# A LINEAR-TIME ALGORITHM FOR FINDING A MAXIMAL PLANAR SUBGRAPH *

HRISTO N. DJIDJEV †

Los Alamos National Laboratory
Basic and Applied Simulation Science (CCS-5)
Los Alamos, NM 87545, USA

**Abstract.** We construct an optimal linear-time algorithm for the maximal planar subgraph problem: given a graph $G$, find a planar subgraph $G'$ of $G$ such that adding to $G'$ an extra edge of $G$ results in a non-planar graph. Our solution is based on a fast data structure for incremental planarity testing of triconnected graphs and a dynamic graph search procedure. Our algorithm can be transformed into a new optimal planarity testing algorithm.

**Key words.** Planar graphs, planarity testing, incremental algorithms, graph planarization, data structures, triconnectivity.

**AMS subject classifications.** 05C10, 05C85, 68R10, 68Q25, 68W40

**1. Introduction.** A graph is *planar* if it can be drawn in the plane so that no two edges intersect except at a common endpoint. Planar graphs arise naturally in many applications of graph theory, e.g., in circuit and VLSI design, network design and analysis, computational geometry, and are one of the most intensively studied classes of graphs [21]. Many problems that are computationally hard for arbitrary graphs have efficient solutions for the case of planar graphs. Testing an $n$-vertex $m$-edge graph for planarity takes $O(n+m)$ time [16, 2].

If the graph is not planar, then often a problem arises of how to find a planar subgraph that is as close to the given graph, as possible. A problem of this type is called a *graph planarization problem.* For any $n$ vertex graph $G$ of genus $g$ there exists a vertex set of size $O(\sqrt{ng})$ whose removal leads to a planar graph [9]. However, the linear-time implementation of the algorithm that finds such planarizing set requires a genus-$g$ embedding of $G$ as input; the best algorithm that finds such embedding [10] is polynomial in $n$, but doubly exponential in $g$. Another version of the graph planarization problem, the problem of finding the smallest number of edges whose removal leaves a planar graph, is known to be NP-complete [13].

Since finding a maximum planar subgraph is very hard, many researchers have investigated the problem of constructing, for a given $n$-vertex $m$-edge graph $G$, a planar subgraph $G'$ of $G$ such that adding to $G'$ any edge of $E(G) - E(G')$ results in a non-planar graph. Such graph $G'$ is called a *maximal planar subgraph* of $G$. This problem has been intensively investigated in relation to its applications to circuit layout [23, 20, 5, 22, 19]. More recently, Cai, Han, and Tarjan [4] developed an $O(m \log n)$ algorithm for the maximal planar subgraph problem based on the Hopcroft-Tarjan planarity testing algorithm. Their result improved (if $m = o(n^2/\log n)$) the best previous $O(n^2)$ algorithm from [19] (based on the PQ-tree technique [2]). An algorithm with the same complexity bound of $O(m \log n)$ can also be derived from the incremental

*A preliminary version of this paper was presented at WADS '95 [8].

†This work is partially supported by NSF grant CCR-9409191 and by EPA grant R82-5207-01-0.

planarity testing algorithm of Di Battista and Tamassia [6]. Using an approach similar to [6], Westbrook [26] described an algorithm that works in $O(n \log n + m\alpha(m,n))$ worst-case time plus an additional $O(n)$ expected time. La Poutré [24] recently gave an incremental planarity testing algorithm that takes $O(\alpha(m,n))$ amortized time per operation, which can be transformed into an $O(n + m\alpha(m,n))$ time algorithm for the maximal planar subgraph problem.

In this paper we describe a linear-time $O(n+m)$ time algorithm for the maximal planar subgraph problem. Our algorithm uses a tree-represented decomposition of a biconnected graph into triconnected components, a common feature of the incremental planarity testing algorithms [6, 7, 26, 24]. We use a variation of the decomposition tree of Di Battista and Tamassia; however, any of the alternative representations could be used instead. Our algorithm has the following structure: (i) it initially constructs a depth-first spanning tree of $G$ (we can assume that w.l.o.g. that $G$ is connected) and uses it as an initial approximation of the maximal planar subgraph; (ii) it adds the edges one by one, making an on-line choice of the next edge to be added so that the testing time will be appropriately small.

Note that our ability to make a choice of the order in which to insert, while possible, the edges into the subgraph so that planarity is preserved is essential for achieving $O(1)$ amortized time per test and insert operation. As noted by Westbrook [26], there is an $\Omega(\alpha(m,n))$ lower bound on the amortized time per operation of any algorithm that maintains a decomposition of the triconnected components of a graph subject to *arbitrary* edge insertions, which gave rise to the conjecture that $O(\alpha(m,n))$ is the best possible time bound for the incremental planarity testing problem [24].

Another technique we use is maintaining in each bicomponent a special dynamic path of nodes of the decomposition tree such that all testing and updating operations are performed on nodes of that path. This makes it possible to implement data structures supporting set union and set split operations in a constant amortized time. Also, we develop a new efficient data structure used for incremental planarity testing of triconnected graphs which works in $O(1)$ amortized time per operation, an improvement over the best previous $O(\alpha(m,n))$ time algorithms.

Independently of our result, Hsu [17] has constructed a linear-time algorithm for the maximal planar subgraph problem that is based on the modified version of the Hopcroft-Tarjan's planarity testing algorithm.

Our algorithm for the maximal planar subgraph problem can be transformed into a linear-time algorithm for planarity testing based on an approach entirely different from the existing ones. The previous algorithms of Hopcroft and Tarjan [16] and Booth and Lueker [2] are based on the Jordan Curve Theorem, which states that any closed curve in the plane divides it into exactly two disjoint connected regions, while our planarity testing algorithm exploits the fact that any triconnected planar graph has a unique embedding in the plane.

This paper is organized as follows. In Section 2 we give some definitions and review a dynamic data structure that maintains a decomposition of a connected graph into biconnected and triconnected components. In Section 3 we develop an algorithm for on-line planarity testing in triconnected graphs in a constant amortized time, which we use as a subroutine in the main algorithm. In Section 4 we give the overall structure of the algorithm as well as more details about individual data structures and update operations.

**2. Preliminaries.** In this section we give some basic definitions related to graph connectivity and graph orientation and describe briefly the data structure for maintaining the biconnected and triconnected components of a graph developed by Di Battista and Tamassia [6, 7].

**2.1. Definitions.** We use standard graph terminology [14]. An undirected graph $G$ is *connected* if any two vertices of $G$ are connected by a path. The maximal connected subgraphs of $G$ are the *connected components* of $G$. A vertex $v$ is a *cutvertex* if the removal of $v$ increases the number of components. $G$ is *biconnected*, if $G$ is connected and $G$ has no cutvertices. The maximal biconnected subgraphs of $G$ are called *bicomponents*. A pair $v, w$ of vertices of $G$ is a *separation pair*, if the deletion of $v$ and $w$ disconnects $G$. $G$ is *triconnected*, if $G$ has no cutvertex and no separation pair.

An essential property of triconnected graphs related to planarity is given in the next lemma. A *subdivision* of a graph $H$ is a graph $H'$ that can be obtained by $H$ by replacing some of the edges of $H$ by paths having at most their endpoints in common.

LEMMA 2.1. *[21] A planar graph $G$ has a unique embedding in the plane if and only if $G$ is a subdivision of a triconnected graph.*

The *triconnected components (or tricomponents)* of $G$ are produced by a recursive procedure that, if $G$ has a separation pair $v, w$, divides $G$ into two subgraphs $G_1$ and $G_2$ defined by the separation pair. Each of $v$ and $w$ is included in both $G_1$ and $G_2$. For the precise definition and a linear-time algorithm that finds the tricomponents of a graph see [15].

An *st-graph* is a directed acyclic graph with exactly one source and exactly one sink. Any biconnected graph can be converted into an *st*-graph using the linear-time *st*-numbering algorithm of [11]. A *planar st-graph* is an *st*-graph that is embedded in the plane such that the source and the sink belong to the external face of the embedding.

Let $G$ be an *st*-graph. A *split pair* $\{a, b\}$ of $G$ is either a separation pair or a pair of adjacent vertices of $G$. A *split component* of a split pair $\{a, b\}$ is either an edge $(a, b)$, or a maximal subgraph $G'$ of $G$ that is an *st*-graph with a source $a$ and a sink $b$ such that $\{a, b\}$ is not a split pair of $G'$. For instance, the graph $G$ from Figure 2.1 (a) has split pairs (3,8), (3,5), (2,7), and all pairs of adjacent vertices; its split components are the subgraphs $G_2$, $G_4$, and $G_5$ from Figure 2.1, as well as all edges of $G$. If there is no other split pair $\{a', b'\}$ such that $\{a, b\}$ is contained in a split component of $\{a', b'\}$, then $\{a, b\}$ is a *maximal split pair.*

**2.2. Decompositions of biconnected graphs.** First we consider the case where $G$ is biconnected. Let $n$ be the number of vertices and $m$ be the number of edges of $G$.

We recall the definition of SPQR trees from [6]. An *SPQR* tree for $G$ is a recursively defined tree $T$ closely related to the decomposition of $G$ with respect to its split pairs. $T$ has four types of nodes S, P, Q, and R and there is an *st*-graph, *skeleton*$(\mu)$, associated with each node $\mu$ of $T$. The skeletons of the internal nodes of $T$ are in one-to-one correspondence with the tricomponents of $G$ and hence their number is $O(m)$. The endpoints of each edge $e$ in the skeleton of the root of $T$ correspond to a maximal split pair of $G$ and $e$ represents the set of split components of that split pair (see Figure 2.1).

Formally, SPQR trees are defined as follows [6, 7]. Let $G$ be an *st*-graph with source $s$ and sink $t$. An SPQR tree $T$ for $G$ has four types of nodes S, P, Q, and R and there is an *st*-graph, *skeleton*$(\mu)$, associated with each node $\mu$ of $T$. $T$ is defined recursively as follows.

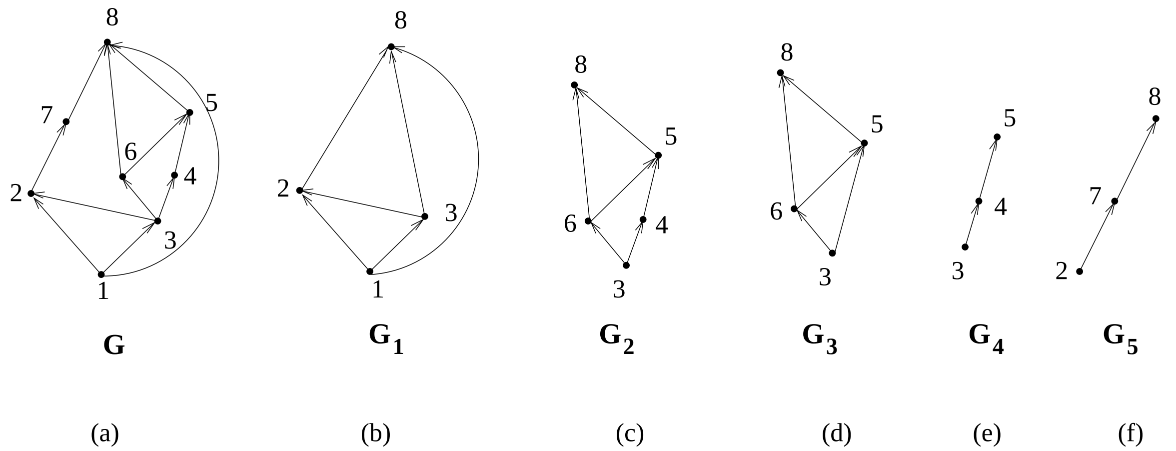


FIG. 2.1. *Illustration to the definition of SPQR trees:*
*(a) A planar st-graph with source 1 and sink 8.*
*(b) The skeleton associated with the root $\mu$ of the SPQR tree for $G$ ($\mu$ is an R node).*
*(c) The split component corresponding to edge (3,8) of $G_1$.*
*(d) The skeleton associated with node($G_2$).*
*(e) The split component corresponding to edge (3,5) of $G_3$.*
*(f) The split component corresponding to edge (2,8) of $G_1$.*

(i) (*trivial case*) If $G$ is a single edge from $s$ to $t$, then $T$ consists of a single Q node $\mu$ (leaf) and $skeleton(\mu)$ is $G$.

(ii) (*parallel case*) If $\{s, t\}$ is a split pair with split components $G_1, \ldots, G_k$, then the root of $T$ is a P node $\mu$ and $skeleton(\mu)$ consists of vertices $s$ and $t$ joined by $k$ parallel edges $e_1, \cdots, e_k$.

(iii) (*series case*) If $G$ has cut vertices $x_1, \ldots, x_{k-1}$, $k \geq 1$, dividing it into components $G_1, \ldots, G_k$ in this order from $s$ to $t$, then the root of $T$ is an S node $\mu$ and $skeleton(\mu)$ is the path $s = x_0, x_1, \ldots, x_{k-1}, t = x_k$. We denote edge $e_i = (x_{i-1}, x_i)$, for $i = 1, \cdots, k$.

(iv) (*rigid case*) If none of the above case applies, let $\{a_1, b_1\}, \ldots, \{a_k, b_k\}$ be the maximal split pairs of $G$ and let $G_i$ be the union of all split components of $\{a_i, b_i\}$. Then the root of $T$ is an R node $\mu$ and $skeleton(\mu)$ is the graph obtained by replacing in $G$ each subgraph $G_i$ by an edge between $a_i$ and $b_i$ with orientation compatible with the orientation of $G_i$ (so that the resulting graph be an *st*-graph.)

In cases (ii)-(iv) $\mu$ has children nodes $\mu_1, \ldots, \mu_k$ which are the roots of the SPQR trees of $G_1, \ldots, G_k$. We denote $node(G) = \mu$ and $node(G_i) = \mu_i$. The edge $e_i$ representing the skeleton of $\mu_i$ in $G$ is called a *virtual edge* of $\mu_i$.

A property of the SPQR trees that is relevant to planarity testing is that the skeleton of any internal node $\mu$ of a SPQR tree has either a unique planar embedding (if $\mu$ is an R node), or *any* two edges can be placed on the same face (if $\mu$ is a P, Q, or S node.) For a more detailed discussion of SPQR trees see [6, 7]. Our next goal is to show how to reduce a planarity testing in a graph to planarity testing in skeletons of nodes of its SPQR tree.

**2.2.1. Planarity testing using SPQR trees.** An *allocation node* of a vertex $v$ of $G$ is a node $\mu$ such that $skeleton(\mu)$ contains $v$. A *proper allocation node* of $v$, denoted by $proper(v)$, is the least common ancestor of all allocation nodes of $v$.

For any pair of vertices $v_1$ and $v_2$ we will define *projections* $pr(v_1)$ and $pr(v_2)$, so either can be a vertex or an edge of the skeleton of an appropriate node $\mu$ of the SPQR tree. The relevant property of the projections of $v_1$ and $v_2$ is that $v_1$ and $v_2$ belong to the same face of $G$ if and only if $pr(v_1)$ and $pr(v_2)$ will belong to the same face of the skeleton of $\mu$.

Let $v_1$ and $v_2$ be two vertices of $G$ and let $proper(v_1)$ and $proper(v_2)$ be the proper allocation nodes of $v_1$ and $v_2$. Assume that $proper(v_1)$ and $proper(v_2)$ belong to a simple tree path to the root of the SPQR tree. (In our algorithm described in Section 4.1 below this will always be the case.) Assume w.l.o.g. that $proper(v_1)$ is an ancestor of $proper(v_2)$. Define $\mu = \mu(v_1, v_2)$ to be the nearest common ancestor of $proper(v_1)$ and $proper(v_2)$ ($proper(v_1)$ in our case). Call a *joining path* $p(v_1, v_2)$ the tree path in the SPQR tree between $proper(v_1)$ and $proper(v_2)$ excluding $\mu$ and its child. For $i = 1$ and $i = 2$, if $skeleton(\mu)$ contains $v_i$, then define $pr(v_i)$ to be the vertex $v_i$ of $skeleton(\mu)$, otherwise define $pr(v_i)$ to be the virtual edge in $skeleton(\mu)$ corresponding to the subgraph containing $v_i$.

We define a *peripheral* vertex (resp. edge) of an *st*-graph to be a vertex (resp. edge) that appears on the external face of some *st*-planar embedding of the graph. A *peripheral node* is a node $\mu$ whose virtual edge is peripheral.

The following lemma relates incremental planarity testing in an arbitrary graph to incremental planarity testing in its tricomponents (assuming $proper(v_1)$ is an ancestor of $proper(v_2)$).

LEMMA 2.2. *[6] There exists a planar embedding of $G$ such that $v_1$ and $v_2$ belong to the same face if and only if*

*(i) $pr(v_1)$ and $pr(v_2)$ are on the same face of some planar embedding of $skeleton(\mu(v_1, v_2))$,*

*(ii) all the nodes on the joining path $p(v_1, v_2)$ are peripheral, and*

*(iii) if $proper(v_1) \neq proper(v_2)$ then $v_2$ is a peripheral vertex of $proper(v_2)$.*

As an example, apply the lemma for vertices $v_1 = 2$ and $v_2 = 4$ of $G$ (Figure 2.1). Their projections (in $G_1$) are $pr(v_1) = 2$ and $pr(v_2) = (3, 8)$. The joining path $p(v_1, v_2)$ consists of a single node $node(G_4)$ whose virtual edge $(3, 5)$ in $G_3$ is peripheral. By Lemma 2.2 edge $(2, 4)$ can be added to $G$ while preserving planarity.

In the next sections we will describe data structures for answering queries of types (i), (ii), and (iii) from Lemma 2.2 in a constant time.

**2.3. Decompositions of connected graphs.** In order to handle connected graphs that are not necessarily biconnected we define the BC trees introduced in [7], which are extensions of the SPQR trees. To construct a BC tree of a connected graph $G$ first find all bicomponents of $G$. Then construct a tree that contains a node of type B for any bicomponent $b$ and a node of type C for any cutvertex $c$ of $G$. Associate with each B node $b$ an SPQR tree representing $b$. Connect a C node $c$ and a B node $b$ iff $c$ belongs to $b$. Finally root the tree at an arbitrary B node. Call the nodes of $B$ *level-1* nodes and the nodes of the SPQR trees *level-2* nodes.

Suppose that an edge $(v_1, v_2)$ has to be added to $G$. If $v_1$ and $v_2$ belong to the same bicomponent $b$ of $G$ then the BC tree of $G$ is not changed after the insertion. In this case we use the SPQR tree associated with $b$ and Lemma 2.2 to determine if $(v_1, v_2)$ can be added while preserving the planarity and do the insertion by modifying the SPQR tree for $b$. Now assume that $v_1$ and $v_2$ belong to different bicomponents $b(v_1)$ and $b(v_2)$. Let $p = \{b_1 = b(v_1), c_1, b_2, c_2, \cdots, c_k, b_{k+1} = b(v_2)\}$ be the unique tree path between $b(v_1)$ and $b(v_2)$. If one of $b_1$ and $b_{k+1}$ is an ancestor to the other and the edge $(c_{i-1}, c_i)$ can be added to $b_i$ while preserving planarity then we call $b_i$

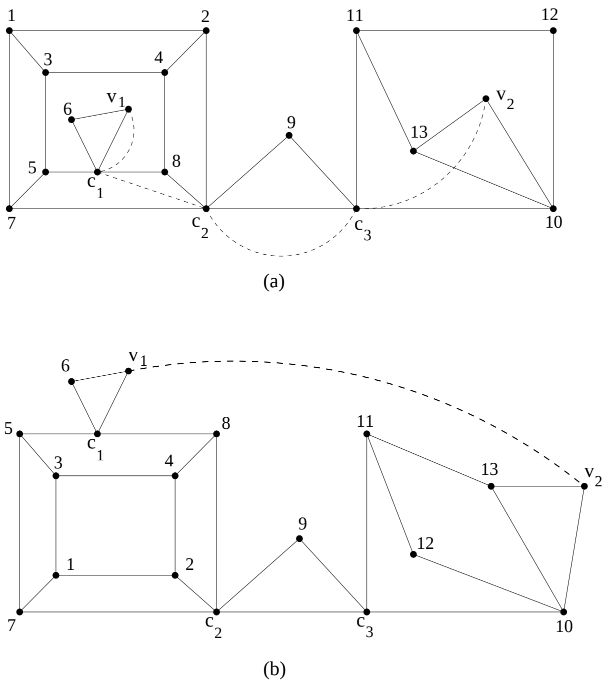


FIG. 2.2. *Illustration to Lemma 2.3.*
*(a) A graph with 4 bicomponents.*
*(b) The result after adding the edge $(v_1, v_2)$ to the planar embedding.*

*peripheral* level-1 node (with respect to path $p$.) We have the following lemma [7].

LEMMA 2.3. *There exists a planar embedding of $G$ such that $v_1$ and $v_2$ belong to the same face if and only if edges $(v_1, c_1)$ and $(c_k, v_2)$ can be added to $G$ while preserving planarity and nodes $b_2, \cdots, b_k$ are peripheral.*

See Figure 2.2 for illustration.

For edges $(v_1, c_1)$ and $(c_k, v_2)$ we do planarity testing using the corresponding SPQR trees and for edges $(c_1, c_2), \ldots, (c_{k-1}, c_k)$ we use dynamically maintained maximal paths of edges $(c_i, c_{i+1})$ whose addition preserves planarity. We will give more details in Section 4.

In order to use the above data structure for the maximal planar subgraph problem we need also algorithms for efficiently updating the data structure after the insertion of any edge. Before discussing the update operations we will describe the data structures for incremental planarity testing in 3-connected graphs and give an outline of the whole algorithm.

**3. The triconnected case.** By Lemma 2.1 the maximal planar subgraph (MPS) problem is easier to solve if a planar spanning triconnected subgraph of the original graph is known. Accordingly, we will first describe a linear-time algorithm for the following restricted version of the MPS problem, which we call the *triconnected maximal planar subgraph* (TMPS) problem.

**Problem:** Let $G$ be a planar triconnected graph and $E'$ be a set of edges between vertices of $G$. Find a maximal set $E'' \subset E'$ such that $G + E''$ is still planar.

No linear-time algorithm for this problem is known. We will use the solution of the TMPS problem, with some little modifications, to the problem of finding a

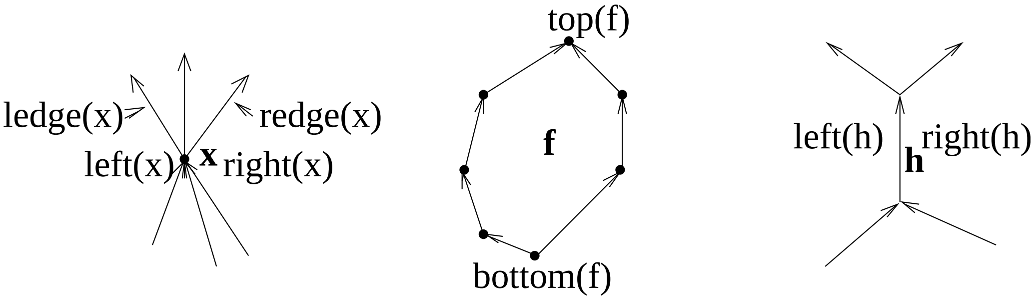


FIG. 3.1. *Illustration to the definition of* $left$*, right, top, and bottom for vertices and* $left$ *and right for edges.*

maximal planar subgraph of a general graph.

Our solution is based on the fact that any triconnected planar graph has a unique planar embedding (Lemma 2.1). Thus an edge $(v, w)$ can be added to a triconnected embedded planar graph so that planarity is preserved, iff $v$ and $w$ belong to the same face of the embedding. To solve the TMPS problem we need a fast procedure that tests if any arbitrary pair of vertices belong to any face of the embedding. We will use the following method of representing planar embeddings from [6].

**3.1. Representation of planar $st$-graphs.** Let $G$ be a planar $st$-graph. For any vertex $x$, the incoming edges in $x$ appear consecutively around $x$ and the edges outgoing from $x$ also appear consecutively around $x$. Thus there is a single face, denoted by $left(x)$, that separates incoming and outgoing edges in a clockwise direction and there is a single face, $right(x)$, that separates incoming and outgoing edges in a counterclockwise direction (Figure 3.1). We also record for $x$ a pair of outgoing edges $ledge(x)$ and $redge(x)$ which are incident to $x$ and respectively to faces $left(x)$ and $right(x)$. Furthermore, the vertices from the boundary of each face $f$ form two directed paths. The common start vertex of these paths will be denoted by $bottom(f)$ and the common endvertex will be denoted by $top(f)$. Furthermore, if $h$ is an edge of $G$, we denote by $right(h)$ the face whose clockwise boundary contains $h$ and by $left(h)$ the other face containing $h$. The values of $left$, $right$, $ledge$, $redge$, $bottom$, and $top$ can be easily computed in linear time and space.

The next lemma concerns testing in a static graph.

LEMMA 3.1. *For any triconnected planar $n$-vertex graph $G$ there exists a data structure for $G$ that can be constructed in $O(n)$ time, uses $O(n)$ space, and that provides answers in $O(1)$ time to the following two types of queries:*

*(a) If $v$ and $w$ are vertices of $G$ and $v$ has degree $O(1)$, check if $v$ and $w$ belong to the same face of $G$.*

*(b) If $v$ is a vertex and $e$ is an edge of $G$, check if $v$ and $e$ belong to the same face of $G$.*

Queries of type (b) will be used in the algorithms described in Section 4.

**Proof:** Consider first a query of type (a). Since the degree of $v$ is $O(1)$, we can answer the query in a constant time by checking if, for some $f_v \in \{left(v), right(v)\}$ and some $f_w \in \{left(w), right(w)\}$, any of the following cases applies:

(3.1) $\quad f_v = f_w;$

(3.2) $\quad top(f_v) = w$ or $bottom(f_v) = w;$

(3.3) $\quad top(f_w) = v$ or $bottom(f_w) = v$;

(3.4) $\quad \{v, w\} = \{top(f), bottom(f)\}$ for some face $f$ incident to $v$.

Similarly, for queries of type (b), the problem is reduced to checking if the following condition is satisfied:

(3.5) $\quad$ There exists $f \in \{left(e), right(e)\}$ such that $f \in \{left(v), right(v)\}$, or $top(f) = v$, or $bottom(f) = v$.

∎

Note that condition (3.4) is the only condition that we would not be able to check in constant time if there were no restrictions on the degree of $v$. Our next goal is to show that for solving the TMPS problem, an edge $(v, w) \in E'$ can always be chosen so that the number of faces $f$ incident to either $v$ or $w$ that are "relevant" in certain context to the planarity testing is $O(1)$. The idea of our solution is related to the observation that any $n$-vertex planar graph $G$ has no more than $3n - 3$ edges and thus there exists a vertex of $G$ of degree less or equal to 6.

Define a graph $G_{tb}$ with vertices $V(G)$ and where the set of edges consists of all pairs $(top(f), bottom(f))$, for each face $f$ of the graph $G$. Note that $G_{tb}$ is a planar graph (since any edge of $G_{tb}$ can be drawn inside a distinct face of $G$) and its edges correspond to all pairs of vertices that satisfy condition (3.4) above. We define a graph $G'_{tb}$ to be the subgraph of $G_{tb}$ induced by the set of vertices incident to at least one edge of $E'$. From the definition of $G'_{tb}$ the next lemma follows.

LEMMA 3.2. *For any edge $(v, w) \in E'$ vertices $v$ and $w$ belong to the same face $f$ of $G$ if and only if $(v, w)$ is in $G'_{tb}$ or at least one of the conditions (3.1)–(3.3) holds. Furthermore, $G'_{tb}$ always contains a vertex of degree at most 6.*

In our algorithm for solving the TMPS problem described below we iteratively choose a new edge $e$ of $E'$ using information about the degrees of $G'_{tb}$. We add $e$ to $G$ if the planarity of the embedding is preserved and we update $G'_{tb}$. The following procedure specifies the details.

**Algorithm** *Triconnected*

{Finds a maximal planar subgraph $G$ of $G_{tr} + E'$ if $G_{tr}$ is triconnected}

**1.** Set initially $G = G_{tr}$ and construct the data structures of Lemma 3.1 for $G$.
**2.** Construct graphs $G_{tb}$ and $G'_{tb}$ for $G$.
**3.** For each vertex $v$ of $G$ construct the linked list of the edges of $E'$ incident to $v$ by a lexicographical sort. For each vertex $v$ of $G$ maintain the values $degree_1(v)$ of the number of edges from $E'$ incident to $v$ and $degree_2(v)$ of the degree of $v$ in $G'_{tb}$. Also maintain a list *SmallDeg* of all vertices whose degree in $G'_{tb}$ is less or equal to 6.
**4.** Repeat until $E' = \emptyset$.
  **4.1.** Pick any vertex $x$ in *SmallDeg* and choose any edge $(x, y) \in E'$.
  **4.2.** If any of the conditions of Lemma 3.1 is satisfied for $x = v$ and $y = w$ then add $(x, y)$ to $G$ and update variables $left, right, top$, $bottom$, and the graph $G'_{tb}$ for $G$.
  **4.3.** Remove $(x, y)$ from $E'$ and update $G'_{tb}$ and variables $degree_1, degree_2$, and *SmallDeg*.

The correctness of Algorithm *Triconnected* follows from Lemma 2.1 and Lemma 3.2. Next we will discuss on how the algorithm can be implemented in linear time.

By the definition of $G'_{tb}$ and Lemma 3.2 the list $SmallDeg$ is non-empty iff $E' \neq 0$ and the conditions from Step 4.2 of Algorithm *Triconnected* can be checked in $O(1)$ time. Furthermore, the initial construction of the data structures in Steps 1 and 2 requires $O(n)$ time. Updating any of the data structures except $left$ and $right$ takes clearly $O(1)$ time.

Maintaining the $left$ and $right$ relations is more complex, because inserting a new edge in $G$ splits a face $f$ of $G$ into two faces $f_1$ and $f_2$ which may require as many as half of the vertices and edges on $f$ to change their $left$ or $right$ pointer from $f$ to $f_1$ or $f_2$. Next we show how to solve this problem making use of the microset technique of Gabow and Tarjan [12].

### 3.2. A find-split-insert data structure.

**3.2.1. Formulation of the problem.** We describe a data structure that maintains a partition of a set of edges of a graph $G$ into a set $\mathcal{P}$ of edge-disjoint paths under the following operations:

- $find(e)$: Return the label of the path containing edge $e$ within it. (Each path is labeled by a distinct integer.)
- $split(v, e)$: Split the path $p$ containing edge $e = (v, w)$ into two paths, the path of the vertices from the start vertex of the path to $v$ and the path from $v$ to the last vertex of the path.
- $insert\_vertex(v, (x, y))$: Replace edge $(x, y)$ by edges $(x, v)$ and $(v, y)$ in the path containing $(x, y)$.
- $add\_vertex(x, (y, z))$: If edge $(y, z)$ belongs to a path $p$ and $y$ is an endpoint of $p$, then add a new edge $(x, y)$ to $p$ making $x$ an endpoints of $p$.
- $insert\_edge(v, w, e)$: Transform the path $p$ containing $e$ into two paths, one consisting of the portion of $p$ between $v$ and $w$ and the other containing the remaining two parts of $p$ joined by an additional edge $(v, w)$. Here it is assumed that $v$ and $w$ belong to $p$ and divide it into exactly three nonempty open paths.
- $new\_path(x, y)$: Create a new path consisting of a single new edge $(x, y)$. It is assumed that no edge $(x, y)$ belongs to a current path in $\mathcal{P}$ .

Each operation will take $O(1)$ amortized time on a random access machine with unit cost measure and $O(\log n)$ bits machine word. The technique we use is similar to the one developed by Gabow and Tarjan [12] for a variation of the set union problem. The same technique was used also by Imai and Asano [18] for the problem of maintaining a partition of a sorted sequence of integers under a sequence of split, find, and insert operations.

**3.2.2. The data structure.** The edges currently in paths of $\mathcal{P}$ are partitioned into subsets of edges occupying consecutive memory locations called *mezzosets*. Each mezzoset contains at most $ln = \lceil \log n \rceil$ edges so that the total number of mezzosets is $O(n/\log n)$ ($n > 2$ is the total number of edges of $G$). Each mezzoset is partitioned into smaller blocks called *microsets* of at most $\lambda(n) = \lceil \log\log n \rceil$ edges each so that the total number of microsets is $O(n/\log\log n)$.

Each path is labeled by a distinct integer. Each mezzoset and each microset will have a label that can be a name of a path in $\mathcal{P}$ or $nil$. A path will consist of a subset of the edges belonging to at most $\lceil n/ln \rceil$ mezzosets with the label of $p$, at most $\lambda(n)$ additional microsets with the label of $p$, and at most 2 microsets with label $nil$.

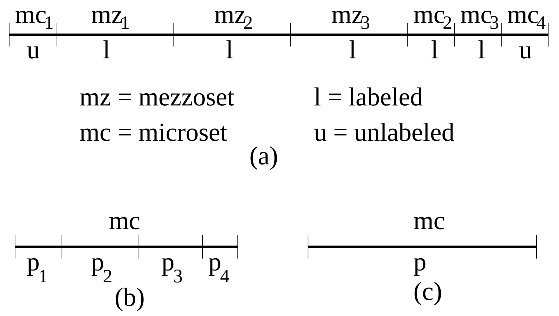


FIG. 3.2. *Examples:*
*(a) A (long) path consisting of 3 mezzosets and 4 microsets.*
*(b) An unlabeled microset containing (short) paths $p_2$ and $p_3$ and portions of paths $p_1$ and $p_4$.*
*(c) A labeled microset containing only internal edges of a path $p$.*

Intuitively, mezzosets and microsets with the label of $p$ will be "internal" for $p$, i.e. they will contain some edge of $p$ but will contain neither the first nor the last edge of $p$. (We will make sure that no more than one such path exists for any microset so the labels of microsets are uniquely defined.)

A microset containing only internal edges will be called a *labeled microset* and a mezzoset containing only identically labeled microsets will be called a *labeled mezzoset.* The other microsets and mezzosets will be called *unlabeled.* See Figure 3.2 for an illustration. The label of any path $p$ is recorded in variables associated with the first and the last edge of $p$ as well.

We will use different strategies for maintaining partitions and implementing the query and update operations on levels of (sequences of) labeled mezzosets or microsets and on the level of individual microsets. To execute an update operation involving an edge $e$, we locate the microset $\mu_1$ and the mezzoset $\mu_2$ containing $e$. Then we perform the necessary updates on $\mu_1$ using the algorithm for microsets. If $\mu_1$ is unlabeled, we are done. If $\mu_1$ is labeled, we apply the algorithm for labeled microsets on $\mu_1$. Finally, if $\mu_2$ is labeled, we apply the algorithm for labeled mezzosets on $\mu_2$. Next we will describe the algorithms applied in each case.

On levels of labeled mezzosets we use the "relabel the smaller half" technique [12]. By this technique we store with each mezzoset its label and maintain the sequence of labeled mezzosets in any path as a doubly linked list. When we split a path the labels of the labeled mezzosets in the smaller half are updated. For a set of $k$ mezzosets this algorithm requires totally $O(k \log k)$ time for all splits and $O(1)$ time for any query [1]. Since $k = O(n/\log n)$, this yields an $O(n)$ bound on the time for operations on the level of labeled mezzosets. A similar method and the same bound apply for maintaining the labels of the labeled microsets.

For maintaining the partition of the edges in the individual microsets we use the following table lookup method. The edges in each microset are arranged in doubly linked lists corresponding to the partition defined by paths in $\mathcal{P}$ . With each edge $e$ of a microset $\mu$ we keep the addresses of the previous and the next edge in the list containing $e$, if any. The addresses are computed relative to the beginning of the memory block corresponding to $\mu$ and therefore each address occupies $\lceil \log \lambda(n) \rceil$ bits. The whole microset can be recorded in one computer word of $O(\lambda(n) \log \lambda(n)) = O(\log n)$ bits.

Next we will explain how this information can be used to answer queries and implement updates in constant time.

**Note:** The main difference between our implementation and those of Gabow and Tarjan [12] and Imai and Asano [18] is that we maintain linked list data structures within each microset, while in previous cases the data structures maintained are arrays. We need lists in order to implement the *insert_edge* operation.

**3.2.3. The *find* operation on microset level.** To answer a $find(e)$ query one can follow the backward or forward pointers to the first or the last edge of the path, if any of these edges is in the same microset. Recall that information about the name of the path is associated with the first and the last edge. If both the first and the last edge are contained in other microsets, then the current microset must be a labeled one and its label gives the name of the path containing $e$.

In order to do these computations faster, we use the table lookup method. The idea of the table lookup approach is to precompute the results of all possible $find$ operations for any possible structure of the microset and record them in a table. Due to the small size of the microsets, the preprocessing will take only linear time and space, as illustrated bellow.

We compute a $first$ and *last* table defined as follows. If $e$ is an edge in a microset $\mu$, if $i$ is the position of $e$ in $\mu$, and if $\mu$ is encoded by an integer $m$, then $first(m,i)$ (resp. $last(m,i)$) denotes the address of the first (resp. last) edge of the path containing $e$ that is also in $\mu$. The number of entries of $first$ are $2^{\lambda(n)\log\lambda(n)}\lambda(n)$ and each entry can be computed in $O(\lambda(n))$ time. Since $\lambda(n) = \lceil \log\log n \rceil$, then

$$2^{\lambda(n)\log\lambda(n)}\lambda(n)^2 = 2^{(\lambda(n)+2)\log\lambda(n)} = 2^{o(\log n)} = o(n).$$

Thus the entire $first$ table occupies $O(n)$ space and can be constructed in $O(n)$ time. Similarly we compute the *last* table.

**3.2.4. The update operations.** Note that updating a pointer in the representation of a microset $\mu$ is equivalent to changing a digit in the radix $\lambda(n)$ representation of $\mu$. Thus the computer word representing $\mu$ can be updated in a constant time if one precomputes the values of $\lambda(n)^k, k = 1, \cdots, \lambda(n) - 1$ and stores them in a table. We implement the update operations on a microset level as follows.

- *split*$(v, e)$: Let $\mu$ be the microset containing $e$. Splitting along $v$ the doubly linked list representing $\mu$ requires updating the corresponding $O(1)$ pointers, which takes a constant time.
- $insert\_vertex(v, (x, y))$: Update the corresponding pointers in the doubly linked list. If in result of the insertion the size of the microset, $\mu$, becomes larger than $\lambda(n)$, divide $\mu$ into two microsets of sizes at most $\lambda(n)/2 + 1$ so that at most one path does not belong to a single microset and compute the encodings of the resulting microsets. The time required for one division is $O(\lambda(n))$. The total time for all divisions is easily shown to be $O(n)$. Similarly divide a mezzoset if its size exceeds $ln$.
- *add_vertex*$(x, (y, z))$: The implementation of this operation is similar to $insert\_vertex$.
- $insert\_edge(v, w, e)$: Recall that for this operation $v$ and $w$ must belong to the same path $p$ determined by $e$. If $v$ and $w$ belong to the same microset, then just update the corresponding pointers in $O(1)$ time. If $v$ and $w$ belong to different microsets, then split $p$ at $v$ and $w$ by changing the pointers. Add the edge $(v, w)$ to the corresponding path in one of the microsets. If that microset becomes too large, divide it into two microsets. Recall that the doubly linked

lists of labeled microsets and mezzosets representing the two resulting paths are updated at the corresponding higher levels (levels of labeled microsets and mezzosets).

- *new_path*$(x, y)$: Choose an arbitrary microset $\mu$ and add the path consisting of the edge $e = (x, y)$ to $\mu$. This is done by adding in $\mu$ the *next* and *previous* pointers from $e$ to itself and storing with $e$ a pointer to its location in microset $\mu$.

We need also to explain how the above update operations affect the labels of mezzosets and microsets. Recall that a microset or a mezzoset is labeled if for some path it contains only internal edges of that path. Consider a *split* in a microset $\mu$. After the operation the labels of $\mu$ and the mezzoset containing $\mu$ are set to *nil*. After an *insert_vertex* in $\mu$, if $\mu$ has not been divided because of its larger size, then no label is changed. If the microset is divided, we have the following two cases: If the original microset has a non-nil label, then copy the same label to both resulting microsets. If the original microset has a *nil* label, label any of the resulting microsets that consists of edges only from one path $p$ but contains no endpoint of $p$ (if there is such a microset) with the label of $p$, or otherwise keep the *nil* label. A similar rule applies when a mezzoset is divided into two mezzosets because of a larger size. *Add_vertex* operation is similar to *insert_vertex*. Finally, for *insert_edge*, if the microset and mezzoset are not divided because of a larger size, we keep the original label (or *nil*). If any of them is divided, we assign labels to the resulting microsets using the rules described for *insert_vertex*.

We summarize the main result from this subsection in the following theorem.

THEOREM 1. *Any sequence consisting of at most $k$ insert_vertex, add_vertex, insert_edge, and new_path operations and at most $l$ $find$ and split operations on a set of edge-disjoint paths $\mathcal{P}$ can be implemented in $O(k + l + m)$ time and $O(k + m)$ space, where $m$ is the original number of edges in the paths of $\mathcal{P}$ .*

**3.2.5. Maintaining the *left* and *right* relations.** Now let us consider the original problem of maintaining the $left$ and $right$ relations in Algorithm *Triconnected*. Let us consider the $right$ relation for vertices (the $right$ relation for edges and the $left$ relations are maintained similarly). We maintain the set of paths $\mathcal{P}$ representing the original triconnected planar $st$-graph $G$. Specifically, for any face $f$, we keep in $\mathcal{P}$ the simple directed path $p$ of all edges $e$ such that $right(e) = f$. We associate with $p$ a variable containing the name of $f$. Denote by $q$ the other directed path on $f$. Let $x \in p$ and $y$ be two vertices on $f$. The addition of a new edge $e = (x, y)$ to the embedding can be implemented as a sequence of $O(1)$ of the above type operations on paths in $\mathcal{P}$ depending on the locations of $x$ and $y$ in $p$ and $q$ (see Figure 3.3). Consider the following cases. By $x'$ and $y'$ we denote two vertices adjacent respectively to $x$ and $y$ as denoted on Figure 3.3.

(a) $x \notin q$, $y \in q$, $y \notin p$ (Figure 3.3(a)). Then we add $e$ to the embedding by performing the following sequence of operations: $split(x, (x, x'))$, $add\_vertex(y, (x, x'))$.
(b) $x \notin q$, $y \in p$, $y \notin q$ (Figure 3.3(b)). Then $add\_edge(x, y)$.
(c) $x \in q, y \in p$, $y \notin q$ (Figure 3.3(c)). Then $split(y, (y, y'))$, $add\_vertex(x, (y, y'))$.
(d) $x \notin q$, $y \in p$, $y \in q$ (Figure 3.3(d)). Then $split(x, (x, x'))$, $add\_vertex(y, (x, x'))$.
(e) $x \in p, q$, $y \in p, q$ (Figure 3.3(e)). In this case we just do $new\_path(x, y)$.

In all cases (a)–(e) we update, if necessary, the variables associated with each of the resulting paths that contain the name of the adjacent face to the right of the path.

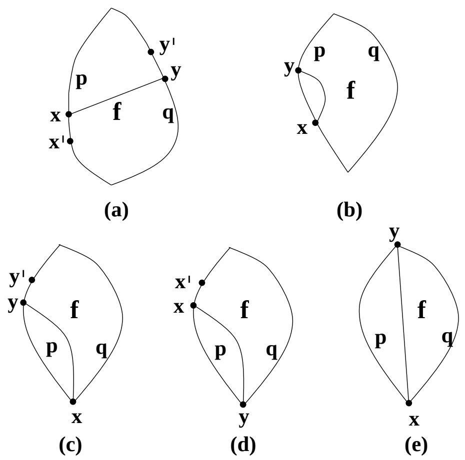


FIG. 3.3. *Cases for inserting edge* $(x, y)$ *in* $f$.

For implementing *right* queries we use the variable $redge(x)$ defined for any vertex $x$ (see Figure 3.1). Then $right(x)$ is the face associated with path $find(redge(x))$ and can be found in $O(1)$ time.

We showed that adding an edge requires $O(1)$ time and any $left$ or $right$ query requires $O(1)$ time. Since any planar $n$-vertex graph has $O(n)$ edges, the total number of edge additions will be $O(n)$. Clearly, the number of $find$ operations is $O(|E'|)$. Thus the total time for all operations connected with maintaining $left$ and $right$ relations for vertices will be $O(n + m)$.

Note that we did not use *insert_vertex* for maintaining $left$ and $right$ relations. We will use this operation in our algorithms given in the next sections.

This concludes our discussion of the triconnected case. We proved the following theorem.

THEOREM 2. *Let $G$ be a planar triconnected $n$-vertex graph and $E'$ be a set of $m$ edges between vertices of $G$. A maximal set $E'' \subset E'$ such that $G + E''$ is planar can be constructed in $O(n + m)$ time.*

## 4. Finding a maximal planar subgraph of an arbitrary graph.

**4.1. Outline of the algorithm.** Our algorithm uses the decomposition tree described in Section 2 to represent the decomposition of the current planar subgraph. Recall that we can assume w.l.o.g. that the input graph is connected, because otherwise we can apply the same algorithm to each connected component. For maintaining the embeddings of skeletons and for answering queries at each node of a SPQR tree we use a procedure similar to Algorithm *Triconnected*. At each iteration the algorithm chooses a new edge and checks if it is possible to add it to the subgraph so that planarity is preserved. The efficiency of our algorithm essentially depends on the order in which the edges are tested for insertion in the subgraph. Another feature of the algorithm is that it maintains a dynamic set *Upaths* of paths in the decomposition tree called *update paths* which will be our "working" paths, i.e. all information we currently could need will be associated with nodes in these paths and all updates will be done on nodes in paths from *Upaths*. By using properties of these paths we will be able to make queries and do updates more efficiently.

**Algorithm** *Maxplanar*
(Outline)

**Input:** A connected $n$-vertex $m$-edge graph $G$.
**Output:** A maximal planar subgraph $G'$ of $G$.

**1.** Construct a depth-first spanning tree $T$ of $G$. Associate a BC tree $B$ with $T$ whose root is the root of $T$. Let $E' = E(G) - E(T)$, $E^* = E(T)$, where $E'$ denotes the set of edges of $G$ not examined yet and $E^*$ denotes the set of edges of the current approximation of the maximal planar subgraph.

**2.** Initialize for the skeleton of each level-2 node of $B$ the data structures for on-line planarity testing from Steps 1, 2, and 3 of Algorithm *Triconnected*.

**3.** Use a variation of a postorder search to visit the nodes of $B$. Denote the current level-1 node by $\lambda^*$ and the current level-2 node by $\mu^*$.
{**Comment:** The postorder will guarantee that any level-2 node that either is a descendant of $\mu^*$ or belongs to a level-1 node which is a proper descendant of $\lambda^*$ is marked. (A level-2 node $\mu$ will be marked if no vertex of $skeleton(\mu)$ is incident to any edge from $E'$.)}

**3.1.** Update *Upaths* and the associated data structures if a new node has been examined in the previous step (details to be given below.)

**3.2.** Pick any vertex $x$ of the skeleton of $\mu^*$ belonging to the *SmallDeg* list of $\mu^*$.

**3.3.** Pick an edge $(x, y) \in E'$ and update $E' := E' - \{(x, y)\}$. If no vertex remains in $skeleton(\mu^*)$ that is incident to an edge of $E'$ then mark $\mu^*$. Denote by $\nu^*$ the proper allocation node $proper(y)$ of $y$. Check if $(x, y)$ can be added to $E^*$ by considering the following cases.

**3.3.1.** If $\nu^* = \mu^*$ then test whether $y$ belongs to any of the faces incident to $x$ by checking whether either $\mu^*$ is a P, S, or Q node, or the conditions of Lemma 3.2 for $v = x$ and $w = y$ are satisfied. If the answer is "yes", then add $(x, y)$ to $E^*$ and update the data structures associated with $\mu^*$.

**3.3.2.** If $\nu^*$ is a proper ancestor of $\mu^*$ then let edge $d$ be the projection $pr(x)$ of $x$ on the skeleton of $\nu^*$. By Lemma 2.2, we have to check if $y$ and $d$ belong to the same face of $skeleton(\nu^*)$ and if all nodes on the joining path $p(x, y)$ are peripheral. If $\nu^*$ is a P, S, or Q node or if Condition (3.5) of the proof of Lemma 3.1 is satisfied for $v = y$ and $e = d$, then $y$ and $d$ belong to the same face of the skeleton of $\nu^*$. Using information stored in *Upaths* we decide whether all nodes on $p(x, y)$ are peripheral. If both queries give positive answers, then we add $(x, y)$ to $E^*$ and update the data structures as will be described in the next subsection.

**3.3.3.** If $\nu^*$ belongs to a level-1 node $\kappa^*$ that is a proper ancestor of $\lambda^*$ then, by Lemma 2.3, we perform on the SPQR trees corresponding to $\kappa^*$ and $\lambda^*$ two test operations of the type described in Step 3.3.2 and (using precomputed information stored with *Upaths*) also check if all other level-1 nodes on the tree path $\pi^*$ between $\kappa^*$ and $\lambda^*$ are peripheral. If all answers are positive we add $(x, y)$ to $E^*$ and merge into the SPQR tree of $\kappa^*$ the SPQR trees of all other level-1 nodes of $\pi^*$.

Next we give more details about the implementation of some of the steps. The search of $B$ in Step 3 is essentially a postorder search applied at two levels: first with respect to level-1 nodes and then, when the level-1 node is chosen, with respect to the level-2 nodes in its SPQR tree. Another feature is that the search is applied on-line to dynamic trees. Thus, if in Steps 3.3.2 or 3.3.3 a path of two or more level-1 or level-2 nodes has been shrunk, then the resulting node, $\mu$, might have unmarked children (which must be visited now before continuing with $\mu$), even if $\mu^*$ may have had no marked children before the shrinking.

In the next subsection we describe the data structures associated with *Upaths* that will allow us to determine in a constant time whether certain path of level-1 or level-2 nodes contains only peripheral nodes. We will also show that it is possible to do all updates on $B$ and on the planar *st*-graphs associated with its nodes after shrinking of paths of level-1 and level-2 nodes in $O(n)$ total time.

**4.2. Updating the data structures.** Our update algorithms are simpler and more efficient than the algorithms of [6, 26, 24] because of our use of *Upaths*.

**4.2.1. The update paths.** *Upaths* is a set of paths called *update paths* that includes the current path $\pi$ of level-1 nodes of $B$ from $\lambda^*$ to the root of the tree and a path of level-2 nodes in the SPQR tree of each node in $\pi$. Let $\pi = \{b_1, c_1, b_2, c_2, \cdots, c_{k-1}, b_k\}$, where the path of level-2 nodes in node $b_1 = \lambda^*$ is called the *top update path* and $b_k$ is the root of $B$. For $\lambda^*$, the corresponding path of level-2 nodes is from $\mu^*$ to the root of the SPQR tree of $\lambda^*$. The update path in $b_i$, for $2 \le i \le k$, is the path from the allocation node of $c_{i-1}$ to the root of the SPQR tree for $b_i$. At any iteration we will use only $\pi$ and the top update path. The other paths in *Upaths* will be needed when we backtrack to an ancestor node during the postorder search. We prove the following important property of *Upaths*.

LEMMA 4.1. *Let $x$ be a vertex in $\mu^*$ and $(x, y) \in E'$ be the edge chosen in Step 3.3 of Algorithm* Maxplanar. *Then the proper allocation node proper$(y)$ of $y$ belongs to the update path of $\lambda^*$ or to the SPQR tree of a node in $\pi$ different from $\lambda^*$.*

**Proof:** Because the tree $T$ constructed in Step 1 is a depth-first spanning tree, then any non-tree edge joins two vertices of $T$ one of which is a descendant to the other. By the definition of a BC tree, such property will also hold for the tree $B$ constructed in Step 1 of Algorithm *Maxplanar*, i.e. any non-tree edge with respect to $T$ joins vertices from two level-1 nodes one of which is a descendant of the other. For any of the following iterations, we prove that the endpoints of any edge not in $B$ have allocation nodes such that either one of them is an ancestor of the other, or they belong to the SPQR trees of level-1 nodes one of which is a proper ancestor of the other. This follows by induction from the fact that each modification of $B$ consists of shrinking a path of either level-1 or level-2 nodes to a single node.

Assume that *proper*$(x)$ and *proper*$(y)$ belong to the same SPQR tree. Then, by the above observation, either $\mu^*$ is a descendant of *proper*$(y)$, or vice versa. However, since the nodes of $B$ are visited in postorder, all descendants of $\mu^*$ have already been visited and marked. Hence *proper*$(y)$ is an ancestor of $\mu^*$ and it belongs to the update path of $\lambda^*$ (the top update path).

The proof in the case when *proper*$(x)$ and *proper*$(y)$ belong to different SPQR trees is similar. In this case the node corresponding to the SPQR tree containing *proper*$(y)$ is a proper ancestor of $\lambda^*$ and by definition that node belongs to $\pi$. ∎

The update paths change during the computation when a new node of $B$ is visited and when a subpath of $B$ is contracted. This requires that some information associated with *Upaths* be dynamically updated, as described below.

**4.2.2. Update algorithms.** We need to dynamically maintain the following types of information:

- The proper allocation nodes of the vertices of $G$;
- For any level-1 or level-2 node $\mu$, the nearest ancestor of $\mu$ that is not peripheral;
- A planar embedding of a triconnected planar $st$-graph $G'$ (a skeleton of a level-2 R node $\nu$) with respect to the structures described in Section 3 subject to the operation replacement of an edge of $G'$ with a planar $st$-graph (the skeleton of a child of $\nu$.)

*1. Proper allocation nodes.* Information about the proper allocation nodes of the vertices of $G$ is needed in Step 3.3 of Algorithm *Maxplanar*. We store for each vertex $v$ of $G$ a pointer to the representative of $v$ in the skeleton of $proper(v)$. Furthermore, we dynamically maintain for each level-2 node $\mu$ belonging to a path in *Upaths* a set of all vertices $x$ of $G$ such that $proper(x) = \mu$. This can be done by using the linear-time incremental set union algorithm of Gabow and Tarjan [12]. Hence one can find the proper allocation node $\mu$ of any vertex of $G$ in $O(1)$ amortized time, provided that $\mu$ belongs to a path of *Upaths*. By the description of Algorithm *Maxplanar* and Lemma 4.1, the proper allocation nodes of the vertices $x$ and $y$ examined in Step 3.3 will always belong to a path of *Upaths*.

*2. Non-peripheral ancestors.* For any level-2 node $\nu$ of a path from *Upaths* we store in a variable $nearest(\nu)$ the value of the nearest non-peripheral ancestor of $\nu$. Whenever a new level-2 node $\mu$ is examined in Step 3 of Algorithm *Maxplanar*, we add $\mu$ to the top update path and check if $\mu$ is peripheral. Depending on the value of *nearest* for the parent of $\mu$ (if $\mu$ is not the root of $B$,) the value of $nearest(\mu)$ is determined. Note that when a subpath of level-2 nodes is contracted, that subpath always includes the endvertex (i.e., the most recently added vertex) of the top update path. Thus in this case at most one value of *nearest* needs to be updated which takes $O(1)$ time. The information about the nearest non-peripheral ancestors of level-1 nodes is maintained in a similar way.

*3. Merging skeletons.* Merging the skeleton of a level-2 node $\mu$ of $B$ with the skeleton of its parent $parent(\mu)$ requires a replacement of an edge $e$ of a planar $st$-graph $G_e$ (the skeleton of $parent(\mu)$) with another $st$-graph (the skeleton of $\mu$.)

Consider the case where both $\mu$ and $parent(\mu)$ are $R$ nodes (the non-trivial case) and $e$ is an internal edge. The set of the faces of the resulting planar $st$-graph is the union of all internal faces of the skeleton of $parent(\mu)$ and all faces of the skeleton of $\mu$, where the two faces, say $f_1$ and $f_2$, incident to $e$, are modified as follows. In each of $f_1$ and $f_2$ we replace $e$ by a path using a sequence of *insert_vertex* operations. Let $p_1$ and $p_2$ be the resulting paths (Figure 4.1). The time needed for these *insert_vertex* operations will be proportional to the sum of the lengths of $p_1$ and $p_2$. However, for the whole execution of the algorithm, the time needed for such type of insertions will be $O(n)$, since any edge of $p_1$ or $p_2$ becomes internal and can not be inserted again.

If $e$ is on the periphery of $G_e$ we apply the same algorithm. But now the above argument for bounding the number of edges that need to be inserted does not apply because same edges may need to be re-inserted more than once. To handle this case we modify our data structure for maintaining the planar embeddings of the skeletons of level-2 nodes by using the same label *outer* to denote the outer face of **any** skeleton

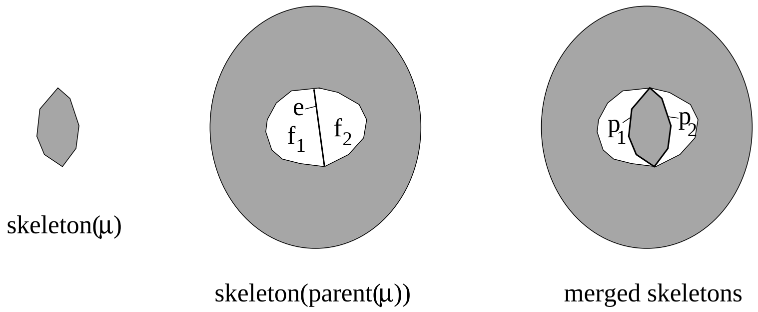


FIG. 4.1. *Merging a skeleton of a level-2 node into the skeleton of its parent.*

of a level-2 node. For example, if $x$ is a vertex on the left boundary path of such a skeleton $\mu$ incident to an internal face $f$, then $right(x) = f$ and $left(x) = outer$. Thus, the value of $left(x)$ does not need to be changed if $x$ becomes a boundary vertex of another skeleton as a result of a merge.

Testing if the addition of an edge $(x, y)$ preserves planarity requires to check whether $left(x) = left(y) = outer$ and whether $x$ and $y$ have the same proper allocation node.

If $x$ becomes an internal edge of a skeleton after some merge, then we set $left(x)$ to the (actual) internal face containing $x$ and maintain its value using the original algorithm.

We can summarize our main result as follows.

THEOREM 3. *Given any $n$-vertex $m$-edge graph $G$, a maximal planar subgraph of $G$ can be found in $O(n + m)$ time.*

**5. Conclusion.** We can also adapt our technique to find a maximal *outerplanar* subgraph of an $n$-vertex graph. Create an additional vertex $z$ and join $z$ to all vertices of $G$. Then find a maximal planar subgraph of the resulting graph by a similar procedure as Algorithm *Maxplanar*, however the initial tree constructed in Step 1 is the star spanning graph with root $z$. This guarantees that the maximal planar graph constructed by the modified algorithm will contain all edges incident to $z$. Removing at the end $z$ and all incident edges clearly will result in a maximal outerplanar graph. We need to show that the time complexity of this algorithm is still $O(n+m)$, since our initial subgraph is not a depth-first tree as in Algorithm *Maxplanar* and the analysis of the new algorithm (e.g., Lemma 4.1) can be not directly applied. In this case, however, we do not need to use the update paths since each level-1 or level-2 node can have at most one ancestor and condition (ii) of Lemma 2.2 and the condition from Lemma 2.3 can be directly checked in a constant time. Thus we have the following theorem.

THEOREM 4. *Given any $n$-vertex $m$-edge graph $G$, a maximal outerplanar subgraph of $G$ can be found in $O(n + m)$ time.*

Also we note that our linear-time algorithm for the MPS problem yields a linear-time algorithm for planarity testing. Given an $n$-vertex $m$-edge graph $G$ we can test in $O(n + m)$ time whether $G$ is planar by finding a maximal planar subgraph $G'$ of $G$. Then $G$ is planar iff $G = G'$. This result is interesting because the new algorithm is based on an approach entirely different from the existing ones. The linear-time algorithms of Hopcroft and Tarjan [16] and Booth and Lueker [2] (and their modifications) essentially use the Jordan Curve Theorem which states that any closed curve in the plane divides it into exactly two connected regions. In contrast, our algorithm is based on the uniqueness of the planar embedding of any triconnected planar graph. It will be of theoretical and practical interest to refine our technique in order to construct a new practical algorithm for planarity testing whose performance is comparable to the algorithms of [16] and [2].

As another approach to the graph planarization problem, other researchers have constructed approximation algorithms for the maximum planar subgraph problem. The algorithm in [3] constructs in $O(m^{3/2} n \log^6 n)$ time a "maximum triangular structure," a planar graph whose bicomponents are single edges or triangles, and prove approximation ratio $2/5$. Although the approximation ratio corresponding to a maximal planar subgraph in the worst case can not be proved to be better than $1/3$, it seems that in most cases our algorithm produces larger subgraphs, e.g., for planar or almost planar graphs, for sparse graphs (e.g., with less than $(5/2)n - 5$ edges), for any bipartite graphs. For practical purposes, probably the best algorithm for constructing large planar subgraphs will be a combination of both approaches: first a planar subgraph is constructed by the approximation algorithm guaranteeing a good approximation ratio, and then the subgraph is augmented to a maximal planar subgraph.

**Acknowledgment.** The author would like to thank anonymous referees for their helpful comments.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
[2] K. Booth, G. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithm*, J. Comp. Syst. Sci. 13, 1976, pp. 335-379.
[3] G. Călinescu, C. G. Fernandes, U. Finkler, and H. Karloff, *A better approximation algorithm for finding planar subgraphs*, In Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 16-25, 1996.
[4] J. Cai, X. Han, and R.E. Tarjan, An O(m log n)-time algorithm for the maximal planar subgraph, *SIAM Journal on Computing*, 22 (1993), pp. 1142-1162, .
[5] T. Chiba, I. Nishioka, and I. Shirakawa, An algorithm of maximal planarization of graphs, *Proc. IEEE Int. Symp. on Circuits and Systems,* 1978, pp. 649-652.
[6] G. Di Battista, R. Tamassia, Incremental planarity testing, *Proc. IEEE Symp. on Found. of Comp. Sci.,* (1989), pp. 436-441.
[7] G. Di Battista, R. Tamassia, On-line graph algorithms with SPQR trees, *Proc. Intern. Colloquium on Automata, Languages and Programming* (1990), 598-611.
[8] H. Djidjev, A Linear Algorithm for the Maximal Planar Subgraph Problem, Proceedings of WADS'95, *Lecture Notes in Computer Science*, vol. 955, pp. 369-380, Springer-Verlag, August, 1995.
[9] H. N. Djidjev, On some properties of nonplanar graphs, *Compt. rend. Acad. bulg. Sci.*, vol. 37 (1984), 9, pp. 1183-1185.
[10] H.N. Djidjev, J. Reif, An efficient algorithm for the genus problem with explicit construction of forbidden subgraphs, *Proc. Annual ACM Symposium on Theory of Computing* (1991), pp.337-347.
[11] S. Even and R.E. Tarjan, Computing an *st*-numbering, Theoretical Computer Science 2 (1976), pp. 339-344.
[12] H. Gabow and R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *J. Compu. Syst. Sci.*, 30 (1985), pp. 209-220.
[13] M.R. Garey and D.S. Johnson, *Algorithms and Intractability: A Guide to the Theory of NP Completeness.* San Francisco, Freeman, 1979.
[14] F. Harary. *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.
[15] J. Hopcroft and R.E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput. 2, 1973, pp. 135-158.
[16] J. Hopcroft and R.E. Tarjan, *Efficient planarity testing*, J.ACM, 21:4, 1974, pp. 549-568.
[17] W.-L. Hsu, *A Linear Time Algorithm for Finding Maximal Planar Subgraphs*, ISAAC'95, Lecture Notes in Computer Science, Vol. 1004, pp. 352–362, December, 1995.
[18] H. Imai and T. Asano, Dynamic orthogonal segment intersection search, in *Journal of Algorithms* 8, 1987, pp. 1-18.
[19] R. Jayakumar, K. Thulasiraman, and M.N.S. Swamy, $O(n^2)$ algorithms for graph planarization, *IEEE Trans. on Comp.-Aided Design* 8 (1989), pp. 257-267.
[20] M. Marek-Sadowska, Planarization algorithm for integrated circuits engineering, *Proc. IEEE Int. Symp. on Circuits and Systems,* 1979, pp. 919-923.
[21] T. Nishizeki, N. Chiba, *Planar Graphs: Theory and Algorithms*, North Holland, 1988.
[22] T. Ozawa and H. Takahashi, A graph-planarization algorithm and its applications to random graphs, in *Graph Theory and Algorithms*, Lecture Notes in Computer Science, vol. 108, 95-107, Springer-Verlag, 1981.
[23] K. Pasedach, Criterion and algorithms for determination of bipartite graphs and their application to planarization of graphs, in *Graphen-Sprachen und Algorithmen in Graphen*, 175-183, Carl Hanser Verlag, Germany, 1976.
[24] J.A. La Poutré, Alpha-algorithms for incremental planarity testing, *Proc. of the Ann. ACM Symp. on Theory of Comput.*, 1994, pp. 706-715.

[25] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. ACM*, 22 (1975), pp. 215-225.

[26] J. Westbrook, Fast incremental planarity testing, *Proc. Int. Col. on Automata, Languages, and Programming*, 1992, pp. 342-353.