SIAM J. COMPUT.
Vol. 25, No. 5, pp. 956–997, October 1996

003

ON-LINE PLANARITY TESTING*

GIUSEPPE DI BATTISTA† AND ROBERTO TAMASSIA‡

Abstract. The *on-line planarity-testing* problem consists of performing the following operations on a planar graph G: (i) testing if a new edge can be added to G so that the resulting graph is itself planar; (ii) adding vertices and edges such that planarity is preserved. An efficient technique for on-line planarity testing of a graph is presented that uses $O(n)$ space and supports tests and insertions of vertices and edges in $O(\log n)$ time, where n is the current number of vertices of G. The bounds for tests and vertex insertions are worst-case and the bound for edge insertions is amortized. We also present other applications of this technique to dynamic algorithms for planar graphs.

Key words. planar graph, on-line algorithm, dynamic algorithm

AMS subject classifications. 68R10, 05C10, 68Q20, 68P05

1. Introduction.

The problems of testing planarity and constructing planar embeddings of graphs have been extensively studied in the past years and find direct application in a variety of areas including circuit layout, graphics, computer-aided design, and automatic graph drawing.

In a static environment, where an n-vertex graph G is entirely known in advance, we can test the planarity of G and compute a planar embedding in optimal $O(n)$ time [5, 8, 18, 20, 31, 38]. In a dynamic environment, where a planar graph G is assembled on-line by insertions of vertices and edges, we would like to determine quickly whether an update causes G to become nonplanar. Namely, the *on-line planarity-testing* problem consists of performing the following operations on a planar graph G: (i) testing if a new edge can be added to G so that the resulting graph is itself planar; (ii) adding vertices and edges such that planarity is preserved.

While many research efforts have been focused on planar graphs and on dynamic graph algorithms, the development of an efficient algorithm for on-line planarity testing has been an elusive goal.

Recent results on planar graphs include algorithms for parallel planarity testing [37, 46], embedding [4, 62], drawing [10, 13, 19, 51], reachability [36, 54, 57], shortest paths [22], and minimum spanning trees [15, 21]. Previous work on dynamic graph algorithms is surveyed in §2. The technique of [53] is a first step toward on-line planarity testing. Namely, it solves the restricted problem of maintaining a planar embedding of a planar graph. It uses $O(n)$ space and supports queries (testing whether two vertices are on the same face of the embedding) and updates (adding vertices and edges to the embedding) in $O(\log n)$ time.

* Received by the editors November 2, 1994; accepted for publication (in revised form) January 24, 1995. This paper includes results presented at the 30*th IEEE Symposium on Foundations of Computer Science* (1989) and the 17*th International Colloquium on Automata, Languages, and Programming* (1990). This research was supported in part by the ESPRIT II Basic Research Actions Program of the EC under contract 3075 (project ALCOM), National Science Foundation grants CCR-9007851 and CCR-9423847, Office of Naval Research/Defense Advanced Research Projects Agency contract N00014-91-J-4052, ARPA order 8225, the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council, and U. S. Army Research Office grant DAAL03-91-G-0035. This research performed in part while G. Di Battista was with the University of Rome "La Sapienza" and with the Universitá della Basiliata, while G. Di Battista was visiting Brown University, and while R. Tamassia was visiting the University of Rome "La Sapienza."

† Dipartimento di Discipline Scientifiche, Sezione Informatica, Terza Universitá degli Studi di Roma, 84 Via della Vasca Navale, Rome 00146, Italy (dibattista@iasi.rm.cnr.it).

‡ Department of Computer Science, Brown University, Providence, RI 02912-1910 (rt@cs.brown.edu).

In this paper, a technique for on-line planarity testing is presented that uses $O(n)$ space and supports tests and updates in $O(\log n)$ time, n being the current number of vertices of the graph. The bounds for tests and vertex insertions are worst-case and the bound for edge insertions is amortized.

The following repertoire of query and update operations is defined for a planar graph G:

Test(v_1, v_2): Determine whether edge (v_1, v_2) can be added to G while preserving planarity, i.e., test whether graph G admits a planar embedding Γ such that v_1 and v_2 are on the boundary of the same face of Γ.

InsertEdge(e, v_1, v_2): Add edge e between vertices v_1 and v_2 to graph G. The operation is allowed only if the resulting graph is itself planar.

InsertVertex(e, v, e_1, e_2): Split edge e into two edges e_1 and e_2 by inserting vertex v.

AttachVertex(e, v, u): Add vertex v and connect it to vertex u by means of edge e.

MakeVertex(v): Add an isolated vertex v.

Our main result is expressed by the following theorem.

THEOREM 1.1. *Let G be a planar graph that is dynamically updated by adding vertices and edges, and let n be the current number of vertices of G. There exists a data structure for the on-line planarity-testing problem in G with the following performance: the space requirement is $O(n)$; operation MakeVertex takes worst-case time $O(1)$; operations Test, AttachVertex, and InsertVertex take worst-case time $O(\log n)$; and operation InsertEdge takes amortized time $O(\log n)$.*

The techniques developed in our work provide new insights on the topological properties of planar *st*-graphs and on the relationship between planarity and the decomposition of a graph into its biconnected and triconnected components.

The rest of this paper is organized as follows. In §2, we survey previous results on dynamic graph algorithms. Section 3 provides basic definitions. In §4, we present a static data structure that supports only operation *Test* in biconnected graphs. Sections 5 and 6 describe the dynamic data structure for on-line planarity testing in biconnected graphs. The data structure is extended to general planar graphs in §7. Finally, some applications of our technique to graph planarization, on-line transitive closure, and on-line minimum spanning trees are given in §8.

2. Dynamic graph algorithms. The development of dynamic algorithms for graph problems has acquired increasing theoretical interest, motivated by many important applications in network optimization, very large-scale integration (VLSI) layout, computational geometry, and distributed computing. In this section, we survey representative dynamic graph algorithms for reachability, shortest paths, minimum spanning trees, and connectivity. Throughout this section, n and m, respectively, denote the number of vertices and edges of the graph being considered. A general lower-bound technique for incremental algorithms, with applications to dynamic graph algorithms, is discussed in [3].

A reachability query in a digraph asks whether there is a directed path between two vertices. For general digraphs, there exist insertions-only semidynamic data structures with $O(n^2)$ space, $O(1)$ query time, and $O(n)$ amortized update time [6, 32, 43]. The same performance is achieved for deletions only in acyclic digraphs [6, 33]. Fully dynamic data structures with $O(n)$ space and $O(\log n)$ query and update time exist for some classes of planar digraphs [11, 34, 54, 56]. The related problem of maintaining a topological ordering of an acyclic digraph is studied in [1].

A shortest-path query in a digraph asks for the length of a shortest path between two vertices. Fully dynamic data structures for shortest-path queries are presented in [16, 48]. They have $O(n^2)$ space, $O(1)$ query time, $O(n^2)$ time for edge insertion, and $O(mn+n^2 \log n)$ time for edge deletion. The best-known semidynamic data structures supporting insertions in digraphs with unit edge lengths use $O(n^2)$ space and have constant query time; the total time to process all edge insertions is $O(n^3 \log n)$, which amortizes to $O(n \log n)$ time per insertion for dense graphs [2, 39]. For series-parallel digraphs with weighted edges, there exists a fully dynamic $O(n)$-space data structure that supports queries and updates in $O(\log n)$ time [9]. This data structure also maintains a maximum flow within the same time bounds.

The dynamic maintenance of minimum spanning trees has the interesting property that, after an update operation consisting of a weight change or adding/deleting an edge, at most one edge needs to be replaced in the minimum spanning tree. For general graphs, the best result is $O(\sqrt{m}\,)$ update time and $O(m)$ space [21]. In the special case of planar graphs, updates can be done in $O(\log n)$ time using an $O(n)$-space fully dynamic data structure [11, 15].

Regarding connectivity problems, a classical result shows that the connected components of a graph can be efficiently maintained in a semidynamic environment where only edge-insertions are performed, by means of a union-find data structure [58]. A sequence of k queries and edge insertions takes time $O(k\alpha(k, n))$, where $\alpha(k, n)$ denotes the slowly growing inverse of Ackermann's function. The same performance is obtained for biconnected components [42, 60], triconnected components [11, 42], and four-connected components [35]. Semidynamic techniques supporting deletions only are studied in [17, 47]. In a fully dynamic environment, the connected components of a general graph can be maintained in time $O(\sqrt{m}\,)$ per update operation [21], while the biconnected components of a planar graph can be maintained in $O(n^{2/3})$ time using $O(n)$ space [25]. The related problems of maintaining the two- and three-edge-connected components are studied in [23, 24, 60].

3. Preliminaries. We assume that the reader is familiar with graph terminology and basic properties of planar graphs (see, e.g., [40]). Throughout this paper, n denotes the number of vertices of the planar graph G currently being considered. Unless otherwise specified, we only consider graphs without self-loops and multiple edges. Recall that a planar graph without self-loops and multiple edges has $O(n)$ edges.

First, we review some definitions on graph connectivity. A separating k-set of a graph G is a set of k vertices whose removal increases the number of connected components of G. Separating 1-sets and 2-sets are called *cutvertices* and *separation pairs*, respectively. A connected graph is said to be *biconnected* if it has no cutvertices. The *blocks* of a connected graph (also called *biconnected components*) are its maximal biconnected subgraphs. A graph is *triconnected* if it is biconnected and has no separation pairs.

A *planar drawing* of a graph is such that no two edges intersect (except possibly at the endpoints). A graph is *planar* if it admits a planar drawing. A planar drawing partitions the plane into topologically connected regions, called *faces*. The unbounded face is called the *external face*. The *boundary* of a face is its delimiting circuit. All the face boundaries of a biconnected graph are simple circuits. For brevity, we sometimes use "face" to mean "face boundary."

The *incidence list* of a vertex v is the set of edges incident upon v. A planar drawing determines a circular ordering on the incidence list of each vertex v according

to the clockwise sequence of the incident edges around v.

Two planar drawings of the same connected graph G are *equivalent* if they determine the same circular orderings of the incidence lists. Two equivalent planar drawings have the same face boundaries. A *planar embedding* or simply *embedding* Γ of G is an equivalence class of planar drawings and is described by circularly sorted incidence lists for each vertex v. The face boundaries of any drawing of Γ are called the faces of Γ. A triconnected planar graph has a unique embedding, up to reversing all the incidence lists.

A *planar st-graph* G is a planar acyclic digraph with exactly one source (vertex without incoming edges) s and exactly one sink (vertex without outgoing edges) t which admits a planar embedding such that s and t are on the same face. Such graphs were first introduced in [38]. Vertices s and t are called the *poles* of G. Henceforth, we shall only consider embeddings of a planar st-graph such that s and t are on the same face. Following the developments of [10, 13], we visualize a planar st-graph with s and t on the external face and all the edges directed upward; see Fig. 1.

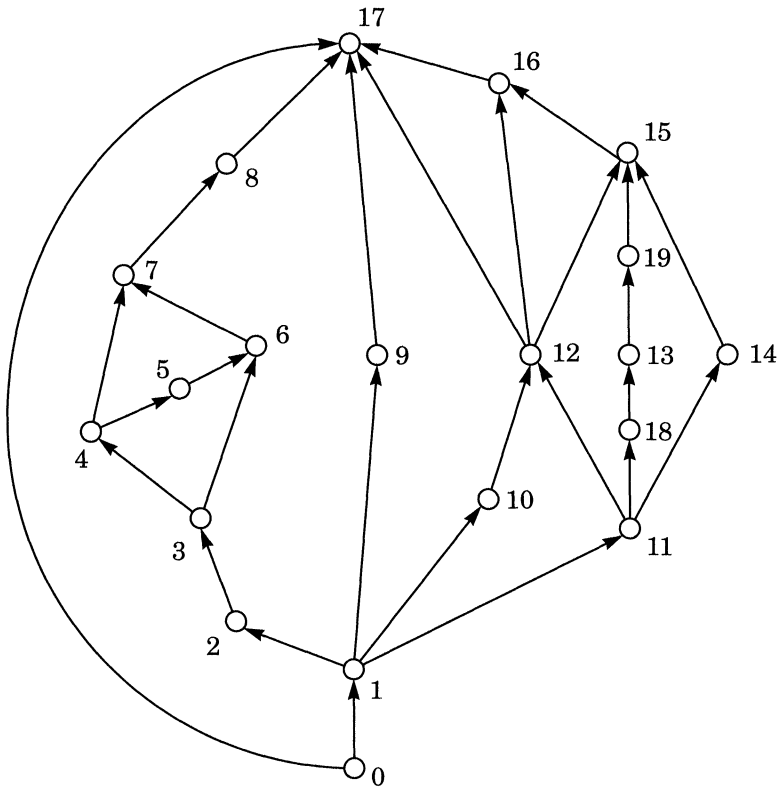

FIG. 1. *Example of a planar st-graph.*

The following properties are demonstrated in [55].

LEMMA 3.1. *Let* Γ *be a planar embedding of a planar st-graph* G.

1. *The incoming edges of each vertex* v *of* G *appear consecutively in the incidence list of* v *sorted according to* Γ, *and so do the outgoing edges.*

2. *Each face of* Γ *consists of the concatenation of two directed paths.*

With reference to the second property, the origin and destination of the paths forming a face f are called the *extreme vertices* of f. The other vertices of f are

called *internal vertices* of f. For example, the planar st-graph of Fig. 1 has a face with extreme vertices 11 and 15 and with internal vertices 12, 18, 13, and 19.

A *planar st-orientation* of an undirected graph G is an orientation of the edges of G such that the resulting directed graph is a planar st-graph. A graph G admits a planar st-orientation if and only if it is *planarly st-biconnectible* [38], i.e., the graph obtained from G by adding the edge (s,t) is planar and biconnected. A planar st-orientation can be computed in $O(n)$ time [18].

4. Tests. In this section, we consider the problem of performing operation *Test* on a biconnected planar graph with n vertices. We assume that the graph has been oriented into a planar st-graph such that s and t are adjacent.

4.1. Decomposition tree. Let G be a planar st-graph. A *split pair* of G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{u,v\}$ is either an edge (u,v) or a maximal subgraph C of G such that C is an uv-graph and $\{u,v\}$ is not a split pair of C. A *maximal split pair* $\{u,v\}$ of G is such that there is no other split pair $\{u',v'\}$ in G such that $\{u,v\}$ is contained in a split component of $\{u',v'\}$.

For example, in the planar st-graph G of Fig. 1, the pair $\{3,7\}$ is a split pair but not a maximal split pair, while $\{0,17\}$ is the only maximal split pair of G. The split components of the split pair $\{1,17\}$ are shown in Fig. 2.

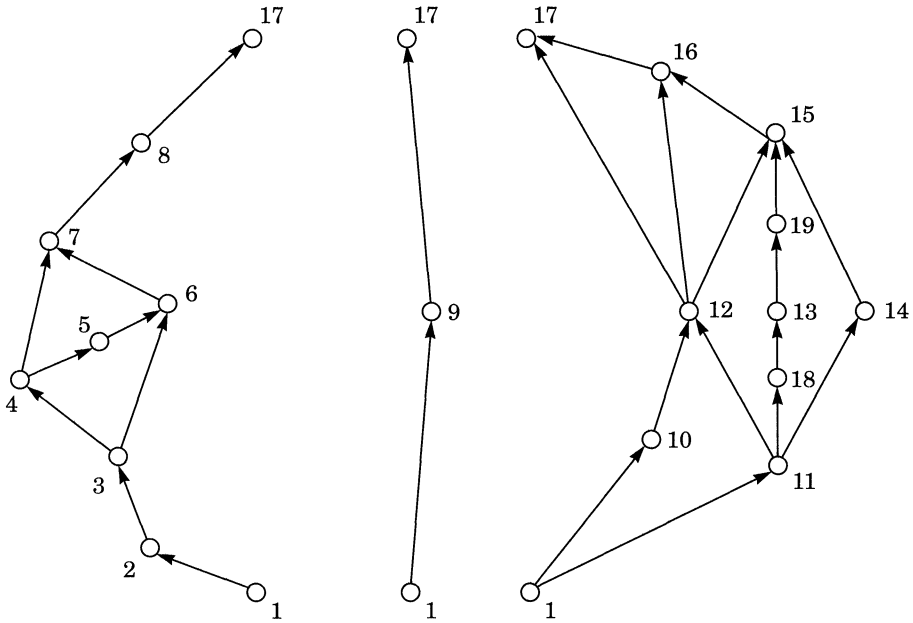

FIG. 2. *Split components of the split pair* $\{1,17\}$ *in the planar st-graph of Fig.* 1.

The *decomposition tree* $\mathcal{T}$ of G describes a recursive decomposition of G with respect to its split pairs and will be used to synthetically represent all the embeddings of G with vertices s and t on the external face. Tree $\mathcal{T}$ is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of $\mathcal{T}$ has an associated planar st-graph (possibly with multiple edges), called the *skeleton* of μ and denoted by *skeleton*(μ). Also, it is associated with an edge of the skeleton of the parent ν of μ, called the *virtual edge* of μ in *skeleton*(ν). Tree $\mathcal{T}$ is recursively defined as follows.

Trivial case. If G consists of a single edge from s to t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.

Series case. If G is not biconnected, let $c_1, \ldots, c_{k-1}$ $(k \geq 2)$ be the cutvertices of G. Since G is planarly st-biconnectible, each cutvertex c_i is contained in exactly two blocks G_i and G_{i+1} such that s is in G_1 and t is in G_k. The root of $\mathcal{T}$ is an S-node μ. Graph *skeleton*(μ) consists of the chain $e_1, \ldots, e_k$, where edge e_i goes from c_{i-1} to c_i, $c_0 = s$, and $c_k = t$, plus the edge (s, t). (See Fig. 3(a))

Parallel case. If s and t are a split pair for G with split components $G_1, \ldots, G_k$ $(k \geq 2)$, the root of $\mathcal{T}$ is a P-node μ. Graph *skeleton*(μ) consists of $k+1$ parallel edges from s to t, denoted $e_1, \ldots, e_{k+1}$. (See Fig. 3(b))

Rigid case. If none of the above cases applies, let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G $(k \geq 1)$, and for $i = 1, \ldots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$. The root of $\mathcal{T}$ is an R-node μ. Graph *skeleton*(μ) is obtained from G by replacing each subgraph G_i with the edge e_i from s_i to t_i and by adding the edge (s, t). Notice that the skeleton of an R-node is triconnected. (See Fig. 3(c))

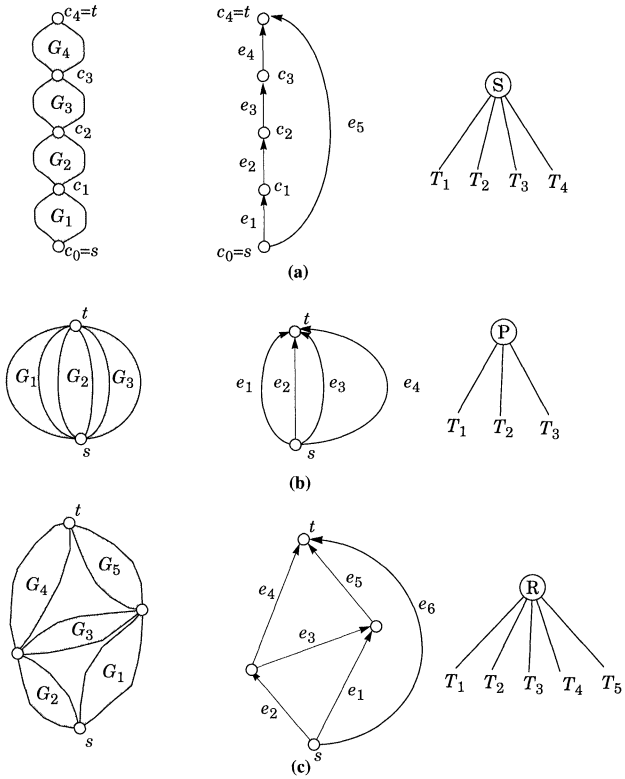

FIG. 3. (a) *Series decomposition.* (b) *Parallel decomposition.* (c) *Rigid decomposition.*

In the last three cases (series, parallel, and rigid), μ has children $\mu_1, \ldots, \mu_k$ (in this order), such that μ_i is the root of the decomposition tree of graph G_i $(i = 1, \ldots, k)$. The virtual edge of node μ_i is edge e_i of *skeleton*(μ). Graph G_i is called the *pertinent graph* of node μ_i, and the *expansion graph* of e_i. (Note that G is the pertinent graph of the root.) We denote with s_μ and t_μ the poles of the skeleton of a node μ. We find it convenient (e.g., in Theorem 4.6 below) to define the expansion graph of a vertex

of *skeleton*(μ) as the vertex itself.

Figure 4 illustrates the decomposition tree and the skeletons of the R-nodes for the planar *st*-graph of Fig. 1. Our definition of decomposition tree is a variation of the one given in [4] and is closely related to the decomposition of biconnected graphs into triconnected components [30].

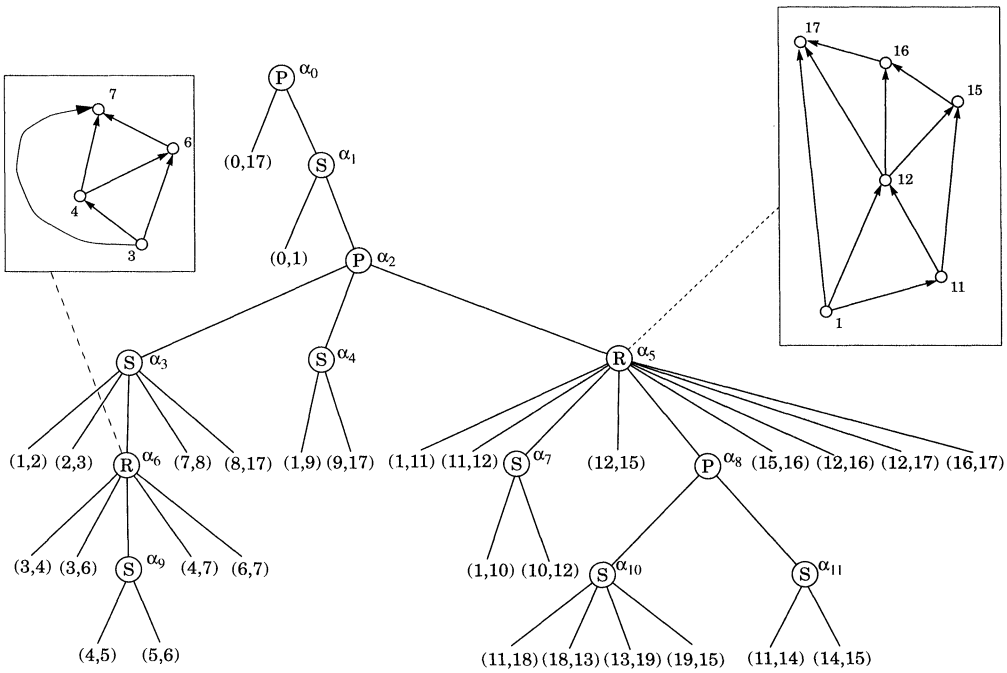

FIG. 4. *Decomposition tree $\mathcal{T}$ for the planar st-graph of Fig. 1 and skeletons of the R-nodes.*

LEMMA 4.1. *The decomposition tree $\mathcal{T}$ of G has $O(n)$ nodes. Also, the total number of edges of the skeletons stored at the nodes of $\mathcal{T}$ is $O(n)$.*

Proof. The leaves of $\mathcal{T}$ are Q-nodes in one-to-one correspondence with the edges of G, and each internal node of $\mathcal{T}$ has at least two children. Hence $\mathcal{T}$ has $O(n)$ nodes. If a node μ of $\mathcal{T}$ has k children, then *skeleton*(μ) has at most $k+1$ edges (one edge for a Q-node, k edges for an S- or P-node, and $k+1$ edges for an R-node). Hence the total number of edges of the skeletons is at most the sum of the number of nodes and edges of $\mathcal{T}$ and thus is $O(n)$. □

Now, we show how the decomposition tree can be used to represent all the planar embeddings of a planar *st*-graph G with the edge (s,t). Let Γ be a planar embedding of G.

Two basic primitives can be used to obtain a new planar embedding from Γ. A *reverse* operation consists of flipping a split component around its poles. A *swap* operation consists of exchanging the position of two split components of the same split pair. For example, Fig. 5 shows the planar embedding obtained from that of Fig. 1 by means of two swap operations and one flip operation.

LEMMA 4.2 (see [12]). *Given a pair of embeddings Γ' and Γ'' of a planar st-graph G with the edge (s,t), Γ'' can be obtained from Γ' by means of a sequence of $O(n)$ reverse and swap operations.*

By Lemma 4.2, the decomposition tree $\mathcal{T}$ can be used to represent an embedding

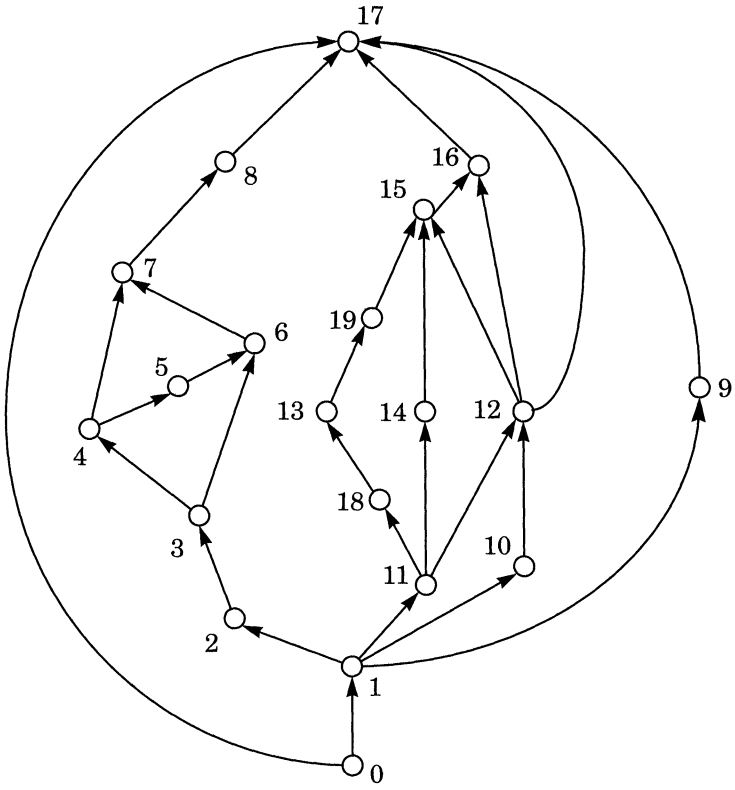

FIG. 5. *Embedding obtained from the one of Fig.* 1 *by means of two swap operations around the split pairs* $(1,17)$ *and* $(11,15)$ *and one flip operation around the split pair* $(1,17)$.

of G by

1. selecting one of the two possible flips of the skeleton of each R-node around its poles; and
2. selecting a permutation of the skeletons of the children of each P-node with respect to their common poles.

Before presenting the algorithm for operation *Test*, we need to introduce additional concepts.

Let v be a vertex of G. The *allocation nodes* of v are the nodes of $\mathcal{T}$ whose skeleton contains v. Note that v has at least one allocation node. For example, with reference to the planar st-graph of Fig. 1 and its decomposition tree shown in Fig. 4, we have that the allocation nodes of vertex 15 are the Q-nodes associated with its incident edges plus nodes α_5, α_8, α_{10}, and α_{11}.

The following facts can be easily proved.

FACT 1. *The pertinent graphs of the children of a node* μ *can only share vertices of skeleton*(μ).

FACT 2. *If* v *is in skeleton*(μ), *then* v *is also in the pertinent graph of all the ancestors of* μ.

FACT 3. *If* v *is a pole of skeleton*(μ), *then* v *is also in the skeleton of the parent of* μ.

FACT 4. *If* v *is in skeleton*(μ) *but is not a pole of skeleton*(μ), *then* v *is not in the skeleton of any ancestor of* μ.

LEMMA 4.3. *Let v be a vertex of G. The least common ancestor μ of the allocation nodes of v is itself an allocation node of v. Also, if $v \neq s, t$, then μ is the only allocation node of v such that v is not a pole of skeleton(μ).*

Proof. For the first part of the lemma, it is sufficient to show that the least common ancestor μ of two allocation nodes μ_1 and μ_2 of v is itself an allocation node of v. This is trivial if one of μ_1 and μ_2 is an ancestor of the other. Otherwise, by Fact 2, vertex v is in the pertinent graphs of the children of μ which are ancestors of μ_1 and μ_2. Hence, by Fact 1, vertex v must be in *skeleton*(μ). The second part of the lemma follows from Facts 3 and 4. □

According to Lemma 4.3, the least common ancestor of the allocation nodes of vertex v is called the *proper* allocation node of v. For example, in Figs. 1 and 4, the proper allocation nodes of vertices 1, 6, and 15 are α_1, α_6, and α_5, respectively.

FACT 5. *If $v \neq s, t$, then the proper allocation node of v is either an R-node or an S-node.*

Let v be a vertex of the pertinent graph of a node μ of $\mathcal{T}$. The *representative* of v in *skeleton*(μ) is the vertex or edge x of *skeleton*(μ) defined as follows: if μ is an allocation node of v, then $x = v$; otherwise, x is the edge of *skeleton*(μ) whose expansion graph contains v. For example, in Figs. 1 and 4, edge $(11, 15)$ of *skeleton*(α_5) is the representative of vertices 13, 14, 18, and 19, while vertex 7 in *skeleton*(α_3) is the representative of itself.

FACT 6. *The nodes of $\mathcal{T}$ whose skeleton has a representative for vertex v are the allocation nodes of v (the representative is a vertex) and their ancestors (the representative is an edge).*

LEMMA 4.4. *Given any two distinct vertices v_1 and v_2 of G, there exists a node χ of $\mathcal{T}$ such that v_1 and v_2 have distinct representatives in skeleton(χ). Also, let μ_1 and μ_2 be the proper allocation nodes of v_1 and v_2, respectively, and let μ be the least common ancestor of μ_1 and μ_2.*

1. *If $\mu_1 = \mu_2 = \mu$, then the common allocation nodes of v_1 and v_2 are exactly those with distinct representatives for v_1 and v_2.*
2. *If $\mu_1 \neq \mu$ and $\mu_2 \neq \mu$, then μ is the only node with distinct representatives for v_1 and v_2.*
3. *If μ_1 is an ancestor of μ_2, then the allocation nodes of v_1 on the path from μ_2 to μ_1 are exactly those with distinct representatives for v_1 and v_2 (such nodes form a path in $\mathcal{T}$).*

Proof. By Fact 6, cases 1, 2, and 3 characterize the set of nodes χ of $\mathcal{T}$ such that v_1 and v_2 have distinct representatives in *skeleton*(χ). □

In the example of Figs. 1 and 4, the nodes with distinct representatives for vertices 6 and 17 are α_3 and α_2 (case 3), while for vertices 6 and 13, node α_2 is the only node with distinct representatives (case 2).

A *peripheral* edge (vertex) of a planar st-graph G is such that it appears on the same face of s and t for some embedding of G. A node μ of $\mathcal{T}$ is said to be peripheral if its virtual edge is peripheral in the skeleton of the parent of μ. Note that the children of S- and P-nodes are always peripheral.

In the example of Figs. 1 and 4, the peripheral edges of *skeleton*(α_5) are $(1, 12)$, $(12, 17)$, $(1, 11)$, $(11, 15)$, $(15, 16)$, and $(16, 17)$, while the only nonperipheral vertex of the entire graph is 5. Also, all the R-, P-, and S-nodes except α_9 are peripheral.

FACT 7. *Let e be an edge of the skeleton skeleton(μ) of node μ. If vertex v is peripheral in the expansion graph of e and e is peripheral in skeleton(μ), then v is peripheral in the pertinent graph G_μ of μ.*

The following lemma gives a method for testing whether a vertex is peripheral in the pertinent graph of some node of $\mathcal{T}$.

LEMMA 4.5. *Let ξ be a node of $\mathcal{T}$, and v be a vertex of the pertinent graph G_ξ of ξ. Let μ be the proper allocation node of v.*

1. *If $\mu = \xi$, then v is peripheral in G_ξ if and only if v is peripheral in skeleton(μ).*
2. *If μ is an ancestor of ξ, then v is always peripheral in G_ξ.*
3. *If μ is a descendant of ξ, then let λ be the child of ξ whose subtree contains μ.*

Then vertex v is peripheral in G_ξ if and only if v is peripheral in skeleton(μ) and all the nodes on the path from μ to λ (inclusive) are peripheral.

Proof. Case 1 is proved by observing that substituting the virtual edges of *skeleton*(μ) with their expansion graphs or vice versa does not change the peripheral status of the vertices of *skeleton*(μ). Case 2 follows from Lemma 4.3 since v is a pole of G_ξ. Case 3 follows by inductively applying Fact 7. □

4.2. Test algorithm. In this section, we show how to perform operation *Test*. The algorithm is based on the following theorem, whose intuition is illustrated in Fig. 6.

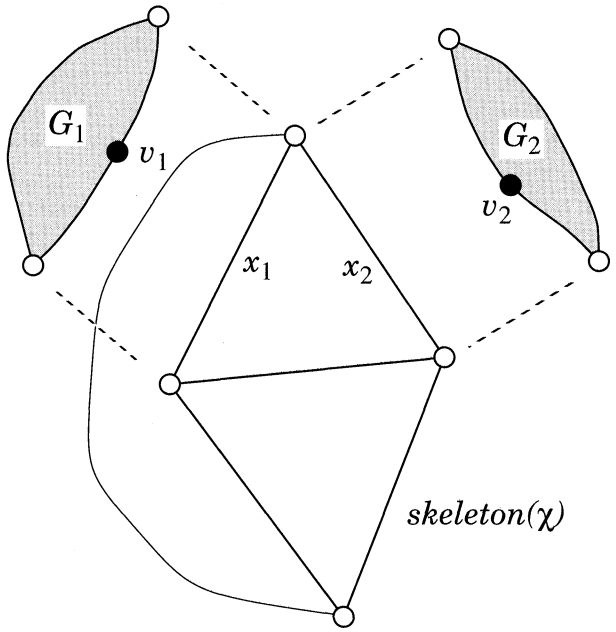

FIG. 6. *Schematic illustration of Theorem* 4.6.

THEOREM 4.6. *Let v_1 and v_2 be vertices of a planar st-graph G with the edge (s,t). There exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ if and only if there exists a node χ of the decomposition tree $\mathcal{T}$ of G such that:*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face of some embedding of skeleton(χ); and*
3. *v_1 and v_2 are peripheral vertices of the expansion graphs G_1 and G_2 of x_1 and x_2, respectively.*

Proof: *If*. From condition 2, let Σ be a planar embedding of *skeleton*(χ) with x_1 and x_2 on the same face. Also, from condition 3, let Γ_1 and Γ_2 be planar embeddings

of G_1 and G_2 with vertices v_1 and v_2 on the same face as the poles of G_1 and G_2, respectively. (See Fig. 6.) We replace x_1 and x_2 in Σ with Γ_1 and Γ_2 and perform at most two reverse operations such that v_1 and v_2 will be on the same face. We replace the remaining edges of $skeleton(\chi)$ with planar embeddings of the corresponding pertinent graphs. This gives a planar embedding of the pertinent graph of χ such that v_1 and v_2 are on the same face. Such an embedding can be easily extended to an embedding of G with the desired property.

Only if. We find node χ using Lemma 4.4. Let μ_1 and μ_2 be the proper allocation nodes of v_1 and v_2, respectively, and let μ be the least common ancestor of μ_1 and μ_2. If one of μ_1 and μ_2 is the ancestor of the other, then we define χ as the lowest node on the path between μ_1 and μ_2 such that the pertinent graph of χ contains both v_1 and v_2. Otherwise, we define $\chi = \mu$. Hence condition 1 is verified.

Consider a planar embedding Γ of G with v_1 and v_2 on the same face. We contract into single edges the pertinent graphs of the children of χ while preserving the embedding. We obtain an embedding of $skeleton(\chi)$ with x_1 and x_2 on the same face (condition 2).

In order to prove condition 3, assume for contradiction that v_1 is not a peripheral vertex of G_1. We have $\mu_1 \neq \chi$ and μ_1 is an R-node. Let s_1 and t_1 be the poles of $skeleton(\mu_1)$. By condition 2, x_1 and x_2 are on the same face of $skeleton(\chi)$, so that there exists a simple undirected path in $skeleton(\chi)$ between s_1 and t_1 that contains x_2. We replace each edge x of such path with a path π_x between the poles of the pertinent graph of node ν such that x is the virtual edge of ν, where, if x_2 is an edge, path π_{x_2} goes through vertex v_2. This gives a simple undirected path π of G between s_1 and t_1. Hence graph G_1^+ consisting of G_1, edge (v_1, v_2), and path π must be planar. Consider a planar embedding of G_1^+. By removing edge (v_1, v_2) and path π, we obtain a planar embedding of G_1 such that v_1, s_1, and t_1 are on the same face. This contradicts the assumption that v_1 is not a peripheral vertex of G_1. A similar argument can be used to show that v_2 must be a peripheral vertex of G_2. This completes the proof of condition 3. □

In the example of Figs. 1 and 4, vertices 6 and 13 verify the hypothesis of Theorem 4.6, while vertices 5 and 18 do not.

We remark that either a skeleton graph admits a unique embedding (R-node) or any two vertices/edges can be placed on the same face (P-, Q-, and S-nodes). Hence Theorem 4.6 reduces the *Test* operation to a test on a fixed embedding. The algorithm for operation *Test* (Algorithm 1) is based on Theorem 4.6 and its proof.

We give two examples for the algorithm *Test* which refer to Figs. 1 and 4.

Consider operation $Test(13, 6)$; namely, $v_1 = 13$ and $v_2 = 6$. We have that $\mu_1 = \alpha_{10}$, $\mu_2 = \alpha_6$, and $\mu = \alpha_2$. Thus we are in case (b) and $\chi = \mu = \alpha_2$, $\lambda_1 = \alpha_5$, $\lambda_2 = \alpha_3$, and $\kappa_1 = \kappa_2 = \alpha_0$. Since χ is a P-node, there exists an embedding of $skeleton(\chi)$ with the representatives of v_1 and v_2 on the same face. Also, both κ_1 and κ_2 are on the path from χ to the root (actually, they are the root). Therefore, $Test(13, 6)$ returns *true*.

Consider operation $Test(17, 5)$; namely, $v_1 = 17$ and $v_2 = 5$. We have that $\mu_1 = \alpha_0$, $\mu_2 = \alpha_9$, and $\mu = \mu_1 = \alpha_0$. Thus we are in case (c). Vertex v_2 is peripheral in $skeleton(\mu_2)$, $\kappa_2 = \alpha_9$, and $\chi = \alpha_6$ and is not an allocation node of v_1. Therefore, $Test(17, 5)$ returns *false*. In this example, vertex 5 is not "peripheral enough" with respect to vertex 17.

The correctness of the algorithm follows from Theorem 4.6 and Lemmas 4.4 and 4.5. In case (a), condition 3 of Theorem 4.6 is always trivially verified. Re-

ALGORITHM 1. *Test*(v_1, v_2)

1. Find the proper allocation nodes μ_1 of v_1 and μ_2 of v_2
2. Find the least common ancestor μ of μ_1 and μ_2.
3. **case of**

(a) $\mu_1 = \mu_2 = \mu$;
let $\chi = \mu$;
if v_1 and v_2 are on the same face of some embedding of *skeleton*(χ)
then return *true*
else return *false.*

(b) $\mu_1 \neq \mu$ **and** $\mu_2 \neq \mu$;
let $\chi = \mu$;
for $i = 1, 2$ **do**
Find the representative x_i of v_i in *skeleton*(χ) as follows: determine the child λ_i of χ on the path from μ_i to χ, and let x_i be the virtual edge of λ_i in *skeleton*(χ).
Find the first nonperipheral node κ_i on the path from μ_i to the root.
endfor
if (x_1 and x_2 are on the same face of some embedding of *skeleton*(χ)) **and** (v_1 and v_2 are peripheral vertices of *skeleton*(μ_1) and *skeleton*(μ_2), respectively) **and** (κ_1 and κ_2 are either children of μ or on the path from μ to the root)
then return *true*
else return *false.*

(c) $\mu_1 = \mu$ **and** $\mu_2 \neq \mu$
if v_2 is not a peripheral vertex of *skeleton*(μ_2)
then return *false*
Determine the first nonperipheral node κ_2 of the path from μ_2 to the root.
if κ_2 is a child of μ **or** κ_2 is on the path from μ_1 to the root
then set $\chi = \mu_1$
else set χ equal to the parent of κ_2.
if χ is not an allocation node of v_1
then return *false*
Find the representative x_2 of v_2 in *skeleton*(χ).
if v_1 and x_2 are on the same face of the embedding of *skeleton*(χ)
then return *true*
else return *false.*

(d) $\mu_2 = \mu$ **and** $\mu_1 \neq \mu$
(This is analogous to the previous case and therefore omitted.)

endcase

garding condition 2, if it is not verified at node μ, then by Lemma 4.4, μ is the only node that verifies condition 1. In case (b), condition 1 of Theorem 4.6 is verified only for node μ, the least common ancestor of the proper allocation nodes of v_1 and v_2. We set $\chi = \mu$, test condition 2 directly, and verify condition 3 by applying Lemma 4.5.

Case (c) is more complex since more than one node may satisfy condition 1 of Theorem 4.6. By Lemma 4.4, such nodes are the allocation nodes of v_1 on the path from μ_2 to μ_1.

In this case, the specific choice of node χ made in the proof of Theorem 4.6 (i.e., the lowest node on the path between μ_1 and μ_2 such that the pertinent graph of χ contains both v_1 and v_2), which satisfies condition 1, appears difficult to compute. Thus we use a slightly different approach, where node χ satisfies condition 3. First, we choose node χ as the highest node where condition 3 is satisfied by applying Lemma 4.5. If χ is not an allocation node of v_1, then condition 1 is not verified at χ; moreover, since condition 1 can be satisfied only at ancestors of χ and condition 3 can be satisfied only at or below χ, there is no node for which both conditions 1 and 3 can be satisfied. Otherwise (χ is an allocation node of v_1), condition 1 is satisfied and we check condition 2 directly. If condition 2 is not verified, then v_1 is not an endpoint of the representative edge x_2 of v_2 in *skeleton*(χ). Hence v_1 is not a pole of the expansion graph of x_2, so that no descendant of χ is an allocation node of v_1. This implies that condition 1 cannot be verified at any descendant of χ.

4.3. Static data structure and time complexity. The following data structure can be used to efficiently perform the *Test* operation in a static environment. We store with each vertex a pointer to its proper allocation node in $\mathcal{T}$. Hence step 1 takes $O(1)$ time. We equip tree $\mathcal{T}$ with a data structure which uses linear space and supports least common ancestor queries in constant time [29, 50]. Hence step 2 takes $O(1)$ time.

Concerning step 3, we set up the following data structures. Each node of $\mathcal{T}$ has a pointer to the corresponding virtual edge in the skeleton of its parent. We mark all the peripheral nodes and the peripheral vertices and edges of each skeleton. Also, each node ζ has a pointer to the first nonperipheral node $\hat{\zeta}$ in the path from ζ to the root (the root node points to itself). Finally, we equip the skeleton of each R-node with the data structure for planar embedding tests described in [53]. This allows us to test whether two vertices/edges are on the same face of the planar embedding of the skeleton in $O(\log n)$ time.

By Lemma 4.1, tree $\mathcal{T}$ uses $O(n)$ space. All the remaining data structures use $O(n)$ space. The decomposition tree $\mathcal{T}$ can be constructed in $O(n)$ time using a variation of the algorithm of [30] for finding the triconnected components of a graph. The planar embeddings of the skeletons of $\mathcal{T}$ and their peripheral vertices and edges can be computed in $O(n)$ time using the planarity-testing algorithm of [30]. We conclude the following.

THEOREM 4.7. *Let G be a biconnected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(\log n)$ and can be constructed in $O(n)$ preprocessing time.*

Proof. Orient G into a planar st-graph, where s and t are adjacent vertices, and then use algorithm *Test* with the data structure described above. □

Note that the $O(\log n)$ bound on the query time depends only on the performance of the data structure of [53] for testing whether two vertices/edges are on the same face of a planar embedding. It can be shown that, applying perfect hashing [59], the query time of [53] can be reduced to $O(1)$ at the expense of using a complicated $O(n)$-space data structure with $O(n^2)$ preprocessing time. We have the following corollary.

COROLLARY 4.8. *Let G be a biconnected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(1)$ and can be constructed in $O(n^2)$ preprocessing time.*

5. Updates. In this section, we show how to perform operations *InsertEdge* and *InsertVertex* on a biconnected planar graph G. As shown in the following theorem, the above repertoire of update operations is complete for biconnected planar graphs.

THEOREM 5.1. *A biconnected planar graph G with $n \geq 3$ vertices and m edges can be assembled starting from the triangle graph (a cycle of three vertices) by means of $m-3$ InsertEdge and InsertVertex operations such that each intermediate graph is planar and biconnected. Also, such a sequence of operations can be determined in $O(n)$ time.*

Proof. We compute an *open-ear decomposition* $D = (P_0, P_1, \ldots, P_r)$ of G, which is a partition of the edges of G into an ordered collection of edge-disjoint simple paths $P_0, P_1, \ldots, P_r$, called *ears*, such that

- P_0 is a simple cycle;
- the two endpoints of ear P_i, for $i \geq 1$, are distinct and contained in some P_j, $j < i$; and
- none of the internal vertices of P_i are contained in any P_j, $j < i$.

A graph G has an open-ear decomposition if and only if it is biconnected; moreover, all intermediate graphs $D_i = P_0 + P_1 + \cdots + P_i$ of an open-ear decomposition of a biconnected graph are biconnected [61]. Further, if G is planar, then each D_i is planar since it is a subgraph of G. An ear decomposition can be computed in $O(n)$ time using the st-numbering technique [18]. We show how to use D to determine the assembly sequence of G. Starting from the initial triangle graph, we construct cycle P_0 by means of a sequence of *InsertVertex* operations. Next, we add the remaining ears $P_1, \ldots, P_r$, each by means of one *InsertEdge* operation followed by zero or more *InsertVertex* operations. Since we start with the triangle graph having three vertices and since each operation adds one edge, the total number of operations is $m-3$. To avoid forming intermediate graphs with multiple edges, we modify the ears as follows. For each edge $e = (u, v)$, let P_{i_0} be the ear containing e, with $P_{i_0} = P'eP''$. If there are ears $P_{i_1}, \ldots, P_{i_k}$ with endpoints u and v and $i_0 < i_1 < \cdots < i_k$, we replace P_{i_0} with $P'P_{i_k}P''$ and P_{i_k} with e. Note that each intermediate graph generated is planar since it is homeomorphic to a subgraph of G. The above modification of the ears can be computed in $O(n)$ time by radix-sorting the edges and ears on their endpoints. □

In our dynamic environment, we maintain a planar st-orientation of a biconnected planar graph G such that s and t are adjacent vertices as follows.

In operation *InsertVertex*(e, v, e_1, e_2), if e goes from s to t, then we orient edge e_1 from s to v and edge e_2 from t to v. Vertex v is the new sink of the orientation. Otherwise, we orient e_1 and e_2 in the same way as e.

In operation *InsertEdge*(e, v_1, v_2), we orient e from v_1 to v_2 if v_2 is reachable from v_1 in the planar st-orientation and from v_2 to v_1 if v_1 is reachable from v_2. If neither vertex is reachable from the other, both orientations of e are possible. To test the condition on reachability, we use the following theorem.

THEOREM 5.2. *Let v_1 and v_2 be vertices of a planar st-graph G with the edge (s, t) and such that there exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ. Let χ be a node of the decomposition tree $\mathcal{T}$ of G such that*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face f of some embedding of skeleton(χ); and*
3. *v_1 and v_2 are peripheral vertices of the expansion graphs G_1 and G_2 of x_1 and x_2, respectively.*

Then there exists a directed path in G from v_1 to v_2 if and only if there exists a directed path in the boundary of face f from x_1 to x_2.

Proof. Note that the existence of node χ is guaranteed by Theorem 4.6. By the definition of a pertinent graph, there exists a directed path in G from v_1 to v_2 if and only if there exists a directed path in *skeleton*(χ) from x_1 to x_2. By the reachability

properties of planar st-graphs given in [54], we have that there exists a directed path in $skeleton(\chi)$ from x_1 to x_2 if and only if there exists a directed path in the boundary of face f from x_1 to x_2. □

By property 2 of Lemma 3.1, face f consists of two directed paths with a common origin and destination. Hence the time for testing reachability in f is dominated by the complexity of determining if two objects of f (each a vertex or an edge) are on the same path and, if so, which object precedes the other. In the rest of this section, we assume that G is a planar st-graph that contains the edge (s,t).

The algorithm for operation *InsertVertex* is as follows. Let ρ be the Q-node storing edge e and let π be the parent of ρ. If π is a P-node or an R-node, we replace ρ with a subtree consisting of an S-node and two child Q-nodes. If π is an S-node, we remove ρ and add two new child Q-nodes to π.

In the rest of this section, we present the algorithm for operation *InsertEdge*(e, v_1, v_2). The algorithm makes use of several types of transformations that modify the decomposition tree. If a transformation produces a node with exactly one child, such node is absorbed into its parent. We assume that operation $Test(v_1, v_2)$ has been already performed and has returned *true*.

The algorithm *InsertEdge* and its subroutines are shown as Algorithm 2 and Procedures 1–6.

ALGORITHM 2. *InsertEdge*(e, v_1, v_2)
Find the proper allocation nodes μ_1 of v_1 and μ_2 of v_2 and their least common ancestor μ.
case of
1. $\mu_1 = \mu_2 = \mu$;
   let $\chi = \mu$;
   { Since G already contains the edge (s,t), by Fact 5, node χ is either an S-node or an R-node. }
   *FinalTransformation*1 (χ)
2. $\mu_1 \neq \mu$ and $\mu_2 \neq \mu$;
   let $\chi = \mu$;
   for $i = 1, 2$ **do**
      PathCondensation (μ_i, λ_i)
   endfor
   *FinalTransformation*2 $(\chi, \lambda_1, \lambda_2)$.
3. $\mu_1 = \mu$ and $\mu_2 \neq \mu$;
   Determine the lowest node χ on the path from μ_2 to μ such that $skeleton(\mu)$ contains v_1.
   if $\chi = \mu_2$
   then *FinalTransformation*1 (χ)
   else
      PathCondensation (μ_2, λ_2);
      *FinalTransformation*3 (χ, λ_2).
4. $\mu_2 = \mu$ and $\mu_1 \neq \mu$
   (This is analogous to the previous case and therefore omitted.)

endcase

PROCEDURE 1. *FinalTransformation*1 (χ)

1. χ *is an R-node.* We have two subcases.
   Graph *skeleton*(χ) does not contain an edge between v_1 and v_2.
   Edge e is inserted in *skeleton*(χ) and we add to the children of χ a new Q-node associated with e.
   Graph *skeleton*(χ) contains an edge between v_1 and v_2.
   The edge (v_1, v_2) of *skeleton*(χ) is the virtual-edge of a child ν of χ. If ν is a P-node, we add to the children of ν a new Q-node associated with e and we insert another edge from v_1 to v_2 in *skeleton*(ν). Else, we replace ν with a new P-node with children ν and a Q-node storing edge e.
2. χ *is an S-node.* We perform at node χ the transformation illustrated in Fig. 7. The sequence of children of χ is partitioned into subsequences α, β, and γ, where β consists of the children of χ associated with the edges of *skeleton*(χ) between vertices v_1 and v_2. We remove the nodes of β from the children of χ and replace them with a new P-node whose children are a Q-node associated with edge e and an S-node whose children are the nodes of β. Graph *skeleton*(χ) is updated by replacing the chain between v_1 and v_2 with a single edge. The skeleton of the new P-node consists of two multiple edges from v_1 to v_2. The skeleton of the new S-node consists of a chain of $|\beta|$ edges from v_1 to v_2. Note that if $|\beta| = 1$, the new S-node is absorbed into its parent, as mentioned above.

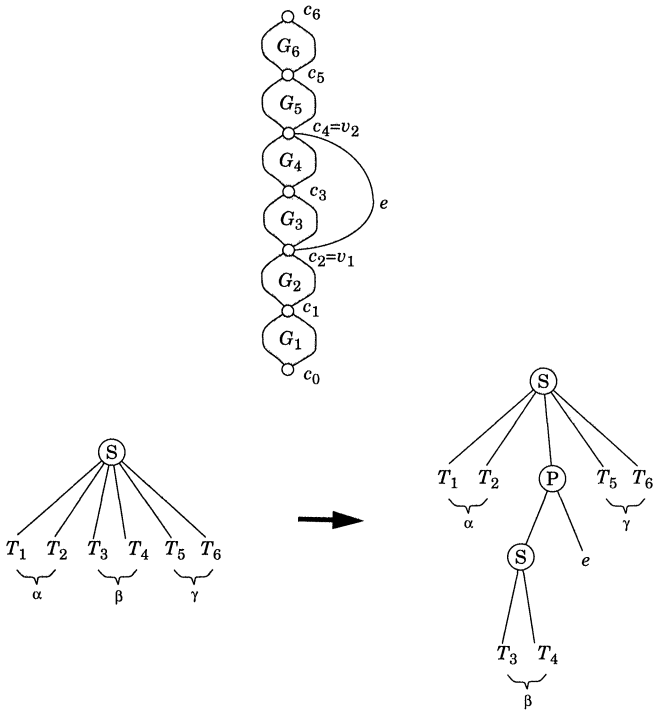

FIG. 7. *The procedure FinalTransformation*1 *when* χ *is an S-node.*

PROCEDURE 2. *PathCondensation* (μ_i, λ_i)

InitialTransformation (μ_i)
Determine the child λ_i of χ on the path from μ_i to χ.
set $\rho = \mu_i$;
while $\rho \neq \lambda_i$ **do**
    set π equal to the parent of ρ;
    ElementaryTransformation $(\rho, \pi; \pi')$;
    set $\rho = \pi'$;
endwhile

PROCEDURE 3. *InitialTransformation* (μ_i)

If μ_i is an S-node, expand μ_i into a structure consisting of an R-node ν and two S-nodes ν' and ν'', such that (see Fig. 8)

- ν has children ν' and ν'' and has the same parent as μ_i;
- graph *skeleton*(ν) consists of edges (s_{μ_i}, v_i), (v_i, t_{μ_i}), and (s_{μ_i}, t_{μ_i});
- graphs *skeleton*(ν') and *skeleton*(ν'') consist of the subchains of *skeleton*(μ_i) from s_{μ_i} to v_i and from v_i to t_{μ_i}, respectively.

Rename $\mu_i = \nu$.

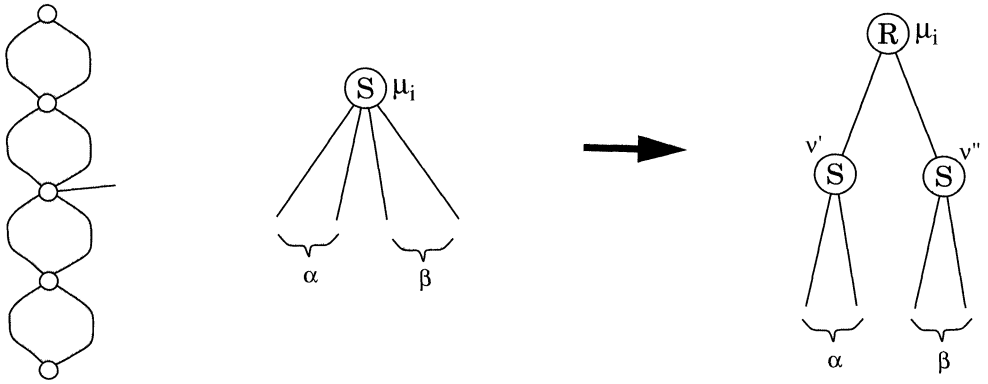

FIG. 8. *Expansion of an S-node in InitialTransformation.*

PROCEDURE 4. *ElementaryTransformation* $(\rho, \pi; \pi')$

Let X be the type of node ρ and let π be the parent of ρ. Perform the RX-transformation described below and shown in Fig. 9.

RR-transformation: Contract nodes ρ and π into a new node π'. Graph *skeleton*(π') is obtained from *skeleton*(π) by replacing the virtual edge of ρ with *skeleton*(ρ) minus the edge (s_ρ, t_ρ). (See Fig. 9(a))

RP-transformation: Rename ρ and π into π' and ρ', respectively. Set the parent of ρ' equal to π'. Set the parent of π' equal to the former parent of π. Graph *skeleton*(ρ') is equal to *skeleton*(π) minus one of the edges (s_ρ, t_ρ) (the former virtual edge of ρ). Graph *skeleton*(π') is equal to *skeleton*(ρ) plus a virtual edge $(s_{\rho'}, t_{\rho'})$. (See Fig. 9(b))

RS-transformation: Split node π into nodes ρ' and ρ'' such that *skeleton*(ρ') and *skeleton*(ρ'') are the subchains of *skeleton*(π) from s_π to s_ρ and from t_ρ to t_π, respectively. Rename ρ into π'. Set the parents of ρ' and ρ'' equal to π'. Set the parent of π' equal to the former parent of π. Graph *skeleton*(π') consists of *skeleton*(ρ) minus edge (s_ρ, t_ρ) plus edges (s_π, s_ρ), (t_ρ, t_π), and (s_π, t_π). (See Fig. 9(c))

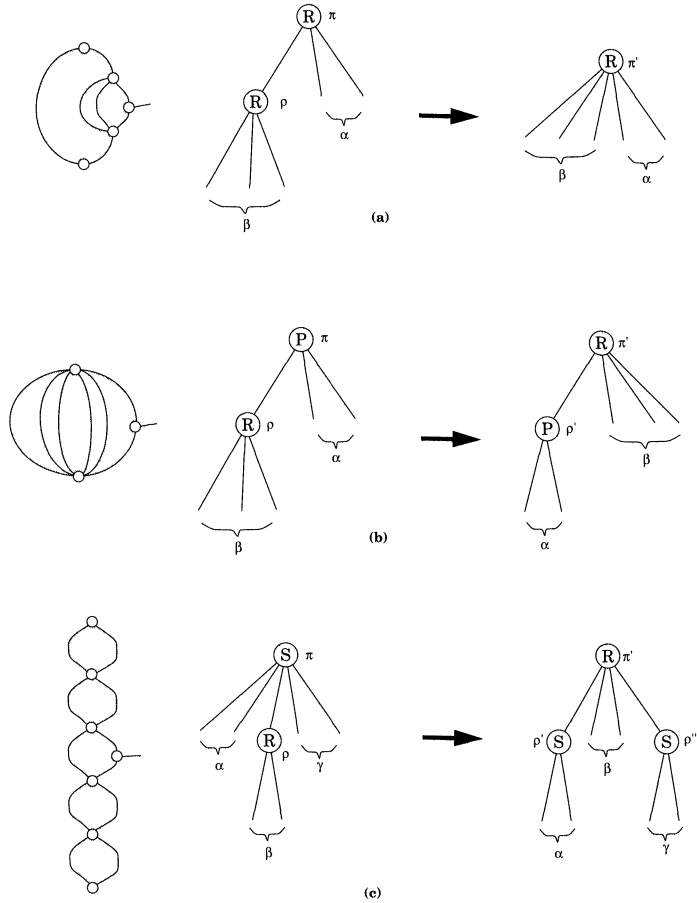

FIG. 9. *Elementary transformations:* (a) *RR*; (b) *RP*; (c) *RS*.

PROCEDURE 5. *FinalTransformation2* $(\chi, \lambda_1, \lambda_2)$

Let X be the type of node χ. Perform the X-transformation described below and shown in Fig. 10. Note that λ_1 and λ_2 are R-nodes.

R-transformation: Contract nodes χ, λ_1 and λ_2 into a new R-node χ'. Graph *skeleton*(χ') is obtained from *skeleton*(χ) by replacing the virtual edges of λ_1 and λ_2 with their skeletons (minus the edge between their poles) and by adding the edge (v_1, v_2).

P-transformation: Contract nodes λ_1 and λ_2 into a new R-node λ. Graph *skeleton*(λ) is obtained by the union of *skeleton*(λ_1), *skeleton*(λ_2), and the edge (v_1, v_2).

S-transformation: Partition the sequence of children of χ into subsequences α, λ_1, β, λ_2, and γ in this order from left to right. We remove the nodes of β from the children of χ and replace them with a new R-node ν. Also, we create a new S-node λ with parent ν and whose children are the nodes of β. Graph *skeleton*(ν) is obtained from *skeleton*(λ_1) and *skeleton*(λ_2) by adding the edges $(s_{\lambda_1}, t_{\lambda_2})$, $(t_{\lambda_1}, s_{\lambda_2})$, and (v_1, v_2).

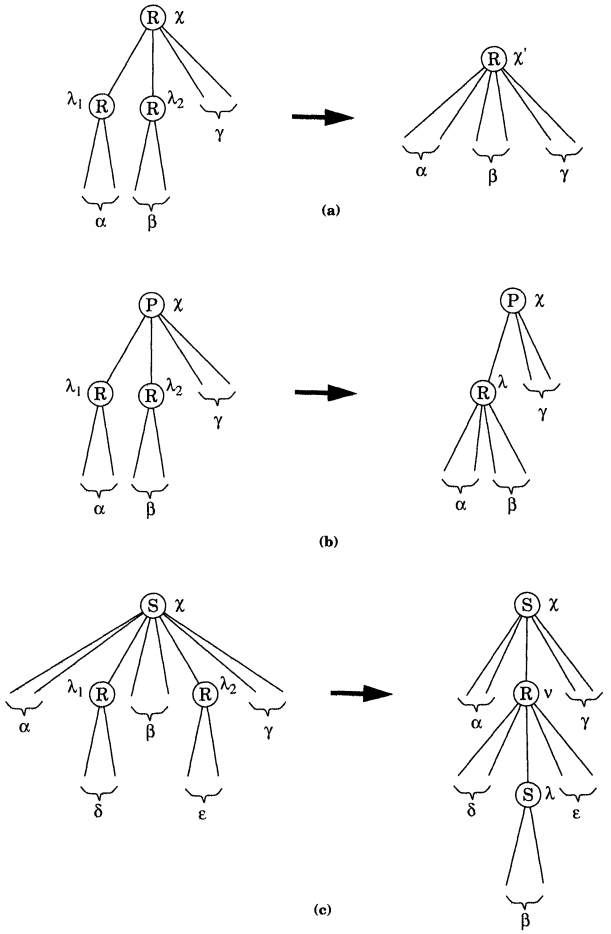

FIG. 10. *The procedure FinalTransformation2:* (a) *R*; (b) *P*; (c) *S*.

PROCEDURE 6. *FinalTransformation3* (χ, λ_2)

Let X be the type of node χ. Perform the X-transformation described below. Note that λ_2 is an R-node.

R-transformation: Contract nodes χ and λ_2 into a new R-node χ'. Graph *skeleton*(χ') is obtained from *skeleton*(χ) by replacing the virtual edge of λ_2 with *skeleton*(λ_2) (minus the edge between the poles) and by adding the edge (v_1, v_2).

S-transformation: Partition the sequence of children of χ into subsequences α, β, λ_2, and γ in this order from left to right, where v_1 is the common pole of the pertinent graphs of the last node of α and the first node of β. We remove the nodes of β from the children of χ and replace them with a new R-node ν. Also, we create a new S-node λ with parent ν and whose children are the nodes of β. Graph *skeleton*(ν) is obtained from *skeleton*(λ_2) by adding the edges (v_1, s_{λ_2}), (v_1, t_{λ_2}), and (v_1, v_2).

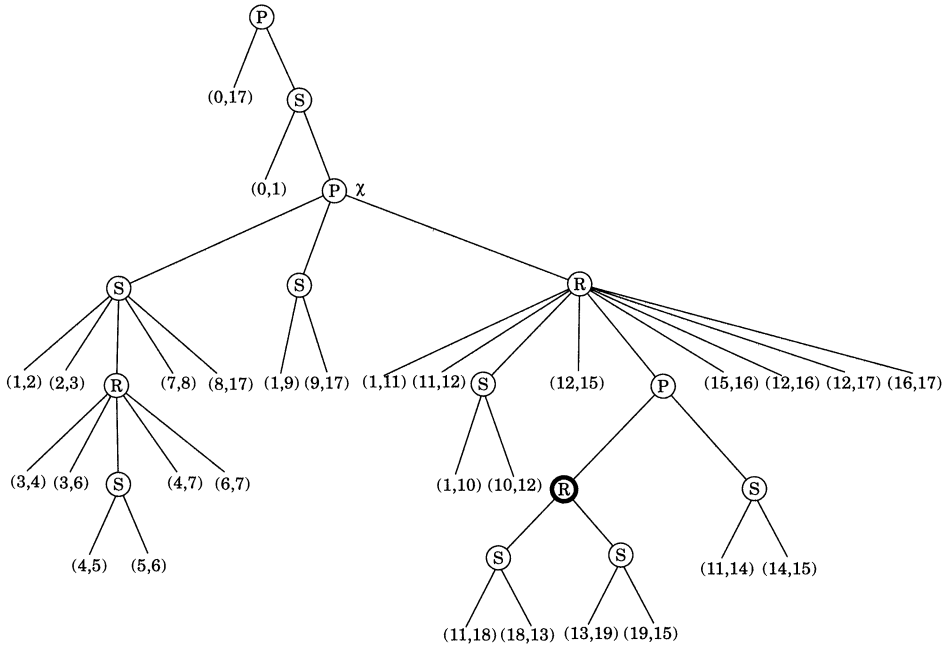

FIG. 11. *Initial S-transformation in operation InsertEdge*$(e, 13, 6)$.

With reference to Figs. 1 and 4, we show how *InsertEdge*$(e, 13, 6)$ is performed. We are in case 2. The initial transformation at μ_1 is shown in Fig. 11. Elementary transformations RP, RR, and RS are shown in Figs. 12–14. The final decomposition tree (after *FinalTransformation2*) is shown in Fig. 15.

We now argue about the correctness of the algorithm *InsertEdge*. We discuss case 2 since it is the most general. Similar considerations hold for cases 1, 3, and 4.

First, we observe that after the insertion of edge (v_1, v_2), the poles of χ remain a separation pair of G. Hence only the subtree of $\mathcal{T}$ rooted at χ is affected by the insertion. Namely, we show that the algorithm *InsertEdge* correctly computes the decomposition tree of the pertinent graph of χ plus the edge (v_1, v_2).

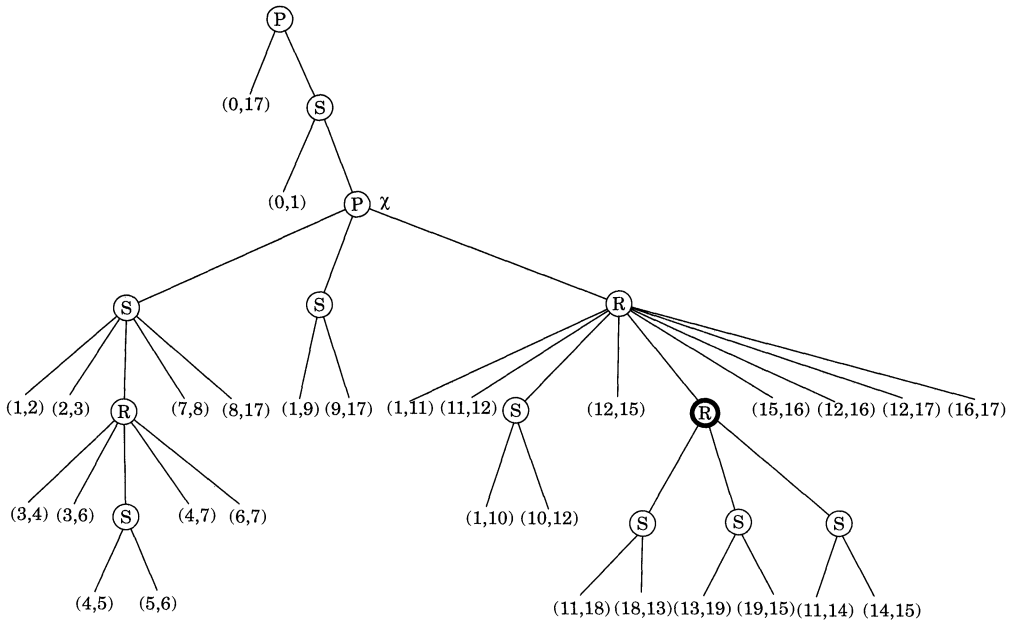

FIG. 12. *Elementary RP-transformation in operation InsertEdge*$(e, 13, 6)$.

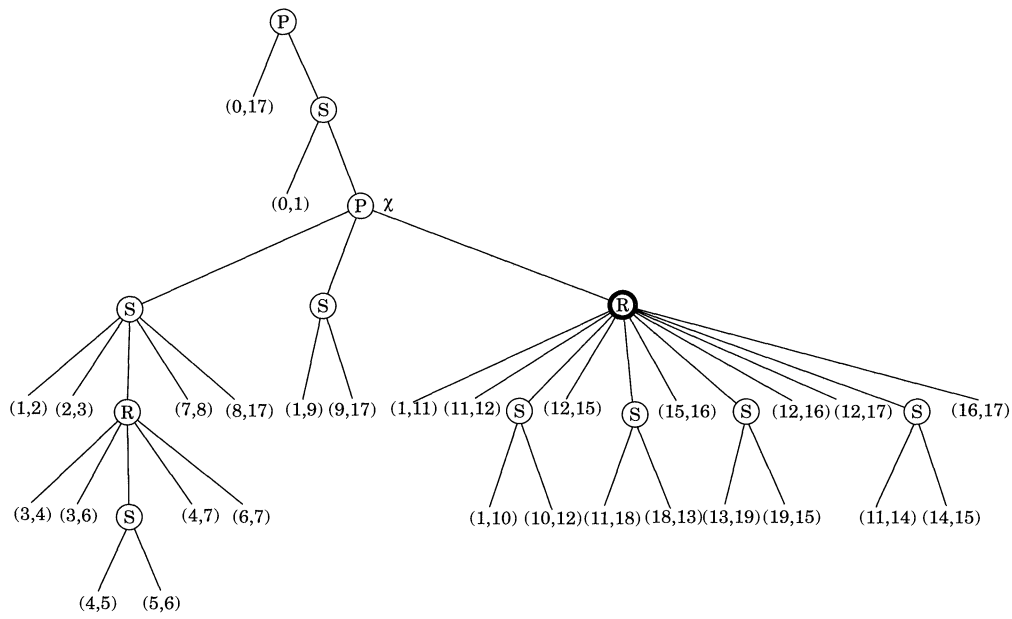

FIG. 13. *Elementary RR-transformation in operation InsertEdge*$(e, 13, 6)$.

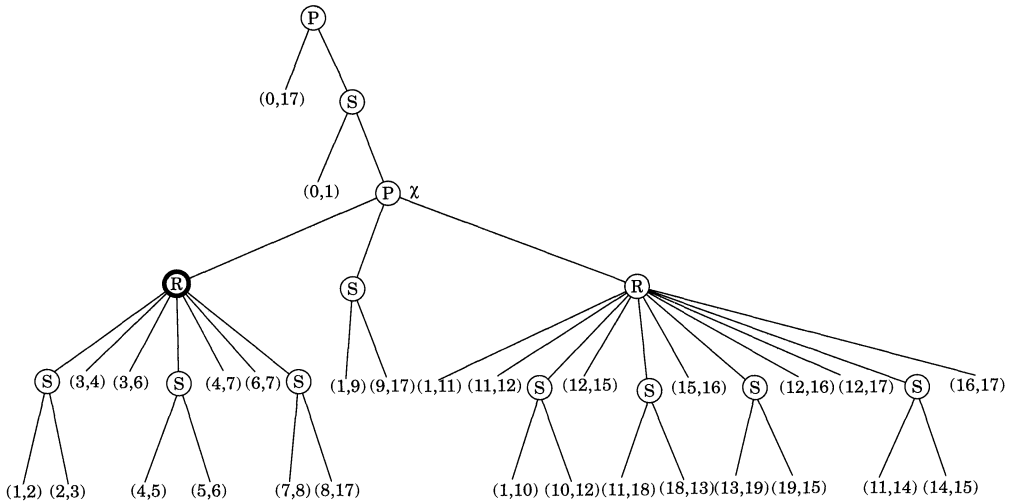

FIG. 14. *Elementary RS-transformation in operation InsertEdge*$(e, 13, 6)$.

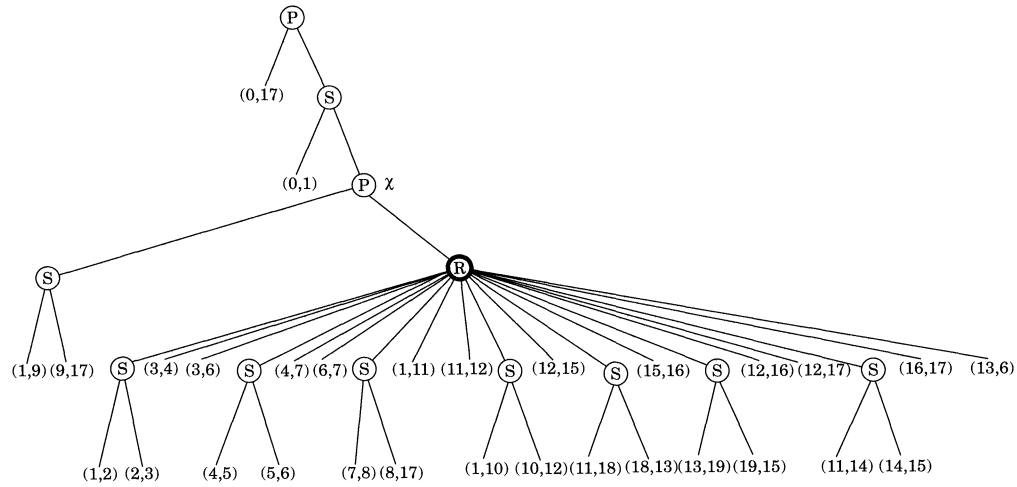

FIG. 15. *Final P-transformation in operation InsertEdge*$(e, 13, 6)$.

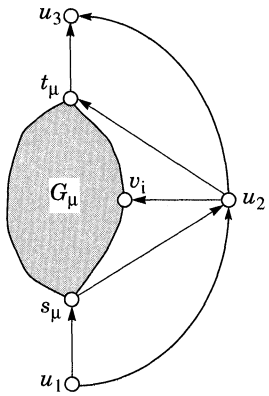

FIG. 16. *Graph Θ^i_μ used to show the correctness of the algorithm InsertEdge.*

Let μ be a node of $\mathcal{T}$ whose pertinent graph G_μ contains vertex v_i. We denote by Θ^i_μ the graph obtained from G_μ by adding three new vertices u_1, u_2, and u_3 and the edges (u_1, s_μ), (u_1, u_2), (s_μ, u_2), (u_2, v_i), (u_2, t_μ), (u_2, u_3), and (t_μ, u_3), as shown in Fig. 16. Intuitively, the "gadget" added to G_μ forces v_i to appear on the external face. The formal proof consists of showing that

1. each transformation (initial or elementary) at a node π on the path from μ_i to λ_i produces the decomposition tree of Θ^i_π, $i = 1, 2$, except for the Q-nodes associated with the extra edges added to G_μ and their virtual edges in the skeletons;
2. the final transformation at node χ produces the decomposition tree of the pertinent graph of χ plus the edge (v_1, v_2).

The first property can be proved by induction. The base case is the initial transformation at μ_i. The inductive steps correspond to the elementary transformations. The second property can be proved by a simple case analysis. Note that the graph Θ^i_π is planar since by Theorem 4.6 vertex v_i is peripheral in the pertinent graph of π. We omit the details of the correctness proof, which are tedious but straightforward.

6. Dynamic data structure. All the information needed to perform the *Test* algorithm must be updated by the *InsertEdge* and *InsertVertex* algorithms. We describe a data structure that represents the decomposition tree $\mathcal{T}$, the skeletons (with their embeddings) of the nodes of $\mathcal{T}$, and the maximal paths of peripheral nodes in $\mathcal{T}$. The interface of the data structure consists of records for the vertices and edges of the graph G.

6.1. Requirements. In this section, we discuss the primitive operations that need to be supported by the dynamic data structure. The data structure for the decomposition tree $\mathcal{T}$ should support finding the parent of a node and the least common ancestor of two nodes. Also, it should support the initial, elementary, and final transformations. Each such transformation is executed by means of a constant number of link/cut and expand/contract operations.

Concerning skeletons and their embeddings, we need to support a repertoire of access, query, and update operations. The access operations are as follows:

1. find the proper allocation node of a vertex;
2. find the poles of the skeleton of a node.

The query operations are as follows:

1. determine if a vertex is peripheral with respect to the skeleton of an R-node;

2. determine if two objects (each a vertex or an edge) are on the same face of the skeleton of an R-node.

3. determine if two objects on the same face f of a skeleton are also on the same directed path forming the boundary of f and, if so, which object precedes the other.

The nontrivial update operations are as follows:

1. add vertices and edges to skeletons;
2. replace an edge of a skeleton with another skeleton;
3. split the skeleton of an S-node (by removing an edge).

Finally, we need to maintain the set of peripheral nodes of $\mathcal{T}$ so that we can efficiently determine the first nonperipheral node κ on the path from a node μ to the root of $\mathcal{T}$.

6.2. Maintaining planar embeddings. Our technique for maintaining the planar embedding of the skeletons extends that of [53], where the latter two update operations are not supported.

We recall from property 2 of Lemma 3.1 that the boundary of each face of the embedding of a planar st-graph G consists of two directed paths with common origin and destination. Also, by property 1 of Lemma 3.1, each vertex of G distinct from the poles s and t is an internal node of exactly two faces.

FACT 8 (see [53]). *Let Γ be a planar embedding of a planar st-graph G. Objects x_1 and x_2 of G, each a vertex or an edge, are on the same face f of Γ if and only if one of the following conditions is verified:*

1. *each of x_1 and x_2 is an edge or an internal vertex of f;*
2. *one of x_1 and x_2 is an edge or an internal vertex and the other is an extreme vertex of f;*
3. *both x_1 and x_2 are extreme vertices of f.*

An *extreme pair* is a pair of vertices that are the extreme vertices of some face f of Γ. For example, $(12, 17)$ and $(11, 15)$ are extreme pairs of the graph of Fig. 1.

By Lemma 3.1, every object is internal in exactly two faces, and every face has exactly two extreme objects (always vertices). Hence conditions 1 and 2 can be tested in constant time after having determined the four faces where x_1 and x_2 are internal and these faces' extreme vertices. Condition 3, on the other hand, is tested by searching for (v_1, v_2) in the set of extreme pairs. The data structure of [53] maintains the set of extreme pairs in a dynamic dictionary and the set of internal vertices of each face in two concatenable queues (associated with the two directed paths forming its boundary).

We now show how to modify the data structure of [53] to support our extended set of operations.

LEMMA 6.1. *The set of extreme pairs is an invariant of a planar st-graph with respect to all its planar embeddings.*

Proof. By Lemma 4.2 we can construct any embedding by means of reverse and swap operations on a given embedding. We show that the set of extreme pairs of an embedding stays unchanged after a reverse or a swap. Consider a reverse operation on a split component C with poles s' and t' (see Fig. 17). The boundaries of the faces internal to C are modified by exchanging their left and right chains, so that their extreme pairs remain the same. Let γ' and γ'' be the left and right chains forming the external boundary of C, where each such chain does not contain s' or t'. The faces f and g on the left and right of C contain γ' and γ'' as subchains of their right and left chains, respectively. After the reverse operation, these faces are modified by replacing γ' with γ'' in the right chain of f and γ'' with γ' in the left chain of g. Hence

the extreme pairs of f and g stay unchanged. Finally, the remaining faces of G are not affected by the reverse operation. Similar considerations show that extreme pairs stay unchanged after a swap operation. □

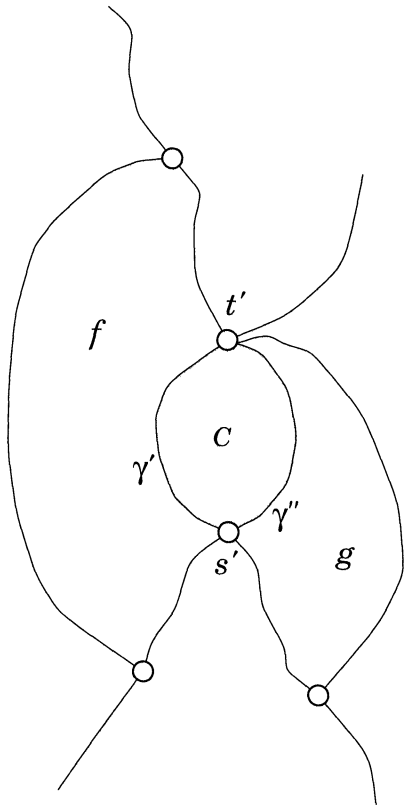

FIG. 17. *Example for the proof of Lemma* 6.1.

According to Lemma 6.1, we denote by $\mathcal{E}$ the set of extreme pairs of G.

LEMMA 6.2. *After performing operation InsertEdge*(e, v_1, v_2)*, the set* $\mathcal{E}$ *is updated by means of at most one deletion and two insertions. Also, after performing operation InsertVertex*(v, e, e_1, e_2)*, the set* $\mathcal{E}$ *stays unchanged.*

Proof. By Lemma 6.1, the update of the set $\mathcal{E}$ is the same in any embedding. Hence we consider adding the edge (v_1, v_2) to an an embedding where v_1 and v_2 are on the same face f. By Lemma 3.1, the boundary face f consists of two directed paths. Let (l, h) be the extreme pair of f. If v_1 and v_2 are on the same directed path of f, then we add to $\mathcal{E}$ the pair (v_1, v_2), unless it is already in $\mathcal{E}$ (when $v_1 = l$ and $v_2 = h$). Otherwise, we remove from $\mathcal{E}$ the pair (l, h) and add to $\mathcal{E}$ the pairs (v_1, h) and (l, v_2). □

The poles of each skeleton and, for R-nodes, the edge between them are directly stored at each node, so that they can be determined in $O(1)$ time. Their update takes $O(1)$ time in each transformation.

The record of each face f of *skeleton*(μ) (except the two faces containing the edge (s_μ, t_μ)) has a bidirectional pointer to the element of $\mathcal{E}$ associated with the extreme pair of f.

6.3. Data structure. The data structure consists of a *main component* and of an *auxiliary component*. The main component is a tree $\mathcal{T}^*$ that represents both the decomposition tree $\mathcal{T}$ and the skeletons of the nodes of $\mathcal{T}$. The auxiliary component is a dictionary (e.g., a balanced search tree) that stores the set $\mathcal{E}$, so that searches and updates in $\mathcal{E}$ take $O(\log n)$ time. The updates to be performed in $\mathcal{E}$ are determined in the final transformations.

The main component $\mathcal{T}^*$ is an *edge-ordered* dynamic tree [15], a variation of the dynamic tree of Sleator and Tarjan [52]. It is a rooted ordered tree with nodes of various types that supports each of the following *primitive tree operations* in logarithmic time:

- Find the parent of a node.
- Find the least common ancestor of two nodes.
- Given two sibling nodes, determine which one precedes the other in the ordered sequence of the children of their parent.
- Given a node ν, find the first node of a given type on the path from ν to the root.
- Link two trees by making the root of one tree a child of a node of the other tree.
- Cut a tree into two trees by removing a tree-edge.
- Expand a node ν into two nodes ν_1 and ν_2 linked by a new tree-edge such that the expansion preserves the ordering of the children. In other words, if $\alpha\beta\gamma$ is the sequence of children of ν, then $\alpha\nu_2\gamma$ is the sequence of children of ν_1 and β is the sequence of children of ν_2.
- Contract a tree-edge (ν_1, ν_2) and merge nodes ν_1 and ν_2 into a new node ν such that the contraction preserves the ordering of the children. In other words, if $\alpha\nu_2\gamma$ is the sequence of children of ν_1 and β is the sequence of children of ν_2, then $\alpha\beta\gamma$ is the sequence of children of ν.

The main component $\mathcal{T}^*$ is obtained from the decomposition tree $\mathcal{T}$ by expanding each node μ of $\mathcal{T}$ into a tree rooted at μ, called a *skeleton tree*, which describes the embedding of *skeleton*(μ), as follows (see Fig. 18):

1. First, we make children of μ a set of *f-nodes* representing the faces of *skeleton*(μ) (their order is irrelevant). The f-node associated with a face f is also said to be a *p-node* ("peripheral" node) if f contains an edge (s_μ, t_μ), and otherwise it is said to be a *b-node* ("blocking" node). Note that if μ is a P-node or an S-node, then all the children of μ are p-nodes. Also, if μ is an S-node or an R-node, it has two child p-nodes.
2. Next, we attach to each f-node two subtrees, called *boundary trees*, that represent the two directed paths forming the boundary of the face (see Lemma 3.1), excluding the extreme vertices. Each boundary tree is a two-level tree whose leaves are an alternating sequence of *e-nodes* and *v-nodes* representing edges and vertices of the path, respectively, and are ordered according to the direction of the path.
3. Finally, for each former child ν of μ, we make ν child of one of the two e-nodes of the skeleton tree of μ associated with the virtual edge of ν in *skeleton*(μ). If the closest f-node ancestor of one of such e-nodes is a p-node, then we make μ a child of that e-node.

The data structure is completed by the following additional pointers. Each R-, P-, and S-node stores pointers to the poles of its skeleton, so that they can be determined in $O(1)$ time. Their update takes $O(1)$ time in each transformation. Each f-node has

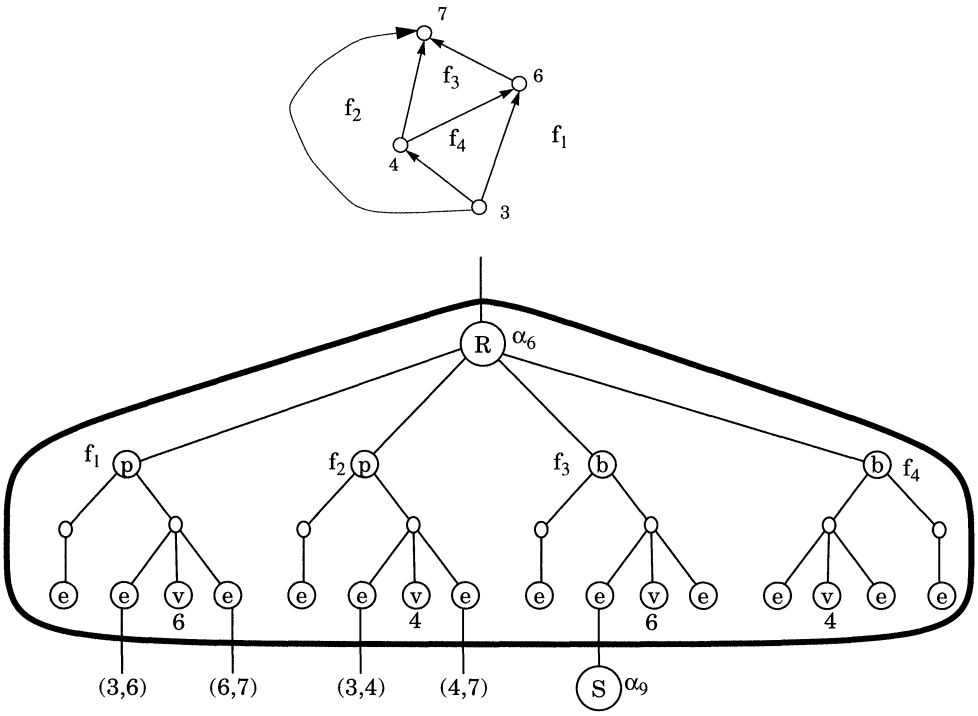

FIG. 18. *Portion of tree $\mathcal{T}^*$ representing the skeleton tree for node α_6 of the decomposition tree $\mathcal{T}$ shown in Fig. 4.*

a bidirectional pointer to the element of $\mathcal{E}$ storing the extreme pair of its associated face. Finally, we establish a pointer from each edge to its Q-node and two pointers from each vertex v to its two representative v-nodes in the boundary trees of the skeleton tree of *skeleton*(μ), where μ is the proper allocation node of v. Note that tree $\mathcal{T}$ can be obtained from tree $\mathcal{T}^*$ by contracting each skeleton tree into its root. We call the S-, P-, Q-, and R-nodes of $\mathcal{T}^*$ *primary nodes.*

6.4. Complexity analysis. In this section, we analyze the performance of the data structure described in §6.3.

LEMMA 6.3. *The above data structure for on-line planarity testing uses $O(n)$ space.*

Proof. The data structure uses space proportional to the total size of the skeletons stored at the nodes of $\mathcal{T}^*$. Thus, by Lemma 4.1, the space requirement is $O(n)$. □

We now consider operation *Test*.

LEMMA 6.4. *The above data structure supports operation Test in time $O(\log n)$.*

Proof. To find the proper allocation node of a vertex v, we access one of the two v-nodes η of v and find the closest primary node ancestor of η. This takes time $O(\log n)$.

To determine whether two objects x_1 and x_2 (each a vertex or an edge) are on the same face of a skeleton, we use Fact 8. Conditions 1 and 2 of Fact 8 are checked in $O(\log n)$ time by finding the closest f-node ancestors of the e-nodes of x_1 and x_2. Namely, x_1 and x_2 are both on face f if the f-node of f is the closest f-node ancestor for both an e-node of x_1 and an e-node of x_2. Condition 3 is verified by searching for the pair (x_1, x_2) in $\mathcal{E}$, again in $O(\log n)$ time.

To determine whether a vertex v is peripheral in the skeleton of its proper allocation node μ, we find the closest f-node ancestors of the two v-nodes of v and test if at least one of them is a p-node. This takes time $O(\log n)$.

To determine the first nonperipheral primary node κ on the path from a node μ to the root of $\mathcal{T}^*$, we find the closest b-node ζ ancestor of μ and then the closest primary node ancestor of ζ. This takes time $O(\log n)$. □

Operation *InsertVertex* is very simple to analyze. It takes worst-case time $O(\log n)$. In the rest of this section, we discuss the time complexity of operation *InsertEdge*.

In operation *InsertEdge*, we need to restructure the tree $\mathcal{T}^*$ and the dictionary $\mathcal{E}$. Each transformation (initial, elementary, or final) of the algorithm *InsertEdge* can be performed in $O(\log n)$ time by means of $O(1)$ link/cut and expand/contract operations on tree $\mathcal{T}^*$ and $O(1)$ updates (insertions or deletions) of $\mathcal{E}$. Hence the time complexity of operation *InsertEdge* is $O((1+T)\log n)$, where T is the number of transformations performed.

THEOREM 6.5. *The amortized time complexity of operation InsertEdge over a sequence of update operations is* $O(\log n)$.

Proof. Let R, S, and P denote the sets of R-, S-, and P-nodes of $\mathcal{T}$, respectively, and let $\deg(\mu)$ denote the number of children of node μ. We define the following *potential function* associated with the data structure:

$$\Phi = |R| + \sum_{\mu \in S \cup P} \deg(\mu).$$

Define the amortized number of transformations performed by *InsertEdge* as $A = T + \Delta\Phi$, where $\Delta\Phi$ is the variation of potential. An RR-transformation decreases by one the number of R-nodes and does not change the degrees of S- and P-nodes. An RP-transformation does not change the number of R-nodes and decreases by one the sum of the degrees of S- and P-nodes. An RP-transformation does not change the number of R-nodes and decreases by one the sum of the degrees of S- and P-nodes. The initial and final transformations in *InsertEdge* change the potential by a constant. Hence, we conclude that $A = O(1)$. Since $|\Phi| = O(n)$, the total time complexity of a sequence of n update operations starting from a graph with $O(1)$ vertices is $O(n \log n)$. □

We conclude the following.

THEOREM 6.6. *There exists a data structure for on-line planarity testing of a biconnected planar graph G whose current number of vertices is n with the following performance: the space requirement is $O(n)$; operations Test and InsertVertex take worst-case time $O(\log n)$, and operation InsertEdge takes amortized time $O(\log n)$.*

Proof. The space and time complexity bounds follow immediately from Lemmas 6.3 and 6.4 and Theorem 6.5. □

7. Tests and updates in general graphs. In this section, we consider on-line planarity testing for general (nonbiconnected) planar graphs. We first consider connected graphs and then disconnected graphs.

7.1. Tests in connected graphs. We consider a connected planar graph G with n vertices. We use the data structure of the previous section for each block (biconnected component) of G and represent the relationship between blocks by means of the block-cutvertex tree.

The *block-cutvertex tree* of a connected graph G has a B-node for each block (biconnected component) of G, a C-node for each cutvertex of G, and edges connecting

each B-node μ to the C-nodes associated with the cutvertices in the block of μ (see, e.g., [28]). The block-cutvertex tree was previously used in [53, 60] for maintaining biconnected components.

We construct an augmented block-cutvertex tree $\mathcal{B}$ for G as follows (see Fig. 19). We root $\mathcal{B}$ at an arbitrary B-node. Next, we add n new leaf nodes, called V-nodes, to $\mathcal{B}$, each associated with a vertex of G. The parent of the V-node representing vertex v it is the C-node associated with v if v is a cutvertex, and is the B-node associated with the unique block containing v otherwise. The number of nodes of $\mathcal{B}$ is $O(n)$. We store at each B-node μ a secondary structure consisting of the data structure of the previous section for on-line planarity testing in the block B of μ.

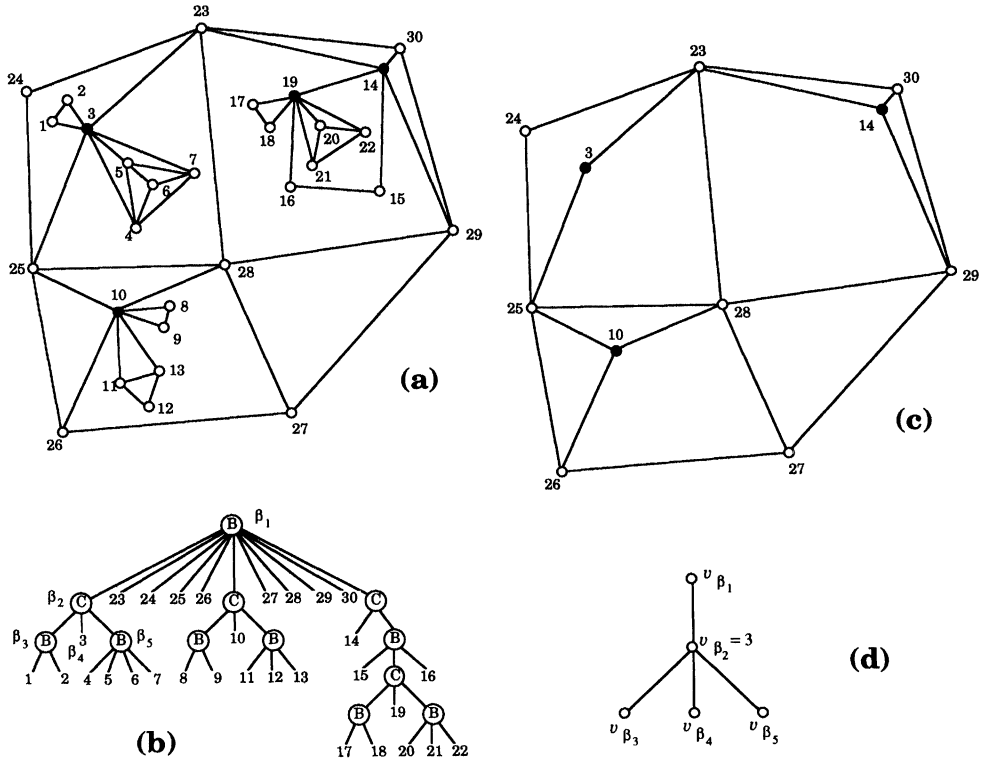

FIG. 19. (a) *A 1-connected graph G.* (b) *The augmented block-cutvertex tree $\mathcal{B}$ of G.* (c) *skeleton*(β_1). (d) *skeleton*(β_2).

The following definitions are analogous to those given for the decomposition tree in §4.

We define graph *skeleton*(μ) for a node μ of $\mathcal{B}$ as follows (see Fig. 19):

- If μ is a V-node representing vertex v, then *skeleton*(μ) consists of a single vertex v.
- If μ is a B-node, then *skeleton*(μ) is the block corresponding to μ. Each child ν of μ is a V- or C-node and hence uniquely associated with a vertex v. The virtual vertex of ν in *skeleton*(μ) is v.
- If μ is a C-node, let c be the cutvertex associated with μ and k be the number of children of μ in $\mathcal{B}$; *skeleton*(μ) is a "star" tree with $k+2$ vertices, where the center vertex is c and the other vertices are the virtual vertices of the

children of μ, plus a vertex representing the parent of μ.

Note that each child of μ is uniquely associated with a vertex of *skeleton*(μ). It is easy to see that two vertices are in the same block if and only if the path in $\mathcal{B}$ between their V-nodes has exactly one B-node.

The *pertinent graph* G_μ of a node μ is *skeleton*(μ) if μ is a V-node, and it is the union of all the blocks of the B-nodes in the subtree of $\mathcal{B}$ rooted at μ otherwise. The *pivot* of a B-node μ distinct from the root is the cutvertex whose C-node is the parent of μ. The pivot of the C-node of a cutvertex c is c itself. In the example of Fig. 19, vertex 3 is the the pivot of nodes β_2, β_3, β_4, and β_5.

The *expansion graph* of a vertex v of *skeleton*(μ) is the pertinent graph of the child of μ with virtual vertex v, or it is v itself if no such child exists. Observe that if v is not a a cutvertex of G, then its expansion graph is v itself.

A vertex v is *pivotal* in the pertinent graph G_μ of a node μ if it appears in the same face of the pivot of μ in some embedding of G_μ. We say that a node ν is *pivotal* if the pivot of ν is pivotal in the pertinent graph of the parent of ν. Note that a child of a C-node is always pivotal. In the example of Fig. 19, the V-node of vertex 5 is pivotal while the V-node of vertex 6 is not pivotal.

Let v be a vertex of the pertinent graph G_μ of a node μ of $\mathcal{B}$. The *representative* of v in the skeleton of μ is the vertex x of *skeleton*(μ) defined as follows: if v is in *skeleton*(μ), then $x = v$; otherwise, x is the vertex of *skeleton*(μ) whose expansion graph contains v. In the example of Fig. 19, the representative of vertex 5 is vertex 3 in *skeleton*(β_1) and is vertex v_{β_5} in *skeleton*(β_2).

We now show how to perform operation *Test*(v_1, v_2) for vertices in distinct blocks (see Fig. 19).

THEOREM 7.1. *Let v_1 and v_2 be vertices of a connected planar graph G. There exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ if and only if there exists a node χ of the block-cutvertex tree $\mathcal{B}$ of G such that*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face of some embedding of skeleton(χ); and*
3. *v_1 and v_2 are pivotal vertices of the expansion graphs of x_1 and x_2, respectively.*

Proof. The proof is essentially the same as that of Theorem 4.6, except for the different meaning of the terminology. □

We provide now examples of application of Theorem 7.1 referring to the graph of Fig. 19. Regarding *Test*(1, 4), we have $v_1 = 1$, $v_2 = 4$, $\chi = \beta_2$, $x_1 = v_{\beta_3}$, and $x_2 = v_{\beta_5}$, so that all the conditions of Theorem 7.1 are verified. Regarding *Test*(5, 20), we have $v_1 = 5$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 3$, and $x_2 = 14$, and again all the conditions of the theorem are verified. Indeed, see in Fig. 20 how edge (5, 20) can be inserted. Regarding *Test*(6, 20), we have $v_1 = 6$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 3$, and $x_2 = 14$, and condition 3 is not verified since v_1 is not pivotal in the expansion graph of x_1. Finally, regarding *Test*(12, 20), we have $v_1 = 12$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 10$, and $x_2 = 14$, and condition 2 is not verified since x_1 and x_2 are not on the same face of some embedding of *skeleton*(χ).

The following lemma will be used to efficiently test condition 3.

LEMMA 7.2. *Vertex v is pivotal in G_χ if and only if the first nonpivotal node on the path of $\mathcal{B}$ from the V-node of v to the root is either a child of χ or a node of the path from χ to the root.*

It is interesting to observe the analogy between the concepts of peripheral (defined in §4) and pivotal. The proof of Lemma 7.2 is analogous to that of Lemma 4.5.

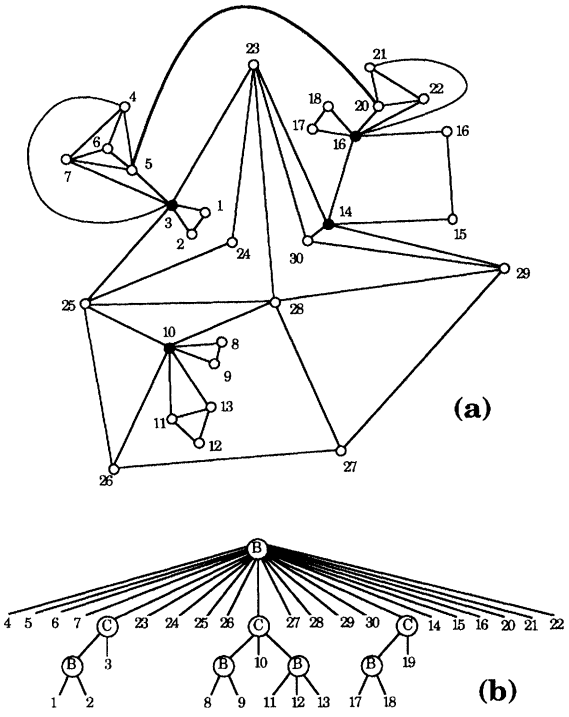

FIG. 20. *Example of operation InsertEdge for vertices in distinct blocks:* (a) *graph of Fig.* 19 *after performing operation InsertEdge*$(e, 5, 20)$; (b) *its augmented block-cutvertex tree.*

THEOREM 7.3. *Let G be a connected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(\log n)$ and can be constructed in $O(n)$ preprocessing time.*

Proof. Condition 1 of Theorem 7.1 is verified for at most two nodes of $\mathcal{B}$: always for the least common ancestor μ of the V-nodes of v_1 and v_2 and possibly for a child of μ. The latter case arises when v_1 is a cutvertex and v_2 is in a block whose B-node is a child of the C-node of v_1. Condition 2 is equivalent to performing operation *Test*(x_1, x_2) in *skeleton*(χ), which is either a biconnected graph or consists of a single vertex.

By Lemma 7.2, we can test condition 3 of Theorem 7.1 using for each vertex v a pointer to the first node η in the path from the V-node of v to the root of $\mathcal{B}$ such that v is not pivotal in G_η.

The time and space complexity bounds follow from Theorem 4.7 and from the fact that tree $\mathcal{B}$ has $O(n)$ nodes. □

COROLLARY 7.4. *Let G be a connected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(1)$ and can be constructed in $O(n^2)$ preprocessing time.*

7.2. Updates in connected graphs. We consider a connected planar graph G with n vertices. It is easy to see that operations *InsertEdge* and *AttachVertex* form a complete repertoire of operations for connected planar graphs.

LEMMA 7.5. *A connected planar graph G with n vertices and m edges can be assembled starting from a single vertex by means of m InsertEdge and AttachVertex*

operations, such that each intermediate graph is planar and connected. Also, such a sequence of operations can be determined in $O(n)$ time.

Proof. First, construct a spanning tree of G by means of $n-1$ *AttachVertex* operations, and then add the remaining edges with $m-n+1$ *InsertEdge* operations. □

Operation *AttachVertex*(e, v, u) is performed as follows. First, if u is a cutvertex, let β' be its C-node; otherwise, replace the V-node of u with a new C-node β' and a child V-node. Second, create a new B-node β'' (associated with the block consisting of the newly added edge e) with a child V-node (associated with the newly added vertex v) and make β'' a child of β'.

We now examine the structural changes of the block-cutvertex tree when operation *InsertEdge*(e, v_1, v_2) is performed on G. If v_1 and v_2 are in the same block B of G, then the primary structure of the block-cutvertex tree stays unchanged, and we process the insertion in the secondary structure (decomposition tree) of the B-node of B. Otherwise (see Fig. 20), let μ_1 and μ_2 be the V-nodes of v_1 and v_2, respectively, and let χ the least common ancestor of μ_1 and μ_2. The effect of *InsertEdge* is to merge the "old" blocks corresponding to the B-nodes of $\mathcal{B}$ on the paths of $\mathcal{B}$ from μ_1 to χ and from μ_2 to χ (inclusive) into a "new" block B'.

The primary structure of $\mathcal{B}$ is updated by means of a sequence of primitive tree operations (see §6.3). To update the secondary structure, we need to efficiently merge the decomposition trees of the old blocks into the decomposition tree of the new block B'. We reorient all the old blocks, except the largest one, denoted B^*. Avoiding the reorientation of B^* is the key to the efficient amortized behavior of the *InsertEdge* algorithm. Each old block distinct from B^* is reoriented into a planar st-graph with poles given by consecutive cutvertices in the chain from μ_1 to μ_2. By adding these orientations to the orientation of B^*, we obtain a planar st-orientation of the B' whose poles are the same as those of B^*. See a schematic example in Fig. 21.

The decomposition tree of the new block B' is obtained as follows. Let β_1 and β_2 be the nodes of $\mathcal{B}$ adjacent to the B-node of B^* in the path of $\mathcal{B}$ between the V-nodes of v_1 and v_2, with β_i on the side of v_i, $i = 1, 2$. Let u_i be the vertex associated with node β_i, $i = 1, 2$. (Note that β_i is either a C-node or a V-node.) We perform *InsertEdge*(e, u_1, u_2) in B^* and then replace the Q-node of e in the decomposition tree of B^* with an S-node whose subtrees are the newly built decomposition trees of the other old blocks. Note that one of u_1 and u_2 is the former pivot c^* of B^*.

In our dynamic environment, we maintain the forest $\mathcal{P}$, called *pivotal forest*, obtained from $\mathcal{B}$ by removing all the edges from nodes that are not pivotal to their parents. Hence the first nonpivotal node on the path of $\mathcal{B}$ from the V-node of v to the root (see Lemma 7.2) is the root of the tree of $\mathcal{P}$ containing the V-node of v.

To update the pivotal forest, we observe that the pivot c of the new block B' is the cutvertex parent of χ and is in general (when B^* is not the block of χ) different from the former pivot c^* of B^* (see Fig. 21). Let μ be the proper allocation node of c^* in the decomposition tree of B'. The new pivot c is in the pertinent graph of an edge of *skeleton*(μ) incident upon c^*. Such a pertinent graph is an orientation with poles u_1 and u_2 of the union of edge e and the old blocks except B^*. Hence we have that nodes of $\mathcal{B}$ can go from pivotal to nonpivotal but not vice versa.

To efficiently maintain the pivotal forest $\mathcal{P}$, we use the following auxiliary data structure. Consider a B-node of $\mathcal{P}$ associated to a block B, and let c be its pivot. Let $\mathcal{T}(B)$ be a new copy of the decomposition tree of B associated with a planar st-orientation of B with $t = c$. We modify $\mathcal{T}$ into a forest $\mathcal{P}^*(B)$ as follows:

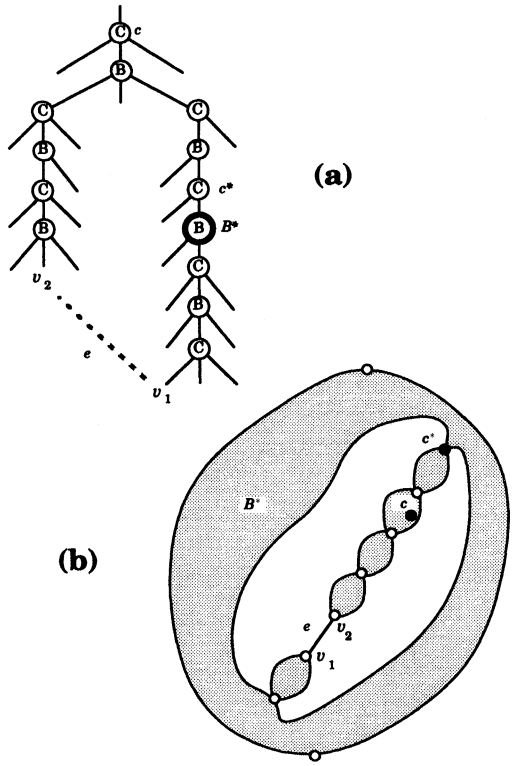

FIG. 21. *Schematic example of operation InsertEdge for vertices in distinct blocks:* (a) *augmented block-cutvertex tree*; (b) *reorientation of the blocks, except the largest one, denoted* B^*.

1. Let $\mathcal{P}^*(B) = \mathcal{T}(B)$.
2. Remove from $\mathcal{P}^*(B)$ the Q-nodes.
3. For each (remaining) node μ of $\mathcal{P}^*(B)$, if μ is nonperipheral and the virtual edge of μ is not in the same face as c in *skeleton*(ν), we remove the edges from μ to its parent ν.
4. For each vertex $u \neq c$, create a U-node μ. Let ν be the proper allocation node of u and (s_ν, t_ν) be the edge between the poles of ν. Make μ a child of ν if one of the following cases applies: (i) c is in *skeleton*(ν), and c and u are on a common face of *skeleton*(ν); or (ii) c is not in *skeleton*(ν), and (s_ν, t_ν) and u are on a common face of *skeleton*(ν).
5. For each allocation node ν of c, let $e_0 = (c, u_0), \ldots, e_{k-1} = (c, u_{k-1})$ be the edges incident on c in *skeleton*(ν) in clockwise order around c; μ_i be the child of ν whose virtual edge in *skeleton*(ν) is e_i; f_i $(i = 0, \ldots, k-1)$ be the face of *skeleton*(ν) containing edges e_i and $e_{(i+1) \bmod k}$; and E_i be the set of edges of f_i excluding e_i and $e_{(i+1) \bmod k}$.
   (a) Expand node ν into a node ν' with k new children $\nu_0, \ldots, \nu_{k-1}$.
   (b) For $i = 0, \ldots, k-1$, make μ_i and the U-node κ_i of u_i be children of ν'.
   (c) Let the order of the children of ν' from left to right be $\mu_0, \kappa_0, \nu_0, \ldots, \mu_{k-1}, \kappa_{k-1}, \nu_{k-1}$.
   (d) For $i = 0, \ldots, k-1$, make each former child of ν whose virtual edge is in E_i be a child of ν_i

(e) For $i = 0, \ldots, k-1$, make each U-node that is a former child of ν whose associated vertex u is in face f_i and is not adjacent to c a child of ν_i.
(f) For $i = 0, \ldots, k-1$, order the children of ν_i from left to right according to the clockwise sequence of vertices and edges in face f_i of *skeleton*(ν).

6. For each node ν that is not an allocation node of c, let f_0 and f_1 be the faces on the two sides of the edge between the poles of ν (s_ν, t_ν) in *skeleton*(ν), and let E_i $(i = 0, 1)$ be the set of edges of f_i excluding (s_ν, t_ν).
(a) Expand node ν into a node ν' with children ν_0 and ν_1.
(b) For $i = 0, 1$, make each former child of ν whose virtual edge is in E_i be a child of ν_i
(c) For $i = 0, 1$, make each U-node that is a former child of ν whose associated vertex u is in face f_i a child of ν_i.
(d) For $i = 0, 1$, order the children of ν_i from left to right according to the clockwise sequence of vertices and edges in face f_i of *skeleton*(ν).

We replace the B-node of block B in $\mathcal{B}$ with forest $\mathcal{P}^*(B)$ and identify each V- and C-node that is a former child of the B-node with the U-node of $\mathcal{P}^*(B)$ associated with the same vertex. We denote by $\mathcal{P}^*$ the resulting forest. The correspondence between $\mathcal{P}$ and $\mathcal{P}^*$ is given by the following lemma.

LEMMA 7.6. *Let ν be the B-node of $\mathcal{P}$ associated with a block B and μ be the child of ν associated with a vertex u of B. There is an edge in $\mathcal{P}$ between μ and ν (i.e., u is pivotal in B) if and only if there is a path in $\mathcal{P}^*(B)$ between the U-node of u and the root of $\mathcal{P}^*(B)$.*

Proof. The proof follows from Lemma 7.2 and the above construction. □

By Theorem 7.1 and Lemmas 7.2 and 7.6, forest $\mathcal{P}^*$ is equivalent to $\mathcal{P}$ regarding *Test* operations. We represent forest $\mathcal{P}^*$ with an edge-ordered dynamic tree [15]. Hence operation *Test* can be performed in $O(\log n)$ time.

Now we examine how to modify forest $\mathcal{P}^*$ in consequence of update operations on G. We consider operation *InsertEdge* first for vertices in the same block and then for vertices in distinct blocks. We omit the discussion of operations *InsertVertex* and *AttachVertex*.

When operation *InsertEdge* joins vertices in the same block B, the modifications of $\mathcal{P}^*$ are in exact correspondence with the transformations performed on the decomposition tree of B and take additional $O(\log n)$ time (amortized).

When operation *InsertEdge* joins vertices in distinct blocks, referring to the terminology developed earlier in this section, we construct $\mathcal{P}^*(B')$ by rebuilding $\mathcal{P}^*(B)$ for each old block B except the largest block B^* and by restructuring $\mathcal{P}^*(B^*)$ as follows:

1. Restructure $\mathcal{P}^*(B^*)$ in consequence of *InsertEdge*(e, u_1, u_2). Without loss of generality, assume that $c^* = u_1$.
2. Let T be the tree of $\mathcal{P}^*(B^*)$ containing the U-node κ_2 of u_2, and let ρ be the root of T.
3. Reroot T at node κ_2.
4. Perform local updates along the path of T from ρ to κ_2.
5. Link T to the the rest of the newly reconstructed $\mathcal{P}^*(B')$.

Let ℓ be the length of the path between ρ and κ_2, and let $\Delta_\mathcal{P}$ be the variation of the number of edges of $\mathcal{P}$ in consequence of *InsertEdge*. We have the following result.

LEMMA 7.7. *The restructuring of $\mathcal{P}^*$ takes time $O(\log n - \Delta_\mathcal{P} + \ell)$.*

Proof. The rerooting ot T and local updates are performed using a variation of operation *evert* of edge-ordered dynamic trees. This takes $O(\log n)$ time plus $O(-\Delta_\mathcal{P} + \ell)$

time to perform the local updates. □

We conclude this section with the amortized analysis of the time complexity of operation *InsertEdge* for vertices in distinct blocks. Let $\mathcal{O}$ be the set of old blocks. Denote by n_B the number of vertices of block B and by $\Delta_{\mathcal{B}}$ the variation of the number of nodes of $\mathcal{B}$ in consequence of *InsertEdge*. (Note that $\Delta_{\mathcal{B}} \geq -2 \cdot |\mathcal{O}| + 1$.) Let t^* be the time to perform *InsertEdge*(e, u_1, u_2) in B^*.

From the above discussion, we have that, with an appropriate choice of the time unit, the total time t for operation *InsertEdge* is

$$t = t^* + t_{\mathcal{B}} + t_{\mathcal{O}} + t_{\mathcal{P}} + \log n,$$

where

$$\begin{aligned} t_{\mathcal{O}} &= \sum_{B \in \mathcal{O} - \{B^*\}} n_B, \\ t_{\mathcal{B}} &= -\Delta_{\mathcal{B}} \cdot \log n, \\ t_{\mathcal{P}} &= -\Delta_{\mathcal{P}} + \ell. \end{aligned}$$

Namely, $t_{\mathcal{O}}$ is the time to reorient the old blocks and rebuild their secondary structures; $t_{\mathcal{B}}$ is the time to update the primary structure of $\mathcal{B}$ by means of primitive tree operations; and $t_{\mathcal{P}}$ is the time to update forest $\mathcal{P}^*$.

Let $\mathcal{A} = \bigcup_B \mathcal{A}_B$, where $\mathcal{A}_B$ is the set of allocation nodes of the cutvertex-c parent of B in $\mathcal{B}$, in the decomposition tree of B. Let $|\mathcal{P}|$ denote the total number of edges of $\mathcal{P}$. Let Φ_B be the potential of the data structure for block B, as given in the proof of Theorem 6.5. We define the potential Φ of the data structure as

$$\Phi = \sum_B \left[n_B \cdot \log \frac{1}{n_B} + (\Phi_B + a + 2) \cdot \log n \right] + |\mathcal{P}| + |\mathcal{A}|,$$

where the constant $a > 0$ denotes the maximum variation of potential of a block in consequence of an *InsertEdge* operation. (Recall that the proof of Theorem 6.5 shows that such variation is bounded by a positive constant.) Since $|\mathcal{P}| + |\mathcal{A}| = O(n)$, we have that $|\Phi| = O(n \log n)$.

Denoting by $\Delta_{\mathcal{A}}$ the variation of cardinality of $\mathcal{A}$, the variation $\Delta\Phi$ of potential in consequence of *InsertEdge* is given by

$$\Delta\Phi = \Delta\Phi_{\mathcal{O}} + \Delta\Phi_{\mathcal{B}} + \Delta\Phi_{\mathcal{P}},$$

where

$$\begin{aligned} \Delta\Phi_{\mathcal{O}} &= n_{B'} \cdot \log \frac{1}{n_{B'}} - \sum_{B \in \mathcal{O}} \left(n_B \cdot \log \frac{1}{n_B} \right), \\ \Delta\Phi_{\mathcal{B}} &= (\Phi_{B'} + a + 2) \cdot \log n - \sum_{B \in \mathcal{O}} (\Phi_B + a + 2) \cdot \log n, \\ \Delta\Phi_{\mathcal{P}} &= \Delta_{\mathcal{P}} + \Delta_{\mathcal{A}}. \end{aligned}$$

The following lemma is proved in [12]. Its proof is sketched here for the reader's convenience.

LEMMA 7.8 (see [12]). *Consider the function* $f(x) = x \log \frac{1}{x}$, *and let* $1 \leq x_1 \leq \cdots \leq x_k$. *We have*

$$f(x_1 + \cdots + x_k) - (f(x_1) + \cdots + f(x_k)) \leq -2(x_1 + \cdots + x_{k-1}).$$

Proof. This is a proof by induction on k. The base case ($k = 2$) is easy to prove by a simple analysis of the binary entropy function $h(x) = f(x) + f(1-x)$ for $0 < x < 1$. □

By Lemma 7.8, $t_{\mathcal{O}} + \Delta\Phi_{\mathcal{O}} \leq 0$. By Theorem 6.5, we have

$$t^* = -\Delta\Phi_{B^*} \cdot \log n + \log n$$

and

$$\Phi_{B'} \leq \sum_{B \in \mathcal{O}} (\Phi_B + a) + \Delta\Phi_{B^*}.$$

Thus

$$t^* \leq -\Phi_{B'} \cdot \log n + \sum_{B \in \mathcal{O}} (\Phi_B + a) \cdot \log n.$$

This implies that $t^* + t_{\mathcal{B}} + \Delta\Phi_{\mathcal{B}} = O(\log n)$. Finally, we have that $\ell \leq -\Delta_{\mathcal{A}}$, and therefore $t_{\mathcal{P}} + \Delta\Phi_{\mathcal{P}} \leq 0$. We conclude that $t + \Delta\Phi = O(\log n)$, so that the amortized time complexity of *InsertEdge* is $O(\log n)$.

7.3. Disconnected graphs. For graphs that are not connected, we complete our repertoire with operation *MakeVertex*, which can clearly be performed in $O(1)$ time. Let G be a general planar graph (possibly disconnected). We consider the *block-cutvertex forest* of G, which is the forest of the block-cutvertex trees of the connected components of G. When an *InsertEdge* operation joins two old connected components C' and C'' into a new connected component C, we rebuild the block-cutvertex tree of the smaller old component, so that it can be linked as a subtree of the block-cutvertex tree of the larger component.

Again, we perform an amortized analysis. Let n_C denote the size of the connected component C. The time for *InsertEdge* is $t = \log n + \min(n_{C'}, n_{C''})$. The potential of the data structure is defined as

$$\Phi = \sum_{\text{all components } C} \left(b \cdot n_C \cdot \log \frac{1}{n_C} + \Phi_C \right),$$

where $b > 1$ is a constant such that $|\mathcal{P}| + |\mathcal{A}| \leq b \cdot n$, and Φ_C is the potential of connected component C as defined in §7.2. We can immediately verify that joining C' and C'' affects the part of the potential associated with the size of $\mathcal{P}$ and $\mathcal{A}$ so that it increases by at most $b \cdot \min(n_{C'}, n_{C''})$. By Lemma 7.8, we conclude that $t + \Delta\Phi = O(\log n)$.

The above analysis concludes the proof of Theorem 1.1 stated in §1.

8. Some applications.

8.1. Graph planarization. Let G be a graph with n vertices and m edges. Given a set of weights on the edges of G, a *maximum-weight planar subgraph* of G is a planar subgraph of G with the maximum total edge weight. Finding a maximum-weight planar subgraph is NP-hard even if all the edges have unit weights [26]. Given an ordering $e_0, \ldots, e_{m-1}$ of the edges of G, each subgraph S of G is identified by an integer $k(S) = (b_{m-1} \cdots b_0)_2$ such that bit $b_i = 1$ if and only if edge e_i is in S. The *lexicographically maximum planar subgraph* of G with respect to the given

edge ordering is the planar subgraph S of G with maximum $k(S)$. Several heuristics have been developed for computing maximum-weight planar subgraphs; see, e.g., [14, 41]. There is experimental evidence that if the edges are sorted by decreasing weight, a lexicographically maximum planar subgraph provides a good approximation of a maximum-weight planar subgraph [14]. As a corollary of Theorem 1.1, we can construct a lexicographically maximum planar subgraph in $O(m \log n)$ time. The same result has been obtained in [7] with a different technique. A maximal planar subgraph of G can instead be constructed in $O(n+m)$ time [49] using the technique of [20].

8.2. Transitive closure. In this section, we study the maintenance of reachability information in planar *st*-graphs, an important class of planar digraphs that find several applications in computational geometry [27, 44, 45] and graph layout [10, 13]. Queries are of the following type: "Is there a directed path from v_1 to v_2?" This problem was previously best solved using a data structure for general digraphs with $O(n^2)$ space, $O(1)$ query time, and $O(n)$ amortized update time [32, 43].

However, this problem admits an efficient algorithm if we assume that the digraph is embedded and restrict the *InsertEdge* operation to join vertices on the same face of the embedding. That is, in the fixed-embedding problem, an edge that preserves planarity cannot be added if this requires a change of the embedding. A data structure for the fixed-embedding version of reachability in planar *st*-graphs is presented in [54]. The space requirement is $O(n)$, and the query/update time is $O(\log n)$ (worst-case). We show that the fixed-embedding restriction of the above technique can be removed by using the decomposition tree (see §4) to maintain a hierarchical representation of the embedding.

More formally, the *on-line reachability problem* for a planar *st*-graph G consists of performing on G a sequence of update operations *InsertEdge*(e, v_1, v_2) and *InsertVertex*(v_1, v_2), intermixed with queries of the following type:

Reachable(v_1, v_2): Determine whether there exists a directed path from v_1 to v_2.

It is shown in [54] that an embedded planar *st*-graph admits two total orderings of its vertices, edges, and faces, called *left-sequence* and *right-sequence*, such that there exists a path from v_1 to v_2 if and only if v_1 precedes v_2 in both the left- and right-sequences. The update of the left- and right sequences after an *InsertEdge* operation that preserves the embedding consists of a simple exchange of subsequences, so that it can be efficiently supported by means of concatenable queues. See [54] for details.

The extension of the technique of [54] to *InsertEdge* operations that arbitrarily modify the embedding is based on the following properties, whose proof is left to the reader.

LEMMA 8.1. *Let μ be a node of the decomposition tree of a planar st-graph G. The left-sequence (right-sequence) of the pertinent graph of μ can be obtained from the left-sequence (right-sequence) of skeleton(μ) by replacing each edge e with the left-sequence (right-sequence) of the expansion graph of e minus its poles.*

LEMMA 8.2. *Flipping the embedding of a planar st-graph around the poles exchanges the left-sequence with the right-sequence.*

We consider an (arbitrary) embedding of graph G, and we maintain two copies of the decomposition tree, denoted $\mathcal{T}_L$ and $\mathcal{T}_R$, which differ only in the order of the children at each node. In tree $\mathcal{T}_L$, the children of each node are ordered according to the left-sequence of the corresponding virtual edges. Tree $\mathcal{T}_R$ is similarly defined with respect to the right-sequence. There is a one-to-one correspondence between the nodes of $\mathcal{T}_L$ and $\mathcal{T}_R$. By Lemma 8.1, the left-to-right order of the Q-nodes of $\mathcal{T}_L$ ($\mathcal{T}_R$)

yields the subsequence of edges of the left-sequence (right-sequence). Since vertices are not explicitly described in our representation of the left- and right-sequences, we perform the query *Reachable*(v_1, v_2) by considering any edge e_1 incoming to v_1 and any edge e_2 outgoing from v_2, and we test whether e_1 precedes e_2 in both the left- and right-sequence. If v_1 has no incoming edges, then it must be the source of G, and hence it reaches v_2. A similar argument applies if v_2 has no outgoing edges. If *Reachable*$(v_1, v_2) = true$, a path from v_1 to v_2 can be reported in time $O(k)$ with a visit of the decomposition tree, where k is the path length.

When adding an edge to G, we may have to modify the embedding. This is done by means of the primitive topological operations reverse and swap; see Lemma 4.2. A reverse operation flips the embedding of the pertinent graph of a node around its poles. A swap operation restructures the embedding of the pertinent graph of a P-node μ by embedding the pertinent graph of a child of μ in a different position. By Lemma 8.2 the reverse operation corresponds to exchanging the subtrees of $\mathcal{T}_L$ and $\mathcal{T}_R$ rooted at the corresponding nodes. It is performed by two cuts followed by two links. The swap operation is performed by means of two link and cut operations at two corresponding P-nodes of $\mathcal{T}_L$ and $\mathcal{T}_R$.

After the embedding has been modified so that v_1 and v_2 are on the same face, the exchange of subsequences in the left-sequence and right-sequence caused by *InsertEdge*(e, v_1, v_2) is performed only at node χ associated with v_1 and v_2 (see *InsertEdge* in Algorithm 2), and can be done with $O(1)$ primitive tree operations. We represent $\mathcal{T}_L$ and $\mathcal{T}_R$ as ordered dynamic trees [15].

THEOREM 8.3. *Let G be a planar st-graph with n vertices. There exists an $O(n)$-space data structure for the on-line reachability problem in G that supports operations Reachable, InsertEdge, and InsertVertex in $O(\log n)$ time, where the bound is amortized for InsertEdge and worst-case for the other operations. Also, a directed path between two vertices can be reported in time $O(\log n + k)$, where k is the path length.*

8.3. Minimum spanning tree. In this section, we investigate the maintenance of a minimum spanning tree of a planar graph under weight changes and insertions of vertices and edges. Queries are of the following type: "Is edge e in the current minimum spanning tree?" The previous best result for this problem is an $O(m)$-space data structure for general graphs supporting queries in $O(1)$ time and updates in $O(\sqrt{m})$ time [21]. Also, this problem admits a more efficient algorithm if we assume that the graph is embedded and restrict the *InsertEdge* operation to join vertices on the same face of the embedding. Namely, $O(n)$ space and $O(\log n)$ query/update time (worst-case) can be achieved [15].

The *on-line minimum-spanning-tree problem* consists of maintaining the minimum spanning tree of a graph. First, we consider the case of a biconnected planar graph that is subject to a sequence of updates *InsertVertex* and *InsertEdge*, intermixed with the following operations:

InMst(e): Determine whether edge e belongs to the current minimum spanning tree.

Reweight(e, w): Set the weight of edge e equal to w.

The fixed-embedding technique of [15] is based on the fact that the edges of G not in the minimum spanning tree T dualize to a maximum spanning tree T^* of the dual graph G^*. The cocycle (partition of the vertices) induced by the deletion of edge e from T corresponds to the cycle induced by the insertion of e^* into T^*, and vice versa. Hence the dynamization of the minimum spanning tree can be done by representing both T and T^* by ordered dynamic trees that support the usual tree

operations plus queries on the minimum-/maximum-weight edge on the tree path between two vertices.

As previously described, the decomposition tree can be supplemented with embedding information so that the modifications of the embedding required by an *InsertEdge* operation are carried out in amortized time $O(\log n)$. Regarding the update of the dual tree T^*, we observe that the faces to the left and right of a pertinent graph are a separation pair of G^*. Hence T^* is updated in a reverse or swap operation by means of a sequence of $O(1)$ expand, cut, link, and contract operations performed at the nodes of T^* representing the faces to the left and right of the pertinent graph being flipped or moved. Figure 22 shows a schematic example of the update of T^* in a reverse operation.

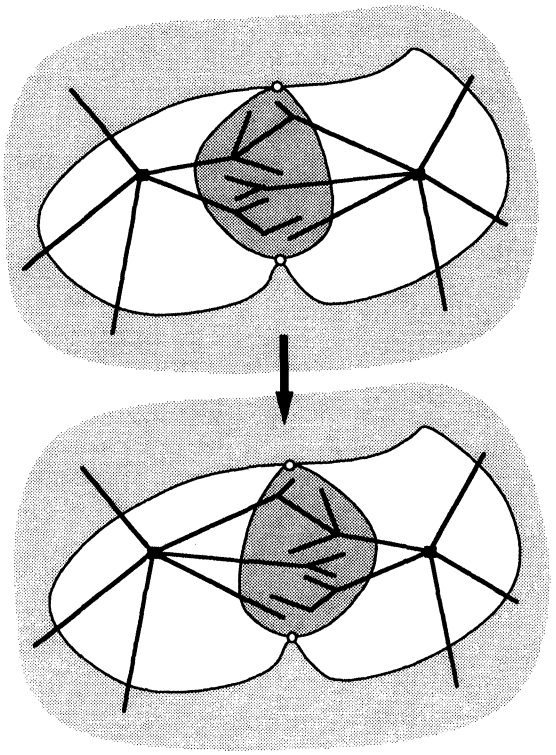

FIG. 22. *Schematic example of the update of the dual tree T^* in a reverse operation. The pertinent graph being reversed and the portion of T^* in it are shown. Dual nodes are represented by filled squares, and dual edges are represented by thick lines.*

THEOREM 8.4. *Let G be a planar biconnected graph with n vertices. There exists a data structure for the on-line minimum-spanning-tree problem that supports operation InMst in $O(1)$ time and operations InsertVertex, InsertEdge, and Reweight in $O(\log n)$ time. The time bound is amortized for InsertEdge and worst-case for the other operations.*

The minimum spanning tree of a nonbiconnected graph is the union of the minimum spanning trees of its blocks. Hence we have the following result.

THEOREM 8.5. *Let G be a planar graph with n vertices. There exists a data structure for maintaining on-line a minimum spanning forest of G that supports operation InMst in $O(1)$ time and operations InsertVertex, MakeVertex, InsertEdge, and*

Reweight in $O(\log n)$ time. The time bound is amortized for InsertEdge and worst-case for the other operations.

REFERENCES

[1] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F. K. Zadeck, *Incremental evaluation of computational circuits*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 32–42.

[2] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, *Incremental algorithms for minimal length paths*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 12–21.

[3] A. M. Berman, M. C. Paull, and B. G. Ryder, *Proving relative lower bounds for incremental algorithms*, Acta Inform., 27 (1990), pp. 665–683.

[4] D. Bienstock and C. L. Monma, *On the complexity of of covering vertices by faces in a planar graph*, SIAM J. Comput., 17 (1988), pp. 53–76.

[5] K. Booth and G. Lueker, *Testing for the consecutive ones property property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[6] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter, *A data structure for arc insertion and regular path finding*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 22–31.

[7] J. Cai, X. Han, and R. E. Tarjan, *An $O(m \log n)$-time algorithm for the maximal subgraph problem*, SIAM J. Comput., 22 (1993), pp 1142–1162.

[8] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, *A linear algorithm for embedding planar graphs using PQ-trees*, J. Comput. System Sci., 30 (1985), pp. 54–76.

[9] R. F. Cohen and R. Tamassia, *Dynamic expression trees and their applications*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1991, pp. 52–61.

[10] G. Di Battista and R. Tamassia, *Algoritms for plane representations of acyclic digraphs*, Theoret. Comput. Sci., 61 (1988), pp. 175–198.

[11] ———, *On-line graphs algorithms with SPQR-trees*, in Automata, Languages, and Programming (Proc. 17th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 442, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 598–611.

[12] ———, *On-line maintenance of triconnected components with SPQR-trees*, Algorithmica, 15 (1996), pp. 302–318.

[13] G. Di Battista, R. Tamassia, and I. G. Tollis, *Area requirement and symmetry display in drawing graphs*, in Proc. ACM Symposium on on Computational Geometry, Association for Computing Machinery, New York, 1989, pp. 51–60.

[14] P. Eades, L. Foulds, and J. Giffin, *An efficient heuristic for identifying a maximal weight planar subgraph*, in Combinatorial Mathematics IX, Lecture Notes in Math. 952, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 239–251.

[15] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, *Maintenance of a minimum spanning forest in a dynamic plane graph*, J. Algorithms, 13 (1992), pp. 33–54.

[16] S. Even and H. Gazit, *Updating distances in dynamic graphs*, Methods Oper. Res., 49 (1985), pp. 371–387.

[17] S. Even and Y. Shiloach, *An on-line edge deletion problem*, J. Assoc. Comput. Mach., 28 (1981), pp. 1–4.

[18] S. Even and R. E. Tarjan, *Computing an st-numbering*, Theoret. Comput. Sci., 2 (1976), pp. 339–344.

[19] H. de Fraysseix, J. Pach, and R. Pollack, *Small sets supporting Fary embeddings of planar graphs*, in Proc. 20th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 426–433.

[20] H. de Fraysseix and P. Rosenstiehl, *A depth-first-search characterization of planarity*, Ann. Discrete Math., 13 (1982), pp. 75–80.

[21] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[22] ———, *Fast algoritms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[23] ———, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning*

trees, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp 632–641.

[24] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for edge-connectivity problems*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 317–327.

[25] ———, *Maintaining biconnected components of dynamic planar graphs*, in Automata, Languages, and Programming (Proc. 18th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 510, Springer-Verlag, Berlin, New York, Heidelberg, 1991, pp. 339–350.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[27] M. T. Goodrich and R. Tamassia, *Dynamic trees and dynamic point location*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 523–533.

[28] F. Harary, *Graph Theory*, Addison–Wesley, Reading, MA, 1969.

[29] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338-355.

[30] J. Hopcroft and R. E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.

[31] ———, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[32] G. F. Italiano, *Amortized efficiency of a path retrieval data structure*, Theoret. Comput. Sci., 48 (1986), pp. 273–281.

[33] ———, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.

[34] G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, *Dynamic data structures for series-parallel graphs*, in Algorithms and Data Structures (Proc. 1989 WADS), Lecture Notes in Comput. Sci. 382, Springer-Verlag, Berlin, New York, Heidelberg, 1989, pp. 352–372.

[35] A. Kanevsky, R. Tamassia, J. Chen, and G. Di Battista, *On-line maintenance of the four-connected components of a graph*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 793–801.

[36] M.-Y. Kao and P. N. Klein, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proc. 22nd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 181–192.

[37] P. N. Klein and J. H. Reif, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[38] A. Lempel, S. Even, and I. Cederbaum, *An algorithm for planarity testing of graphs*, in Theory of Graphs, International Symposium, Gordon and Breach, New York, 1967, pp. 215–232.

[39] C. C. Lin and R. C. Chang, *On the dynamic shortest path problem*, in Proc. International Workshop on Discrete Algorithms and Complexity, 1989, pp. 203–212.

[40] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Ann. Discrete Math. 32, North–Holland, Amsterdam, 1988.

[41] T. Ozawa and H. Takahashi, *A graph-planarization algorithm and its applications to random graphs*, in Graph Theory and Algorithms, Lecture Notes in Comput. Sci. 108, Springer-Verlag, Berlin, New York, Heidelberg, 1981, pp. 95–107.

[42] J. A. La Poutré, *Dynamic graph algorithms and data structures*, Ph.D. thesis, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1991.

[43] J. A. La Poutré and J. van Leeuwen, *Maintenance of transitive closures and transitive reductions of graphs*, in Graph-Theoretic Concepts in Computer Science (Proc. 1987 WG), Lecture Notes in Comput. Sci. 314, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 106–120.

[44] F. P. Preparata and R. Tamassia, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[45] ———, *Efficient point location in a convex spatial cell complex*, SIAM J. Comput., 21 (1992), pp. 267–280.

[46] V. Ramachandran and J. H. Reif, *An optimal parallel algorithm for graph planarity*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 282–293.

[47] J. H. Reif, *A topological approach to dynamic graph connectivity*, Inform. Process. Lett., 25 (1987), pp. 65–70.

[48] H. Rohnert, *A dynamization of the all-pairs least cost problem*, in Proc. 1985 Symposium on

Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, New York, Heidelberg, 1985, pp. 279–286.

[49] P. ROSENSTIEHL, personal communication.

[50] B. SCHIEBER AND U. VISHKIN, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[51] W. SCHNYDER, *Embedding planar graphs on the grid*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 138–148.

[52] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.

[53] R. TAMASSIA. *A dynamic data structure for planar graph embedding*, in Automata, Languages, and Programming (Proc. 15th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 317, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 576–590.

[54] R. TAMASSIA AND F. P. PREPARATA, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.

[55] R. TAMASSIA AND I. G. TOLLIS, *A unified approach to visibility representations of planar graphs*, Discrete Comput. Geom., 1 (1986), pp. 321–341.

[56] ———, *Dynamic reachability in planar digraphs with one source and one sink*, Theoret. Comput. Sci., 119 (1993), pp. 331–343.

[57] R. TAMASSIA AND J. S. VITTER, *Parallel transitive closure and point location in planar structures*, SIAM J. Comput., 20 (1991), pp. 708–725.

[58] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set-union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[59] R. E. TARJAN AND A. C.-C. YAO, *Storing a sparse table*, Comm. Assoc. Comput. Mach., 22 (1979), pp. 606–611.

[60] J. WESTBROOK AND R. E. TARJAN, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.

[61] H. WHITNEY, *Non-separable and planar graphs*, Trans. Amer. Math. Soc., 34 (1932), pp. 339–362.

[62] M. YANNAKAKIS, *Four pages are necessary and sufficient for planar graphs*, in Proc. 18th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 104–108.