e = [a, b]$ , we define  $S(e)$ , the *segment* of  $e$ , to be the subgraph of  $G$  that consists of all the descendants of  $e$ . We use  $ATT(e)$  to denote the set of back edges  $[c, d]$  in  $S(e)$  such that  $d$  is an ancestor of  $a$ , including  $a$  itself. Each back edge in  $ATT(e)$  is called an *attachment* of  $e$ .

For any edge  $e = [a, b]$ , we define  $cycle(e)$  as follows: if  $e$  is a back edge, then  $cycle(e) = \{e\} \cup \{e': e' \text{ belongs to the tree path from } b \text{ to } a\}$ ; if  $e$  is a tree edge and  $low_1(e) > a$ , then  $cycle(e) = \{\}$ ; otherwise,  $cycle(e) = cycle(e_1)$ , where  $e_1$  is the first successor of  $e$ . We use  $sub(e)$  to denote the subgraph  $S(e) \cup cycle(e)$ . It is easy to see that if  $cycle(e)$  is not empty, then the vertex  $low_1(e)$  is always on  $cycle(e)$ . Also, if  $low_1(e) \geq a$ , then  $sub(e) = S(e)$ ; if  $low_1(e) < a$ , then  $sub(e) - S(e) = \{e': e' \text{ belongs to the tree path from } low_1(e) \text{ to } a\}$ .

Fig. 6 illustrates some of these definitions, where  $low_1(e) = 1$ ;  $low_2(e) = 2$ ;  $cycle(e) = \{[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 1]\}$ ;  $S(e)$  contains all the edges in the graph except  $[1, 2], [2, 3], [3, 4]$ ;  $sub(e)$  is the whole graph;  $ATT(e) = \{[8, 1], [9, 3], [12, 1], [14, 2], [13, 4]\}$ .



embedded on the same side of  $\text{cycle}(e)$  as  $B_i$  must be in the same block.

The back edge on  $\text{cycle}(e)$  is the only attachment of  $e$  that will not be embedded on either side of  $\text{cycle}(e)$ . By convention, this back edge forms a block by itself, called the *singular block* of  $e$ , which does not interlace other blocks of  $\text{ATT}(e)$ .

In Fig. 6,  $\text{ATT}(e)$  consists of four blocks:  $B_1 = \{[8, 1]\}$ ,  $B_2 = \{[12, 1], [14, 2]\}$ ,  $B_3 = \{[9, 3]\}$ , and  $B_4 = \{[13, 4]\}$ .  $B_1$  is singular.  $B_2$  and  $B_3$  are interlacing.

If  $e' = [u, v]$  is an attachment of  $e$ , then  $\text{low}_1(e) \leq v \leq a$ . If  $\text{low}_1(e) < v < a$ , then we say that  $e'$  is *normal*. Otherwise we say that  $e'$  is *special*. A block of attachments of  $e$  is *normal* if it contains some normal attachment of  $e$ . Otherwise we say that it is *special*. In Fig. 6,  $B_1$  and  $B_4$  are special, and other blocks are all normal. We say that  $\text{sub}(e)$  is *strongly planar* w.r.t.  $e$  if  $e$  is planar and if all the normal blocks of  $\text{ATT}(e)$  can be embedded on the same side of  $\text{cycle}(e)$ . If  $\text{sub}(e)$  is strongly planar (w.r.t.  $e$ ), then we say that  $e$  is strongly planar. We have

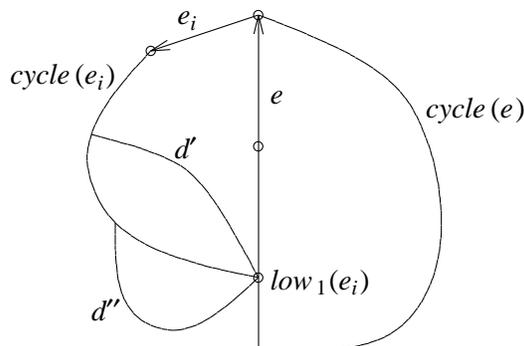
LEMMA 1. *Let  $e_i$  be the  $i$ th successor of  $e$ , where  $i > 1$ . Then  $e_i$  is strongly planar iff the subgraph  $S(e_i) \cup \text{cycle}(e)$  is planar.*

**Proof**

$\Rightarrow$  If  $e_i$  is strongly planar, then there is an embedding of  $\text{sub}(e_i)$  such that all its normal blocks are on the same side of  $\text{cycle}(e_i)$ . Thus we can add  $\text{cycle}(e)$  to the other side of  $\text{cycle}(e_i)$  to get an embedding of  $S(e_i) \cup \text{cycle}(e)$ .

$\Leftarrow$  If  $S(e_i) \cup \text{cycle}(e)$  is planar, then in any embedding of  $S(e_i) \cup \text{cycle}(e)$ , all the normal blocks of  $\text{ATT}(e_i)$  must be on the same side of  $\text{cycle}(e_i)$ . ■

Note that in an embedding of  $S(e_i) \cup \text{cycle}(e)$ , the special blocks of  $e_i$  do not have to be on the same side of  $\text{cycle}(e_i)$ . (See Fig. 7.)



The two special attachments  $d'$  and  $d''$  of  $e_i$  can be on different sides of  $\text{cycle}(e_i)$ , although they are on the same side of  $\text{cycle}(e)$ .

Fig. 7

We will represent a block of back edges  $H = \{[b_1, a_1], [b_2, a_2], \dots, [b_t, a_t]\}$  by a list  $K = [a_1, a_2, \dots, a_t]$ , where  $a_1 \leq a_2 \leq \dots \leq a_t$ . Frequently, we will identify blocks with their list representations. Define  $first(H) = first(K) = a_1$ , and  $last(H) = last(K) = a_t$ . If  $K$  is empty, we define  $first(H) = first(K) = n+1$ , and  $last(H) = last(K) = 0$ . We can further organize the blocks of  $ATT(e)$  as follows: if two blocks  $X$  and  $Y$  interlace, we put them into a pair  $[X, Y]$ , assuming  $last(X) \geq last(Y)$ ; if a nonempty block  $X$  does not interlace any other block, we form a pair  $[X, []]$ . Let  $[X_1, Y_1]$  and  $[X_2, Y_2]$  be two pairs of interlacing blocks. We say  $[X_1, Y_1] \leq [X_2, Y_2]$  iff  $last(X_1) \leq \min(first(X_2), first(Y_2))$ .

We say a list of interlacing pairs  $[q_1, \dots, q_s]$  is *well-ordered* if  $q_1 \leq \dots \leq q_s$ . Empty lists or lists of one pair are well-ordered by convention. We will see that all the interlacing pairs of  $ATT(e)$  can be organized into a well-ordered list  $[p_1, \dots, p_t]$ . We call this list  $att(e)$ .

In Fig. 6,  $att(e) = [p_1, p_2, p_3]$ , where  $p_1 = [[1], []]$ ,  $p_2 = [[3], [1, 2]]$ , and  $p_3 = [[4], []]$ .

### 3.2. Computing $att(e)$

Now we are ready to compute  $att(e)$ . The planarity of  $e$  will be decided at the same time.

Consider any edge  $e = [a, b]$ . If  $e$  is a back edge, then its only attachment is  $e$  itself. Therefore  $att(e) = [[[b], []]]$ . Otherwise, let  $e_1, \dots, e_k$  be the successors of  $e$  in increasing order by their  $\phi$  values. We first recursively compute  $att(e_i)$  for each successor  $e_i$  of  $e$ . Then we compute  $att(e)$  in four steps:

#### Algorithm 1

**Step 1** For  $i = 1 \dots k$ , delete all occurrences of  $b$  appearing in blocks within  $att(e_i)$ . Because these occurrences appear together at the end of the blocks that are contained in the last pairs of  $att(e_i)$  only, a simple list traversal suffices to delete all these occurrences in time  $O(k + \text{number of deletions})$ . After this, initialize  $att(e)$  to be  $att(e_1)$ .

**Step 2.** For  $i = 2 \dots k$ , merge all the blocks of  $att(e_i)$  into one intermediate block  $B_i$ . See Fig. 8.

According to Lemma 1, this step can only be done for a given value of  $i$  if the normal blocks of  $att(e_i)$  do not interlace. (If a pair of normal blocks of  $att(e_i)$  interlace, the graph is not planar, and the computation fails.) To merge the blocks for a given value of  $i$ , we traverse the list of pairs  $att(e_i)$ , concatenating blocks to form  $B_i$ . Initially,  $B_i$  is empty. To process pair  $[X, Y]$ , if  $X$  and  $Y$  are both normal, the computation fails, since the graph is not planar. Otherwise, we concatenate  $X$  and  $Y$  onto the end of  $B_i$  in order and continue. In case that  $Y$  is a special block, we know that  $Y$  has only attachments ending in  $low_1(e_i)$ , since  $b$  was deleted from all blocks of  $att(e_i)$  in Step 1. Thus the correct ordering of attachments is maintained by this process. This

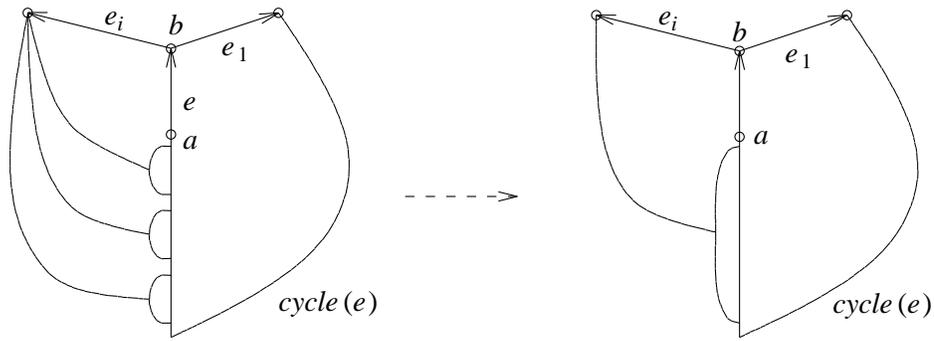


Fig. 8

step takes  $O(1 + \text{number of blocks in } att(e_i))$ , resulting in one block for each  $i$ .

**Step 3.** Merge blocks in  $att(e)$ . See Fig. 9.

By Observation 3, all blocks  $D$  in  $att(e)$  with  $last(D) > low_1(e_2)$  must be merged into one block  $B_1$ . (If any two of these blocks interlace, the graph is not planar, and the computation fails.) This is achieved by merging from the high end of  $att(e)$ . The time is  $O(1 + \text{reduction in number of blocks})$ . This step turns  $att(e)$  into a list of pairs  $p_1 \leq \dots \leq p_h$  with only  $p_h$  possibly having a block  $D$  with  $last(D) > low_1(e_2)$ . Note that  $low_1(e_2)$  is the lowest among the vertices  $low_1(e_2), \dots, low_1(e_k)$ .

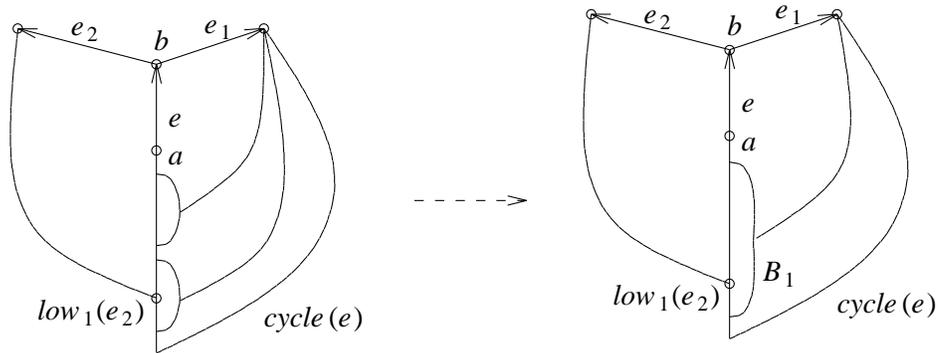


Fig. 9

**Step 4.** For  $i = 2, \dots, k$ , add blocks  $B_i$  into  $att(e)$ .

To process  $B_i$ , consider the highest pair  $P: [X, Y]$  of  $att(e)$ . Consider three subcases:

- i. If  $B_i$  cannot be embedded on either side of  $cycle(e)$ , then the computation of  $att(e)$  fails.
- ii. If  $B_i$  interlaces  $X$  only, then merge  $B_i$  into  $Y$  by concatenating their ordered list representations. Next, switch  $X$  and  $Y$  if  $last(X) < last(Y)$ .

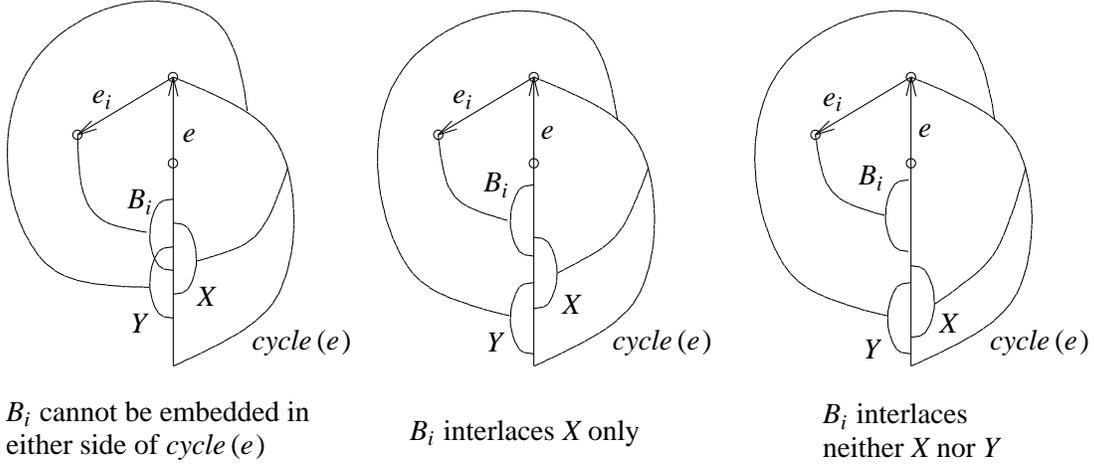


Fig. 10

iii. If  $B_i$  interlaces neither  $X$  nor  $Y$ , then add  $[B_i, [ ]]$  to the high end of  $\text{att}(e)$ ;  $P := [B_i, [ ]]$ .

By the following lemma, testing whether  $B_i$  interlaces  $X$  or  $Y$  takes  $O(1)$  time. Also by that lemma, it is not possible that  $B_i$  interlaces  $Y$  only, since  $\text{last}(X) \geq \text{last}(Y)$  ( see Fig. 10 ). ■

LEMMA 2. In Step 4,  $B_i$  and  $D$  can be embedded on the same side of  $\text{cycle}(e)$  iff  $\text{low}_1(e_i) \geq \text{last}(D)$ , where  $D = X$  or  $D = Y$ .

**Proof**

$\Rightarrow$  Assume  $\text{low}_1(e_i) < \text{last}(D)$ . Then there must be a path  $P_1$  in  $S(e_i)$  from  $b$  to  $\text{low}_1(e_i)$  containing some back edge in  $B_i$ , and another path  $P_2$  in  $S(e_1) \cup \dots \cup S(e_{i-1})$  from a vertex  $w$  on  $\text{cycle}(e)$  to  $\text{last}(D)$  containing some back edge in  $D$  but no edge on  $\text{cycle}(e)$ . We consider two cases ( see Fig. 11 ). If  $w > b$ , then by Observation 3,  $P_1$  and  $P_2$  cannot be embedded on the same side of  $\text{cycle}(e)$ . If  $w = b$ , then the first edge on  $P_2$  is  $e_j$  for some  $1 < j < i$ , which implies  $i > 2$  and  $\phi(e_j) \leq \phi(e_i)$ . Consequently,  $\text{low}_1(e_j) \leq \text{low}_1(e_i) < \text{last}(D) < b$ , which implies that  $\text{low}_2(e_j) < b$ . If  $\text{low}_1(e_j) < \text{low}_1(e_i)$ , then there must be an undirected simple path  $P_3$  between  $\text{last}(D)$  and  $\text{low}_1(e_j)$  containing back edges in  $D$  but no edges on  $\text{cycle}(e)$ . By observation 3 again,  $P_1$  and  $P_3$  cannot be embedded on the same side of  $\text{cycle}(e)$ . If  $\text{low}_1(e_j) = \text{low}_1(e_i)$ , then  $\text{low}_2(e_i) < b$  (recall  $\text{low}_1(e_j) \leq \text{low}_1(e_i)$ ). By observation 4,  $S(e_i)$  and  $S(e_j)$  cannot be embedded on the same side of  $\text{cycle}(e)$ . All of the above cases imply that  $B_i$  and  $D$  cannot be embedded on the same side of  $\text{cycle}(e)$ .

$\Leftarrow$  See the proof of Lemma 4 in the next section. ■

THEOREM 1.

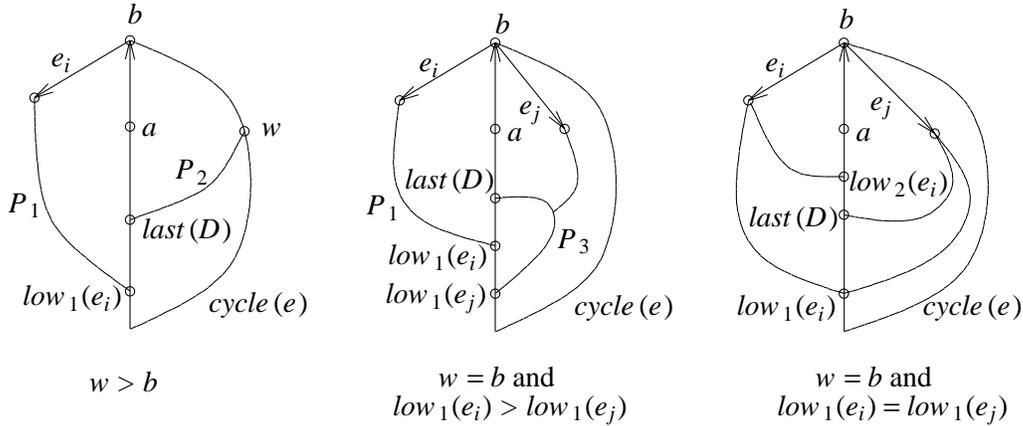


Fig. 11

1. Algorithm 1 computes  $att(e)$  successfully iff  $e$  is planar.
2. If  $e$  is planar, then Algorithm 1 computes  $att(e)$  correctly.

**Proof** See the next section. ■

### 3.3. Correctness

In the following proofs, unless stated otherwise, we will use  $att(e)$  to mean the list  $att(e)$  computed by Algorithm 1. But we will prove that this  $att(e)$  correctly implements the  $att(e)$  defined in Section 3.1.

During the presentation of Algorithm 1, we explained that two nonempty blocks form a pair within  $att(e)$  only if they cannot be embedded on the same side of  $cycle(e)$ , and the computation of  $att(e)$  fails only when  $e$  is not planar. Also we can see that the singular block of  $e$  is not merged with any other block. To prove Theorem 1, we still have to show that the following assertions are true:

- (1) if computation of  $att(e)$  succeeds, then  $e$  is planar;
- (2) if any two nonempty nonsingular blocks of  $att(e)$  do not form a pair, then these blocks can be embedded on the same side as well as on different sides of  $cycle(e)$ ;
- (3)  $att(e)$  is well-ordered.

We first prove (3), *i.e.*,

LEMMA 3. *The list of pairs  $att(e)$  computed by Algorithm 1 is well-ordered.*

**Proof** We prove this lemma by induction on the number of descendants of  $e$ . If  $e$  has no successor, then  $e$  is a back edge, and the lemma is trivially true. Now assume that  $e$  is a tree edge with successors  $e_1, \dots, e_k$  in increasing order by  $\phi$  value, and that  $att(e_1), \dots, att(e_k)$  are all well-ordered. After Steps 1 and 2 are executed,  $att(e_1), \dots, att(e_k)$  are still well-ordered. Thus,  $att(e)$

is well-ordered when it is initialized to  $att(e_1)$ . In Step 3, only blocks in the highest pairs of  $att(e)$  are merged, and therefore  $att(e)$  is still well-ordered after the merge. Then consider the moment in Step 4 just before  $B_i$  is added to  $att(e)$ . Assume  $att(e)$  is well-ordered at this moment. Let  $P: [X, Y]$  be the last pair of  $att(e)$ . We need only consider the two cases in which the computation does not fail.

1.  $last(Y) \leq low_1(B_i) < last(X)$ . Then  $B_i$  is merged with  $Y$ . If  $P$  is the only pair in  $att(e)$ , then  $att(e)$  is well-ordered by definition. Otherwise, let  $Q: [X_1, Y_1]$  be the pair next to  $P$ . Then we have  $Q \leq P$  before merge. We need only to show that this is still true after the merge. If  $i = 2$ , then Step 3 guarantees that  $first(B_2) = low_1(e_2) \geq \max(last(X_1), last(Y_1))$ . If  $i > 2$ , then  $B_{i-1}$  is contained in either  $X$  or  $Y$ . Since  $first(B_i) = low_1(e_i) \geq low_1(e_{i-1}) = first(B_{i-1}) \geq \min(first(X), first(Y))$ , then merging  $B_i$  into  $Y$  does not change the value of  $\min(first(X), first(Y))$ . Thus, after merging  $Y$  and  $B_i$ , we still have  $Q \leq P$ .

2.  $low_1(B_i) \geq last(X)$ . Then  $[B_i, [ ]]$  becomes the last pair of  $att(e)$ . Since  $last(X) \leq low_1(e_i) = first(B_i)$  in this case, we have  $P \leq [B_i, [ ]]$ .

Thus,  $att(e)$  is still well-ordered after each  $B_i$  is added,  $i = 2 \dots k$ . Therefore  $att(e)$  is well-ordered after Step 4. ■

Next we prove the *if* part of Lemma 2: In Step 4 of Algorithm 1, if  $low_1(e_i) \geq last(D)$ , then  $B_i$  and  $D$  can be embedded on the same side of  $cycle(e)$ , where  $D = X$  or  $D = Y$ .

**Proof** Consider an embedding of  $sub(e)$  before  $B_i$  is added. Assume without loss of generality that  $X$  is embedded on the left hand side of  $cycle(e)$  in this embedding. Let  $h = last(X)$ . Then  $h = \max\{last(Z): Z \text{ is a block in } att_i\}$ , and there is a face  $F$  on the left of  $cycle(e)$  in the current embedding of  $sub(e)$  such that the tree path from  $h$  to  $b$  is on the boundary of  $F$ . See Fig. 12.

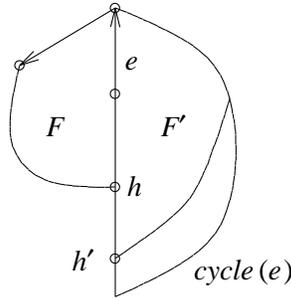


Fig. 12

Thus, if  $low_1(B_i) \geq last(X)$ , then  $B_i$  can be embedded in  $F$ . Similarly, let  $h' = \max\{last(U): U \text{ is a block in } att_i \text{ embedded on the right hand side of } cycle(e)\}$ . Then there is a face  $F'$  on the right of  $cycle(e)$  in the current embedding of  $sub(e)$  such that the tree path from  $h'$  to  $b$  is on the

boundary of  $F'$ . According to the proof of Lemma 3, if  $low_1(B_i) \geq last(Y)$ , then  $low_1(B_i) \geq h'$  also. Therefore  $B_i$  can be embedded in  $F'$  in this case. ■

Then we prove the following lemma that implies the assertions (1) and (2). We say that a set  $W$  of blocks of  $att(e)$  is *consistent* w.r.t.  $e$  if for all  $X, Y \in W$ , neither  $[X, Y]$  nor  $[Y, X]$  is in  $att(e)$ .

LEMMA 4.. *If Algorithm 1 does not fail, and  $D_1$  and  $D_2$  are two disjoint consistent sets of nonsingular blocks from  $att(e)$ , then there is an embedding of  $sub(e)$  such that blocks of  $D_1$  are on one side of  $cycle(e)$  and blocks of  $D_2$  are on the other side of  $cycle(e)$ .*

**Proof** The lemma is trivially true if  $e$  has no successors. If  $e$  has successors, let  $e_1, \dots, e_k$  be the list of successors of  $e$  in increasing order by their  $\phi$  values. Assume that the lemma holds for each of these successors. We want to construct an embedding of  $sub(e)$  such that  $D_1$  and  $D_2$  are embedded on different sides of  $cycle(e)$ .

If  $W$  is a set of blocks, then a  $W$ -attachment is an attachment contained in some block of  $W$ . For  $j = 1, 2$ , let  $D_j' = \{ X : [X, Y] \text{ or } [Y, X] \text{ is a pair in } att(e) \text{ and } Y \in D_j \}$ . Let  $H_1 = \{ X : ([X, Y] \text{ is a pair in } att(e)) \text{ and } (X \text{ and } Y \text{ are not in } D_1 \cup D_2) \}$ , and let  $H_2 = \{ Y : ([X, Y] \text{ is a pair in } att(e)) \text{ and } (X \text{ and } Y \text{ are not in } D_1 \cup D_2) \}$ . Let  $C_1 = D_1 \cup D_2' \cup H_1$ , and  $C_2 = D_2 \cup D_1' \cup H_2$ . For  $j = 1, 2$ , let  $K_j = \{ X : X \text{ is a block in } sub(e_1) \text{ containing some } C_j\text{-attachment} \}$ . Then  $K_1$  and  $K_2$  are two disjoint consistent subsets of blocks of  $sub(e_1)$ .

Initially, we construct an embedding of  $sub(e_1)$  such that  $K_1$  and  $K_2$  are on different sides of  $cycle(e_1)$ . As a result, those  $C_1$ -attachments and  $C_2$ -attachments contained in  $sub(e_1)$  are on different sides of  $cycle(e)$  (which is  $cycle(e_1)$ ). This embedding exists by the induction hypothesis. Take this embedding to be the initial embedding of  $att(e)$ . Then for  $i = 2, \dots, k$ , we add  $sub(e_i)$  to this embedding one by one as follows.

Since the normal blocks of  $att(e_i)$  do not interlace, we can, by induction, find an embedding of  $sub(e_i)$  such that all of its normal blocks are embedded on the same side of  $cycle(e_i)$ . We call this embedding  $E_i$ , and its mirror image  $E_i'$ . Let  $B_i, P$  and  $[X, Y]$  be the same as in Step 4 of algorithm 1. Assume without loss of generality that  $X$  is embedded on the left hand side of  $cycle(e)$ . Consider two cases:

Case 1:  $last(Y) \leq low_1(e_i) < last(X)$ . Then  $B_i$  is merged with  $Y$ . According to Lemma 2, one of  $E_i$  or  $E_i'$  can be embedded on the right of  $cycle(e)$ . If  $B_i$  contains any  $C_1$ -attachment, then  $X$  contains some  $C_2$ -attachment; if  $B_i$  contains any  $C_2$ -attachment, then  $X$  contains some  $C_1$ -attachment. In any case,  $C_1$ -attachments and  $C_2$ -attachments are still on different sides of  $cycle(e)$  after  $B_i$  is embedded.

Case 2:  $low_1(e_i) \geq last(X)$ . Again by to Lemma 2, one of  $E_i$  or  $E_i'$ , say  $E_i$ , can be embedded on the left of  $cycle(e)$ , and the other,  $E_i'$ , can be embedded on the right of  $cycle(e)$ . One of

these two choices will result in an embedding such that  $C_1$ -attachments are on one side of  $\text{cycle}(e)$ , and  $C_2$ -attachments are on the other side.

When all the  $B_i$ 's are added, we get an embedding of  $\text{sub}(e)$  such that  $D_1$  is on one side of  $\text{cycle}(e)$ , and  $D_2$  is on the other side of  $\text{cycle}(e)$ . This is true because every  $D_1$ -attachment is a  $C_1$ -attachment, and every  $D_2$ -attachment is a  $C_2$ -attachment. ■

This completes the proof of Theorem 1, and establishes that the list  $\text{att}(e)$  computed by Algorithm 1 has the properties discussed at the end of Section 3.1.

Let  $e$ ,  $e_i$ ,  $B_i$ ,  $\text{att}_i$ ,  $X$ ,  $Y$ , and  $h'$  be the same as in the above proofs. Let  $L = \{Z: [Z, U] \text{ is a pair in } \text{att}_i\}$ ,  $R = \{U: [Z, U] \text{ is a pair in } \text{att}_i\}$ , and  $h_2 = \{\text{last}(U): U \text{ is a block in } R\}$ . Then it is easy to see that  $h' \geq h_2$ . According to Lemma 4, there exists an embedding of  $\text{sub}(e)$  (before adding  $B_i$ ) such that  $L$  is embedded in one side of  $\text{cycle}(e)$  and  $R$  is embedded on the other side. According to the proof of Lemma 2,  $\text{low}_1(e_i) \geq \text{last}(Y)$  iff  $\text{low}_1(e_i) \geq h_2$ . Therefore we also have

**COROLLARY 1.1.**  $B_i$  cannot be embedded in either side of  $\text{cycle}(e)$  iff  $\text{low}_1(e_i) < h_2$ . ■

Corollary 1.1 gives a test whether  $B_i$  can be added to  $\text{att}_i$  without referring to the top pair of  $\text{att}_i$ . This is useful in our maximal planar subgraph algorithm, where we need to test whether  $B_2$  can be added to  $\text{att}(e)$  before Step 3 is performed.

### 3.4. Data structure and running time

As suggested in [7], we can implement blocks as linked lists. An interlacing pair of blocks can be represented as a record containing two pointers to the two linked lists representing these two blocks. Then  $\text{att}(e)$  can be represented as a linked list of such records. In this way, the time cost for Step 1 is  $O(k + \text{number of deletions})$ . The cost for Step 2, 3 and 4 is  $O(k + \text{reduction in number of blocks})$ . The cumulative expense of executing these steps over the whole graph is  $O(m)$ . The initial DFS in which  $\text{low}_1$  values are computed takes time  $O(m)$ . Arranging the successors in increasing order by  $\phi$  value for all tree edges takes  $O(m)$  time using a bucket sort. Thus the whole algorithm runs in  $O(m)$  time. It is well known that any  $O(m)$ -time algorithm for planarity testing can be implemented in  $O(n)$  time since  $m = O(n)$  for a planar graph [7].

### 3.5. A modification to Algorithm 1

Consider Step 4 of Algorithm 1. Lemma 2 requires that the successors of each tree edge be ordered by  $\phi$  values. Maintaining this ordering causes difficulties in solving the maximal planar subgraph problem. Fortunately, we can modify Algorithm 1 so that it requires only the  $\text{low}_1$  ordering of the successors of each tree edge.

Let  $e = [a, b]$  be a tree edge, and  $e_1, \dots, e_k$  be the list of its successors in increasing order by  $low_1$  values. Still define  $cycle(e) = cycle(e_1)$ . Then Step 1, 2, and 3 can be performed w.r.t. this ordering without any modification.

Next we want to merge  $B_2, \dots, B_k$  into  $att(e)$  in that order. In general, successors ordered by  $low_1$  values may not be ordered by  $\phi$  values. Consequently, there may be some  $1 < i \leq k$  such that  $\phi(e_{i-1}) > \phi(e_i)$ . But if this happens, we know that  $low_1(e_{i-1}) = low_1(e_i)$  and  $low_2(e_i) \geq b$ . If  $i = 2$ , Lemma 2 still applies, and we can merge  $B_2$  into  $att(e)$  as before. Otherwise, the following lemma says that we do not have to merge  $B_i$  into  $att(e)$  at all:

**LEMMA 5.** *If for some  $2 < i \leq k$ ,  $low_1(e_{i-1}) = low_1(e_i)$ ,  $low_2(e_i) \geq b$ , and  $e_i$  is planar, then  $G$  is planar iff  $G - S(e_i)$  is planar.*

**Proof** The *only if* part is trivial, so we just prove the *if* part. Consider an embedding  $E_i$  of  $G_i = G - S(e_i)$ . Under the condition of the lemma,  $e_i$  has no normal attachments. Since  $e_i$  is planar, then  $e_i$  is strongly planar. Also,  $b$  and  $low_1(e_i)$  are the only two vertices shared by  $S(e_i)$  and  $G_i$ . Therefore  $S(e_i)$  can be embedded in any face of  $E_i$  whose boundary contains the two vertices  $b$  and  $low_1(e_i)$ .

Let  $P$  be the tree path  $cycle(e_{i-1}) \cap cycle(e)$  and let  $C$  be the closed curve  $cycle(e_{i-1}) \cup cycle(e) - P$ . Then  $C$  contains edges from both  $S(e_{i-1})$  and  $G_i - S(e_{i-1})$ . By Observation 2,  $P$  is on one side of  $C$ . Call this side of  $C$   $S_1$ , and the other side  $S_2$ . Let  $U$  be the set of faces in  $S_2$  whose boundaries contain edges from  $S(e_{i-1})$  only, and let  $W$  be the set of faces in  $S_2$  whose boundaries contain edges from  $G_i - S(e_{i-1})$  only. Then faces in  $U$  and faces in  $W$  do not share common boundaries. Thus, within  $S_2$  there must be some face  $F$  whose boundary contains edges from both  $S(e_{i-1})$  and  $G_i - S(e_{i-1})$ , and therefore contains at least two vertices common to  $S(e_{i-1})$  and  $G_i - S(e_{i-1})$ . But all the vertices common to  $S(e_{i-1})$  and  $G_i - S(e_{i-1})$  are on  $P$ , and among them only  $b$  and  $low_1(e_{i-1})$  are on the boundary of  $S_2$ . Therefore these two vertices must be on the boundary of  $F$ . Thus we can embed  $S(e_i)$  in  $F$  to get an embedding of  $G$ . ■

Therefore, under the conditions of Lemma 5, in deciding the planarity of  $G$ , we can ignore its subgraph  $S(e_i)$ . Since the condition  $low_1(e_{i-1}) = low_1(e_i)$  and  $low_2(e_i) \geq b$  is implied by  $low_1(e_{i-1}) \leq low_1(e_i)$  and  $\phi(e_{i-1}) > \phi(e_i)$ , we can modify Step 4 as follows:

**Step 4'.** Add blocks  $B_2, \dots, B_k$  into  $att(e)$  in that order, assuming  $low_1(e_1) \leq low_1(e_2) \leq \dots \leq low_1(e_k)$ . Initially, let  $j = 1$  and  $i = 2$ . To process  $B_i$ , we consider two cases. If  $j = 1$  or  $\phi(e_j) \leq \phi(e_i)$ , we do the same thing as in Step 4, and then let  $j = i$ ; otherwise, we do nothing.

The list  $att(e)$  computed by the modified algorithm may not contain all the attachments of  $e$ . Some attachments may be omitted by Step 4', because their existence does not affect the planarity of the whole graph  $G$ .

#### 4. The maximal planar subgraph problem

Now we consider the maximal planar subgraph problem: find a minimal set of edges whose deletion results in a planar graph. The resulting graph is called a *maximal planar subgraph* of  $G$ . We can always find a maximal planar subgraph of  $G$  by deleting back edges only, since all the tree edges form a forest, which is planar.

We will not assume that the input graph is biconnected, since deletion of back edges may turn a biconnected graph into a graph with articulation points. But without loss of generality we can assume that the input graph is connected. Thus the tree edges of  $G$  form a single tree with root  $r$ . Let  $t_1, \dots, t_s$  be the tree edges leaving the root. If  $s = 1$ , then  $sub(t_1)$  is the whole graph  $G$ . If  $s > 1$ , then  $r$  is the only vertex common to  $sub(t_1), \dots, sub(t_s)$ . Thus, to find a maximal planar subgraph of  $G$ , we can just find a maximal planar subgraph for each of the subgraphs  $sub(t_1), \dots, sub(t_s)$ , and then simply put these subgraphs together. Therefore, what we need is a procedure that can find a maximal planar subgraph of  $sub(e)$  for any given edge  $e$  of  $G$ .

##### 4.1. Maximal $l$ -planar subgraphs

We cannot build a maximal planar subgraph of  $sub(e)$  by simply putting together the maximal planar subgraphs of  $sub(e_1), \dots, sub(e_k)$ , and deleting those back edges causing failure in Algorithm 1. The reason is that after these edges are deleted, it may turn out that some other edges, which we deleted for making  $sub(e_1), \dots, sub(e_k)$  planar, would not have had to be deleted at all. We avoid this difficult situation by constructing such maximal subgraphs  $S_1, \dots, S_k$  of  $sub(e_1), \dots, sub(e_k)$  that they can be used to construct a planar subgraph  $S$  of  $sub(e)$  without further deletion of edges. Two measures are taken for this purpose. Firstly, those back edges in  $sub(e_i)$  that can cause failure in Step 3 or Step 4 of Algorithm 1 are deleted before a maximal subgraph of  $S_i$  is recursively computed. Secondly, the information where blocks of  $sub(e_i)$  are allowed to interlace is passed to the recursive call that computes  $S_i$ , so that when the returned  $S_i$  is merged to  $sub(e)$ , Step 2 of Algorithm 1 can also be performed successfully without deletion. Since the planar subgraph  $S$  of  $sub(e)$  computed by our algorithm may be used to build a larger planar subgraph of  $G$  in the same way as we use  $S_1, S_2, \dots, S_k$  to build  $S$ , we also need to know where in  $S$  blocks are allowed to interlace. This approach leads naturally to the concept of  *$l$ -planar subgraphs*, which is a generalization of the concept of strongly planar subgraphs.

Consider an edge  $e = [a, b]$  and a vertex  $l$  on the tree path from  $low_1(e)$  to  $a$ . An attachment  $[u, v]$  of  $e$  is  *$l$ -normal* if  $low_1(e) < v < l$ . A block of attachments is  *$l$ -normal* if it contains some  $l$ -normal attachment. Let  $D$  be the list representation of a nonempty block of attachments. Define  $second(D)$  to be the second smallest element in the set  $\{x: x \in D\} \cup \{n+1\}$ , and define  $second(\emptyset) = n+1$ . Then  $D$  is  *$l$ -normal* iff  $low_1(e) < first(D) < l$  or  $second(D) < l$ . The two mappings *first* and *second* can be maintained during the computation of  $att(e)$  in  $O(1)$  time

for each modification to  $att(e)$ .

We say that the subgraph  $sub(e)$  is  $l$ -planar if  $e$  is planar and the  $l$ -normal blocks of  $att(e)$  do not interlace. (c.f. Fig. 13, where (a) is  $l$ -planar, but (b) is not.) Edge  $e$  is  $l$ -planar if  $sub(e)$  is  $l$ -planar.

If  $H$  is a subgraph obtained from  $sub(e)$  by deleting back edges only, then we can define the  $l$ -planarity for  $H$  (w.r.t.  $e$ ) in the same way as we did for  $sub(e)$ . We will talk about  $l$ -planar subgraphs of  $sub(e)$  in this sense. An  $l$ -planar subgraph of  $sub(e)$  is *maximal* if it can be obtained from  $sub(e)$  by deleting a minimal set of back edges.

Consider edge  $e = [a, b]$ . According to our definition,  $e$  is planar iff  $e$  is  $low_1(e)$ -planar, and  $e$  is strongly planar iff  $e$  is  $a$ -planar. Therefore, if we can find a maximal  $l$ -planar subgraph of  $sub(e)$  for any  $l$  with  $low_1(e) \leq l \leq a$ , then we can compute a maximal planar subgraph of  $sub(e)$ .

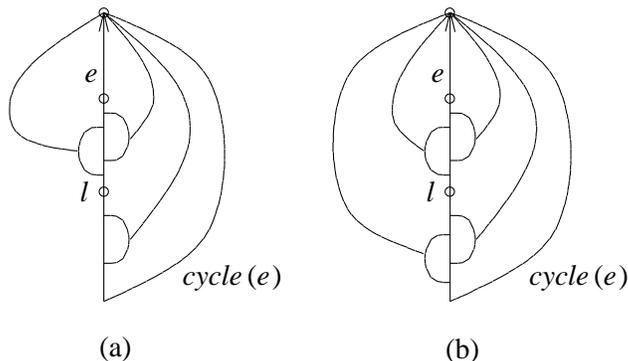


Fig. 13

The following is an outline of our maximal  $l$ -planar subgraph algorithm, where  $l$  is a given integer with  $low_1(e) \leq l \leq a$  and remains fixed during the processing of an edge. Let  $e = [a, b]$ , and consider three cases:

Case 1:  $e$  is a back edge. Assign  $[[[b], [ ]]]$  to  $att(e)$ , and return.

Case 2:  $e$  is a tree edge with no successors. Assign  $[ ]$  to  $att(e)$ , and return.

Case 3:  $e$  is a tree edge with successors  $e_1, \dots, e_k$ , among which  $e_1$  has the smallest  $low_1$  value. We construct a sequence  $G_1, \dots, G_k$  of  $l$ -planar subgraphs of  $sub(e)$  such that  $G_1$  is a maximal  $l$ -planar subgraph of  $sub(e_1)$  and  $low_1(e_1)$  remains unchanged;  $G_k$  is a maximal  $l$ -planar subgraph of  $sub(e)$ ; and each  $G_i$ ,  $1 < i \leq k$ , is obtained from  $G_{i-1}$  by adding to it a strongly planar subgraph  $S_i$  of  $sub(e_i)$ , where  $e_i$  is some successor of  $e$  not contained in  $G_{i-1}$ . During the construction, we compute  $att(e)$  using the modified version of Algorithm 1. We describe below in rough terms how we compute  $S_i$ :

- 1 select an edge  $e_i$  with the smallest  $low_1$  value from successors of  $e$  not contained in  $G_{i-1}$ ;

```

2   while there exists a maximal strongly planar subgraph of  $sub(e_i)$  whose addition to  $G_{i-1}$ 
      destroys its  $l$ -planarity do
3     delete some attachments from  $sub(e_i)$ ;
4     if the deletion changes the  $low_1$  value of  $e_i$  then
5       select a possibly new edge  $e_i$  with the smallest  $low_1$  value from
          successors of  $e$  not contained in  $G_{i-1}$ ;
6     end if;
7   od;
8   recursively construct a maximal strongly planar subgraph of  $sub(e_i)$  without changing
       $low_1(e_i)$  further. We take this subgraph as  $S_i$ .

```

In the procedure sketched above, lines 1, 4, 5, 6, and 8 guarantee that subgraphs  $S_i$  are generated in increasing order by new  $low_1$  values of the corresponding successors. For each  $1 < i \leq k$ , once  $S_i$  is computed, no edges will be deleted further from it. There are still two questions remaining to be answered: how the testing in line 2 can be done without constructing a maximal strongly planar subgraph of  $sub(e_i)$ , and how the attachments are chosen so that the deletion in line 3 makes the set of deleted edges minimal. These two questions are closely related and will be explained together in the next section.

**Remark** In Algorithm 1, we do not need the concept of  $l$ -planarity, since our purpose is to check the planarity of  $G$ . If some interlacing blocks of  $sub(e)$  are found not being able to fit in the whole graph after returning from several levels of recursive calls, we simply declare that the graph is not planar. But if we want to construct a maximal planar subgraph of  $G$ , then it is too late to delete edges efficiently by that time. Therefore we use the parameter  $l$  to pass the information where blocks of  $sub(e)$  are allowed to interlace, to the recursive calls, so that the correct edges are already deleted during the processing of  $sub(e)$ .

The need to generalize to  $l$ -planarity arises in the following way in the algorithm sketched above. To compute a maximal planar subgraph of the input graph, the recursive calls that construct  $S_i$  for  $i = 2, \dots, k$  must construct maximal strongly planar graphs. Within one of these recursive calls, the initial second-level recursive call (to construct a maximal  $b$ -planar subgraph of  $sub(e_{i1})$ , where  $e_{i1}$  is the first successor of edge  $e_i$ ) and more deeply nested recursive calls of the same kind construct maximal  $l$ -planar subgraphs for general values of  $l$ . ■

#### 4.2. Algorithm for deleting back edges

Let  $e = [a, b]$ , and consider the **while**-loop in the procedure sketched above. If  $low_1(e_i) \geq b$ , then  $b$  is the only vertex common to  $sub(e_i)$  and  $G - sub(e_i)$ . In this case, we can apply the maximal planar subgraph to  $sub(e_i)$  separately, and do not have to consider the effect on the

whole graph. Next, we consider the case when  $low_1(e_i) < b$ . Assume that  $sub(e_i)$  is made strongly planar by deleting some back edges. Suppose that the  $low_1$  value and the  $low_2$  value of  $e_i$  are not changed by these deletions. We want to see whether the union of  $sub(e_i)$  and  $G_{i-1}$  is  $l$ -planar.

As in planarity testing, let  $B_i$  be the block of attachments obtained by merging  $att(e_i)$ ; let  $att_i$  be the current value of  $att(e)$ ; let  $B_j$  be the last block merged into  $att_i$  by Step 4'; let  $h_1 = \max\{last(Z): [Z, U] \text{ is a pair in } att_i\}$  and  $h_2 = \max\{last(U): [Z, U] \text{ is a pair in } att_i\}$ . (Initially, we set  $j=1$  and  $att(e) = att(e_1)$  after removing all the occurrences of  $b$  from  $att(e_1)$ .) Finally, let  $h_3 = \max\{last(Z): Z \text{ is an } l\text{-normal block of } att_i\}$ .

The two variables  $h_1, h_2$  can be maintained in  $O(1)$  time per modification to  $att(e)$  by maintaining two lists  $L$  and  $R$  ( as suggested in [7] ), where  $L$  is the ordered list of nonempty blocks  $X$  such that  $[X, Y]$  is a pair in  $att(e)$ , and  $R$  is the ordered list of nonempty blocks  $Y$  such that  $[X, Y]$  is a pair in  $att(e)$ . If  $B_L$  and  $B_R$  are the highest blocks of  $L$  and  $R$  respectively, then  $h_1 = last(B_L)$  and  $h_2 = last(B_R)$ . Lists  $L$  and  $R$  also let  $h_3$  be maintained easily. If  $e$  is a back edge,  $h_3 = 0$  from its definition. If  $e$  is a tree edge, we get the initial value of  $h_3$  from the computation of  $att(e_1)$ , and modify it in  $O(1)$  time for each modification of  $att(e)$ . The details will not be discussed here.

By Lemma 5, in case that  $e_i$  is strongly planar,  $sub(e_i)$  can affect the planarity of  $G$  only if any of the following conditions holds:

- a.  $j = 1$ , i.e., no block  $B_j$  has been merged into  $att(e)$  yet,
- b.  $low_1(e_j) < low_1(e_i)$ , or
- c.  $low_2(e_i) < b$ .

If any of these conditions is true, we consider two additional cases:

1. The union of  $sub(e_i)$  and  $G_{i-1}$  is not planar. By Corollary 1.1, this happens iff  $low_1(e_i) < h_2$ .
2. The union of  $sub(e_i)$  and  $G_{i-1}$  is planar, but not  $l$ -planar. Then  $B_i$  is  $l$ -normal, and it interlaces an  $l$ -normal block of  $att_i$ . We know that  $B_i$  is  $l$ -normal iff  $low_1(e_1) < low_1(e_i) < l$  or  $low_2(e_i) < l$ . Also, it is easy to see that  $B_i$  interlaces an  $l$ -normal block of  $att_i$  iff  $low_1(e_i) < h_3$ .

This means, under the conditions a, b, or c, that the union of  $sub(e_i)$  and  $G_{i-1}$  is not  $l$ -planar iff any of the following conditions holds

- i.  $low_1(e_i) < h_2$
- ii.  $low_1(e_1) < low_1(e_i) < \min \{h_3, l\}$
- iii.  $low_2(e_i) < l$  and  $low_1(e_i) < h_3$

Therefore, if any of the condition i, ii, and iii holds, some back edge has to be deleted to change

either the  $low_1$  value or the  $low_2$  value of  $e_i$ . Such testing and deletion can be done even before making  $sub(e_i)$  strongly planar. For this purpose, we combine the above conditions (a or b or c) and (i or ii or iii) into two groups according to whether they involve  $low_2(e_i)$  or not:

Condition AA.

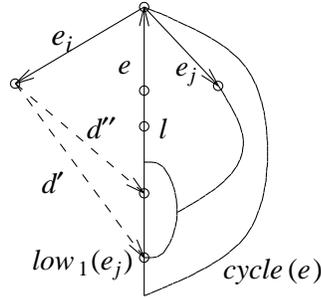
- ( $j = 1$  and  $low_1(e_i) < h_2$ ) or
- ( $low_1(e_j) < low_1(e_i) < h_2$ ) or
- ( $low_1(e_j) < low_1(e_i) < \min\{h_3, l\}$ )

Condition BB.

- ( $low_2(e_i) < b$  and  $low_1(e_i) < h_2$ ) or
- ( $low_2(e_i) < b$  and  $low_1(e_1) < low_1(e_i) < \min\{h_3, l\}$ ) or
- ( $low_2(e_i) < l$  and  $low_1(e_i) < h_3$ )

It can be checked that the condition ((a or b or c) and (i or ii or iii)) is equivalent to the condition (AA or BB).

If Condition AA is true, we can make it false only by changing the value  $low_1(e_i)$ . In this case, we delete all the back edges of  $sub(e_i)$  entering the vertex  $low_1(e_i)$ . After the deletion, we choose a possibly new  $e_i$  with the smallest  $low_1$  value.



The edge  $e_i$  satisfies condition BB. If we choose to delete  $d'$ , then  $d''$  will also be deleted later because of Condition AA, and the resulting graph will not be maximal.

Fig. 14

If Condition AA is false, then we test Condition BB. If the result is true, we know that  $low_1(e_i) = low_1(e_j)$ . This is because  $low_1(e_i) < low_2(e_i)$ , which means that BB implies that  $low_1(e_i) < h_2$  or  $low_1(e_i) < \min\{h_3, l\}$ , from which it follows that  $low_1(e_j) = low_1(e_i)$ ; otherwise AA would be true. (We have  $low_1(e_j) \leq low_1(e_i)$  by the ordering of the successors of  $e$ .) To make Condition BB false, we can change the value of either  $low_1(e_i)$  or  $low_2(e_i)$ . If we choose to change  $low_2(e_i)$  consistently, then at least one of the back edges  $[u, v]$  of  $sub(e_i)$  with  $v = low_1(e_i)$  will survive. But if we choose to change  $low_1(e_i)$ , it may happen that all the attachments in  $ATT(e_i) \cap ATT(e)$  are deleted eventually and that the resulting graph is not maximal.

Therefore, in this case we choose to delete all the back edges  $[u, v]$  of  $sub(e_i)$  with  $v = low_2(e_i)$  ( see Fig. 14 ).

We test and delete repeatedly as described above until we find an edge  $e_i$  that does not satisfy AA or BB. Then we can construct  $S_i$  recursively from  $sub(e_i)$  and merge it into  $G_{i-1}$ . Since no edge is added to  $sub(e_i)$  during the construction of  $S_i$ , conditions AA and BB remain false after the construction. Thus, the resulting graph  $G_i$  will be planar, and no  $l$ -normal blocks will interlace.

To see that the deleted set of back edges is minimal, let  $[u, v]$  be an edge deleted by the above algorithm, and add it back to  $G_i$ . If  $[u, v]$  was deleted because of Condition AA, then  $low_1(e_i) = v$  now, and Condition AA is true again. If  $[u, v]$  was deleted because of Condition BB, then  $low_2(e_i) = v$  now, and Condition BB is true again. Notice that, in the latter case, the  $low_1$  value of  $e_i$  has remained unchanged since the deletion of  $[u, v]$ . In either case,  $G_i$  will not be  $l$ -planar.

### 4.3. Data structures and running time

In the algorithm described above, we need to repeatedly select an unprocessed successor of  $e$  with the smallest  $low_1$  value, and the  $low_1$  values of tree edges are constantly changing. Therefore we maintain a heap [13] based on  $low_1$  values of the unprocessed successors of the tree edge  $e$  currently being processed. Since the algorithm is recursive, we actually maintain simultaneously a heap of unprocessed successors for each tree edge along the path to the currently active tree edge. The total size of all such heaps is  $O(m)$ . The initialization of all these heaps takes a total of  $O(m)$  time. When the  $low_1$  value of some element in a heap increases, we modify the heap accordingly. It is important to note that any two edges in active heaps are unrelated; thus deletion of a single attachment can modify the  $low_1$  value of only a single such edge. It follows that the total number of modifications to and deletions from heaps is  $O(m)$ . The time for the heap operations is  $O(\log n)$  time per operation, for a total of  $O(m \log n)$  time. (Since  $m < n^2$ ,  $\log m = O(\log n)$ .)

We also need a data structure for the back edges of  $sub(e)$  so that the following operations can be done efficiently:

1. delete an attachment  $[u, v]$  of  $e$  with  $v = low_1(e)$  or  $v = low_2(e)$ ;
2. maintain the  $low_1$  and  $low_2$  values of  $e$ ;
3. split the data structure into several pieces, one for each successor of  $e$ .

One easy solution that meets these requirements is the *selection tree* [8]. To represent a set of edges  $E_0$  as a selection tree  $T_0$ , we store edges of  $E_0$  inside the leaves of  $T_0$  from left to right in increasing order (by DFS number) of their tails. Edges with the same tail are ordered

arbitrarily. Each internal node  $w$  of  $T_0$  has two children  $w.lchild$  and  $w.rchild$ . Let  $S_w$  be the set of edges stored in the leaves of the subtree rooted at  $w$ ; let  $lb = \min \{x : [x, y] \in S_w\}$  and  $rb = \max \{x : [x, y] \in S_w\}$ ; let  $low_1 = \min \{y : [x, y] \in S_w\}$ , and  $low_2 = \min (\{y : [x, y] \in S_w \mid y \neq low_1\} \cup \{n+1\})$ . Then the four values  $lb$ ,  $rb$ ,  $low_1$  and  $low_2$  are stored in the four fields  $w.lb$ ,  $w.rb$ ,  $w.low_1$ , and  $w.low_2$  of  $w$  respectively. If  $w$  is a leaf storing the edge  $[x, y]$ , then  $w.lb = x$ ,  $w.rb = x$ ,  $w.low_1 = y$ , and  $w.low_2 = n+1$ . The values in each internal node can be computed from the values in the children ( in constant time ).

In the following discussion, we will refer to a tree by its root. Let  $r_1$  and  $r_2$  be two selection trees representing the two disjoint sets of edges  $E_1$  and  $E_2$ . If  $u_1 \leq u_2$  for all  $[u_1, v_1] \in E_1$  and  $[u_2, v_2] \in E_2$ , then we can merge  $r_1$  and  $r_2$  to get the selection tree for  $E_1 \cup E_2$  in  $O(1)$  time:

```

procedure merge( $r_1, r_2$ );
begin if  $r_1 = null$  then
    return  $r_2$ ;
end if;
if  $r_2 = null$  then
    return  $r_1$ ;
end if;
 $r := newnode ()$ ;
 $r.lchild := r_1$ ;
 $r.rchild := r_2$ ;
 $r.lb := r_1.lb$ ;
 $r.rb := r_2.rb$ ;
 $r.low_1 := \min\{r_1.low_1, r_2.low_1\}$ ;
 $r.low_2 := \min(\{r_1.low_1, r_2.low_1, r_1.low_2, r_2.low_2\} - \{r.low_1\})$ ;
return  $r$ ;
end;

```

Let  $r$  be a selection tree representing a set of edges  $E_0$ . To split  $E_0$  into two sets  $E_1 = \{[u, v] \in E_0 \mid u \leq u_x\}$  and  $E_2 = \{[u, v] \in E_0 \mid u > u_x\}$ , we split  $r$  with respect to  $u_x$  as follows:

```

procedure split( $r, u_x$ );
begin if  $u_x < r.lb$  then
    return [ $null, r$ ];
elseif  $u_x \geq r.rb$  then
    return [ $r, null$ ];
else [ $r_l, r_r$ ] := [ $r.lchild, r.rchild$ ];
    if  $u_x < r_l.rb$  then
        [ $r_{l1}, r_{l2}$ ] := split( $r_l, u_x$ );
        return [ $r_{l1}, merge(r_{l2}, r_r)$ ];
    else [ $r_{r1}, r_{r2}$ ] := split( $r_r, u_x$ );
        return [ $merge(r_l, r_{r1}), r_{r2}$ ];
    end if;
end if;
end;

```

The height of any tree that results from splitting a tree  $r$  can be no greater than the height of  $r$ . To select and delete an edge  $[x, v]$  from a tree  $r$ , where  $v \in \{r.low_1, r.low_2\}$ , we do the following:

```

procedure delete( $r, v$ );
begin if  $r$  is a leaf then
    mark the back edge stored in  $r$  as 'deleted';
    return null;
else  $[r_l, r_r] := [r.lchild, r.rchild]$ ;
    if  $v = r_l.low_1$  or  $v = r_l.low_2$  then
        return merge(delete( $r_l, v$ ),  $r_r$ );
    else return merge( $r_l$ , delete( $r_r, v$ ));
    end if;
end if;
end;

```

Assuming the input graph  $G$  is connected, we know that all the tree edges form a tree. Let the root be 1. For technical reasons, we add a dummy edge  $e_0 = [0, 1]$  to the tree edges. To get a maximal planar subgraph of  $G$ , we just construct a 0-planar subgraph of  $sub(e_0)$ , and then delete  $e_0$  from it. Initially, we construct a balanced selection tree  $tree(e_0)$  to store all the back edges of  $G$ . The height of this tree is  $O(\log n)$ . The time and space needed to initialize  $tree(e_0)$  are both  $O(m)$ .

When we begin to construct a maximal  $l$ -planar subgraph for a tree edge  $e$ , we first split  $tree(e)$  into several pieces  $tree(e_1), \dots, tree(e_k)$ , where  $e_1, \dots, e_k$  are the successors of  $e$  not marked as 'deleted'. For each such successor  $e_i$ ,  $tree(e_i)$  is a selection tree representing the set of back edges in  $sub(e_i)$ , and can be obtained as follows. If  $e_i$  is a back edge, then  $tree(e_i)$  consists of a single edge, and can be constructed in  $O(1)$  time. If  $e_i$  is a tree edge, let  $e_i = [b, c_i]$ , and let  $n_i$  be the number of descendants of  $c_i$ . It is well known that a back edge  $[u, v]$  is a descendant of  $e_i$  iff  $c_i \leq u < c_i + n_i$ . Then we can use the procedure *split* to get  $tree(e_i)$  from  $tree(e)$  in  $O(\log n)$  time. Since there are at most  $n$  tree edges in  $G$ , then the splitting takes  $O(n \log n)$  time for the whole algorithm. After each split, the total size of the trees is still  $O(m)$ .

To select and delete an attachment  $[x, v]$  of  $e_i$ , where  $v \in \{low_1(e_i), low_2(e_i)\}$ , we execute *delete*( $tree(e_i), v$ ), which takes  $O(\log n)$  time. There can be at most  $O(m)$  such invocations of *delete*, so the total cost for executing *delete* is  $O(m \log n)$ . Given the selection tree  $tree(e_i)$ , the values  $\phi(e_i)$ ,  $low_1(e_i)$ , and  $low_2(e_i)$  can be computed from  $tree(e_i)$  in  $O(1)$  time: if  $tree(e_i)$  is null, we just set these values to  $n+1$ ; otherwise, they can be computed from  $tree(e_i).low_1$  and  $tree(e_i).low_2$ . Thus, the total cost of selection tree operations is  $O(m \log n)$ .

We have mentioned that the total cost of heap operations is also  $O(m \log n)$ . The other costs of the algorithm are the same as in planarity-testing. Thus the total cost of our maximal planar subgraph algorithm is  $O(m \log n)$ .

#### 4.4. The complete algorithm

Now we summarize our maximal planar subgraph algorithm. We take a connected undirected graph as input, and convert it into a DFS representation  $G = (V, T, B)$ . At the same time, we compute the two mappings  $succ$  and  $N$ , where, for each  $e \in T$ ,  $succ(e)$  gives the successor edges of  $e$  in increasing order of their tails, and for each  $v \in V$ ,  $N(v)$  gives the number of descendants of  $v$ . We assume that there is a dummy edge  $e_0 = [0, 1]$  such that  $succ(e_0)$  gives the list of tree edges leaving the root. The whole preprocessing takes  $O(m)$  time.

We summarize the maximal  $l$ -planar subgraph algorithm below.

```

procedure  $lplanar(e, l)$ ;
begin let  $e = [a, b]$ ;
    if  $e \in B$  then
        return  $[[[b], []]]$ ;
    end if;
    if  $e$  has no successors then
        return  $[]$ ;
    end if;
    let  $e_1, \dots, e_k$  be the successors of  $e$  not marked as 'deleted';
    split  $tree(e)$  into  $tree(e_1), \dots, tree(e_k)$ ;
    organize  $e_1, \dots, e_k$  into a heap based on their  $low_1$  values, with the smallest one on the top;
    let  $e_1$  be the edge on the top of the heap;
    delete  $e_1$  from the heap;
1    $att(e) := lplanar(e_1, l)$ ;
    delete all the occurrences of  $b$  from the top blocks of  $att(e)$ ;
     $j := 1$ ;
     $i := 2$ ;
    while heap is not empty do
        let  $e_i$  be the edge on the top of the heap;
        if  $low_1(e_i) \geq b$  then
            delete  $e_i$  from the heap;
2            $dummy := lplanar(e_i, b)$ ;
        elseif Condition AA is true then
             $v := tree(e_i).low_1$ ;
            while  $v = tree(e_i).low_1$  do
3                $tree(e_i) := delete(tree(e_i), v)$ ;
            end while;
            if  $e_i$  is a back edge then
                delete  $e_i$  from heap;
            else modify heap;
            end if;
        elseif Condition BB is true then
             $v := tree(e_i).low_2$ ;
            while  $v = tree(e_i).low_2$  do
4                $tree(e_i) := delete(tree(e_i), v)$ ;
            end while;
        else delete  $e_i$  from the heap;
5            $att(e_i) := lplanar(e_i, b)$ ;

```

```

6           merge blocks of  $att(e_i)$  into one block  $B_i$ ;
           delete all the occurrences of  $b$  from the top blocks of  $att(e_i)$ ;
           if  $i = 2$  then
7               perform Step 3 of Alg.1;
           end if;
8           merge  $B_i$  into  $att(e)$  as described in Step 4';
            $i := i + 1$ ;
           end if
       end while;
       return  $att(e)$ ;
end;

```

The procedure  $lplanar(e, l)$  implicitly constructs a maximal  $l$ -planar subgraph of  $sub(e)$  by deleting a minimal set of back edges. The parameter  $l$  specifies where the blocks of  $att(e)$  are not allowed to interlace in the resulting subgraph, so that this subgraph can be used to build a larger planar subgraph without further deletion when we process the predecessor of  $e$ . For the initial call where  $e = e_0$ , we have  $l = 0$ , meaning that we need to construct a maximal planar subgraph of  $sub(e_0)$ . In the recursive calls for the successors of  $e$ , the  $l$  values are determined as follows. Since no  $l$ -normal blocks are allowed to interlace in  $sub(e)$ , then no  $l$ -normal blocks are allowed to interlace in  $sub(e_1)$  either. Thus the recursive call of  $lplanar$  for  $e_1$  (line 1) has the same parameter  $l$  as for the edge  $e$ . The remaining calls for  $e_2, \dots, e_k$  (line 5) just construct maximal strongly planar subgraphs, therefore have  $b$  as their  $l$  values. Thus when we merge blocks at line 6, no normal blocks of  $att(e_i)$  interlace. At line 7, we merge all normal blocks of  $att(e)$  above  $low_1(e_2)$  in to one block; at line 8, we merge  $B_i$  into  $sub(e)$ . Because of the deletions at lines 3 and 4, these steps can be performed successfully (without any further deletion). At line 2,  $sub(e_i)$  is detected to be a biconnected component and is processed separately.

To compute a maximal planar subgraph, we simply do the following:

1. Organize  $B$  into a selection tree  $tree(e_0)$ ;
2. Execute  $lplanar(e_0, 0)$ ;

Then  $T \cup B - B'$  gives a maximal planar subgraph of  $G$ , where  $B'$  is the set of back edges deleted by the procedure *delete* in the preceding algorithm.

**Remark** The procedure  $lplanar$  can be greatly simplified if we know that all the tree edges of  $G$  are on a same cycle. In this case, the  $low_1$  values need not be dynamically maintained: if  $e = [a, b]$  is a back edge, then  $low_1(e) = b$ ; otherwise  $low_1(e) = 1$ . The  $low_2$  values, which are used only for testing condition BB when  $i > 1$ , need not be maintained either, since for  $i > 1$ ,  $e_i$  is a back edge. As a result, the selection trees are no longer useful, and the heaps storing the successors of tree edges can be replaced by lists precomputed as in Algorithm 1. With these simplifications, our algorithm gives the following result which is first reported in [10] by *Kobayashi et al.*: A maximal planar subgraph can be constructed in linear time provided that a

Hamiltonian tour of the graph is given. ■

## 5. Summary

The problem of drawing graphs in the plane arises naturally in circuit layout. Since finding a maximum planar subgraph is *NP*-complete [5], a maximal planar subgraph seems to be a reasonable approximation. Because planarity-testing can be done in linear time, it is easy to solve the maximal planar subgraph problem in  $O(mn)$  time: start with a graph  $H$  with no edge; for each edge of the input graph  $G$ , add it to  $H$  if the resulting graph is planar, and reject it otherwise. The resulting graph  $H$  will be a maximal planar subgraph of  $G$ . However, a better solution seemed to be hard to find for a long time. Jayakumar *et al.* [9] even made the conjecture that "no maximal planarization algorithm of complexity better than  $O(mn)$  will be possible." Our  $O(m \log n)$  solution disproves this conjecture, as does the method of Di Battista and Tamassia [3].

We have assumed that the input graph to our algorithm is connected. For a more general graph, we can find a maximal planar subgraph by applying our algorithm to each of its connected components.

**Acknowledgment** We thank the referees for their careful and invaluable comments on the earlier versions of this paper.

## References

1. Booth, K. S. and Lueker, G. S., "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms," *Journal of Computer and System Science*, vol. 13, no. 3, pp. 335-379, 1976.
2. Chiba, N., Nishizeki, T., Abe, S., and Ozawa, T., "A Linear Algorithm for Embedding Planar Graphs Using PQ-trees," *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 54-76, 1985.
3. Di Battista, G. and Tamassia, R., "Incremental Planarity Testing (Extended Abstract)," in *Proc. 30th Annual I.E.E.E. Symposium on Foundations of Computer Science*, pp. 436-441, 1989.
4. Even, S. and Tarjan, R. E., "Computing an st-numbering," *Th. Comp. Sci.*, vol. 2, no. 3, pp. 339-344, 1976.
5. Garey, M. and Johnson, D., *Computers and Intractability*, Freeman, 1979.
6. Hall, D. and Spencer, G., *Elementary Topology*, Wiley, New York, 1955.
7. Hopcroft, J. and Tarjan, R., "Efficient Planarity Testing," *JACM*, vol. 21, no. 4, pp. 549-568, October, 1974.
8. Horowitz, E. and Sahni, S., in *Fundamentals of Data Structures*, Computer Science Press, 1983.
9. Jayakumar, R., Thulasiraman, K., and Swamy, M. N. S., " $O(n^2)$  Algorithms for Graph Planarization," *IEEE Transactions on CAD*, vol. 8, no. 3, pp. 257-267, March, 1989.
10. Kobayashi, N., Masuda, S., and Kashiwabara, T., "Algorithms to Obtain a Maximal Planar Hamilton Subgraph," *IEICE Transactions E74*, no. 4, pp. 657-664, April, 1991.
11. Lempel, A., Even, S., and Cederbaun, I., "An Algorithm for Planarity Testing of Graphs," in *Theory of Graphs, International Symposium*, pp. 215-232, Rome, July, 1966.
12. Tarjan, R., "Depth-First Search and Linear Graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146-160, April, 1972.

13. Tarjan, R., *Data Structures and Network Algorithms*, SIAM, 1984.
14. Thron, W.T., *Introduction to the Theory of Functions of a Complex Variable.*, Wiley, New York, 1953.
15. Wu, W., "On the Planar Imbedding of Linear Graphs," *J. Sys. Sci. & Math. Scis.*, vol. 5, no. 4, pp. 290-302, Institute of Systems Science, Academia Sinica, Beijing, 1985.