JOURNAL OF ALGORITHMS **21,** 201–239 (1996)
ARTICLE NO. 0044

On-line Planar Graph Embedding*

Roberto Tamassia

Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910

Received May 5, 1989

We present a dynamic data structure for the incremental construction of a planar embedding of a planar graph. The data structure supports the following operations: (i) testing if a new edge can be added to the embedding without introducing crossing; and (ii) adding vertices and edges. The time complexity of each operation is $O(\log n)$ (amortized for edge insertion), and the memory space and preprocessing time are $O(n)$, where n is the current number of vertices of the graph. © 1996 Academic Press, Inc.

1. INTRODUCTION

Embedding a graph in the plane is a fundamental problem in several areas of computer science, including circuit layout, graphics, and computer-aided design. The problems of testing the planarity and of constructing a planar embedding of a graph have been extensively studied in the past years, and the development of linear time algorithms for them has brought significant advances in algorithm design and analysis [4, 24, 33, 46].

In this paper we consider the problem of incrementally constructing a planar embedding of a graph. We investigate a dynamic data structure that

*A preliminary version of this paper has been presented at the ''15th International Colloquium on Automata, Languages and Programming,'' Tampere, Finland, 1988. Work performed in part while the author was with the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Research supported in part by the National Science Foundation under Grants ECS-84-10902 and CCR-9007851, by the U.S. Army Research Office under Grant DAAL03-91-G-0035, by the ''Office of Naval Research and the Advanced Research Projects Agency under Contract N00014-91-J-4052, ARPA Order 8225, and by Cadre Technologies, Inc.

allows us to perform efficiently the following operations:

- *queries*: given two vertices u and v, determine whether there is a face of the current embedding whose boundary contains both u and v;

- *updates*: modify on-line the current embedding by adding and/or removing vertices and edges.

The performance of such a data structure will be measured in terms of the *space* requirement, the *query* and *update* times, and the *preprocessing* time.

The development of dynamic algorithms and data structures is a challenging area of research that has attracted increasing interest in the past years. While considerable progress has been made in dynamic computational geometry (see, e.g., [7]), considerably fewer results exist on the dynamization of graph algorithms. Existing work is of a preliminary nature and limited to connectivity [23, 53], minimum spanning tree [25], transitive closure [35, 36, 45, 59, 69], and shortest path [54]. In fact, in several of the above data structures the capability of handling update operations is limited, and the space/time performance appears to be far from optimal.

Formally, our problem can be defined as follows: Let G be a planar graph embedded in the plane, referred to henceforth as a *plane graph*. We assume that G is connected and has no multiple edges, and we denote with n the number of vertices of G. By Euler's formula, G has $O(n)$ edges. We consider the *incremental embedding problem*, which consists of performing the following operations on G:

test(u, v): Test whether there is a face f that has both vertices u and v on its boundary. If such a face exists, output its name.

insert-edge$(e, u, v, f; f_1, f_2)$: Add edge $e = (u, v)$ inside face f, which is decomposed into faces f_1 and f_2. Vertices u and v must both be on the boundary of face f.

insert-vertex$(e, v; e_1, e_2)$: Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$, by adding vertex v.

attach-vertex(e, v, u, f): Add vertex v and edge (u, v) inside face f.

The incremental embedding problem naturally arises in several applications. For example, in a graph drawing system, one may want to add a new edge without introducing crossings (see, e.g., [48, 58]); in a CAD system for circuit layout, one may want to route a wire between two terminals without crossing other wires; and in a motion planning system, one may want to determine the existence of a path that avoids obstacles (here, the edges of the graph represent obstacles).

We present a data structure for the incremental embedding problem that uses $O(n)$ space and supports operations **test**, **insert-edge**, **insert-vertex**, and **attach-vertex** in $O(\log n)$ time, amortized for **insert-edge**, and worst-case for the other operations. In addition to the good space/time performance from a theoretical viewpoint, our data structure is also practical and easy to implement, and therefore suited for real-world applications.

We also consider the *dynamic embedding problem*, where the following "deletion" operations are also considered:

remove-edge$(e, u, v, f_1, f_2; f)$: Remove the edge $e = (u, v)$, and merge faces f_1 and f_2 formerly on the two sides of e into face f.

remove-vertex$(e_1, e_2, v; e)$: Let v be a vertex of degree two. Remove v and replace its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$ with edge $e = (u, w)$.

detach-vertex(e, v, u, f): Remove the degree-1 vertex v and its incident edge $e = (u, v)$, which lie in face f.

We extend our results for the incremental embedding problem to a restricted version of the dynamic embedding problem, called *hierarchical embedding problem*, where deletions are allowed only if they "undo" an **insert-edge** or **insert-vertex** operation performed in the past (not just the last operation).

These results are obtained by maintaining on-line an orientation of the graph, called *spherical st-orientation* and exploiting the partial order among the vertices, edges, and faces induced by this orientation. Besides its relevance to this problem, the concept of spherical *st*-orientation is of theoretical interest in its own right, and extends the results on *bipolar orientations* and *cylindric orientations* of planar graphs presented in [55, 60−62].

The problem of testing whether two vertices are on the same face can be considered a topological version of the *point-location* problem in the plane [50]. While our results show that "topological location" can be efficiently dynamized, it is an outstanding open problem to devise a dynamic data structure for point-location that uses linear space and supports query and update operations in logarithmic time. Recent results on dynamic point-location that come close this goal are presented in [2, 5, 6, 28, 51].

This work constitutes also a first step toward the development of an efficient data structure for the *dynamic planarity testing problem*, which 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 and removing vertices and edges. The related *incremental planarity testing problem* restricts the updates to insertions of vertices and edges.

The rest of this paper is organized as follows: Definitions and preliminary results are given in Section 2. Section 3 deals with orientations of planar graphs. The static *topological location problem*, which consists of performing only operation **test** on a fixed embedding, is studied in Section 4. Section 5 describes the full-fledged data structure for the incremental embedding problem. Deletions and the hierarchical embedding problem are studied in Section 6. Finally, further extensions are discussed in Section 7.

Note. After the presentation of a preliminary version of this paper [57], there has been considerable progress on dynamic graph algorithms (see, e.g., [1, 10–13, 20, 21, 26, 27, 31, 37, 38, 41–43, 52, 63]). In particular, efficient solutions have been given for the dynamic and incremental planarity testing problem.

Regarding incremental planarity testing, Di Battista and Tamassia [15–17] present a data structure with $O(\log n)$ test and update times (amortized for edge insertion), and Westbrook [67] shows that that a sequence of k test and update operations can be performed in expected amortized time $O(k\alpha(k,n))$. Very recently, La Poutré obtained $O(k\alpha(k,n))$ deterministic time [44]. Variations of the data structure given in this paper are used as components of the data structures developed in [15, 16, 67].

Regarding dynamic planarity testing, the best result is by Eppstein *et al.* [19], who provide a data structure that supports edge insertions and deletions, and queries that test whether the graph is currently planar in $O(\sqrt{n})$ amortized time.

Very recently, Italiano *et al.* [39] presented a data structure for dynamic planar embedding with $O(\log^2 n)$ test and update time.

The results of Theorems 5 and 12 have been improved by Westbrook and Tarjan [68], who show how to maintain on-line the biconnected components of a general graph under insertions of vertices and edges, such that a sequence of k test and update operations takes time $O(k\alpha(k,n))$.

2. PRELIMINARIES

We consider only finite connected graphs without self-loops and multiple edges. For the basic terminology about graphs and planarity, see [3, 22, 49]. Unless otherwise specified, paths and cycles of a directed graph are assumed to be directed. Also, if w is a vertex of a directed graph, $\mathbf{deg}^{+}(w)$ and $\mathbf{deg}^{-}(w)$ denote the *outdegree* and *indegree* of w, respectively.

First, we recall some definitions on graph connectivity. A *cutvertex* of a graph G is a vertex whose removal disconnects G. Graph G is said to be

2-connected if it has no cutvertices, and *1-connected* otherwise. A *block* of a 1-connected graph G is a maximal 2-connected subgraph of G. The *proper* vertices of a block B of G are the vertices of B that are not cutvertices. The *block-cutvertex tree* of G is a tree whose nodes represent the blocks and cutvertices of G, and whose edges connect each cutvertex v to the blocks that contain v (see Fig. 1). A graph G is *st-2-connectible* if adding the edge (s, t) to G makes G 2-connected [46]. Clearly, a 2-connected graph is also *st*-2-connectible for every pair of vertices s and t. An *st*-numbering of a graph G with vertex set V is a bijection ξ: $V \to \{1, 2, \ldots, |V|\}$ such that every vertex $v \neq s, t$ has neighbors u and w with $\xi(u) < \xi(v) < \xi(w)$. A graph admits an *st*-numbering if and only if it is *st*-2-connectible [46].

Let G be a 2-connected plane graph. A *separating pair* of G is a pair of distinct vertices of G whose removal disconnects G. Graph G is *3-connected* if it has no separation pairs. A 3-connected planar graph has a unique embedding [22].

Regarding the dynamic embedding problem, we assume that the vertices, edges, and faces of the graph are identified by *names*, which are elements of an ordered set. For example, names can be integers, alphanumeric strings, or pairs of coordinates. The total order among names will be referred to as *alphabetic order*. Regarding the complexity analysis, we assume that a name uses $O(1)$ space, and that the alphabetic comparison between two names can be done in $O(1)$ time. Also, for generality, we

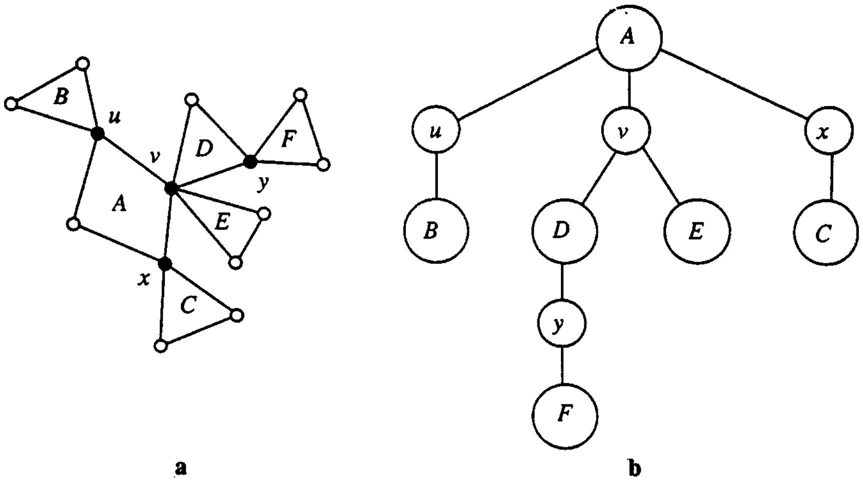

FIG. 1. (a) A 1-connected graph, and (b) its block-cutvertex tree.

assume that the query and update operations use the names as input parameters.

In the following, we denote with n the current number of edges of the graph being considered. Also, we denote the memory space by **space**(n) and the preprocessing time by **preprocess**(n). The time complexity of the various operations is denoted by **test**(n), **insert-edge**(n), **insert-vertex**(n), **remove-edge**(n), and **remove-vertex**(n). Finally, throughout this paper, $\log x$ means $\max\{1, \log_2 x\}$.

3. ORIENTATIONS OF PLANAR GRAPHS

A *spherical st-graph* is a plane digraph G with the following properties:

Property 1. G has exactly one source (vertex without incoming edges), s, and exactly one sink (vertex without outgoing edges), t.

Property 2. Every vertex v of G is on some directed simple path from s to t.

Property 3. Every directed cycle γ separates s from t; i.e., one of s and t is inside the region of the plane bounded by γ, and the other is outside.

We can visualize a spherical *st*-graph as embedded on the surface of a sphere, with s and t at the South and North poles, respectively (see Fig. 2).

The concept of spherical *st*-graph extends that of *planar st-graph* introduced in [46], which has important applications in the test of graph planarity [46] and the construction of planar drawings [14, 18, 55, 60]. A spherical *st*-graph differs from a planar *st*-graph because it admits (directed) cycles. However, such cycles must verify the aforementioned Property 3.

The following two lemmas show that spherical *st*-graphs have the same properties as planar *st*-graphs (see [60]) with regard to the circular sequences of edges that are incident upon a vertex and that form the boundary of a face (see Figs. 2b, 2c).

LEMMA 1. *For every vertex v of G, the incoming* (*outgoing*) *edges appear consecutively around v.*

Proof. Assume, for a contradiction, that there is a vertex v, $v \neq s, t$, for which the lemma is not true (see Fig. 3). Then there must be four edges incident upon v, denoted $e_1 = (w_1, v)$, $e_2 = (v, w_2)$, $e_3 = (w_3, v)$, and $e_4 = (v, w_4)$, which appear in this order counterclockwise around v. By Property 2, there are (directed) paths from s to w_1 and w_3. Let s' be the vertex farthest from s that is on both these paths. We denote with π_1 and π_3 the portions of such paths from s' to w_1 and w_3, respectively. The

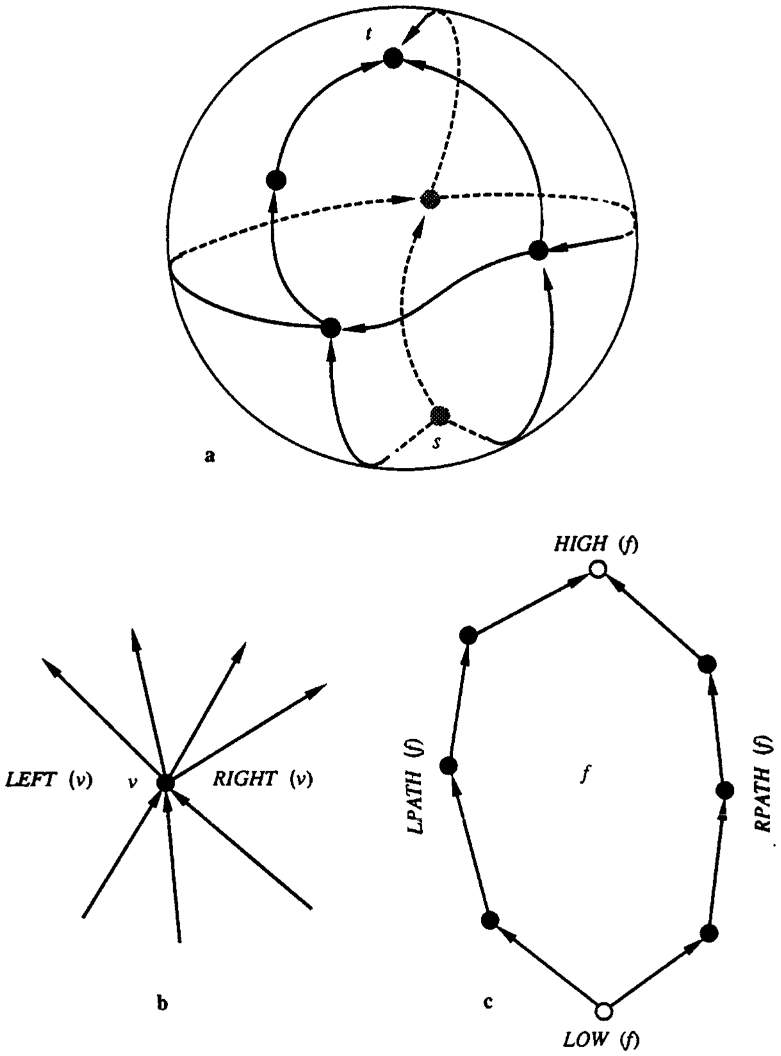

FIG. 2. (a) Example of spherical *st*-graph. (b) Example for Lemma 1. (c) Example for Lemma 2.

union of π_1, π_3, e_1, and e_3 forms an undirected cycle γ, which separates w_2 from w_4. The two regions of the plane delimited by cycle γ will be denoted by A and B, where A is the region that contains vertex w_2. We assume that both A and B contain cycle γ. By Property 2, there must be paths π_2 and π_4 from w_2 and w_4 to t, respectively. Now, we have four cases for the relative placement of s and t with respect to cycle γ: If both s and t are in A, then π_4 intersects γ at some vertex (see Fig. 3a). This creates a cycle that does not separate s from t. If s is in A and t is in B, then π_2 intersects γ, and again we have a cycle that does not separate s from t (see Figs. 3b, 3c). The cases when both s and t are in B, or s is in B and t is in A, are treated similarly. We conclude the proof by observing that in all cases we have a contradiction to Property 3. ▮

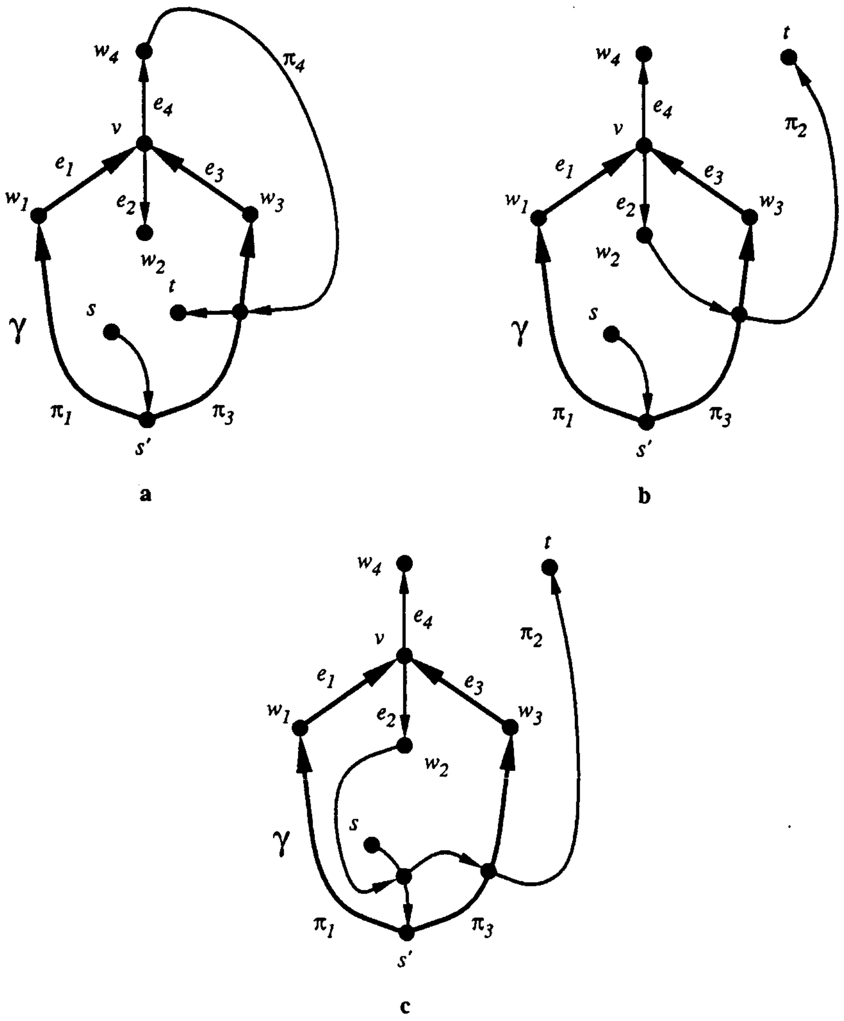

FIG. 3. Examples for the proof of Lemma 1: (a) s and t in A; (b, c) s in A and t in B.

LEMMA 2. *For every face f of G, the boundary of f consists of two directed paths with common origin and destination.*

Proof. Assume, for a contradiction, that there is a face f for which the lemma is not true (see Fig. 4). Then there are distinct vertices u and v on the boundary of f such that the edges of the boundary of f incident upon them are all outgoing. We denote these edges with $e_1 = (u, w_1)$, $e_2 = (u, w_2)$, $e_3 = (v, w_3)$, and $e_4 = (v, w_4)$, in counterclockwise order on the boundary of f. From Property 2, there are directed paths from s to u and v. Let s' be the vertex farthest from s that is on both these paths. We denote with π_u and π_v the portions of such paths from s' to u and v, respectively. Also, we denote by π' the portion of one of such paths from s to s'. The union of π_u, π_v, and the portion of the boundary of f clockwise from v to u forms an undirected cycle γ, which contains vertices w_2 and w_3. The two regions of the plane delimited by cycle γ will be denoted by A and B, where A is the region that does not contain face f. We assume that both A and B contain cycle γ. From Property 2, there must be paths π_1, π_2, π_3, and π_4 from w_1, w_2, w_3, and w_4 to t, respectively. Now, we have four cases for the relative placement of s and t with respect to cycle γ:

(i) s and t are both in A. Path π_1 must intersect at least one of π_u and π_v. If it intersects first π_u, then we have immediately a cycle that does not separate s from t (see Fig. 4a). Otherwise, let r be the intersection vertex of π_1 with π_v. Path π_4 must intersect either π_v, or the portion of π_1 from w_1 to r and then π_u. In both cases, we have again a cycle that does not separate s from t (see Figs. 4b, 4c).

(ii) s is in A and t is in B. Let π' be a path from s to s'. Path π_2 must intersect at least one of π_u and π_v. If it intersects first π_u at vertex r, then we have a cycle formed by edge $e_2(u, w_2)$, the subpath of π_2 from w_2 to r, and the subpath of π_u, from r to u. If this cycle does not separate s from t, we are done (see Fig. 4d). Otherwise, path π_2 must intersect π' at some vertex q and path π_3 must intersect the directed path consisting of the portion of π_2 from w_2 to q, the portion of π' from q to s', and path π_v. Hence, we have a cycle that does not separate s from t (see Fig. 4e). The last subcase to examine is when π_2 intersects first π_v. Let r be the first intersection of π_2 with π_v. Path π_3 must intersect the directed path consisting of the portion of π_2 from w_2 to r and the portion of path π_v from r to v. Hence, π_3 forms a cycle with the above path. If this cycle separates s from t, then π_2 must intersect π' at some vertex q and form a cycle that does not separate s from t (see Fig. 4f).

(iii) and (iv) s and t are both in B, or s is in B and t is in A. These cases are analogous to those above, and their treatment is omitted for brevity.

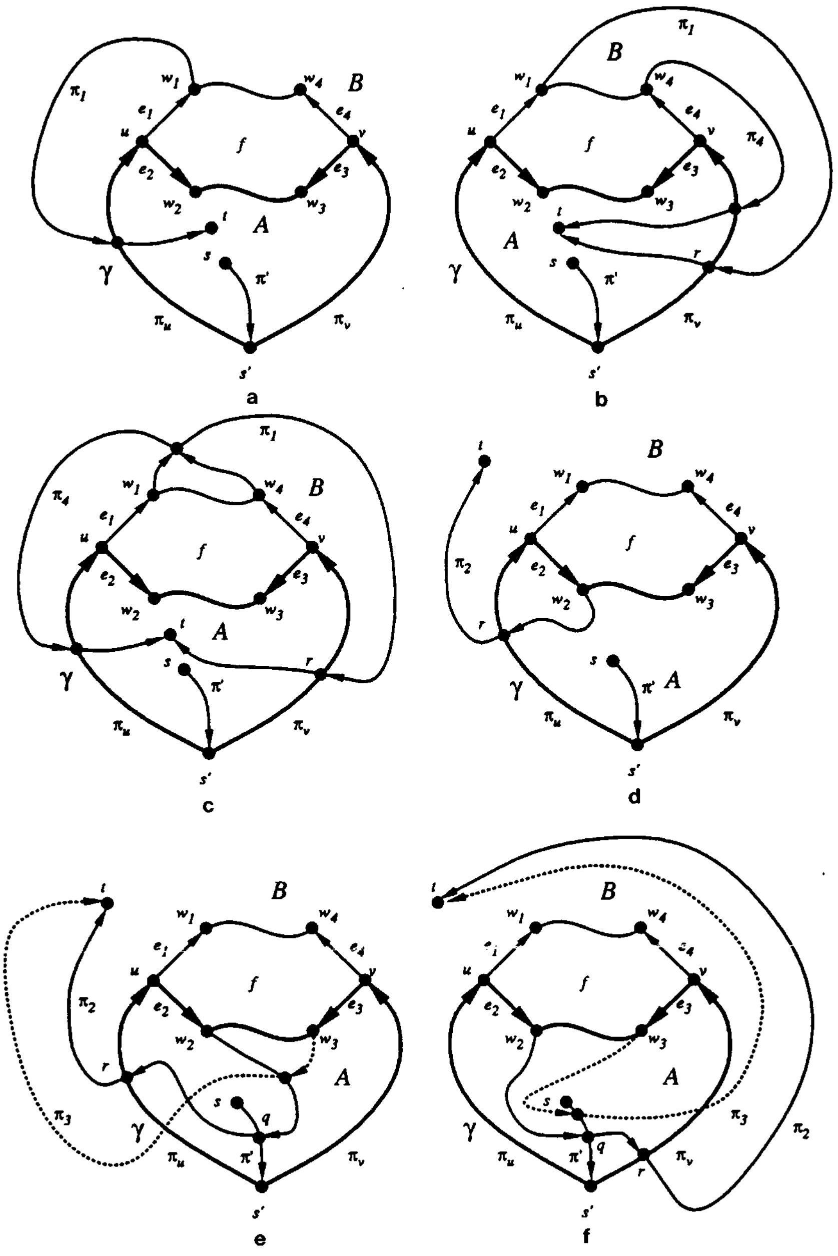

FIG. 4. Examples for the proof of Lemma 2: (a–c) Case (i). (d–f) Case (ii).

In all cases we have a contradiction to Property 3, and the proof is completed. ∎

Motivated by the previous lemmas, we introduce for the following terminology. Let V, E, and F denote the set of vertices, edges, and faces of G, respectively. For each element $x \in V \cup E \cup F$, we define vertices **low**(x) and **high**(x), and faces **left**(x) and **right**(x), as follows:

- If $x = v \in V$, we define **low**(v) = **high**$(v) = v$. Also, with reference to Lemma 1 and Fig. 2b, we denote by **left**(v) and **right**(v) the two faces that separate the incoming and outgoing edges of a vertex $v \neq s, t$, where **left**(v) is the face to the left of the leftmost incoming and outgoing edges, and **right**(v) is the face to the right of the rightmost incoming and outgoing edges.

- If $x = e \in E$, we define **low**(e) and **high**(e) as the tail and head vertices of e, respectively. Also, we denote by **left**(e) and **right**(e) the faces on the left and right side of e, respectively.

- If $x = f \in F$, we denote by **low**(f) and **high**(f) the two vertices that are the common origin and destination of the two paths forming the boundary of f (see Lemma 2 and Fig. 2c). Vertices **low**(f) and **high**(f) are called the *extreme* vertices of face f. Also, we define **left**(f) = **right**$(f) = f$. Finally, the two directed paths forming the boundary of f are called the *left path* and *right path* of f, respectively.

We assume that the left and right paths of f do not include their endpoints, so that these paths and the extreme vertices of f form a partition of the boundary of f. Note that a vertex v is on the left (respectively, right) path of face f if and only if **right**$(v) = f$ (respectively, **left**$(v) = f$). Hence, from **left**(v), **right**(v), **low**(f), and **high**(f) we can decide in $O(1)$ time whether vertex v is on the boundary of face f.

Let G be a plane graph, and s and t two distinct vertices of G. A *spherical st-orientation* of G is a spherical *st*-graph whose undirected version is isomorphic to G. Graph G is said to be *st-orientable* if it admits a spherical *st*-orientation. The following theorem provides a characterization of *st*-orientable graphs, and is similar to the characterization of *st*-numerable graphs given in [46].

THEOREM 1. *Let G be a plane graph with n vertices. The following statements are equivalent*:

1. *G is st-orientable*;
2. *G admits an acyclic spherical st-orientation*;

3. *G admits an st-numbering*;

4. *G is st-2-connectible*.

Also, there are $O(n)$-time algorithms for testing if G is st-orientable and constructing a spherical st-orientation for G.

Proof. It is proved in [46] that (4) ⇒ (3). Given an *st*-numbering for G, we can construct an acyclic spherical *st*-orientation by orienting each edge from the lowest to the highest numbered vertex. We have thus (3) ⇒ (2). Clearly, (2) ⇒ (1). To complete the proof of the characterization, we show that (1) ⇒ (4). Assume, for a contradiction, that G is not *st*-2-connectible. Then there is a cutvertex v of G such that one of the components generated by the removal of v, denoted by C, does not contain either s or t. Let u be a vertex of C. Any path from s to t through u is not simple, which is a contradiction.

The algorithm for testing if G is *st*-orientable consists of verifying that each cutvertex of G belongs to exactly two blocks (connected components) of G and that each block of G contains no more than two cutvertices. This takes $O(n)$ time. Finally, since computing the *st*-numbering of a planar graph can be done in $O(n)$ time [24], we also have the result that constructing a spherical *st*-orientation takes $O(n)$ time. ▮

Now, we turn our attention to operations that update a spherical *st*-graph by additions and deletions of vertices and edges. The operations **insert-edge**, **insert-vertex**, **remove-edge**, and **remove-vertex**, defined in the Introduction, are suitable for this purpose. However, further restrictions must be imposed on their applicability in order to ensure that the resulting graph is itself a spherical *st*-graph.

LEMMA 3. *Let G be a spherical st-graph, and G' be the plane digraph obtained by performing operation Π on G. Depending on Π, G' is a spherical st-graph if and only if*

1. *for* $\Pi =$ **insert-edge**$(e, u, v, f; f_1, f_2)$ *edge e must not create a (directed) cycle with the edges of face f*;

2. *for* $\Pi =$ **insert-vertex**$(e, v; e_1, e_2)$, *there is no restriction*;

3. *for* $\Pi =$ **remove-edge**$(e, u, v, f_1, f_2; f)$, *edge e must be such that* $f_1 \neq f_2$, $\mathbf{deg}^+(u) \geq 2$, *and* $\mathbf{deg}^-(v) \geq 2$;

4. *for* $\Pi =$ **remove-vertex**$(e_1, e_2, v; e)$, *v must be a (degree-2) vertex distinct from s and t*.

Proof. The proof of (2), (3), and (4) is straightforward. For operation **insert-edge**, we consider two cases. First, assume that there is a path π on the boundary of f from u to v. If edge $e = (u, v)$ creates a cycle that does not separate s from t, then by replacing e with π we have that G

already had a nonseparating cycle, a contradiction. Now, if there is no path on the boundary of f from u to v, we are in the situation shown in Fig. 5. Let γ be the cycle that does not separate s from t, and π_1 and π_2 be paths from s to **low**(f) and from **high**(f) to t, respectively. One of these two paths, say π_1, must intersect γ at some vertex x. This implies that G already had a nonseparating cycle, formed by the subpath of γ from v to x, the subpath of π_1 from x to **low**(f), and the subpath of the right path of f from **low**(f) to v. Again, we reach a contradiction. ▮

COROLLARY 1. *Let G be a spherical st-graph, and G_1 and G_2 be the graphs obtained by performing operations* **insert-edge**$(e, u, v, f, f_1; f_2)$ *and* **insert-edge**$(e, v, u, f, f_1; f_2)$ *on G, respectively, where both vertices u and v are on the boundary of face f. Then at least one of G_1 and G_2 is a spherical st-graph.*

Proof. By Lemma 3, we only have to ensure that the edge to be added does not form directed cycles with the boundary of f. Hence, if there is a (directed) path in the boundary of f from u to v, we perform operation **insert-edge**$(e, u, v, f, f_1; f_2)$, while if there is a (directed) path from v to u, we perform operation **insert-edge**$(e, v, u, f, f_1; f_2)$. Both operations are allowed when no directed path exists between u and v. ▮

It is a relatively simple exercise to show that the repertory of operations **insert-edge**, **insert-vertex**, **remove-edge**, and **remove-vertex** is com-

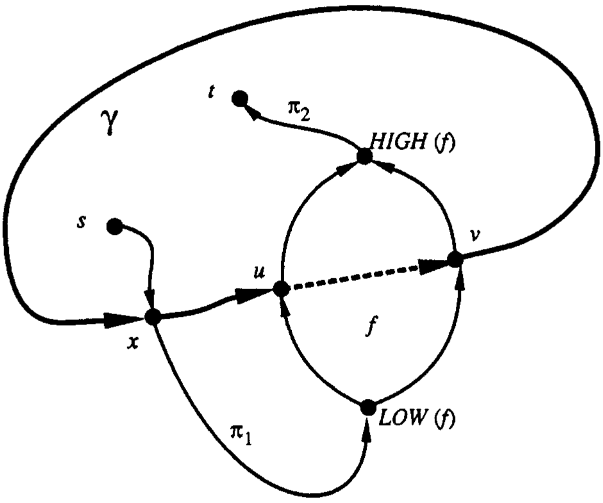

FIG. 5. Example for the proof of Lemma 3.

plete for the class of *st*-2-connectible planar graphs, as stated by the following theorem.

THEOREM 2. *Let* G_0 *be the trivial graph consisting of two vertices connected by an edge. Any st-2-connectible planar graph with m edges can be assembled from* G_0 *by means of* $m - 1$ **insert-edge** *and* **insert-vertex** *operations, and can be disassembled to yield* G_0 *by means of* $m - 1$ **remove-edge** *and* **remove-vertex** *operations.*

4. TOPOLOGICAL LOCATION

In this section we consider the *topological location problem*, which consists of performing efficiently operation **test** on a plane graph G. First, we present a geometric technique valid for 3-connected graphs, and then describe the more general approach based on spherical *st*-orientations. Notice that straightforward approaches to the topological location problem require either quadratic space (storing all possible answers in an $n \times n$ matrix) or linear time to perform **test**(u, v) (searching for vertex v in all the faces containing vertex u).

4.1. *A Geometric Construction*

If G is 3-connected, we can construct a *convex drawing* for it, i.e., a planar straight-line drawing such that the boundary of each face is a convex polygon. The existence of convex drawings for 3-connected planar graphs follows from Steinitz's theorem on convex polytopes in three dimensions [56] and was explicitly proved by Tutte [65, 66]. A linear-time algorithm for constructing convex drawings is presented in [9].

Given a convex drawing Γ of G, the point of the plane associated with a vertex u is denoted by $\Gamma(u)$, and the region associated with a face f of G is denoted by $\Gamma(f)$. We store with each vertex v the list of edges (v, w) incident upon v, sorted by the slope of the segment $\overline{\Gamma(v)\Gamma(w)}$. Using an array representation for such lists, we can compute in time $O(\log n)$ the internal regions $\Gamma(f)$ and $\Gamma(g)$ of Γ that contain the first and last portion of segment $\overline{\Gamma(u)\Gamma(v)}$. Because of convexity, u and v are on the same internal face if and only if $f = g$ (see Fig. 6). We obtain:

THEOREM 3. *There exists a data structure for the topological location problem on* 3*-connected planar graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n); \qquad \textbf{test}(n) = O(\log n).$$

A disadvantage of the above data structure is that the aforementioned algorithms for constructing a convex drawing Γ use real arithmetic for the

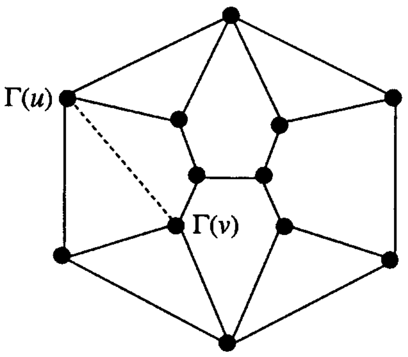

FIG. 6. Operation **test**(u, v) using a convex drawing.

computation of the vertex coordinates, and if integer coordinates are required, they might be exponential in n. Hence, Theorem 3 holds only in a model of computation where real numbers can be stored in $O(1)$ space and arithmetic operations on them take $O(1)$ time.

4.2. *st-Orientable Graphs*

Let G be a spherical *st*-graph. From Lemma 2, vertices u and v of G are on the boundary of face f if and only if one of the following cases occurs (see Fig. 7):

Case 1. Both u and v are nonextreme vertices of f, i.e.,

$$(f = \textbf{left}(u) \text{ or } f = \textbf{right}(u)) \quad \text{and} \quad (f = \textbf{left}(v) \text{ or } f = \textbf{right}(v));$$

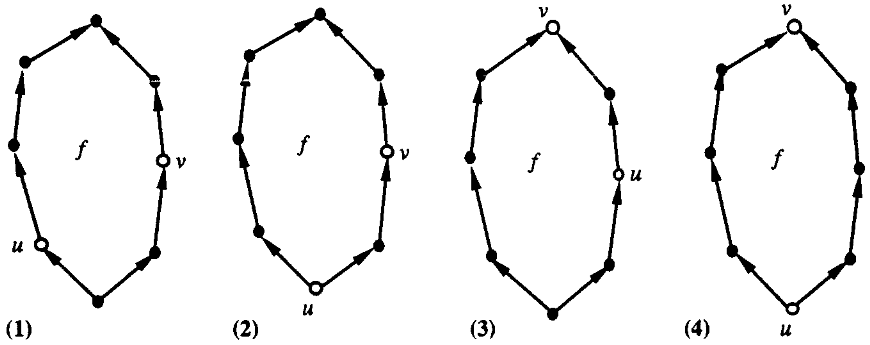

FIG. 7. The four cases for two vertices on the same face.

Case 2. u is an extreme vertex of f and v is not, i.e.,

$$(u = \mathbf{low}(f) \text{ or } u = \mathbf{high}(f)) \quad \text{and} \quad (f = \mathbf{left}(v) \text{ or } f = \mathbf{right}(v));$$

Case 3. v is an extreme vertex of f and u is not, i.e.,

$$(v = \mathbf{low}(f) \text{ or } v = \mathbf{high}(f)) \quad \text{and} \quad (f = \mathbf{left}(u) \text{ or } f = \mathbf{right}(u));$$

Case 4. Both u and v are extreme vertices of f, i.e.,

$$(u = \mathbf{low}(f) \text{ and } v = \mathbf{high}(f)) \quad \text{or} \quad (u = \mathbf{high}(f) \text{ and } v = \mathbf{low}(f)).$$

In order to check Cases 1–3, we store for each vertex v the faces $\mathbf{left}(v)$ and $\mathbf{right}(v)$, and for each face f the vertices $\mathbf{high}(f)$ and $\mathbf{low}(f)$. For each of these cases, the test is carried out in $O(1)$ time. To check the remaining Case 4, we store with each vertex v a search table $\mathbf{below}(v)$ whose elements are the pairs $(f, \mathbf{low}(f))$ such that $v = \mathbf{high}(f)$, sorted according to the alphabetic order of the name of $\mathbf{low}(f)$. Hence, the test for Case 4 consists of searching for vertex u in $\mathbf{below}(v)$ and for vertex v in $\mathbf{below}(u)$, which takes $O(\log n)$ time. This proves the following theorem:

THEOREM 4. *There exists a data structure for the topological location problem on spherical st-graphs with the performance*

$$\mathbf{space}(n) = \mathbf{preprocess}(n) = O(n); \qquad \mathbf{test}(n) = O(\log n).$$

COROLLARY 2. *There exists a data structure for the topological location problem on st-orientable graphs with the performance*

$$\mathbf{space}(n) = \mathbf{preprocess}(n) = O(n); \qquad \mathbf{test}(n) = O(\log n).$$

Proof. Construct a spherical *st*-orientation for G and apply Theorem 4. ∎

4.3. *General Plane Graphs*

For plane graphs that are not *st*-orientable, the algorithm of the preceding subsection cannot be directly applied. Instead, we will combine the above technique with a data structure that takes into account the plane arrangement of the blocks of the graph.

In the following, we will be interested in preprocessing a graph G in order to determine quickly whether there is a block that contains two given vertices u and v. This can be done efficiently by orienting the block-cutvertex tree T of G so that it becomes a rooted source tree with an arbitrarily selected block at the root. In the example of Fig. 1, the edges of the block-cutvertex tree are oriented from bottom to top. Also, we store with every vertex v a pointer **node**(v), where, if v is a cutvertex, **node**(v) points to the representative node of cutvertex v in T, while if v is a proper vertex of a block B, **node**(v) points to the representative node of block B in T. It is simple to verify that there is a block containing vertices u and v if and only if one of the following cases is verified:

1. **node**$(u) =$ **node**(v);
2. **node**(u) is the parent of **node**(v) in T;
3. **node**(v) is the parent of **node**(u) in T;
4. the parents of **node**(u) and **node**(v) are the same node of T, associated with a block;
5. **node**(u) is the grandparent of **node**(v) in T, and both u and v are cutvertices;
6. **node**(v) is the grandparent of **node**(u) in T, and both u and v are cutvertices.

Note that the above discussion applies also to nonplanar graphs. We conclude:

THEOREM 5. *Let G be a graph with n vertices and m edges. There exists a data structure with $O(n)$ space and $O(n + m)$ preprocessing time that allows to test if two vertices belong to the same block of G in $O(1)$ time.*

Now, let G^* be the dual graph of a plane graph G. A face of G that is a cutvertex of G^* is called a *cutface* (of G). The following simple lemma shows that there is a bijection between the blocks of G and the ones of G^*.

LEMMA 4 [30, p. 124, ex. 11.4]. *A subset of edges of G forms a block of G if and only if their duals form a block of G^*.*

We define now a partial order on the set of blocks, vertices, and faces of G. In the following definitions we adopt the convention that the faces of a block B of G have the same names as the corresponding faces in G.

Let B be a block of G. The *outer face* of B, denoted **outer**(B), is the face f whose boundary includes the external boundary of B. In turn, B is called an *inner block* of face **outer**(B). Note that **outer**(B) is either a cutface or the external face. Now, let f be a face distinct from the external face. We denote with **outer**(f) the block B such that f is an internal face of B. Finally, let v be a vertex. If there is a block B containing v such that v is not on the external boundary of B, then we define **outer**$(v) = B$.

Otherwise, there is a (unique) face f containing v such that, for no block B containing v, $B = \textbf{outer}(f)$, and we define $\textbf{outer}(v) = f$. As a straightforward consequence of the above definitions, we have:

LEMMA 5. *The graph of relation* **outer** *is a directed source tree, whose root is the external face.*

Proof. Given a planar straight-line drawing of the graph, consider the geometric containment relation among the blocks, vertices, and faces induced by the drawing, where we identify each block with its external boundary. This relation is a partial order, and includes the $\textbf{outer}(v)$ relation. Hence, the $\textbf{outer}(v)$ relation is acyclic. To complete the proof, we observe that every vertex, block, and face, except the external face, has a unique **outer** object. ▮

Figure 8 shows a 1-connected graph and the corresponding **outer** relation for the blocks, cutvertices, and cutfaces.

THEOREM 6. *Let u and v be vertices of G that are on the boundary of the same face f. If u and v belong to the same block B, then they are also on the boundary of face f in B. Otherwise, one of the following cases arises*:

1. $f = \textbf{outer}(u)$ *and* $f = \textbf{outer}(v)$;
2. $f = \textbf{outer}(u)$ *and v is on face f of block* $\textbf{outer}(f)$;
3. $f = \textbf{outer}(v)$ *and u is on face f of block* $\textbf{outer}(f)$.

Proof. A straightforward consequence of the definition of the **outer** relation. ▮

The data structure for the topological location problem in general plane graphs consists of:

1. A data structure to test whether two vertices belong to the same block, see Theorem 5.
2. A separate data structure for performing operation **test** in a 2-connected graph (see Corollary 2), for each block B of G.
3. **outer** pointers for the blocks, vertices, and faces of G.

From Theorem 6 and Corollary 2, we conclude:

THEOREM 7. *There exists a data structure for the topological location problem on plane graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n); \qquad \textbf{test}(n) = O(\log n).$$

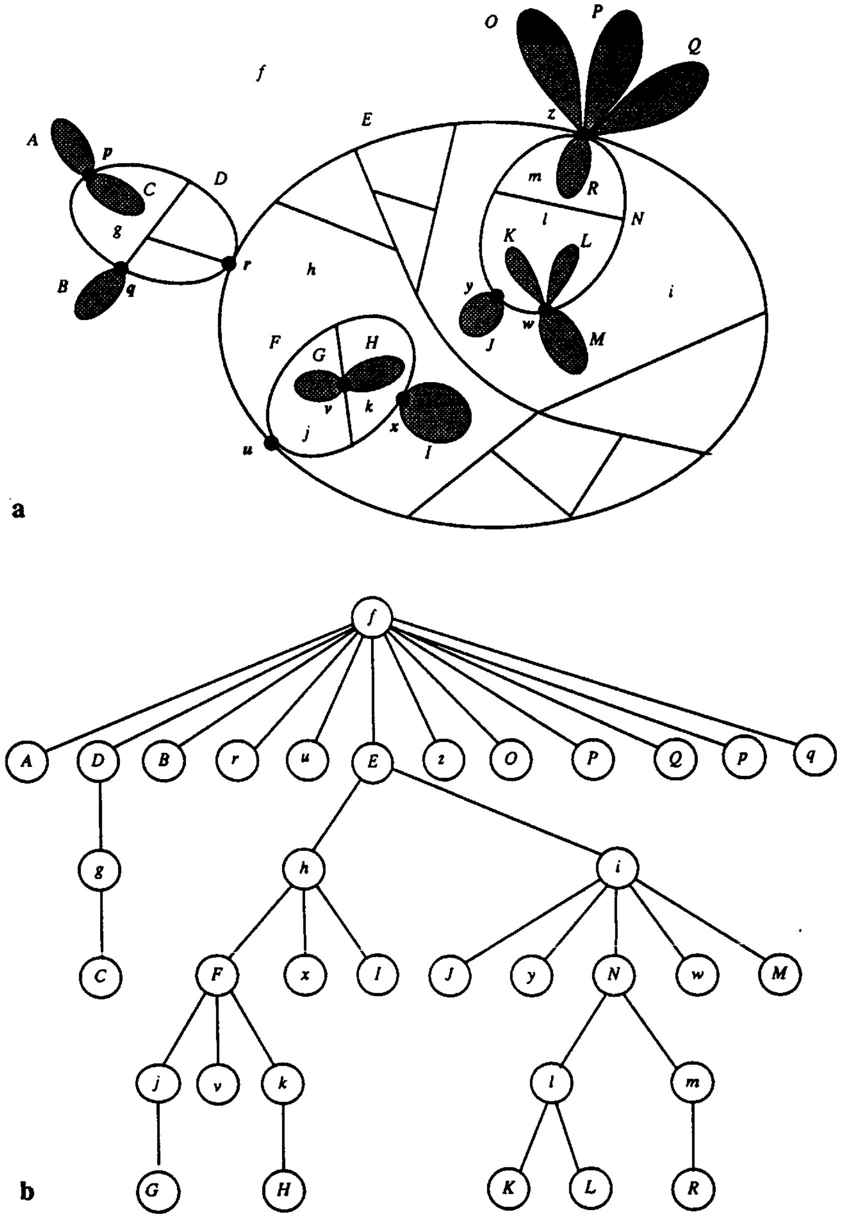

FIG. 8. (a) A 1-connected graph and (b) its **outer** relation. Uppercase letters denote blocks; lowercase letters $f \cdots m$ denote faces; and lowercase letters $p \cdots z$ denote vertices.

4.4. *Average Query Time*

Let $Q_{uv}(n)$ be the time to perform the query operation **test**(u, v). We have shown in the previous subsection that in the worst case $Q_{uv}(n) = O(\log n)$. Here, we consider the average query time over all possible queries, defined by

$$\overline{Q}(n) = \frac{1}{\binom{n}{2}} \sum_{u \neq v} Q_{uv}(n).$$

THEOREM 8. *In the data structure of Theorem 7 for the topological location problem, the average time for the* **test** *operation over all possible queries is* $O(1)$.

Proof. From the description of the algorithm for the **test** operation (see Section 4.2), we have that $Q_{uv}(n) = O(\log \mathbf{deg}^-(u) + \log \mathbf{deg}^-(v))$, where $\mathbf{deg}^-$ and $\mathbf{deg}^+$ refer to the spherical *st*-orientation. Hence, $\binom{n}{2}\overline{Q}(n) = O(n\sum_v \log \mathbf{deg}^-(v))$. The proof is completed observing that $\sum_v \mathbf{deg}^-(v) = O(n)$ and that $\sum_v \mathbf{deg}^-(v)$ is maximized when $\mathbf{deg}^-(v)$ is the same for all v distinct from the source. ▮

5. INCREMENTAL EMBEDDING

In this section, we present the complete data structure for the incremental embedding problem. First, we consider the dynamic embedding problem in spherical *st*-graphs, and then extend the results to undirected graphs using spherical *st*-orientations.

5.1. *Spherical st-Graphs*

In this subsection we describe a data structure for efficiently solving the dynamic embedding problem for spherical *st*-graphs. Let G be a spherical *st*-graph.

The data structure has a record for each vertex, edge, and face of G. The records for the vertices are arranged in a *vertex-tree* $\mathscr{V}$, which is a balanced search tree whose nodes are ordered according to the alphabetic order of the names of the vertices. Similarly, the records of the edges and faces are arranged in alphabetic order in a *face-tree* $\mathscr{F}$, and in an *edge-tree* $\mathscr{E}$, respectively. The above trees allow us to access in $O(\log n)$ time the records associated with the vertices, edges, and faces involved in the current operation.

The record for a face f stores the following information:

1. f: name of the face;
2. **high**(f): pointer to the record of the topmost vertex of f;
3. **low**(f): pointer to the record of the bottommost vertex of f;
4. pointers to two balanced search trees, **lpath**(f) and **rpath**(f), associated with the left and right paths of face f, respectively. The nodes of each such tree represent the vertices and edges of the corresponding path, and are sorted according to the direction of the path. The roots of **lpath**(f) and **rpath**(f) point back to the record of f.

The record for a vertex v stores the following information:

1. v: name of the vertex;
2. **deg**$^{+}(v)$, **deg**$^{-}(v)$: outdegree and indegree of v.
3. **pleft**(v): pointer to the representative of v in the tree **rpath**(f), where $f =$ **left**(v);
4. **pright**(v): pointer to the representative of v in the tree **lpath**(g), where $g =$ **right**(v);
5. **below**(v): pointer to a balanced search tree whose nodes store pointers to the records of the faces f such that $v =$ **high**(f). The nodes of **below**(v) are sorted according to the alphabetic order of the names of the vertices **low**(f).

The record for an edge e stores the following information:

1. e: name of the edge;
2. pointers to the records of the vertices **low**(e) and **high**(e);
3. pointers **pleft**(e) and **pright**(e) to the representatives of e in the trees **rpath**(f) and **lpath**(g), where $f =$ **left**(e) and $g =$ **right**(e).

We show in Fig. 9 a spherical *st*-graph and a fragment of the data structure for it.

Using the above data structure, the **test** operation can be performed with the same strategy as in the static case. The only difference is that now the faces to the left and right of u and v are not immediately available and must be retrieved by walking up to the roots of the trees that contain the representatives of u and v pointed to by **pleft**(u), **pright**(u), **pleft**(v), and **pright**(v).

With regard to the **insert-edge** operation, testing for its applicability can be done by a simple modification of the **test** algorithm.

Now, assume that f_1 is to the left of e and f_2 is to the right of e. We partition the left path of f into subpaths L_1, L_2, and L_3, where L_2 is the

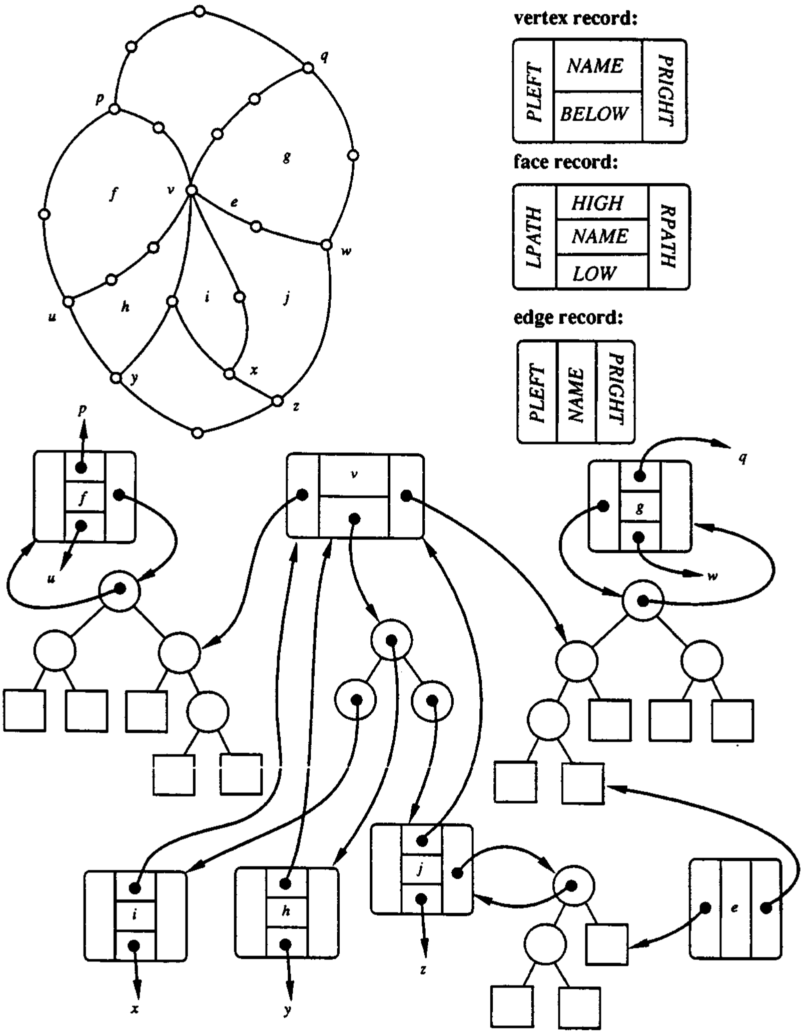

FIG. 9. A spherical *st*-graph and a fragment of the dynamic data structure for it. The edges are oriented upward.

left path of f_1. Note that L_1 and/or L_3 might be empty. Analogously, we partition the right path of f into subpaths R_1, R_2, and R_3, where R_2 is the right path of f_2. After the insertion of edge e, the new boundaries of f_1 and f_2 are as follows (see Fig. 10):

$$\textbf{lpath}(f_1) = L_2; \qquad \textbf{rpath}(f_1) = R_1 e R_3;$$
$$\textbf{lpath}(f_2) = L_1 e L_3; \qquad \textbf{rpath}(f_2) = R_2.$$

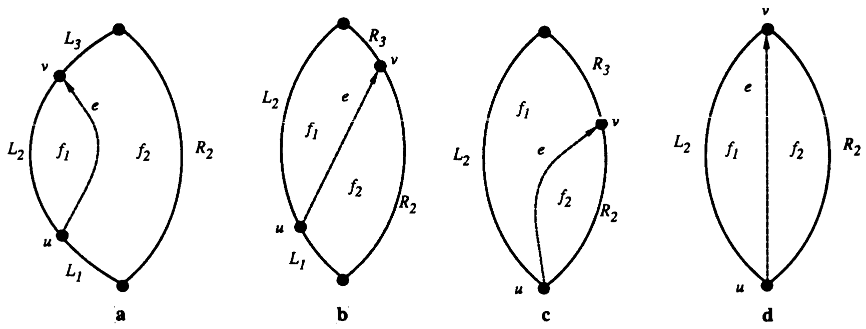

FIG. 10. Restructuring of the face boundaries after an **insert-edge** operation. (a) u and v on left path; (b) u on left path and v on right path; (c) u bottommost and v on right path; (d) u bottommost and v topmost.

Hence, the **lpath** and **rpath** trees of the new faces are obtained by *splitting* **lpath**(f) and **rpath**(f) at vertices u and v, and *splicing* appropriately the resulting trees. Standard techniques allow us to perform the above split and splice operations in logarithmic time (see, for example, [47, pp. 213–216]).

The remaining updates of the data structure are as follows:

1. Insert into $\mathscr{E}$ a new record for edge e, and increment counters **deg**$^{+}(u)$ and **deg**$^{-}(v)$.
2. Delete from $\mathscr{F}$ the record of face f, and insert new records for faces f_1 and f_2.
3. Delete from **below**(**high**(f)) the node pointing to f, and insert into **below**(**high**(f_1)) and **below**(**high**(f_2)) nodes pointing to f_1 and f_2, respectively.

The **insert-vertex**(e, v, e_1, e_2) operation is performed as follows:

1. Delete from $\mathscr{E}$ the record for edge e, and insert new records for edges e_1 and e_2.
2. Insert into $\mathscr{V}$ a new record for vertex v.
3. Find the faces f and g respectively to the left and right of edge e by waking up to the roots of the trees that contain **pleft**(e) and **right**(e).
4. Replace the leaf representative of e in **rpath**(f) by a subtree with root v and children e_1 and e_2, and rebalance **rpath**(f).
5. Replace the leaf representative of e in **lpath**(g) by a subtree with root v and children e_1 and e_2, and rebalance **lpath**(g).
6. Set **below**$(v) = \emptyset$.

The above data structure also supports operations **remove-edge** and **remove-vertex**. In fact, the **remove-edge** operation can be performed by reversing the transformations on the data structure realized by the algorithm for the **insert-edge** operation. Similarly, the **remove-vertex** operation is the reverse of the **insert-vertex** operation. Note also that, by Lemma 3, the counters $\mathbf{deg}^{+}(v)$ and $\mathbf{deg}^{-}(v)$ allow us to test the feasibility of each such operation in $O(1)$ time. The time complexity analysis of the various operations is straightforward, and we conclude:

THEOREM 9. *There exists a data structure for the dynamic embedding problem on spherical st-graphs with the following performance*:

$$\mathbf{space}(n) = \mathbf{preprocess}(n) = O(n);$$

$$\mathbf{test}(n) = \mathbf{insert\text{-}edge}(n) = \mathbf{insert\text{-}vertex}(n) = \mathbf{remove\text{-}edge}(n) = \mathbf{remove\text{-}vertex}(n) = O(\log n).$$

In the execution of an update operation we can distinguish the *search time* spent in finding the nodes of the various trees involved in the operation, and the *restructuring time* that takes into account the update and rebalancing of the trees. The next theorem shows that in our data structure the *amortized* restructuring time for a sequence of **insert-edge** and **insert-vertex** operations is optimal. For the definition of amortized time complexity, see [64].

THEOREM 10. *There exists a data structure for the dynamic embedding problem on spherical st-graphs such that*:

1. *The space and time complexity of the various operations are the same as in Theorem* 9.

2. *In a sequence of* **insert-edge** *and* **insert-vertex** *operations, the amortized restructuring time complexity of each such operation is* $O(1)$.

Proof. Use *2−4 trees* (which are equivalent to *red−black trees* [29]) to realize trees $\mathscr{V}$, $\mathscr{F}$, $\mathscr{E}$, and $\mathbf{below}(v)$. Such trees have $O(1)$ amortized rebalancing time for insertions and deletions [34]. With regard to the **lpath** and **rpath** trees, their manipulation in a sequence of **insert-edge** and **insert-vertex** operations involves insertions and the kind of generalized splittings considered in [32]. It is shown there that *circular level-linked 2−4* trees support efficiently a sequence of insertions and generalized splittings. With arguments similar to those developed in [32], we can show that by realizing the **lpath** and **rpath** trees by circular level-linked 2−4 trees, the amortized rebalancing time for **lpath** and **rpath** is $O(1)$. ∎

5.2. *st-Orientable Graphs*

For *st*-orientable plane graphs, we maintain on-line a spherical *st*-orientation and use the data structure previously described. Operations **test** and

insert-vertex do not require any modifications. In connection with operation **insert-edge**$(e, u, v, f; f_1, f_2)$, we have to select the direction of edge e so that it does not introduce cycles internal to face f.

By Corollary 1, this can be done easily by reversing the direction whenever the **insert-edge** algorithm rejects the operation. This proves:

THEOREM 11. *There exists a data structure for the incremental embedding problem on st-orientable plane graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n);$$

$$\textbf{test}(n) = \textbf{insert-edge}(n) = \textbf{insert-vertex}(n) = O(\log n).$$

5.3. *General Plane Graphs*

In this section, we study the incremental embedding problem on general plane graphs. We add operations **attach-vertex** to our repertory, which allows to add a degree-1 vertex to the graph.

Regarding operation **insert-edge**, if the vertices joined are not in the same block, the new edge and a sequence of blocks of G, called *old-blocks*, are merged into a new block, called *new-block*. In the example of Fig. 11b, the old-blocks are B_1, B_2, and B_3, and the new-block is the union of B_1, B_2, B_3, and edge $e = (u, v)$.

In the following, we will provide an *amortized* $O(\log n)$ time bound for operation **insert-edge**, and a worst-case $O(\log n)$ time bound for the remaining operations. According to standard conventions in amortized complexity analysis [64], this means that a sequence of n operations starting from an initial graph consisting of a single edge takes $O(n \log n)$ time.

First, we dynamize the data structure of Theorem 5 that tests whether two vertices belong to the same block of graph G. We implement the oriented block-cutvertex tree T of G using a balanced search tree to represent the set of children of each node. Also, we store with each block B a pointer **proper**(B) to a balanced search tree whose nodes represent the proper vertices of B. In turn, each proper vertex v of B has a pointer to its representative in the tree **proper**(B).

THEOREM 12. *Let G be a graph with n vertices. There exists a data structure with $O(n)$ space and $O(n + m)$ preprocessing time that allows to test whether two vertices belong to the same block of G in $O(\log n)$ time (worst-case), and supports operations* **insert-vertex** *and* **attach-vertex** *in $O(\log n)$ worst-case time, and operation* **insert-edge** *in $O(\log n)$ amortized time.*

Proof. We use the same query algorithm presented in Section 4.3. To access **node**(v) for a vertex v, and to access the parent of a node of T, we

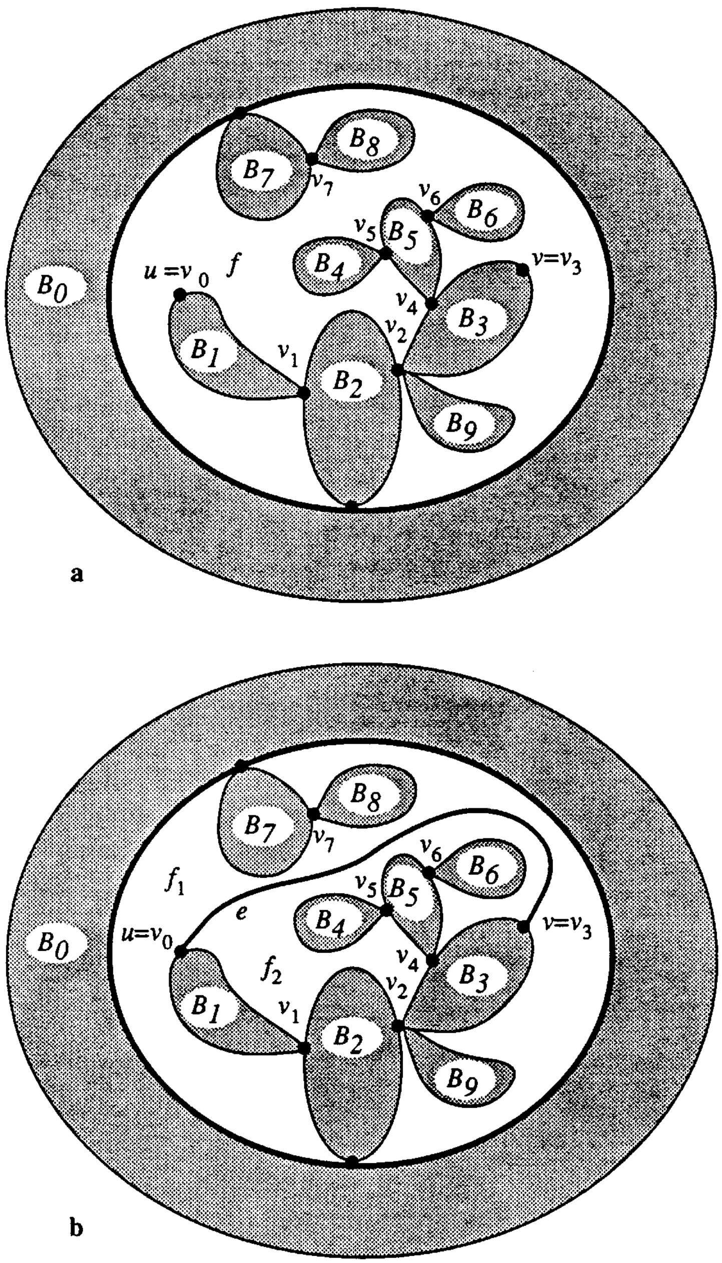

FIG. 11. Example of **insert-edge** operation in a cutface: (a) Before insertion. (b) After insertion.

traverse a leaf-to-root path in a balanced tree, which takes $O(\log n)$ time. Hence, a query takes $O(\log n)$ time. Operations **insert-vertex** and **attach-vertex** take $O(\log n)$ time since they can be performed with $O(1)$ insertions in balanced trees. Regarding operation **insert-edge**, merging two blocks can be done in time $O(\log n)$. Also, note that in any sequence of operations, there can be at most $n - 1$ merges of blocks. Hence the amortized time complexity of **insert-edge** is $O(\log n)$. ▮

The data structure for maintaining the **outer** relation is as follows. For each face f, we store block **outer**(f) directly. Also, for each vertex v such that **outer**(v) is a block, we store **outer**(v) directly. For each face f, we store the circuit **inner**(f) forming the boundary of f in a balanced tree. Note that cutvertices of blocks inside f are repeated in **inner**(f). For each vertex v with **outer**$(v) = f$, we store in a balanced tree **ext**(v) bidirectional pointers to the representatives of v in **inner**(f), sorted along the boundary of f. Also, for each block B such that **outer**$(B) = f$, we store in a balanced tree **ext**(B) bidirectional pointers to the representatives of the vertices of B that are in **inner**(f), sorted along the boundary of f. The above data structure makes it possible to compute **outer**$(\cdot)$ in $O(\log n)$ time. Hence, the complexity of the **test** operation is the same as in the static case.

Now, we describe the transformations of the data structure due to the execution of operation **insert-edge**$(e, u, v, f; f_1, f_2)$. If face f is not a cutface and is part of block B, we simply perform the operation in the data structure of B. Otherwise, edge e and the old-blocks have to be merged into the new block. Specifically, let $\mu(w)$ be the node of the block-cutvertex tree T of G associated with vertex w (i.e., if w is a cutvertex then $\mu(w)$ is the node representing w, and otherwise $\mu(w)$ is the node representing the block of w). The old-blocks are exactly those on the path of T from $\mu(u)$ to $\mu(v)$. Also, in general both f_1 and f_2 will be cutfaces in the updated embedding.

The modification of the block structure caused by the **insert-edge** operation requires that we set up a new data structure for the new-block B. We show that the data structure for B can be obtained by reorienting all but one old-block, called *base-block*, which is chosen as an old-block with maximum number of edges.

Indeed, let the old-blocks be $B_1, \dots, B_l$, with $u \in B_1$ and $v \in B_l$, and let v_i, $i = 1, \dots, l - 1$, be the cutvertex between B_i and B_{i+1}. Also, let $v_0 = u$ and $v_l = v$. (In the example of Fig. 11b, $l = 3$). Denoting by B_j the base-block, at least one of the directed edges (v_{j-1}, v_j) and (v_j, v_{j-1}), say (v_j, v_{j-1}), can be added to B_j while preserving the spherical *st*-orientation (see Corollary 1). In this case, we construct a new acyclic spherical *st*-orientation for each B_i, $i \neq j$, using v_{i-1} as the source and v_i as the sink (see Theorem 1), and we direct e from v_l to v_0. The case when (v_{j-1}, v_j) can be added to B_j is analogous.

Since replacing an edge (v', v'') of a spherical *st*-graph with an acyclic spherical *st*-graph with source v' and sink v'' yields a spherical *st*-graph, the above orientation of the new-block B is a spherical *st*-orientation. The data structure for B can be constructed from the spherical *st*-orientation of B in time

$$O\left(\log n + \sum_{i=1}^{l} m_i - \max_{i=1,\ldots,l} m_i\right),$$

where m_i is the number of edges of B_i.

Blocks **outer**(f_1) and **outer**(f_2) can be recomputed in $O(\log n)$ time. Note that one of them is the new block B. Lists **inner**(f_1) and **inner**(f_2) are obtained from **inner**(f) by $O(1)$ split and splice operations, each taking $O(\log n)$ time. List **ext**(B) for the new block is obtained by $O(l)$ split and splice operations on the **ext** lists of the old blocks. For each vertex w in the **ext** lists of the old blocks that does not appear in **ext**(B), we set **outer**$(w) = B$. Note that the latter operation is performed at most once for each vertex, so that it contributes $O(1)$ to the amortized time complexity of **insert-edge**.

THEOREM 13. *There exists a data structure for the incremental embedding problem on plane graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n);$$

$$\textbf{test}(n) = \textbf{insert-vertex}(n) = \textbf{attach-vertex}(n) = O(\log n);$$

$$\textbf{insert-edge}(n) = O(\log n)\ (\textit{amortized}).$$

Proof. The amortized complexity analysis makes use of the *potential function*

$$\mathbf{\Phi} = \sum_{i=1}^{b} m_i \log \frac{1}{m_i}$$

where $B_1, \ldots, B_b$ are the current blocks of G, and m_i is the number of edges of B_i ($\sum_{i=1}^{b} m_i = m$). Note that $-n \log n \leq \mathbf{\Phi} \leq 0$. It is interesting to observe that the function $\mathbf{\Phi}$ is similar to the *entropy function* used in information theory. Informally, we can say that when the old-blocks are merged together into the new-block, $\mathbf{\Phi}$ decreases to compensate for the work spent in the merge process. For simplicity, consider the case when two old-blocks, B_1 and B_2, are merged into a new-block B^*, where m_i is the number of edges of B_i ($i = 1, 2$) and m^* is the number of edges of B^*.

By an appropriate choice of the time unit, we have that operation **insert-edge** is executed in time

$$t = \log n + 2\min\{m_1, m_2\}.$$

The variation $\Delta\Phi$ of potential is given by

$$\Delta\Phi = m^* \log\frac{1}{m^*} - \left(m_1 \log\frac{1}{m_1} + m_2 \log\frac{1}{m_2}\right).$$

Define $x_1 = m_1/m^*$ and $x_2 = m_2/m^*$. Clearly, $x_1 + x_2 = 1$. We can write

$$\frac{\Delta\Phi}{m^*} = (x_1 + x_2)\log\frac{1}{m^*} - \left(x_1 \log\frac{1}{m_1} + x_2 \log\frac{1}{m_2}\right).$$

Thus,

$$\frac{\Delta\Phi}{m^*} = -\left(x_1 \log\frac{1}{x_1} + x_2 \log\frac{1}{x_2}\right) = -H(x_1, x_2),$$

where $H(x_1, x_2)$ is the *binary entropy function* of information theory.

We show in Fig. 12 a plot of $H(x_1, 1 - x_1)$ in the interval $x_1 \in [0, 1]$. It is easy to see that

$$H(x_1, 1 - x_1) \geq 2\min\{x_1, 1 - x_1\}.$$

Therefore, we obtain

$$\Delta\Phi \leq -2m^* \min\{x_1, x_2\} = -2\min\{m_1, m_2\}.$$

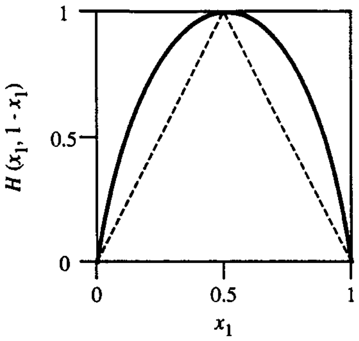

FIG. 12. Plot of the binary entropy function $H(x_1, 1 - x_1)$.

Hence, the amortized time complexity of operation **insert-edge** is

$$\bar{t} = t + \Delta\Phi \leq \log n + 2\min\{m_1, m_2\} - 2\min\{m_1, m_2\} = \log n.$$

When **insert-edge** joins vertices in the same block B, it takes time $t = O(\log n)$. Also, the variation of potential is given by

$$\Delta\Phi = (m+1)\log\frac{1}{m+1} - m\log\frac{1}{m} \leq 0,$$

where m is the number of edges of B.

Hence, also in this case the amortized time complexity $\bar{t} = t + \Delta\Phi$ is $O(\log n)$. ∎

6. DELETIONS

In this section, we show how to add a limited deletion capability to the data structures of Section 5.

6.1. *Vertices and Edges That Can Be Deleted*

Let G be a spherical *st*-graph with n vertices. We say that a vertex (edge) of G is *free* if it can be removed by operation **remove-vertex** (**remove-edge**) such that the resulting digraph is also a spherical *st*-graph, and we say that it is *locked* otherwise. Lemma 3 characterizes the free and locked edges.

The data structure of Section 5.1 for the incremental embedding problem on G also supports deletion operations **remove-vertex** and **remove-edge** on free vertices and edges. Namely, the transformations of the data structure involved in a **remove-vertex** or **remove-edge** operation are exactly the reverse of those for **insert-vertex** or **insert-edge**, and can be performed in $O(\log n)$ time by means of:

- $O(1)$ insertions/deletions in $\mathscr{V}$, $\mathscr{E}$, and $\mathscr{F}$, and the **below** trees;
- $O(1)$ updates of the **high**/**low** pointers and the **deg**$^+$/**deg**$^-$ counters; and
- $O(1)$ split/splice operations on the **lpath** and **rpath** trees.

Also, by Lemma 3, testing whether a vertex or an edge is free or locked can be done in $O(\log n)$ time. We have:

THEOREM 14. *The data structure of Theorem* 9 *supports also operations* **remove-edge** *and* **remove-vertex** *on the free vertices and edges of a spherical st-graph in* $O(\log n)$ *time*.

For an *st*-orientable plane graph, we say that a vertex (edge) is free if it is free in the corresponding spherical *st*-orientation. By Theorem 14, we have:

COROLLARY 3. *The data structure of Theorem* 11 *supports also operations* **remove-edge** *and* **remove-vertex** *on the free vertices and edges of an st-orientable plane graph in $O(\log n)$ time*.

Finally, we consider a general plane graph G. With regard to operations **remove-edge**, **remove-vertex**, and **detach-vertex**, again we can partition the edges of the graph into free and locked, as follows. We recall that the data structure for G maintains the block-cutvertex tree of G and spherical *st*-orientations for the blocks of G. Let (u, v) be an edge of G, and B the block containing (u, v). We say that edge (u, v) is *free* if operation **remove-edge** on (u, v) does not change the block-cutvertex tree of G and can be performed in the spherical *st*-orientation of B. Otherwise, we say that edge (u, v) is *locked*. We have

COROLLARY 4. *The data structure of Theorem* 13 *also supports operations* **remove-edge**, **remove-vertex**, *and* **detach-vertex** *on the free edges and vertices of a plane graph in $O(\log n)$ time* (*worst-case*).

6.2. *Hierarchical Embedding*

In several layout applications, design methodologies limit the freedom of the designer in making arbitrary updates to the layout. For example, well-known hierarchical design strategies for VLSI circuits build a layout in a top-down fashion by means of successive refinements.

In the following, we show that the class of free vertices and edges in an *st*-orientable plane graph is sufficiently large to support a hierarchical deletion scheme that allows us to ‘‘undo’’ any **insert-edge** and **insert-vertex** operation performed in the past (not just the last operation).

The *hierarchical embedding problem* for an *st*-orientable plane graph G consists of performing a sequence of operations **test**, **insert-edge**, **insert-vertex**, **remove-edge**, and **remove-vertex**, starting from an initial *st*-orientable plane graph G_0, where the following restrictions are placed on the **remove-edge** and **remove-vertex** operations:

- An edge can be deleted by a **remove-edge** operation only if it was created (at any time in the past) by means of an **insert-edge** operation.
- A vertex can be deleted by a **remove-vertex** operation only if it was created (at any time in the past) by an **insert-vertex** operation.

In Fig. 13, we show a sequence of update operations in an instance of the hierarchical embedding problem.

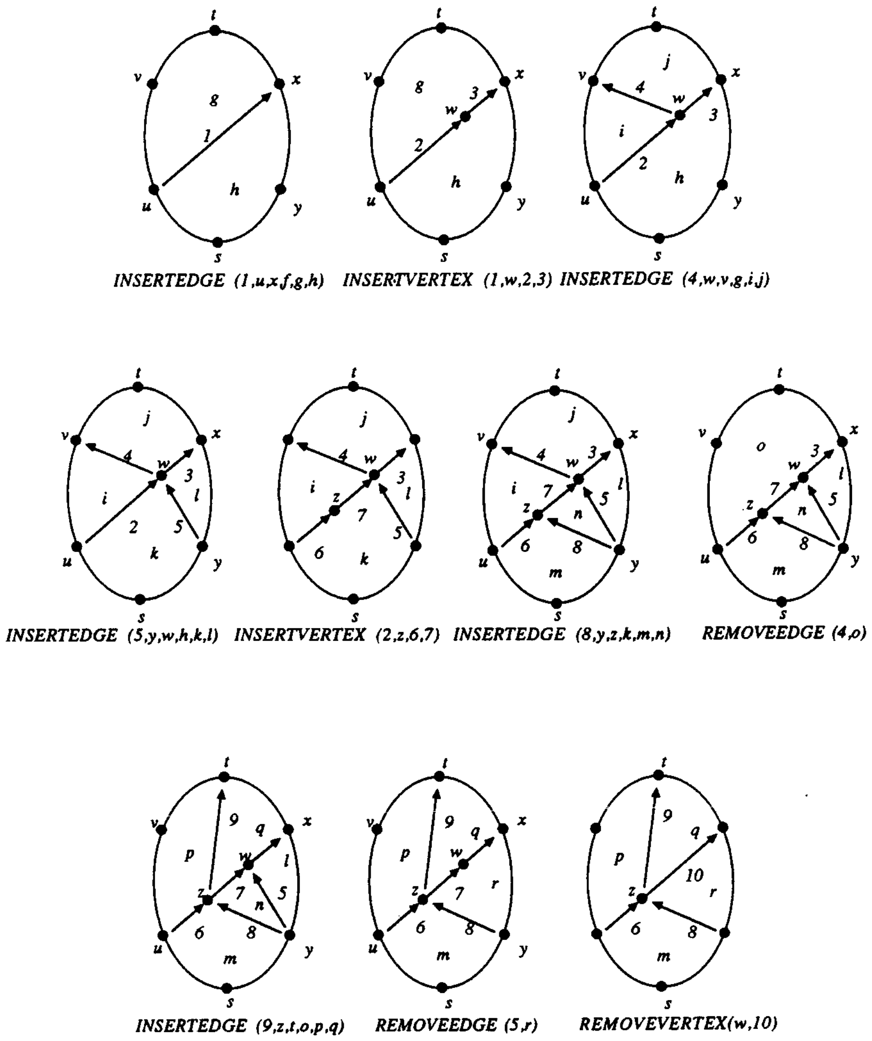

FIG. 13. Sequence of hierarchical update operations on an *st*-orientable plane graph. Flags in the "set" condition are shown by disconnecting the corresponding endpoint.

The aforementioned restrictions on the **remove-edge** and **remove-vertex** operations can be enforced by storing with each edge e two flags, denoted **free-head**(e) and **free-tail**(e), which are associated with the head and tail of edge e in the spherical *st*-orientation, respectively. We use these flags to maintain the invariant that an edge e can be removed if and only if both **free-tail**(e) and **free-head**(e) are set. The manipulation of the flags in the various operations is straightforward. In the example of Fig. 13, a flag in the "set" condition is indicated by showing the corresponding endpoint unconnected.

It is not difficult to show that the free edges and vertices include those that can be deleted in an instance of the hierarchical embedding problem.

LEMMA 6. *Let G be an st-orientable plane graph that is subject to a sequence of hierarchical update operations, starting from an initial st-orientable plane graph G_0. If a vertex (edge) of G can be deleted, then it is free.*

Hence, we have the following theorem:

THEOREM 15. *There exists a data structure for the hierarchical embedding problem on st-orientable plane graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n);$$
$$\textbf{test}(n) = \textbf{insert-edge}(n) = \textbf{insert-vertex}(n) = \textbf{remove-edge}(n) = \textbf{remove-vertex}(n) = O(\log n).$$

7. EXTENSIONS

In this section we outline several extensions of our data structure. First, it is straightforward to extend the results to plane graphs that are not necessarily connected and that may have multiple edges. In this case, the statements of the theorems hold provided n denotes the number of vertices plus the number of edges of the graph.

7.1. *Reporting All Faces*

The following operation extends operation **test**:

list(u, v): Report all the faces that have both vertices u and v on their boundary.

The algorithm for operation **list** is a simple variation of the one for operation **test**. Namely, we *report* all the faces that verify the four cases described in Section 4.2.

We have:

COROLLARY 5. *The data structure of Theorem* 15 *also supports operation* **list** *in $O(\log n + k)$ time, where k is the number of faces reported.*

7.2. *Computing Separating Pairs*

The identification of separating pairs is important in problems of fault-tolerance of networks. Previous work [40] shows that the number of separating pairs in a graph is $O(n^2)$, and that this bound is tight and can be achieved with a planar graph (a cycle).

Our data structure can be easily extended to support the following query that detects separating pairs (This problem is not addressed in [40]):

separating-pair(u, v): Test whether u and v form a separating pair.

We use the following lemma, which follows immediately from the results of [8].

LEMMA 7. *Let G be a 2-connected plane graph. Vertices u and v form a separating pair of G if and only if*

- *u and v are adjacent and there are at least three faces whose boundary contains u and v; or*
- *u and v are not adjacent and there are at least two faces whose boundary contains u and v.*

By Lemma 7, we can test whether two vertices form a separating pair by a simple modification of the algorithm for operation **test**(u, v), which *counts* the number of occurrences of u in **below**(v) and of v in **below**(u) (see Section 4.2). We obtain:

COROLLARY 6. *The data structure of Theorem 15 also supports operation* **separating-pair** *in $O(\log n)$ time.*

7.3. *A Dual Embedding Problem*

The *dual dynamic embedding problem* is a variation of the dynamic embedding problem that consists of performing the following operations on a plane graph G:

test*(f, g): Test if there is a vertex v that is on the boundaries of both faces f and g. In case such a vertex v exists, output its name.

expand-vertex$(e, f, g, v; v_1, v_2)$: Expand vertex v into vertices v_1 and v_2 connected by an edge e on the boundary of faces f and g.

contract-vertex$(e, f, g, v_1, v_2; v)$: Contract the edge $e = (v_1, v_2)$ on the boundary of faces f and g, and call v the vertex resulting from the contraction of v_1 and v_2.

duplicate-edge$(e, f; e_1, e_2)$: Replace the edge e with two multiple edges, e_1 and e_2, with the same endpoints, and call f the resulting face between them.

merge-edge$(e_1, e_2, f; e)$: Let f be a face whose boundary consists of two multiple edges, e_1 and e_2. Remove f by merging e_1 and e_2 into a new edge e.

We can show that this problem is the *dual* of the dynamic embedding problem by extending the notion of duality of undirected plane graphs to spherical *st*-graphs.

The *dual graph* G^* of a plane digraph G is the plane digraph defined as follows: the vertices of G^* are the faces of G; for each edge e of G there is an edge e^* of G^* from face **left**(e) to face **right**(e). A *cylindrical* s^*t^**-graph* is a plane digraph G whose dual G^* is a spherical *st*-graph. We can visualize a cylindrical s^*t^*-graph as embedded in a cylinder, with external faces s^* and t^* (see Fig. 14).

An (undirected) plane graph is said to be s^*t^**-orientable* if it can be oriented to become a cylindrical s^*t^*-graph. Note that every 2-connected graph is s^*t^*-orientable. Using Theorem 9 and duality arguments, we obtain:

THEOREM 16. *There exists a data structure for the dual dynamic embedding problem on cylindrical* s^*t^**-graphs with the performance*

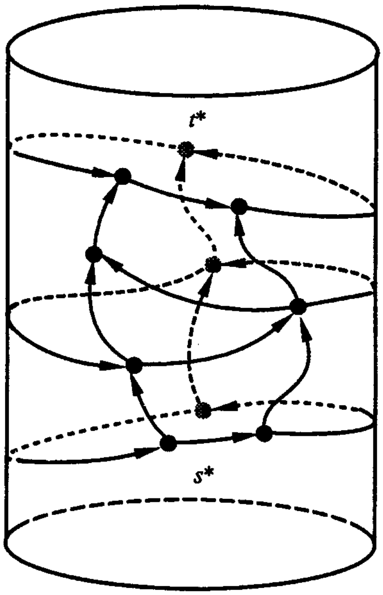

FIG. 14. Example of cylindrical s^*t^*-graph.

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n);$$
$$\textbf{test}^*(n) = \textbf{expand-vertex}(n) = \textbf{duplicate-edge}(n)$$
$$= \textbf{contract-vertex}(n) = \textbf{merge-edge}(n) = O(\log n).$$

The *hierarchical dual embedding problem* is a variation of the dynamic embedding problem where the **contract-vertex** and **merge-edge** operations can be performed only on edges (respectively, faces) previously inserted by the **expand-vertex** (respectively, **duplicate-edge**) operation. By Theorem 15, we have:

THEOREM 17. *There exists a data structure for the hierarchical dual embedding problem on s*t*-orientable plane graphs with the performance*

$$\textbf{space}(n) = \textbf{preprocess}(n) = O(n);$$
$$\mathit{Test}^*(n) = \textbf{expand-vertex}(n) = \textbf{duplicate-edge}(n)$$
$$= \textbf{contract-vertex}(n) = \textbf{merge-edge}(n) = O(\log n).$$

ACKNOWLEDGMENT

I thank Herbert Edelsbrunner, Franco Preparata, and Ioannis Tollis for comments and useful discussions.

REFERENCES

1. G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, Incremental algorithms for minimal length paths, *in* "Proc. ACM-SIAM Sympos. Discrete Algorithms," pp. 12–21, 1990.
2. H. Baumgarten, H. Jung, and K. Mehlhorn, Dynamic point location in general subdivision, *in* "Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms," pp. 250–258, 1992.
3. J. A. Bondy and U. S. R. Murty, "Graph Theory with Applications," North-Holland, New York, 1976.
4. K. Booth and G. Lueker, Testing for the consecutive ones property interval graphs and graph planarity using PQ-tree algorithms, *J. Comput. System Sci.* **13** (1976), 335–379.
5. S. W. Cheng and R. Janardan, New results on dynamic planar point location, *SIAM J. Comput.* **21** (1992), 972–999.
6. Y.-J. Chiang, F. P. Preparata, and R. Tamassia, A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps, *in* "Proc. 4th ACM-SIAM Sympos. Discrete Algorithms," pp. 44–53, 1993.
7. Y.-J. Chiang and R. Tamassia, Dynamic algorithms in computational geometry, *Proc. IEEE* **80**(9) (1992), 1412–1434.
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**(1) (1985), 54–76.

9. N. Chiba, K. Onoguchi, and T. Nishizeki, Drawing planar graphs nicely, *Acta Inform*. **22** (1985), 187–201.
10. R. F. Cohen, G. Di Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi, A framework for dynamic graph drawing, *in* ‘‘Proceedings 8th Annual ACM Symposium Computational Geometry,’’ pp. 261–270, 1992.
11. R. F. Cohen, S. Sairam, R. Tamassia, and J. S. Vitter, Dynamic algorithms for optimization problems in bounded tree-width graphs, *in* ‘‘Proceedings of the Third Conference on Integer Programming and Combinatorial Optimization,’’ pp. 99–112, 1993.
12. R. F. Cohen and R. Tamassia, Dynamic expression trees, *Algorithmica* **13** (1995), 245–265.
13. R. F. Cohen and R. Tamassia, Combine and conquer: A general technique for dynamic algorithms, *in* ‘‘1st Annual European Symposium on Algorithms (ESA ’93),’’ Lecture Notes in Computer Science, Vol. 726, pp. 97–108, Springer-Verlag, Berlin/New York, 1993.
14. G. Di Battista and R. Tamassia, Algorithms for plane representations of acyclic digraphs, *Theoret*. *Comput*. *Sci*. **61** (1988), 175–198.
15. G. Di Battista and R. Tamassia, Incremental planarity testing, *in* ‘‘Proceedings 30th Annual IEEE Symposium on the Foundations of Computer Science,’’ pp. 436–441, 1989.
16. G. Di Battista and R. Tamassia, On-line graph algorithms with SPQR-trees, *in* ‘‘Automata, Languages and Programming (Proc. 17th ICALP), Lecture Notes in Computer Science, Vol. 442, pp. 598–611, 1990.
17. G. Di Battista and R. Tamassia, On-line planarity testing, *SIAM J*. *Comput*., **25** (1996) to appear. Preprint: Technical Report CS-92-39, Computer Science Department, Brown University, 1992.
18. G. Di Battista, R. Tamassia, and I. G. Tollis, Area requirement and symmetry display of planar upward drawings, *Discrete Comput*. *Geom*. **7** (1992), 381–401.
19. D. Eppstein, Z. Gail, G. Italiano, and T. Spencer, Separator based sparsification for dynamic planar graph algorithms, *in* ‘‘Proceedings 25th Annual ACM Symposium on the Theory of Computing, pp. 208–217, 1993.
20. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, Sparsification: A technique for speeding up dynamic graph algorithms, *in* ‘‘Proceedings 33rd Annual IEEE Symposium on the Foundations of Computer Science,’’ pp. 60–69, 1992.
21. D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, Maintenance of a minimum spanning forest in a dynamic planar graph, *J*. *Algorithms* **13** (1992), 33–54.
22. S. Even, ‘‘Graph Algorithms,’’ Computer Science Press, Potomac, MD, 1979.
23. S. Even and Y. Shiloach, An on-line edge deletion problem, *J*. *Assoc*. *Comput*. *Mach*. **28** (1981), 1–4.
24. S. Even and R. E. Tarjan, Computing an st-numbering, *Theoret*. *Comput*. *Sci*. **2** (1976), 339–344.
25. G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, with applications, *SIAM J*. *Comput*. **14** (1985), 781–798.
26. G. N. Frederickson, Ambivalent data structures for dynamic 2-edge connectivity and *k*-smallest spanning trees, *in* ‘‘Proceedings 32nd Annual IEEE Symposium on the Foundations of Computer Science,’’ pp. 632–641, 1991.
27. Z. Galil, G. F. Italiano, and N. Sarnak, Fully dynamic planarity testing, *in* ‘‘Proceedings 24th Annual ACM Symposium on the Theory of Computing, pp. 495–506, 1992.
28. M. T. Goodrich and R. Tamassia, Dynamic trees and dynamic point location, *in* ‘‘Proceedings 23rd Annual ACM Symposium on the Theory of Computing,’’ pp. 523–533, 1991.

29. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *in* ''Proceedings 19th Annual IEEE Symposium on the Foundations of Computer Science,'' Lecture Notes in Computer Science, pp. 8–21, 1978.
30. F. Harary, ''Graph Theory,'' Addison-Wesley, Reading, MA, 1969.
31. J. Hershberger, M. Rauch, and S. Suri, Fully dynamic 2-edge connectivity in planar graphs, *in* ''Proceedings 3rd Scand. Workshop Algorithm Theory,'' Vol. 621 of Lecture Notes in Computer Science, pp. 233–244, Springer-Verlag, Berlin/New York, 1992.
32. K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan, Sorting Jordan sequences in linear time using level-linked search trees, *Inform. Control* **68** (1986), 170–184.
33. J. Hopcroft and R. E. Tarjan, Efficient planarity testing, *J. Assoc. Comput. Mach.* **21**(4) (1974), 549–568.
34. S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Inform.* **17** (1982), 157–184.
35. T. Ibaraki and N. Katoh, On-line computation of transitive closure of graphs, *Inform. Process. Lett.* **16** (1983), 95–97.
36. G. F. Italiano, Amortized efficiency of a path retrieval data structure, *Theoret. Comput. Sci.* **48** (1986), 273–281.
37. G. F. Italiano, Finding paths and deleting edges in directed acyclic graphs, *Inform. Process. Lett.* **28** (1988), 5–11.
38. G. F. Italiano and Z. Galil, Fully dynamic algorithms for edge connectivity problems, *in* ''Proceedings 23rd Annual ACM Symposium on the Theory of Computing, pp. 317–327, 1991.
39. G. F. Italiano, J. A. La Pourtré, and M. H. Rauch, Fully dynamic planarity testing in planar embedded graphs, *in* ''1st Annual European Symposium on Algorithms (ESA '93),'' Lecture Notes in Computer Science, Vol. 726, pp. 212–223. Springer-Verlag, Berlin/New York, 1993.
40. A. Kanevsky, A characterization of separating pairs and triplets in a graph, *Cong. Numer.* **74** (1990), 213–232.
41. A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen, On-line maintenance of the four-connected components of a graph, *in* ''Proceedings of Annual IEEE Symposium on the Foundations of Computer Science,'' pp. 793–801, 1991.
42. P. N. Klein and S. Subramanian, A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs, *in* ''Proceedings 1993 Workshop Algorithms Data Structure,'' pp. 442–451, 1993.
43. J. A. La Poutré, Maintenance of triconnected components of graphs, *in* ''Automata, Languages and Programming (Proc. 19th ICALP),'' Lecture Notes in Computer Science, Vol. 623, 1992.
44. J. A. La Poutré, Alpha-algorithms for incremental planarity testing, *in* ''Proceedings 26th Annual ACM Symposium on the Theory of Computing (STOC 94),'' pp. 706–715, 1994.
45. 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. WG '87)'' (H. Gottler and H. J. Schneider, Eds.), Lecture Notes in Computer Science, Vol. 314, pp. 106–120, Springer-Verlag, Berlin/New York, 1988.
46. A. Lempel, S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, *in* ''Theory of Graphs: Internat. Symposium (Rome 1966),'' pp. 215–232, Gordon and Breach, New York, 1967.
47. K. Mehlhorn, ''Sorting and Searching,'' Data Structures and Algorithms, Vol. 1, Springer-Verlag, Heidelberg, 1984.
48. K. Miriyala, S. W. Hornick, and R. Tamassia, An incremental approach to aesthetic graph layout, *in* ''Proceedings International Workshop on Computer-Aided Software Engineering (CASE '93),'' 1993.

49. T. Nishizeki and N. Chiba, Planar graphs: Theory and algorithms, *Ann. Discrete Math.* **32**, 1988.
50. F. P. Preparata and M. I. Shamos, ''Computational Geometry: An Introduction,'' Springer-Verlag, New York, 1985.
51. F. P. Preparata and R. Tamassia, Fully dynamic point location in a monotone subdivision, *SIAM J. Comput.* **18** (1989), 811–830.
52. M. Rauch, Fully dynamic biconnectivity in graphs, *in* ''Proceedings 33rd Annual IEEE Symposium on the Foundations of Computer Science,'' pp. 50–59, 1992.
53. J. H. Reif, A topological approach to dynamic graph connectivity, *Inform. Process. Lett.* **25** (1987), 65–70.
54. H. Rohnert, A dynamization of the all-pairs least cost problem, *in* ''Proceedings STACS '85,'' Lecture Notes in Computer Science, Vol. 182, pp. 279–286, Springer-Verlag, Berlin/New York, 1985.
55. P. Rosenstiehl and R. E. Tarjan, Rectilinear planar layouts and bipolar orientations of planar graphs, *Discrete Comput. Geom.* **1**(4) (1986), 343–353.
56. E. Steinitz and H. Rademacher, ''Vorlesungen über die Theorie der Polyeder,'' Springer, Berlin, 1934.
57. R. Tamassia, A dynamic data structure for planar graph embedding, *in* ''Automata, Languages and Programming (Proc. 15th ICALP)'' (T. Lepisto and A. Salomaa, Eds.), Lecture Notes in Computer Science, Vol. 317, pp. 576–590, Springer-Verlag, Berlin/New York, 1988.
58. R. Tamassia, G. Di Battista, and C. Batini, Automatic graph drawing and readability of diagrams, *IEEE Trans. Systems Man Cybernet.* **SMC-18**(1) (1988), 61–79.
59. R. Tamassia and F. P. Preparata, Dynamic maintenance of planar digraphs, with applications, *Algorithmica* (1990), 509–527.
60. R. Tamassia and I. G. Tollis, A unified approach to visibility representations of planar graphs, *Discrete Comput. Geom.* **1**(4) (1986), 321–341.
61. R. Tamassia and I. G. Tollis, Tessellation representations of planar graphs, *in* ''Proceedings 27th Allerton Conference Commun. Control Comput.,'' pp. 48–57, 1989.
62. R. Tamassia and I. G. Tollis, Representations of graphs on a cylinder, *SIAM J. Discrete Math.* **4**(1) (1991), 139–149.
63. R. Tamassia and I. G. Tollis, Reachability in planar digraphs with one source and one sink, *Theoret. Comput. Sci. A* **119** (1993), 331–343.
64. R. E. Tarjan, Amortized computational complexity, *SIAM J. Algebraic Discrete Methods* **6**(2) (1985), 306–318.
65. W. T. Tutte, Convex representations of graphs, *Proc. London Math. Soc.* **10** (1960), 304–320.
66. W. T. Tutte, How to draw a graph, *Proc. London Math. Soc.* **3**(13) (1963), 743–768.
67. J. Westbrook, Fast incremental planarity testing, *in* ''Automata, Languages and Programming (Proc. 19th ICALP),'' Lecture Notes in Computer Science, Vol. 623, pp. 342–353, 1992.
68. J. Westbrook and R. E. Tarjan, Maintaining bridge-connected and biconnected components on-line, *Algorithmica* (1992), 433–464.
69. D. Yellin, Speeding up dynamic transitive closure for bounded degree graphs, *Acta Inform.* **30** (1993), 369–384.