Lempel, Even, and Cederbaum Planarity Method

John M. Boyer[1], Cristina G. Fernandes[2⋆], Alexandre Noma[2⋆⋆], and José Coelho de Pina[2⋆].

[1] PureEdge Solutions Inc. jboyer@acm.org
[2] University of São Paulo, Brazil {cris,noma,coelho}@ime.usp.br

Abstract. We present a simple pedagogical graph theoretical description of Lempel, Even, and Cederbaum (LEC) planarity method based on concepts due to Thomas. A linear-time implementation of LEC method using the PC-tree data structure of Shih and Hsu is provided and described in details. We report on an experimental study involving this implementation and other available linear-time implementations of planarity algorithms.

1 Introduction

The first linear-time planarity testing algorithm is due to Hopcroft and Tarjan [9]. Their algorithm is an ingenious implementation of the method of Auslander and Parter [1] and Goldstein [8]. Some notes to the algorithm were made by Deo [6], and significant additional details were presented by Williamson [20, 21] and Reingold, Nievergelt, and Deo [16].

The second method of planarity testing proven to achieve linear time is due to Lempel, Even, and Cederbaum (LEC) [13]. This method was optimized to linear time thanks to the *st*-numbering algorithm of Even and Tarjan [7] and the PQ-tree data structure of Booth and Lueker (BL) [2]. Chiba, Nishizeki, Abe, and Ozawa [5] augmented the PQ-tree operations so that a planar embedding is also computed in linear time.

All these algorithms are widely regarded as being quite complex [5, 12, 17]. Recent research efforts have resulted in simpler linear-time algorithms proposed by Shih and Hsu (SH) [10, 17, 18] and by Boyer and Myrvold (BM) [3, 4]. These algorithms implement LEC method and present similar and very interesting ideas. Each algorithm uses its own data structure to efficiently maintain relevant information on the (planar) already examined portion of the graph.

The description of SH algorithm made by Thomas [19] provided us with the key concepts to give a simple graph theoretical description of LEC method. This description increases the understanding of BL, SH, and BM algorithms, all based on LEC method.

⋆ Research partially supported by ProNEx/CNPq 664107/1997-4 (Brazil).
⋆⋆ Supported by fapesp 00/03969-2 (Brazil).

Section 2 contains definitions of the key ingredients used by LEC method. In Section 3, an auxiliary algorithm is considered. LEC method is presented in Section 4 and an implementation of SH algorithm is described in Section 5. This implementation is available at `http://www.ime.usp.br/~coelho/sh/` and, as far as we know, is the unique available implementation of SH algorithm, even though the algorithm was proposed about 10 years ago. Finally, Section 6 reports on an experimental study.

2 Frames, XY-paths, XY-obstructions and planarity

This section contains the definitions of some concepts introduced by Thomas [19] in his presentation of SH algorithm. We use these concepts in the coming sections to present both LEC method and our implementation of SH algorithm.

Let H be a planar graph. A subgraph F of H is a *frame of* H if F is induced by the edges incident to the external face of a planar embedding of H (Figs. 1(a) and 1(b)).

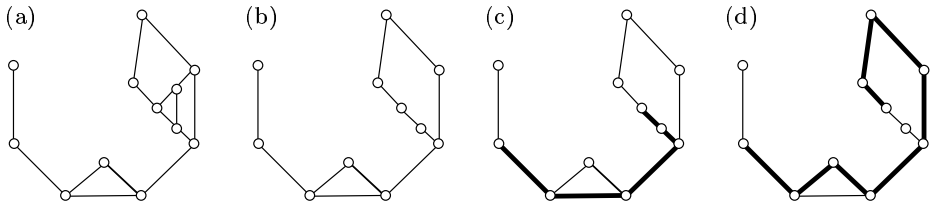

Fig. 1. (a) A graph H. (b) A frame of H. (c) A path P in a frame. (d) The complement of P.

If G is a connected graph, H is a planar induced subgraph of G and F is a frame of H, then we say that F is a *frame of* H *in* G if it contains all vertices of H that have a neighbor in $V_G \setminus V_H$. Neither every planar induced subgraph of a graph G has a frame in G (Fig. 2(a)) nor every induced subgraph of a planar graph G has a frame in G (Fig. 2(b)). The connection between frames and planarity is given by the following lemma.

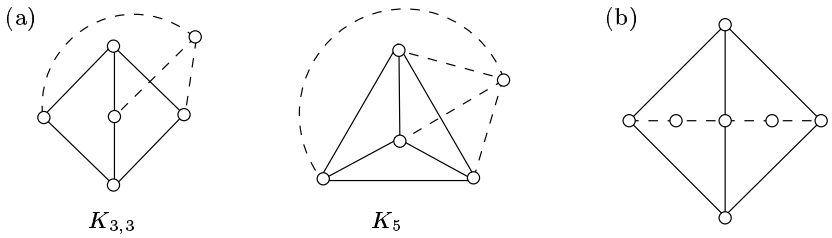

Fig. 2. (a) Subgraphs of $K_{3,3}$ and K_5 induced by the solid edges have no frames. (b) Subgraph induced by the solid edges has no frame in the graph.

Lemma 1 (Thomas [19]). *If H is an induced subgraph of a planar graph G such that $G - V_H$ is connected, then H has a frame in G.* ■

Let F be a frame of H and P be a path in F. The *basis of* P is the subgraph of F formed by all blocks of F which contain at least one edge of P. Let $C_1, C_2, \ldots, C_k$ be the blocks in the basis of P. For $i = 1, 2, \ldots, k$, let $P_i := P \cap C_i$ and, if C_i is a cycle, let $\bar{P}_i := C_i \setminus P_i$, otherwise let $\bar{P}_i := P_i$. The *complement of* P *in* F is the path $\bar{P}_1 \cup \bar{P}_2 \cup \ldots \cup \bar{P}_k$, which is denoted by $\bar{P}$. If $E_P = \emptyset$ then $\bar{P} := P$ (Figs. 1(c) and 1(d)).

Let W be a set of vertices in H and Z be a set of edges in H. A vertex v in H *sees* W *through* Z if there is a path in H from v to a vertex in W with all edges in Z. Let X and Y be sets of vertices of a frame F of H. A path P in F with basis S is an XY*-path* (Fig. 3) if

(p1) the endpoints of P are in X;
(p2) each vertex of S that sees X through $E_F \setminus E_S$ is in P;
(p3) each vertex of S that sees Y through $E_F \setminus E_S$ is in $\bar{P}$;
(p4) no component of $F - V_S$ contains vertices both in X and in Y.

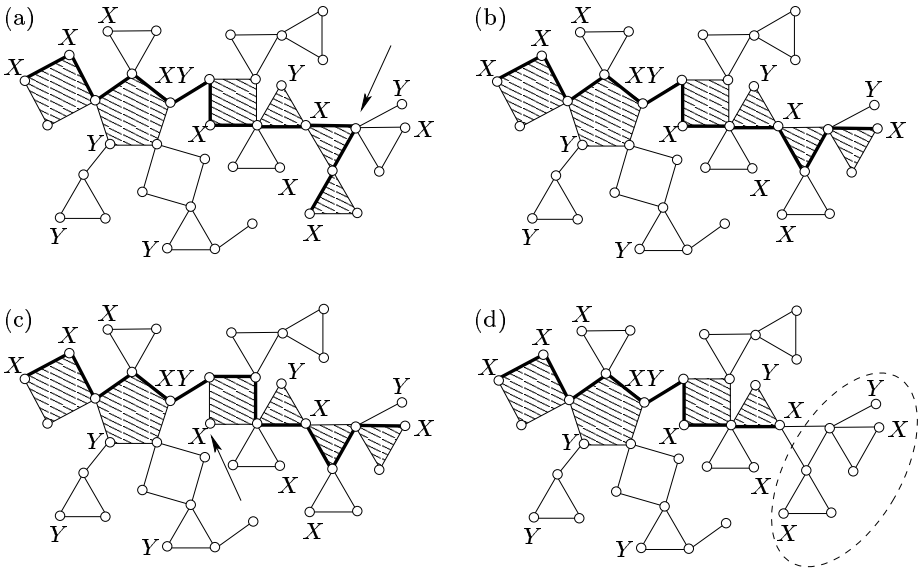

Fig. 3. In (a), (b), (c), and (d), let P denote the thick path; its basis is shadowed. (a) P is not an XY-path since it violates (p3). (b) P is an XY-path. (c) P is not an XY-path since it violates (p2). (d) P is not an XY-path since it violates (p4).

There are three types of objects that obstruct an XY-path to exist. They are called XY*-obstructions* and are defined as

(o1) a 5-tuple (C, v_1, v_2, v_3, v_4) where C is a cycle of F and v_1, v_2, v_3, and v_4 are distinct vertices in C that appear in this order in C, such that v_1 and v_3 see X through $E_F \setminus E_C$ and v_2 and v_4 see Y through $E_F \setminus E_C$;
(o2) a 4-tuple (C, v_1, v_2, v_3) where C is a cycle of F and v_1, v_2, and v_3 are distinct vertices in C that see X and Y through $E_F \setminus E_C$;
(o3) a 4-tuple (v, K_1, K_2, K_3) where $v \in V_F$ and K_1, K_2, and K_3 are distinct components of $F - v$ such that K_i contains vertices in X and in Y.

The existence of an XY-obstruction is related to non-planarity as follows.

Lemma 2 (Thomas [19]). *Let H be a planar connected subgraph of a graph G and w be a vertex in $V_G \setminus V_H$ such that $G - V_H$ and $G - (V_H \cup \{w\})$ is connected. Let F be a frame of H in G, let X be the set of neighbors of w in V_F and let Y be the set of neighbors of $V_G \setminus (V_H \cup \{w\})$ in V_F. If F has an XY-obstruction then G has a subdivision of $K_{3,3}$ or K_5.*

Sketch of the proof: An XY-obstruction of type (o1) or (o3) indicates a $K_{3,3}$-subdivision. An XY-obstruction of type (o2) indicates either a K_5-subdivision or a $K_{3,3}$-subdivision (Fig. 4). ■

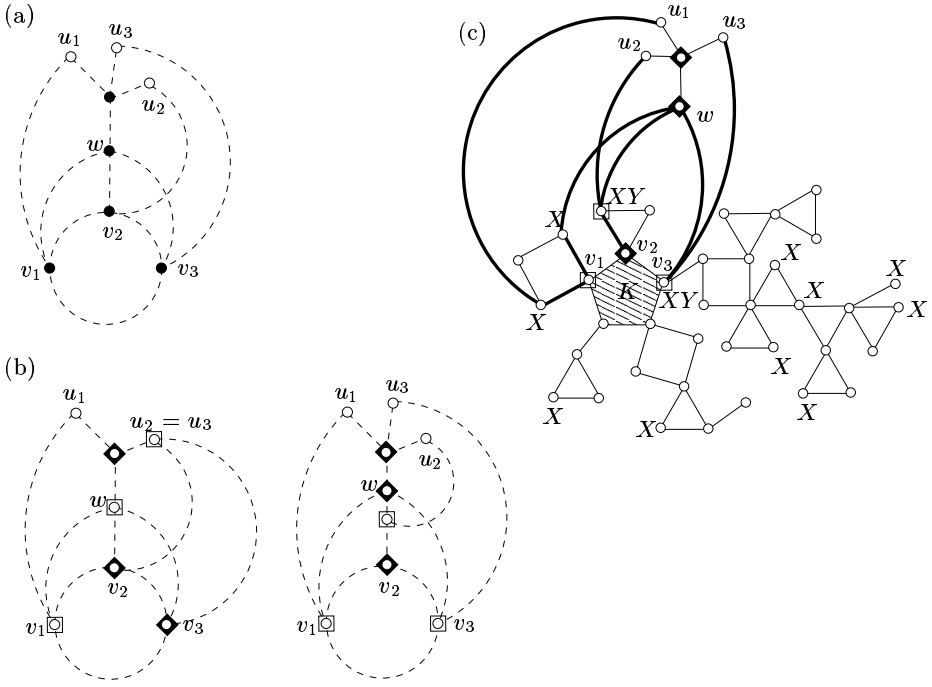

Fig. 4. Some situations with an XY-obstruction (C, v_1, v_2, v_3) of type (o2). The dashed lines indicate paths. In $u_i v_i$-path, u_i is the only vertex in $(V_H \setminus (V_H \cup \{w\}))$, for $i = 1, 2, 3$. (a) Subdivision of K_5 coming from an XY-obstruction. (b) Subdivisions of $K_{3,3}$ coming from an XY-obstruction. (c) Concrete example of an XY-obstruction leading to a $K_{3,3}$-subdivision.

3 Finding XY-paths or XY-obstructions

Let F be a connected frame and let X and Y be subsets of V_F. If F is a tree, then finding an XY-path or an XY-obstruction is an easy task. The following algorithm finds either an XY-path or an XY-obstruction in F manipulating a tree that represents F.

Let $\mathcal{B}$ be the set of blocks of a connected graph H and let T be the tree with vertex set $\mathcal{B} \cup V_H$ and edges of the form Bv where $B \in \mathcal{B}$ and $v \in V_B$. We call T the *block tree*[3] of H (Fig. 5). Each node of T in $\mathcal{B}$ is said a *C-node* and each node of T in V_H is said a *P-node*.

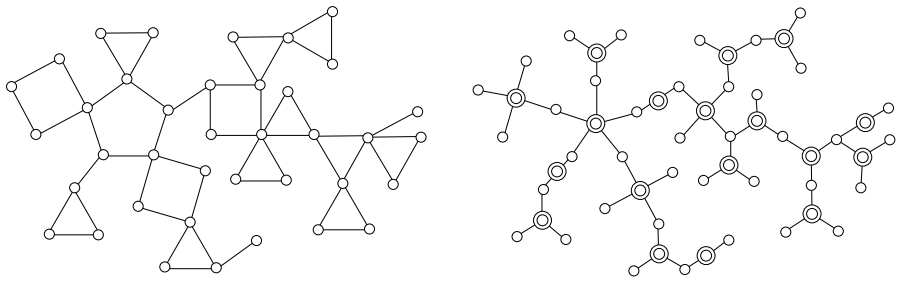

Fig. 5. A graph and its block tree.

Algorithm Central(F, X, Y). Receives a connected frame F and subsets X and Y of V_F and returns either an XY-path or an XY-obstruction in F.

Let T_0 be the block tree of F. The algorithm is iterative and each iteration begins with a subtree T of T_0, subsets X_T and Y_T of V_T and subsets W and Z of V_F. The sets X_T and Y_T are formed by the nodes of T that see X and Y through $E_{T_0} \setminus E_T$, respectively. The sets W and Z contain the P-nodes of T_0 that see X and Y through $E_{T_0} \setminus E_T$, respectively. At the beginning of the first iteration, $T = T_0$, $X_T = X$, $Y_T = Y$, $W = X$, and $Z = Y$. Each iteration consists of the following:

Case 1: Each leaf of T is in $X_T \cap Y_T$ and T is a path.
Let R be the set of P-nodes of T.
For each C-node C of T, let $X_C := V_C \cap (W \cup R)$ and $Y_C := V_C \cap (Z \cup R)$.

Case 1A: Each C-node C of T has a path P_C containing X_C and internally disjoint from Y_C.
Let P_T be the path in F obtained by the concatenation of the paths in $\{P_C : C \text{ is a C-node of } T\}$.
Let P be a path in F with endpoints in X, containing P_T and containing $V_C \cap W$ for each block C in the basis of P.
Return P and stop.

[3] The leaves in V_H make the definition slightly different than the usual.

Case 1B: There exists a C-node C of T such that no path containing X_C is internally disjoint from Y_C.
Let v_1, v_2, v_3, and v_4 be distinct vertices of C appearing in this order in C, such that v_1 and v_3 are in X_C and v_2 and v_4 are in Y_C.
Return (C, v_1, v_2, v_3, v_4) and stop.

Case 2: Each leaf of T is in $X_T \cap Y_T$ and there exists a node v of T with degree greater than 2.

Case 2A: v is a C-node.
Let C be the block of F corresponding to v.
Let v_1, v_2, and v_3 be distinct P-nodes adjacent to v in T.
Return (C, v_1, v_2, v_3) and stop.

Case 2B: v is a P-node.
Let C_1, C_2, and C_3 be distinct C-nodes adjacent to v in T.
Let K_1, K_2, and K_3 be components of $F - v$ such that C_i is a block of $K_i + v$ $(i = 1, 2, 3)$.
Return (v, K_1, K_2, K_3) and stop.

Case 3: There exists a leaf f of T not in $X_T \cap Y_T$.
Let u be the node of T adjacent to f.
Let $T' := T - f$.
Let $X_{T'} := (X_T \setminus \{f\}) \cup \{u\}$ if f is in X_T; otherwise $X_{T'} := X_T$.
Let $Y_{T'} := (Y_T \setminus \{f\}) \cup \{u\}$ if f is in Y_T; otherwise $Y_{T'} := Y_T$.
Let $W' := W \cup \{u\}$ if f is in X_T and u is a P-node; otherwise $W' := W$.
Let $Z' := Z \cup \{u\}$ if f is in Y_T and u is a P-node; otherwise $Z' := Z$.
Start a new iteration with $T', X_{T'}, Y_{T'}, W'$, and Z' in the roles of T, X_T, Y_T, W, and Z, respectively.

The execution of the algorithm consists of a sequence of "reductions" made by Case 3 followed by an occurrence of either Case 1 or Case 2. At the beginning of the last iteration, the leaves of T are called *terminals*. The concept of a terminal node is used in a fundamental way by SH algorithm. The following theorem follows from the correctness of the algorithm.

Theorem 3 (Thomas [19]). *If F is a frame of a connected graph and X and Y are subsets of V_F, then either there exists an XY-path or an XY-obstruction in F.* ■

4 LEC planarity testing method

One of the ingredients of LEC method is a certain ordering $v_1, v_2, \ldots, v_n$ of the vertices of the given graph G such that, for $i = 1, \ldots, n$, the induced subgraphs $G[\{v_1, \ldots, v_i\}]$ and $G[\{v_{i+1}, \ldots, v_n\}]$ are connected. Equivalently, G is connected and, for $i = 2, \ldots, n-1$, vertex v_i is adjacent to v_j and v_k for some j and k such that $1 \leq j < i < k \leq n$. A numbering of the vertices according to such an ordering is called a *LEC-numbering* of G. If the ordering is such that $v_1 v_n$ is an edge of the graph, the numbering is called an *st-numbering* [13]. One can show that every biconnected graph has a LEC-numbering.

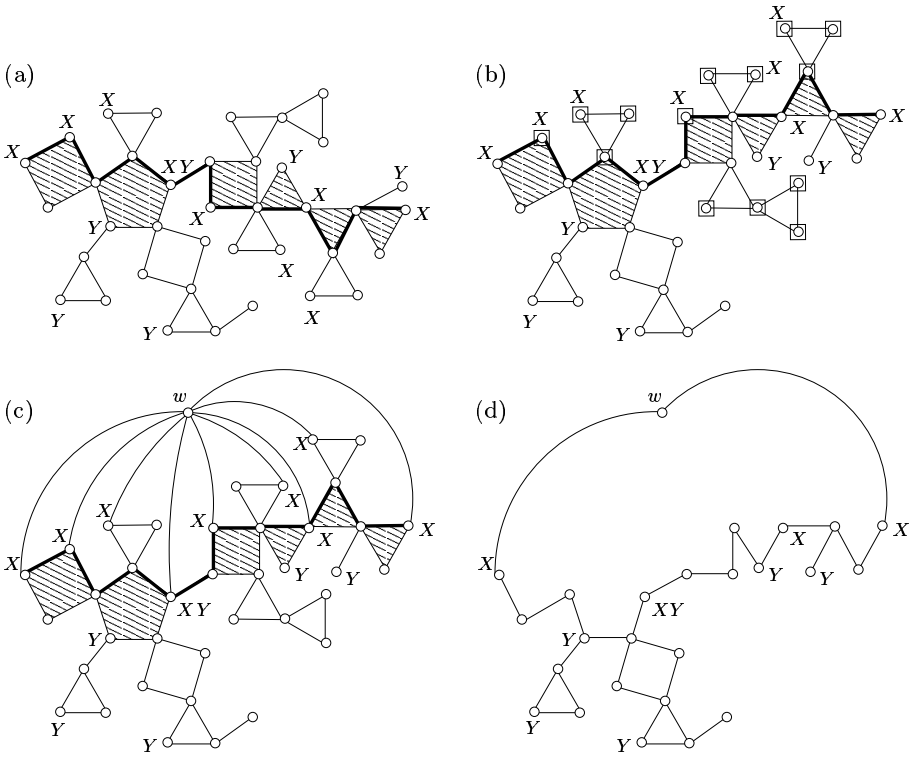

Fig. 6. (a) A frame F and an XY-path P in thick edges. (b) F after moving the elements of X to one side and the elements of Y to the other side of P. Squares mark vertices in $V_F \setminus V_{\bar{P}}$ that do not see $Y \setminus V_{\bar{P}}$ through $E_F \setminus E_S$, where $\bar{P}$ is the complement and S is the basis of P. (c) F together with the edges with one endpoint in F and the other in w. (d) A frame of $H + w$.

LEC method examines the vertices of a given biconnected graph, one by one, according to a LEC-numbering. In each iteration, the method tries to extend a frame of the subgraph induced by the already examined vertices. If this is not possible, the method declares the graph is non-planar and stops.

Method LEC(G). Receives a biconnected graph G and returns YES if G is planar, and NO otherwise.

Number the vertices of G according to a LEC-numbering. Each iteration starts with an induced subgraph H of G and a frame F of H in G. At the beginning of the first iteration, H and F are empty. Each iteration consists of the following:

Case 1: $H = G$.
Return YES e stop.

Case 2: $H \neq G$.
Let w be the smallest numbered vertex in $G - V_H$.
Let $X := \{u \in V_F : uw \in E_G\}$.
Let $Y := \{u \in V_F :$ there exists $v \in V_G \setminus (V_H \cup \{w\})$ such that $uv \in E_G\}$.

Case 2A: There exists an XY-obstruction in F.
Return NO and stop.
Case 2B: There exists an XY-path P in F.
Let $\bar{P} := \langle w_0, w_1, \ldots, w_k \rangle$ be the complement of P and let S be the basis of P.
Let R be the set of vertices in $V_F \setminus V_{\bar{P}}$ that do not see $Y \setminus V_{\bar{P}}$ through $E_F \setminus E_S$ (Figs. 6(a) and 6(b)).
Let F' be the graph resulting from the addition of w and the edges ww_0 and ww_k to the graph $F - R$ (Fig. 6(c)).
Let $H' := H + w$ (Fig. 6(d)).
Start a new iteration with H' and F' in the roles of H and F respectively.

The following invariants hold during the execution of the method.

(lec1) H and $G - V_H$ are connected graphs;
(lec2) F is a frame of H in G.

These invariants together with Lemmas 1 and 2 and Theorem 3 imply the correctness of the method and the following classical theorem.

Theorem 4 (Kuratowski). *A graph is planar if and only if it has no subdivision of $K_{3,3}$ or K_5.* ■

Three of the algorithms mentioned in the introduction are very clever linear-time implementations of LEC method. BL use an *st*-numbering instead of an arbitrary LEC-numbering of the vertices and use a PQ-tree to store F. SH use a DFS-numbering and a PC-tree to store F. BM also use a DFS-numbering and use still another data structure to store F. One can use the previous description easily to design a quadratic implementation of LEC method.

5 Implementation of SH algorithm

SH algorithm, as all other linear-time planarity algorithms, is quite complex to implement. The goal of this section is to share our experience in implementing it.

Let G be a connected graph. A *DFS-numbering* is a numbering of the vertices of G obtained from searching a DFS-tree of G in post-order. SH algorithm uses a DFS-numbering instead of a LEC-numbering. If the vertices of G are ordered according to a DFS-numbering, then the graph $G[\{i+1, \ldots, n\}]$ is connected, for $i = 1, \ldots, n$. As a DFS-numbering does not guarantee that $H := G[\{1, \ldots, i-1\}]$ is connected, if there exists a frame F of H and H is not connected, then F is also not connected. Besides, to compute (if it exists) a frame of $H + i$, it is necessary to compute an XY-path for each component of F that contains a neighbor of i.

Let v be a vertex of F and C be a block of F containing v and, if possible, a higher numbered vertex. We say v is *active* if v sees $X \cup Y$ through $E_F \setminus E_C$.

PC-tree

The data structure proposed by SH to store F is called a *PC-tree* and is here denoted by T. Conceptually, a PC-tree is an arborescence representing the relevant information of the block forest of F. It consists of *P-nodes* and *C-nodes*.

There is a P-node for each active vertex of F and a C-node for each cycle of F. We refer to a P-node by the corresponding vertex of F. There is an arc from a P-node u to a P-node v in T if and only if uv is a block of F. Each C-node c has a circular list, denoted $RBC(c)$, with all P-nodes in its corresponding cycle of F, in the order they appear in this cycle. This list starts by the largest numbered P-node in it, which is called its *head*. The head of the list has a pointer to c. Each P-node appears in at most one RBC in a non-head cell. It might appear in the head cell of several RBCs. Each P-node v has a pointer $nonhead\_RBC\_cell(v)$ to the non-head cell in which it appears in an RBC. This pointer is NULL if there is no such cell. The name RBC extends for *representative bounding cycle* (Figs. 7(a)-(c)).

Let T' be the rooted forest whose node set coincides with the node set of T and the arc set is defined as follows. Every arc of T is an arc of T'. Besides these arcs, there are some *virtual* arcs: for every C-node c, there is an arc in T' from c to the P-node which is the head of $RBC(c)$ and there is an arc to c from all the other P-nodes in $RBC(c)$ (Fig. 7(d)). In the exposition ahead, we use on nodes of T concepts such as *parent*, *child*, *leaf*, *ancestral*, *descendant* and so on. By these, we mean their counterparts in T'.

Forest T' is not really kept by the implementation. However, during each iteration, some of the virtual arcs are determined and temporarily stored to avoid traversing parts of the PC-tree more than once. So, each non-head cell in an RBC and each C-node has a pointer to keep its virtual arc, when it is determined. The pointer is NULL while the virtual arc is not known.

Values $h(u)$ and $b(v)$

For each vertex u of G, denote by $h(u)$ the largest numbered neighbor of u in G. This value can be computed together with a DFS-numbering, and can be stored in an array at the beginning of the algorithm.

For each node v of T, let $b(v) := \max\{h(u) : u \text{ is a descendant of } v \text{ in } T\}$. For a C-node of T, this number does not change during the execution of the algorithm. On the other hand, for a P-node of T, this number might decrease because its set of descendants might shrink when T is modified. So, in the implementation, the value of $b(c)$ for a C-node c is computed and stored when c is created. It is the maximum over $b(u)$ for all u in the path in T corresponding to the XY-path in F that originated c. One way to keep $b(v)$ for a P-node v is, at the beginning of the algorithm, to build an adjacency list for G sorted by the values of h, and to keep, during the algorithm, for each P-node of T, a pointer to the last traversed vertex in its sorted adjacency list. Each time the algorithm needs to access $b(v)$ for a P-node v, it moves this pointer ahead on the adjacency list (if necessary) until (1) it reaches a vertex u which has v as its parent, in which case $b(v)$ is the maximum between $h(v)$ and $b(u)$, or (2) it reaches the end of the list, in which case $b(v) = h(v)$.

Traversal of the PC-tree

The traversal of the PC-tree T, inspired by Boyer and Myrvold [3, 4], is done as follows. To go from a P-node u to a node v which is an ancestral of u in T,

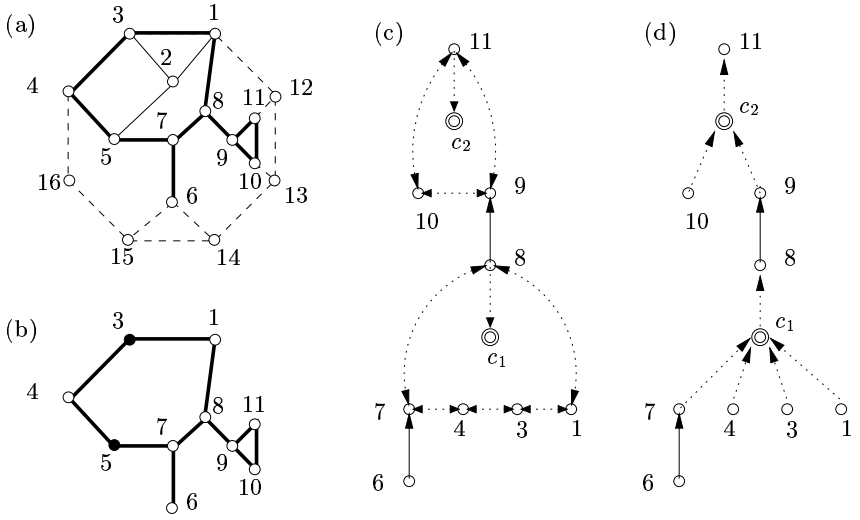

Fig. 7. (a) A graph G, a DFS-numbering of its vertices and, in thick edges, a frame F of $G[1\,..\,11]$ in G. (b) Black vertices in frame F are inactive. (c) The PC-tree T for F, with RBCs indicated in dotted. (d) Rooted tree T' corresponding to T; virtual arcs are dashed.

one starts with $x = u$ and repeats the following procedure until $x = v$. If x is a P-node and *nonhead_RBC_cell*(x) is NULL, move x up to its parent. If x is a P-node and *nonhead_RBC_cell*(x) is non-NULL, either its virtual arc is NULL or not. If it is non-NULL, move x to the C-node pointed by the virtual arc. Otherwise, starting at *nonhead_RBC_cell*(x), search the RBC in an arbitrary direction until either (1) the head of the RBC is reached or (2) a cell in the RBC with its virtual arc non-NULL is reached or (3) a P-node y such that $b(y) > w$ is reached. If (3) happens before (1), search the RBC, restarting at *nonhead_RBC_cell*(x), but in the other direction, until either (1) or (2) happens. If (1) happens, move x to the C-node pointed by the head. If (2) happens, move x to the C-node pointed by the virtual arc. In any case, search all visited cells in the RBC again, setting their virtual arcs to x. Also, set the virtual arc from x to the head of its RBC.

In a series of moments, the implementation traverses parts of T. For each node of T, there is a mark to tell whether it was already visited in this iteration or not. By visited, we mean a node which was assigned to x in the traversal process described above. Every time a new node is included in T, it is marked as unvisited. Also, during each phase of the algorithm where nodes are marked as visited, the algorithm stacks each visited node and, at the end of the phase, unstacks them all, undoing the marks. This way, at the beginning of each iteration, all nodes of T are marked as unvisited.

The same trick with a stack is done to unset the virtual arcs. When a virtual arc for a node v is set in the traversal, v is included in a second stack and, at

the end of the iteration, this stack is emptied and all corresponding virtual arcs are set back to NULL.

Terminals

The next concept, introduced by SH, is the key on how to search efficiently for an XY-obstruction. A node t of T is a *terminal* if

(t1) $b(t) > w$;
(t2) t has a descendant in T that is a neighbor of w in G;
(t3) no proper descendant of t satisfies properties (t1) and (t2) simultaneously.

Because of the orientation of the PC-tree, one of the terminals from Section 4 might not be a terminal here. This happens when one of the terminals from Section 4 is an ancestor in the PC-tree of all others. An extra effort in the implementation is necessary to detect and deal with this possible extra terminal.

The first phase of an iteration of the implementation is the search for the terminals. This phase consists of, for each neighbor v of w such that $v < w$, traversing T starting at v until a visited node z is met. (Mark all nodes visited in the traversal; this will be left implicit from now on.) On the way, if a node u such that $b(u) > w$ is seen, mark the first such node as a *candidate-terminal* and, if z is marked as such, unmark it. The result from this phase is the list of terminals for each component of F.

Search for XY-obstructions

The second phase is the search for an XY-obstruction. First, if there are three or more terminals for some component of F, then there is an XY-obstruction of type either (o2) or (o3) in F (Case 2 of Central algorithm). We omit the details on how to effectively find it because this is a terminal case of the algorithm. Second, if there are at most two terminals for each component of F, then, for each component of F with at least one terminal, do the following. If it has two terminals, call them t_1 and t_2. If it has only one terminal, call it t_1 and let t_2 be the highest numbered vertex in this component. Test each C-node c on the path in T between t_1 and t_2 for an XY-obstruction of type (o1) (Case 1B of Central algorithm). The test decides if the cycle in F corresponding to c plays or not the role of C in (o1). Besides these tests, the implementation performs one more test in the case of two terminals. The least common ancestor m of t_1 and t_2 in T is tested for an XY-obstruction of type (o2), if m is a C-node, or an XY-obstruction of type (o3), if m is a P-node. This extra test arises from the possible undetected terminal.

To perform each of these tests, the implementation keeps one more piece of information for each C-node c. Namely, it computes, in each iteration, the number of P-nodes in $RBC(c)$ that see X through $E_F \setminus E_C$, where C is the cycle in F corresponding to c. This number is computed in the first phase. Each C-node has a counter that, at the beginning of each iteration, values 1 (to account for the head of its RBC). During the first phase, every time an RBC is entered through a P-node which was unvisited, the counter of the corresponding C-node

is incremented by 1. As a result, at the end of the first phase, each (relevant) C-node knows its number.

For the test of a C-node c, the implementation searches $RBC(c)$, starting at the head of $RBC(c)$. It moves in an arbitrary direction, stopping only when it finds a P-node u (distinct from the head) such that $b(u) > w$. On the way, the implementation counts the number of P-nodes traversed. If only one step is given, it starts again at the head of $RBC(c)$ and moves to the other direction until it finds a P-node u such that $b(u) > w$, counting the P-nodes, as before. If the counter obtained matches the number computed for that C-node in the first phase, it passed the test, otherwise, except for two cases, there in an XY-obstruction of type (o1). The first of the two cases missing happens when there are exactly two terminals and c is the lower common ancestor of them. The second of the two cases happens when there is exactly one terminal and c is (potentially) the upper block in which the XY-path ends. The test required in these two cases is slightly different, but similar, and might give raise to an XY-obstruction of type (o1) or (o2). We omit the details.

PC-tree update

The last phase refers to Case 2B in LEC method. It consists of the modification of T according to the new frame. First, one has to add to T a P-node for w. Then, parts of T referring to a component with no neighbor of w remain the same. Parts of T referring to a component with exactly one neighbor of w are easily adjusted. So we concentrate on the parts of T referring to components with two or more neighbors of w. Each of these originates a new C-node. For each of them, the second phase determined the basis of an XY-path, which is given by a path Q in T. Path Q consists basically of the nodes visited during the second phase. Let us describe the process in the case where there is only one terminal. The case of two terminals is basically a double application of this one.

Call c the new C-node being created. Start $RBC(c)$ with its head cell, which refers to w, and points back to c. Traverse Q once again, going up in T. For each P-node u in Q such that *nonhead_RBC_cell*(u) is NULL, if $b(u) > w$ (here we refer to the possibly new value of $b(u)$, as u might have lost a child in the traversal), then an RBC cell is created, referring to u. It is included in $RBC(c)$ and *nonhead_RBC_cell*(u) is set to point to it. For each P-node u such that *nonhead_RBC_cell*(u) is non-NULL, let c' be its parent in T. Concatenate to $RBC(c)$ a part of $RBC(c')$, namely, the part of $RBC(c')$ that was not used to get to c' in any traversal in the second phase. To be able to concatenate without traversing this part, one can use a simple data structure proposed by Boyer and Myrvold [4, 3] to keep a doubled linked list. (The data structure consists of the cells with two indistinct pointers, one for each direction. To move in a certain direction, one starts making the first move in that direction, then, to keep moving in the same direction, it is enough to choose always the pointer that does not lead back to the previous cell.)

During the traversal of Q, one can compute the value of $b(c)$. Its value is simply the maximum of $b(u)$ over all node u traversed. This completes the description of the core of the implementation.

Certificate

To be able to produce a certificate for its answer, the implementation carries still more information. Namely, it carries the DFS-tree that originated the DFS-numbering of the vertices and, for each C-node, a combinatorial description of a planar embedding of the corresponding biconnected component where the P-nodes in its *RBC* appear all on the boundary of the same face. We omit the details, but one can find at `http://www.ime.usp.br/~coelho/sh/` the complete implementation, that also certificates its answer.

6 Experimental study

The main purpose of this study was to confirm the linear-time behavior of our implementation and to acquire a deeper understanding of SH algorithm. Boyer *et al.* [11] made a similar experimental study that does not include SH algorithm.

The LEDA platform has a planarity library that includes implementations of Hopcroft and Tarjan's (HT) and BL algorithms and an experimental study comparing them. The library includes the following planar graph generator routines: `maximal_planar_map` and `random_planar_map`. Neither of them generates plane maps according to the uniform distribution [14], but they are well-known and widely used. The following classes of graphs obtained through these routines are used in the LEDA experimental study:

(G1) random planar graphs;
(G2) graphs with a $K_{3,3}$: six vertices from a random planar graph are randomly chosen and edges among them are added to form a $K_{3,3}$;
(G3) graphs with a K_5: five random vertices from a random planar graph are chosen and all edges among them are added to form a K_5;
(G4) random maximal planar graphs;
(G5) random maximal planar graphs plus a random edge connecting two non-adjacent vertices.

Our experimental study extends the one presented in LEDA including our implementation of SH algorithm made on the LEDA platform and an implementation of BM algorithm developed in `C`. We performed all empirical tests used in LEDA to compare HT and BL implementations [15]. The experimental environment was a PC running GNU/Linux (RedHat 7.1) on a Celeron 700MHz with 256MB of RAM. The compiler was the `gcc 2.96` with options `-DLEDA_CHECKING_OFF -O`.

In the experiments [15, p. 123], BL performs the planarity test 4 to 5 times faster than our SH implementation in all five classes of graphs above. For the planar classes (G1) and (G4), it runs 10 times faster than our SH to do the planarity test and build the embedding. On (G2) and (G3), it is worse than our SH, requiring 10% to 20% more time for testing and finding an obstruction. On (G5), it runs within 65% of our SH time for testing and finding an obstruction. For the planarity test only, HT runs within 70% of our SH time for the planar

classes (G1) and (G4), but performs slightly worse than our SH on (G2) and (G3). On (G5), it outperforms our SH, running in 40% of its time. For the planar classes (G1) and (G4), HT is around 4 times faster when testing and building the embedding. (The HT implementation in question has no option to produce an obstruction when the input graph is non-planar; indeed, there is no linear-time implementation known for finding the obstruction for it [22].) BM performs better than all, but, remember, it is the only one implemented in `C` and not in the LEDA platform. It runs in around 4% of the time spent by our SH for testing and building the embedding and, for the non-planar classes, when building the obstruction, it runs in about 15% of our SH time on (G2) and (G3) and in about 10% of our SH time on (G5). (There is no implementation of BM available that only does the planarity testing.) The time execution used on these comparisons is the average CPU time on a set of 10 graphs from each class.

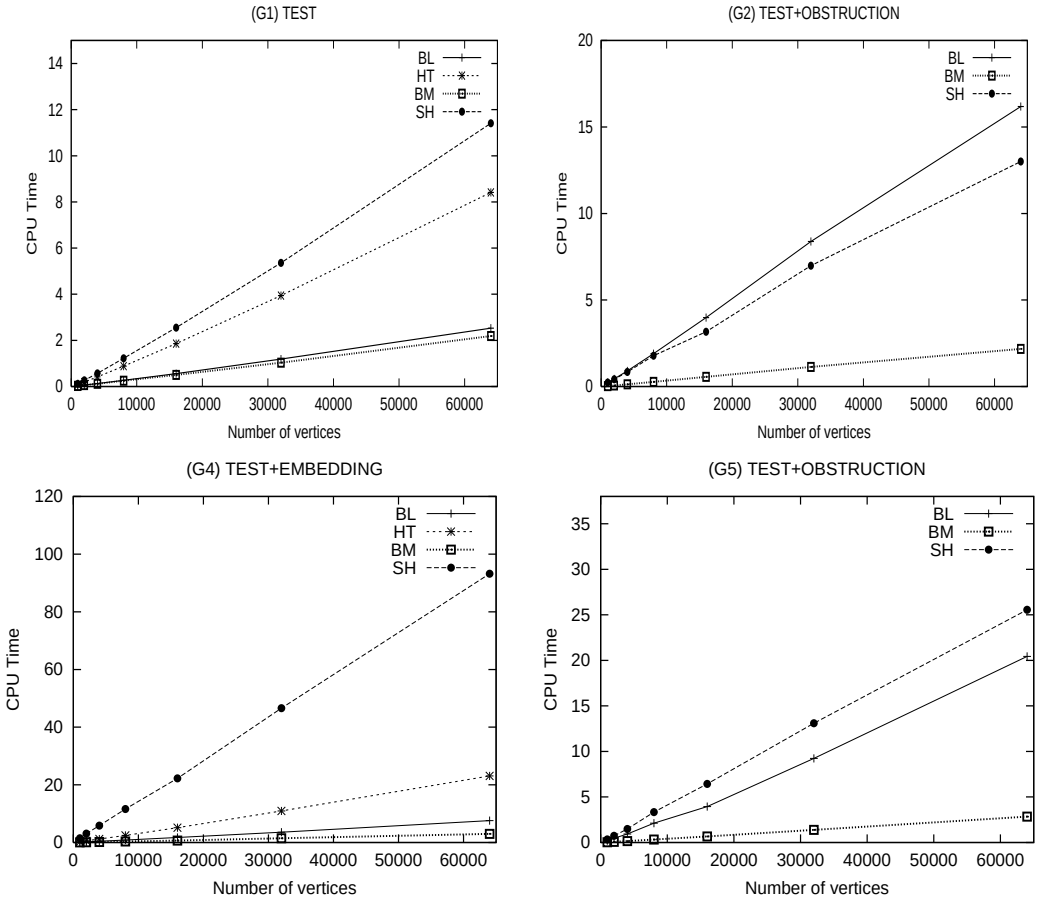

Fig. 8. Empirical results comparing SH, HT, BL, and BM implementations.

Figure 8 shows the average CPU time of each implementation on (a) (G1) for only testing planarity (against BM with testing and building an embedding, as there is no testing only available), (b) (G2) for testing and finding an obstruction (HT is not included in this table, by the reason mentioned above), (c) (G4) for testing and building an embedding, and (d) for testing and finding an obstruction (again, HT excluded).

We believe the results discussed above and shown in the table are initial and still not conclusive because our implementation is yet a prototype. (Also, in our opinion, it is not fair to compare LEDA implementations with `C` implementations.)

Our current understanding of SH algorithm makes us believe that we can design a new implementation which would run considerably faster. Our belief comes, first, from the fact that our current code was developed to solve the planarity testing only, and was later on modified to also produce a certificate for its answer to the planarity test. Building an implementation from the start thinking about the test and the certificate would be the right way, we believe, to have a more efficient code. Second, during the adaptation to build the certificate (specially the embedding when the input is planar) made us notice several details in the way the implementation of the test was done that could be improved. Even though, we decide to go forward with the implementation of the complete algorithm (test plus certificate) so that we could understand it completely before rewriting it from scratch. The description made on Section 5 already incorporates some of the simplifications we thought of for our new implementation. It is our intention to reimplement SH algorithm from scratch.

References

1. L. Auslander and S.V. Parter. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10:517–523, 1961.
2. K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ–tree algorithms. *Journal of Computer and Systems Sciences*, 13:335–379, 1976.
3. J.M. Boyer and W. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. Preprint, 29pp.
4. J.M. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 140–146, 1999.
5. N. Chiba, T. Nishizeki, A. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ–trees. *Journal of Computer and Systems Sciences*, 30:54–76, 1985.
6. N. Deo. Note on Hopcroft and Tarjan planarity algorithm. *Journal of the Association for Computing Machinery*, 23:74–75, 1976.
7. S. Even and R.E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.
8. A.J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conf.* Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dep.of Math., Princeton U., 2 pp., 1963.

9. J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974.
10. W.L. Hsu. An efficient implementation of the PC-tree algorithm of Shih & Hsu's planarity test. Technical report, Inst. of Information Science, Academia Sinica, 2003.
11. M.P̃atrignani J.M.B̃oyer, P.F.C̃ortese and G.D̃i Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Graph Drawing (GD 2003)*, volume 2912 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2004.
12. M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ-trees in automatic graph drawing. In G. Di Battista, editor, *Proc. 5th International Symposium on Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag, Sept. 1997.
13. A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232, New York, 1967. Gordon and Breach.
14. K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge Press, 1997.
15. A. Noma. Análise experimental de algoritmos de planaridade (in Portuguese). Master's thesis, Universidade de São Paulo, 2003. `http://www.ime.usp.br/dcc/posgrad/teses/noma/dissertation.ps.gz`.
16. E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
17. W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.
18. W.K. Shih and W.L. Hsu. A simple test for planar graphs. In *Proceedings of the International Workshop on Discrete Math. and Algorithms*, pages 110–122. University of Hong Kong, 1993.
19. R. Thomas. Planarity in linear time. `http://www.math.gatech.edu/~thomas/`, 1997.
20. S.G. Williamson. Embedding graphs in the plane – algorithmic aspects. *Ann. Disc. Math.*, 6:349–384, 1980.
21. S.G. Williamson. *Combinatorics for Computer Science*. Computer Science Press, Maryland, 1985.
22. S.G. Williamson. Personal Communication, August 2001.