Dynamic Planar Graph Embedding

Roberto Tamassia

Technical Report No. CS-89-28

April 1989

Dynamic Planar Graph Embedding*

Roberto Tamassia

Department of Computer Science
Brown University
Box 1910
Providence, RI 02912
rt@cs.brown.edu

Abstract

We present a dynamic data structure that allows for incrementally constructing a planar embedding of a planar graph with n vertices and m edges. The data structure supports the following operations: (1) testing if a new edge can be added to the embedding without introducing crossings; (2) adding and removing vertices and edges. In each case the time complexity is $O(\log m)$. The space used and the preprocessing time are $O(m)$. If the graph is simple (i.e., it has no self-loops and no multiple edges), the above bounds become $O(\log n)$ and $O(n)$, respectively. This work finds applications in circuit layout, graphics, motion planning, and computer-aided design.

(April 1989)

---

* Work performed in part while the author was with the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.

Research partially supported by the National Science Foundation under grant ECS-84-10902 and by a grant from Cadre Technologies, 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 [2, 15]. Nevertheless, as confirmed by recent results [3, 9, 10, 19, 30], graph planarity is still a vital area of research, rich in interesting issues to be explored.

In this paper we consider the problem of incrementally constructing a planar embedding of a graph. We investigate a dynamic data structure that allows us to perform efficiently the following operations:

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

(2) *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 (1) the *space* requirement, (2) the *query* and *update* times, and (3) the *preprocessing* time.

Although very important in practical applications, only a few dynamic data structures have been devised for graph problems. Existing results are of a preliminary nature, and limited to connectivity [8, 27], minimum spanning tree [11], transitive closure [17, 18, 22, 23, 34], and shortest path [28]. 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*. For generality, we allow G to have multiple edges, and we denote with n and m the number of vertices and edges of G, respectively. We consider the *dynamic 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.

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

INSERTEDGE $(e,u,v,f;f_1,f_2)$: Add the edge $e=(u,v)$ to G 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.

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

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

REMOVEVERTEX $(e_1,e_2,v;e)$: 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 $e=(u,w)$.

The dynamic embedding problem naturally arises in interactive CAD layout environments. Applications include the design of integrated circuits, motion planning in robotics, architectural floor planning, and graphic editing of block diagrams.

We present a data structure that uses $O(m)$ space, supports all of the above operations in $O(\log m)$ time, and can be constructed in $O(m)$ time. If G is simple, i.e. it has no multiple edges and no self-loops, then $m=O(n)$, and the above bounds become $O(n)$ space and preprocessing time, and $O(\log n)$ query and update times. 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.

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 [29, 32, 33].

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 [26]. 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 [23-25].

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: (1) testing if a new edge can be added to G so that the resulting graph is itself planar; (2) adding and removing vertices and edges.

The rest of this paper is organized as follows: Section 2 contains definitions and preliminary results. Section 3 deals with orientations of planar graphs. In Section 4, we study the *topological location problem*, which consists of performing operations *TEST* and *LIST*. Section 5 describes the full-fledged data structure for the dynamic embedding problem. Finally, further applications are discussed in Section 6.

2. Preliminaries

We consider only finite connected graphs without self-loops. We allow multiple edges between two vertices. For the basic terminology about graphs and planarity, see [1, 7]. 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, $\deg^+(w)$ and $\deg^-(w)$ 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 [20]. 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 $\xi: V \rightarrow \{1,2,\cdots,|V|\}$ such that every vertex $v \neq s,t$ has neighbors u and v with $\xi(u) < \xi(v) < \xi(w)$. A graph admits an *st*-numbering if and only if it is *st*-2-connectible [20].

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

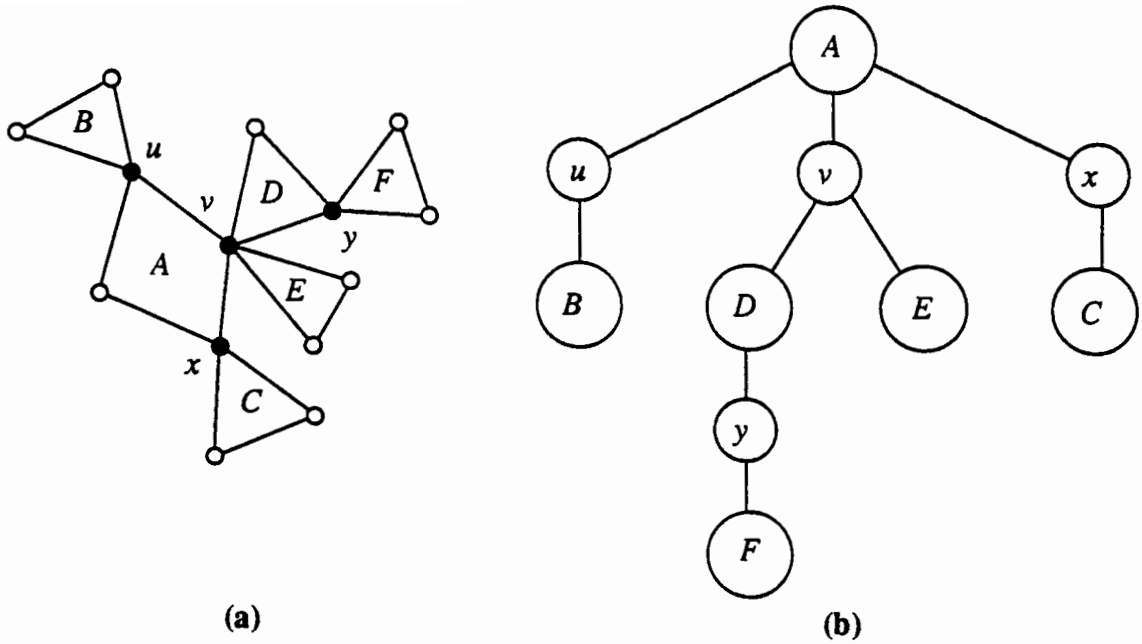


Figure 1 **(a)** A 1-connected graph, and **(b)** its block-cutvertex tree.

3-connected planar graph has a unique embedding.

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

Let m be the current number of edges of the graph. We will denote the storage by *Space* (m) and the preprocessing time by *Preprocess* (m). Also, the time complexity of the various operations will be denoted by *Test* (m), *List* (m,k), *InsertEdge* (m), *InsertVertex* (m), *RemoveEdge* (m), and *RemoveVertex* (m), where k is the number of faces retrieved by the *LIST* operation. 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 such that:

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 the one of *planar st-graph* introduced in [20], which has important applications in the test of graph planarity [20] and the construction of planar drawings [5, 29, 32] 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 with regard to the circular sequences of edges that are incident upon a vertex and that form the boundary of a face (see [32]).

Lemma 1 For every vertex v of G, the incoming (outgoing) edges appear consecutively around v (See Fig. 2(b)).

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 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

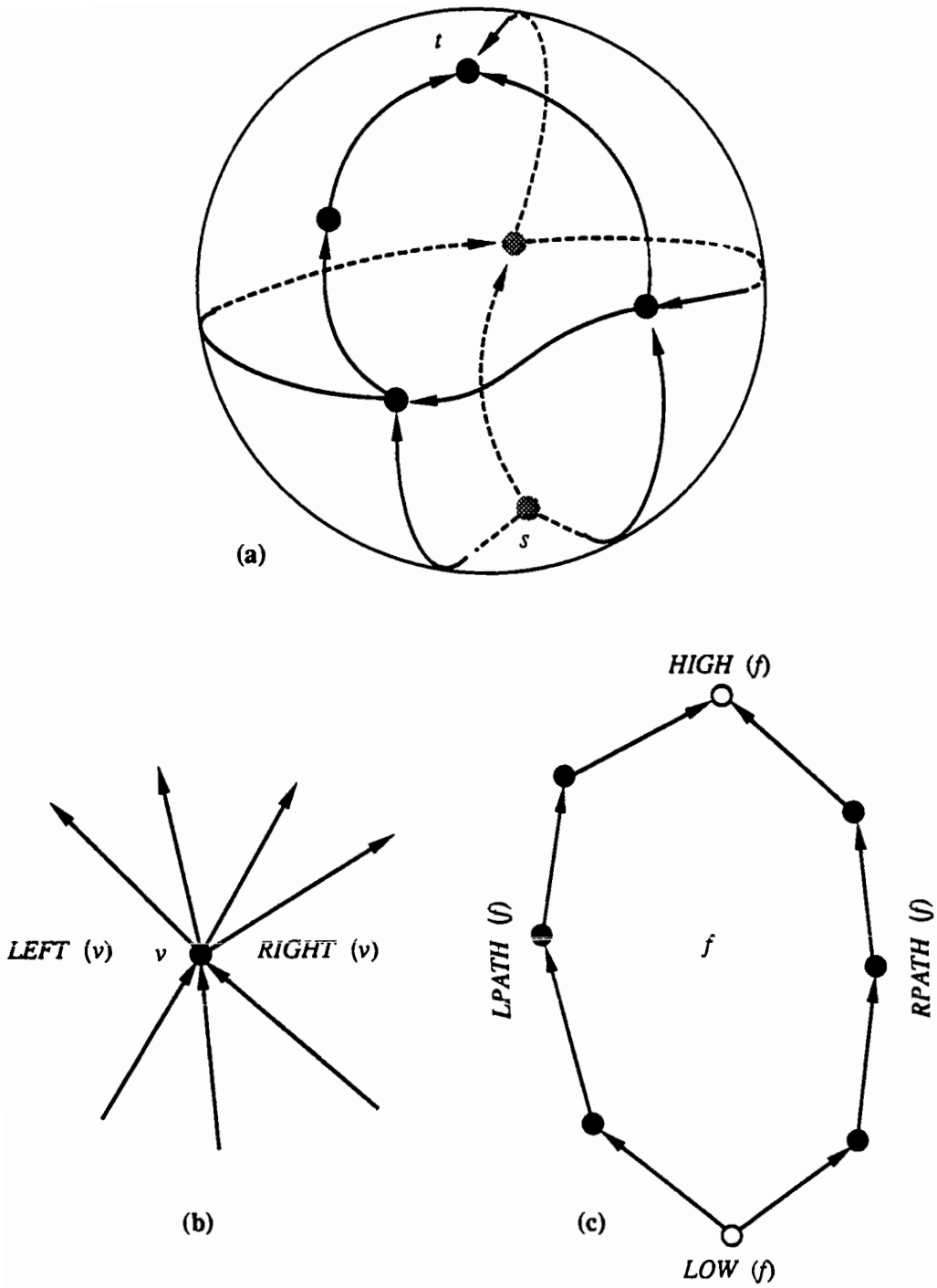


Figure 2 (a) Example of spherical *st*-graph. (b) Example for Lemma 1. (c) Example for Lemma 2.

are in A, then π_4 intersects γ at some vertex (see Fig. 3(a)). 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. 3(b-c)). 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. □

Lemma 2 For every face f of G, the boundary of f consists of two directed paths with common origin and destination (See Fig. 2(c)).

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 π_u 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. 4(a)). 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 Fig. 4(b-c)).

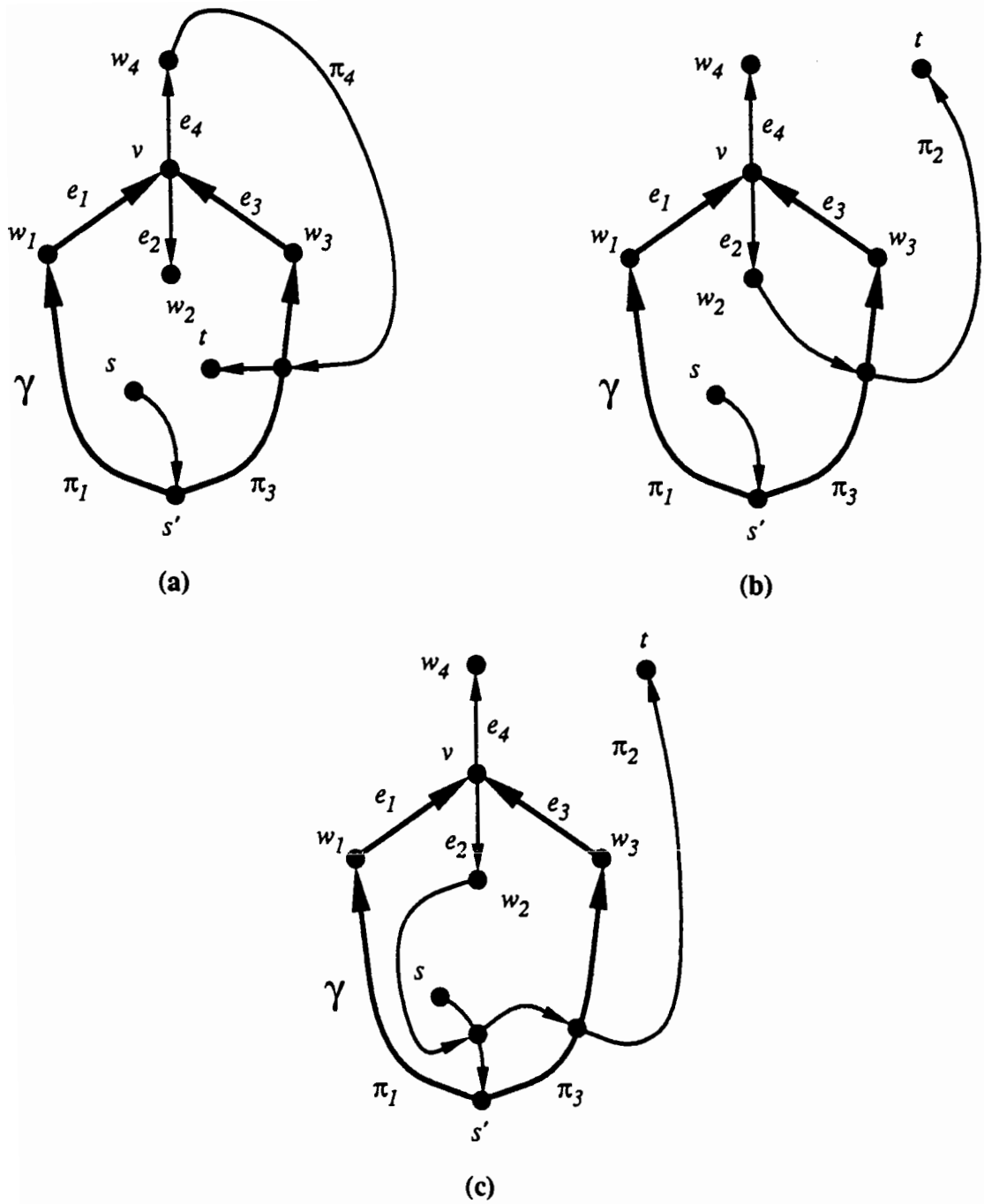


Figure 3 Examples for the proof of Lemma 1 **(a)** s and t in A; **(b-c)** s in A and t in B.

(*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. 4(d)). 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. 4(e)). 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. 4(f)).

(*iii*) and (*iv*) s and t are both in B, or s is in B and t is is in A.

These cases are analogous to the ones above, and their treatment is omitted for brevity. 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 of $V \cup E \cup F$, we define vertices $LOW(x)$ and $HIGH(x)$, and faces $LEFT(x)$ and $RIGHT(x)$, as follows:

(1) If $x = v \in V$, we define $LOW(v) = HIGH(v) = v$. Also, with reference to Lemma 1 and Fig. 2(b), 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.

(2) 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.

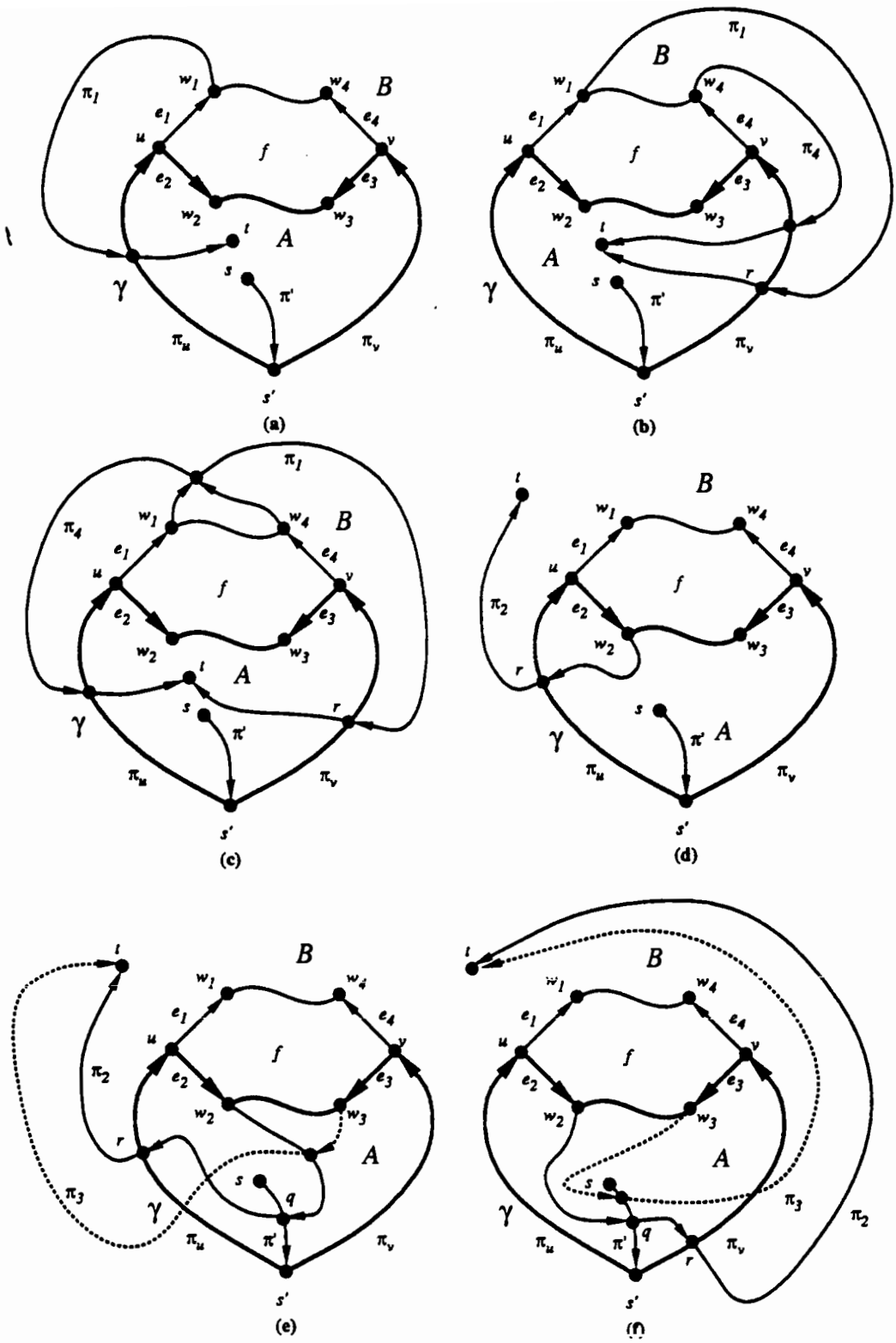


Figure 4 Examples for the proof of Lemma 2 **(a-c)** Case (*i*). **(d-f)** Case (*ii*).

(3) 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. 2(c)). 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. Notice 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. 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 [20].

Theorem 1 Let G be a plane graph. 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 $O(m)$-time algorithms for testing if G is *st*-orientable and constructing a spherical *st*-orientation for G.

Proof: It is proved in [20] that (4) $\Rightarrow$ (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) $\Rightarrow$ (2). Clearly, (2) $\Rightarrow$ (1). To complete the proof of the characterization, we show that (1) $\Rightarrow$ (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 v 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 $O(m)$ time. Finally, since computing the *st*-numbering of a planar graph can be done in $O(m)$ time [6], we also have the result that constructing a spherical *st*-orientation takes $O(m)$ time. □

Now, we turn our attention to operations that update a spherical *st*-graph by additions and deletions of vertices and edges. The operations *INSERTEDGE*, *INSERTVERTEX*, *REMOVEEDGE*, and *REMOVEVERTEX*, 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 directed plane graph obtained by performing operation Π on G. Depending on Π, G' is a spherical *st*-graph if and only if

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

(2) for $\Pi = INSERTVERTEX\ (e, v; e_1, e_2)$, there is no restriction;

(3) for $\Pi = REMOVEEDGE\ (e, u, v, f_1, f_2; f)$, $e = (u, v)$ must be an edge such that $\deg^+(u) \geq 2$ and $\deg^-(v) \geq 2$;

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

Proof: The proof of (2), (3), and (4) is straightforward. For operation *INSERTEDGE*, 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 *INSERTEDGE* $(e,u,v,f,f_1;f_2)$ and *INSERTEDGE* $(e,v,u,f,f_1;f_2)$ 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 *INSERTEDGE* $(e,u,v,f,f_1;f_2)$, while if there is a (directed) path from v to u, we perform operation *INSERTEDGE* $(e,v,u,f,f_1;f_2)$. Both operations are allowed when no directed path exists between u and v. □

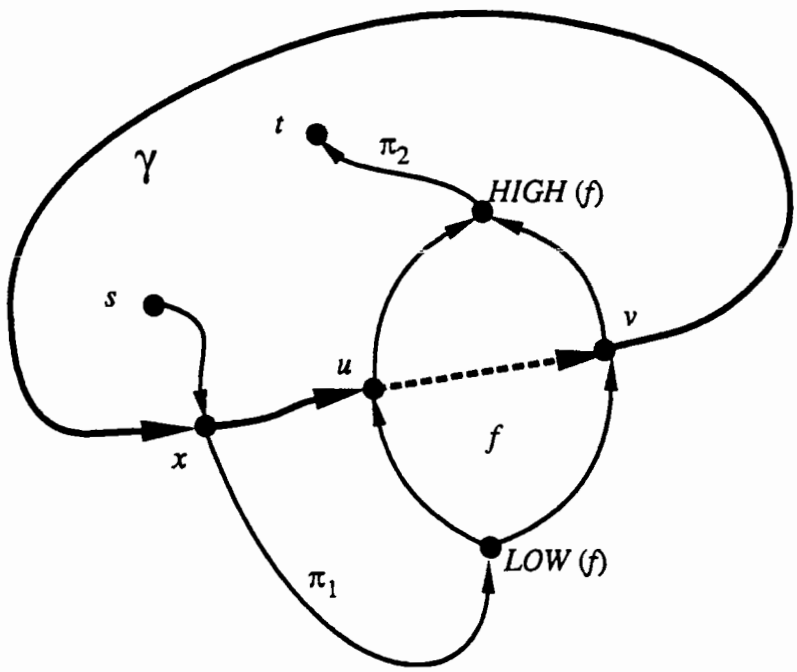


Figure 5 Example for the proof of Lemma 3

It is a relatively simple exercise to show that the above repertory of operations is complete for the class of *st*-2-connectible planar graphs.

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 $m-1$ *INSERTEDGE* and *INSERTVERTEX* operations, and can be disassembled to yield G_0 by means of $m-1$ *REMOVEEDGE* and *REMOVEVERTEX* operations.

4. Topological Location

In this section we consider the *topological location problem*, which consists of performing efficiently the *TEST* and *LIST* operation 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 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 simple and 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 [31], and was explicitly proved by Tutte [36, 37]. A linear-time algorithm for constructing convex drawings is presented in [4].

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 the 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). The extension to nonsimple 3-connected graphs is trivial, and we obtain:

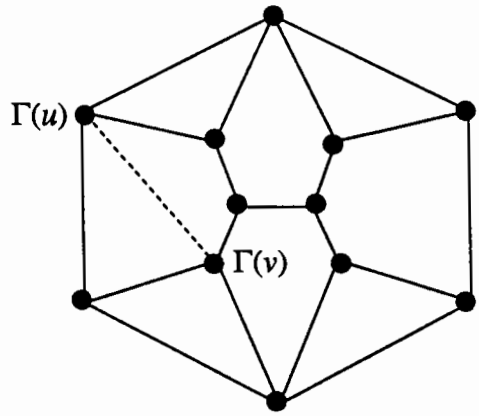


Figure 6 Operation $TEST(u, v)$ using a convex drawing.

Theorem 3 There exists a data structure for the topological location problem in 3-connected planar graph with the following performance:

$$Space(m) = Preprocess(m) = O(m);$$
$$Test(m) = O(\log m); \quad List(m, k) = O(\log m + k).$$

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

$$(f = LEFT(u) \textbf{ or } f = RIGHT(u)) \textbf{ and } (f = LEFT(v) \textbf{ or } f = RIGHT(v));$$

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

$$(u = LOW(f) \textbf{ or } u = HIGH(f)) \textbf{ and } (f = LEFT(v) \textbf{ or } f = RIGHT(v));$$

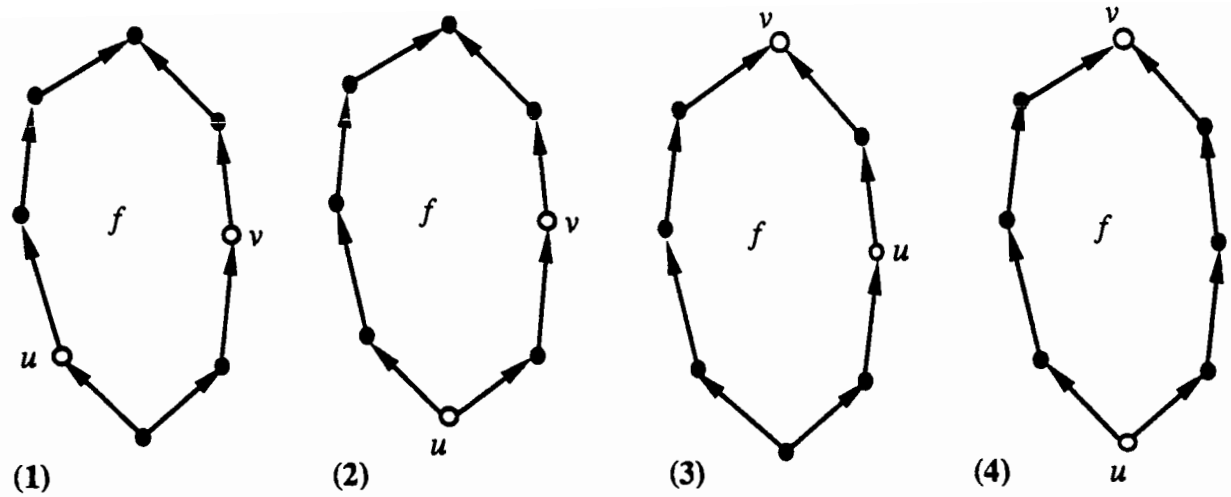


Figure 7 The four cases for two vertices on the same face

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

$$(v = LOW(f) \textbf{ or } v = HIGH(f)) \textbf{ and } (f = LEFT(u) \textbf{ or } f = RIGHT(u));$$

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

$$(u = LOW(f) \textbf{ and } v = HIGH(f)) \textbf{ or } (u = HIGH(f) \textbf{ and } v = LOW(f)).$$

In order to check Cases 1-3, we store for each vertex v the faces $LEFT(v)$ and $RIGHT(v)$, and for each face f the vertices $HIGH(f)$ and $LOW(f)$. 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 $BELOW(v)$ whose elements are the pairs $(f, LOW(f))$ such that $v = HIGH(f)$, sorted according to the alphabetic order of the name of $LOW(f)$. Hence, the test for Case 4 consists of searching for vertex u in $BELOW(v)$ and for vertex v in $BELOW(u)$, which takes $O(\log m)$ time. This proves the following theorem:

Theorem 4 There exists a data structure for the topological location problem in spherical *st*-graphs with the following performance:

$$Space(m) = Preprocess(m) = O(m);$$
$$Test(m) = O(\log m); \quad List(m,k) = O(\log m + k).$$

Corollary 2 There exists a data structure for the topological location problem in *st*-orientable graphs with the following performance:

$$Space\ (m) = Preprocess\ (m) = O(m);$$
$$Test\ (m) = O(\log m);\quad List\ (m,k) = O(\log m + k).$$

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 in 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 father of $node(v)$ in T;

(3) $node(v)$ is the father of $node(u)$ in T;

(4) the fathers of $node(u)$ and $node(v)$ are the same node of T, associated with a block.

We conclude

Lemma 4 There exists a data structure with $O(m)$ space and preprocessing time that allows for testing if two vertices belong to the same block 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 5 [13, 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 $OUTER(B)$. Notice 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, $B=OUTER(f)$, and we define $OUTER(v)=f$. As a straightforward consequence of the above definitions, we have

Lemma 6 The graph of relation $OUTER$ is a directed source tree, whose root is the external face.

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

Theorem 5 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) $f=OUTER(u)$ and $f=OUTER(v)$;

(2) $f=OUTER(u)$ and v is on face f of block $OUTER(f)$;

(3) $f=OUTER(v)$ and u is on face f of block $OUTER(f)$.

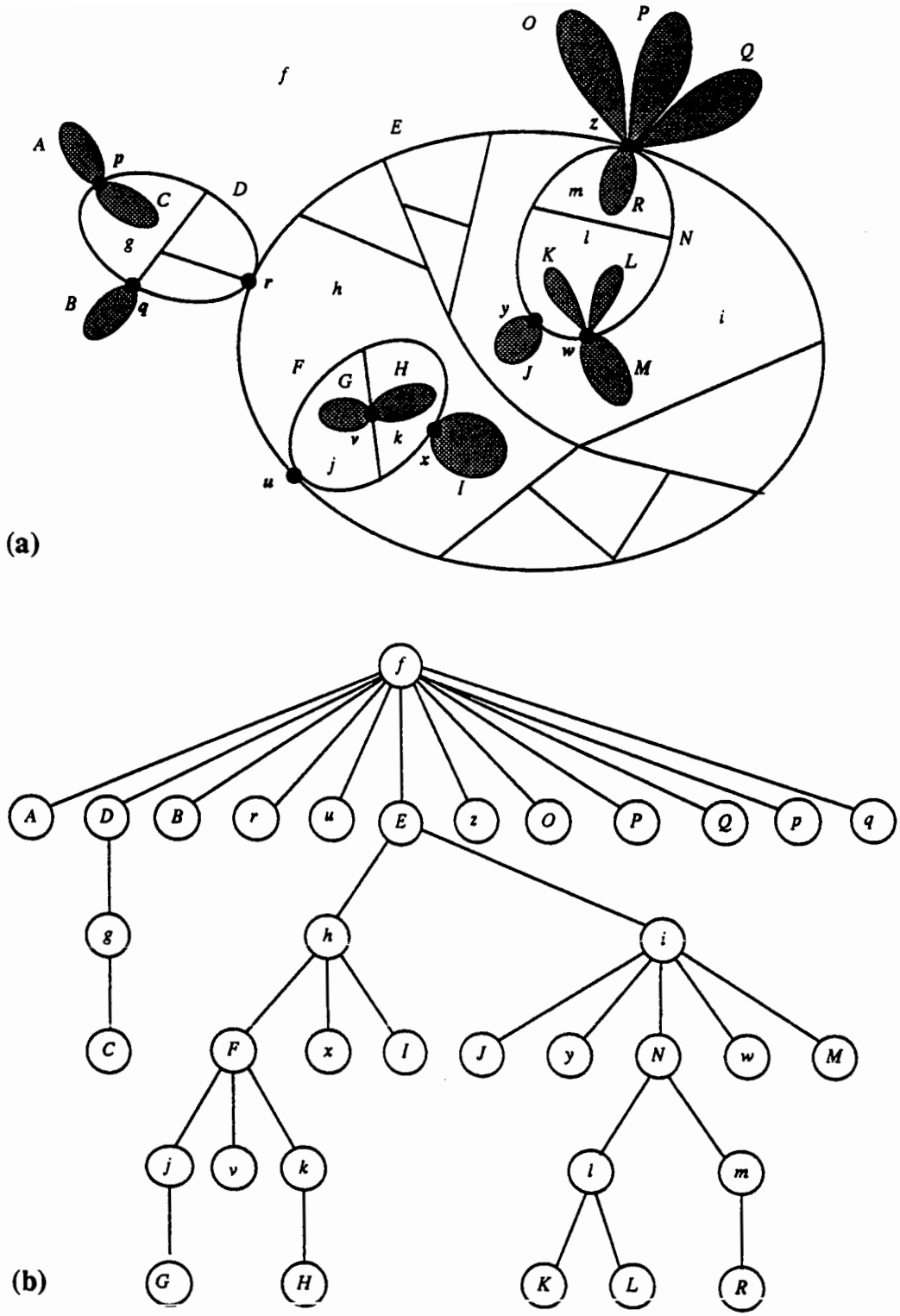


Figure 8 **(a)** A 1-connected graph and **(b)** its *OUTER* relation.

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 Lemma 4.

(2) A separate data structure for performing operations *TEST* and *LIST* in a 2-connected graph (see the previous subsection), for each block B of G.

(3) *OUTER* pointers for the blocks, vertices, and faces of G.

From Theorem 5, we conclude

Theorem 6 There exists a data structure that solves the topological location problem for general plane graphs with the following performance:

$$Space\ (m)=Preprocess\ (m)=O(m);$$
$$Test\ (m)=O(\log m);\quad List\ (m,k)=O(\log m+k).$$

4.4. Average query time Let $Q_{uv}(n,m)$ be the time to perform the query operation $TEST(u,v)$. We have shown in the previous subsection that in the worst case $Q_{uv}(n,m)=O(\log n)$. Here, we consider the average query time over all possible $\binom{n}{2}$ queries, defined by: $\overline{Q}(n,m)=\frac{1}{\binom{n}{2}}\sum_{\substack{u,v\\ u\neq v}} Q_{uv}(n,m)$.

Theorem 7 In the previously described data structure for the topological location problem, the average time for the *TEST* operation over all possible $\binom{n}{2}$ queries is $\overline{Q}(n,m)=O\left(\log\frac{m}{n}\right)$. In particular, if the graph is simple, then $\overline{Q}(n,m)=O(1)$.

Proof: From the description of the algorithm for the *TEST* operation we have that $Q_{uv}(n,m)=O(\log\deg^-(u)+\log\deg^-(v))$. Hence, $\binom{n}{2}\overline{Q}(n,m)=O(n\sum_v \log\deg^-(v))$. The proof is completed observing that $\sum_v \deg^-(v)=m$. □

5. Dynamic Planar Graph Embedding

In this section, we present the complete data structure for the dynamic embedding problem. First, we consider the 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* T_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* T_F, and in an *edge-tree* T_E, respectively. The above trees allow us to access in $O(\log m)$ 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 *INSERTEDGE* 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 left path of f_1. Notice 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):

$$LPATH(f_1) = L_2; \quad RPATH(f_1) = R_1 e R_3; \quad LPATH(f_2) = L_1 e L_3; \quad RPATH(f_2) = R_2.$$

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 [21, pp. 213-216]).

The remaining updates of the data structure are as follows:

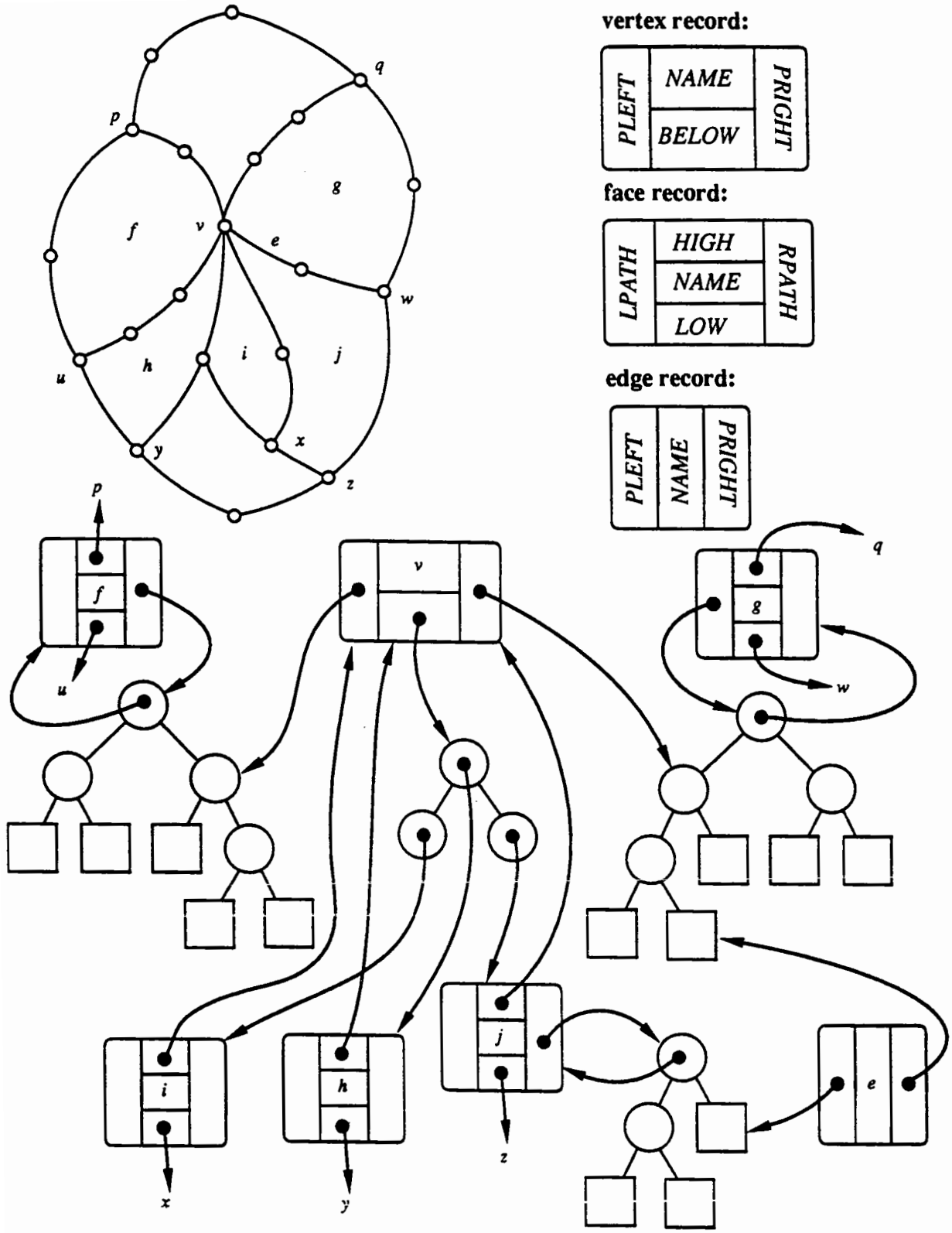


Figure 9 A spherical *st*-graph and a fragment of the dynamic data structure for it. The edges are oriented upward.

(1) Insert into T_E a new record for edge e, and increment counters $\deg^+(u)$ and $\deg^-(v)$.

(2) Delete from T_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 *INSERTVERTEX* (e, v, e_1, e_2) operation is performed as follows:

(1) Delete from T_E the record for edge e, and insert new records for edges e_1 and e_2.

(2) Insert into T_V a new record for vertex v.

(3) Find the faces f and g respectively to the left and right of edge e by walking up to the roots of the trees that contain $PLEFT(e)$ and $PRIGHT(e)$.

(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 $LPATH(g)$.

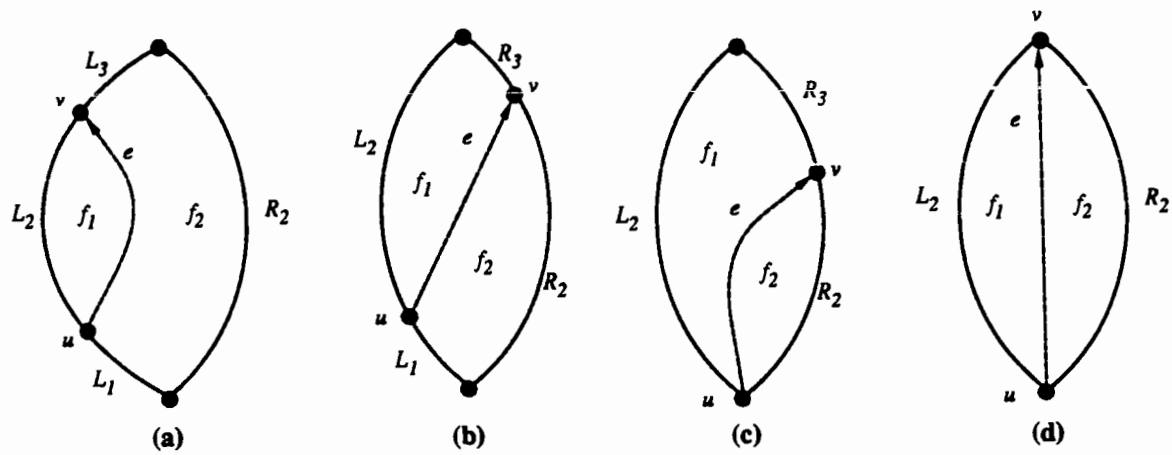


Figure 10 Restructuring of the face boundaries after an *INSERTEDGE* operation. **(a)** u and v on left path; **(b)** u on left path and v on right path; **(c)** u bottommost and v on right path; **(d)** u bottommost and v topmost.

(6) Set $BELOW(v) := \emptyset$.

The above data structure also supports operations *REMOVEEDGE* and *REMOVEVERTEX*. In fact, the *REMOVEEDGE* operation can be performed by reversing the transformations on the data structure realized by the algorithm for the *INSERTEDGE* operation. Similarly, the *REMOVEVERTEX* operation is the reverse of the *INSERTVERTEX* operation. Notice also that, by Lemma 3, the counters $\deg^+(v)$ and $\deg^-(v)$ 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 8 There is a data structure for the dynamic embedding problem in spherical *st*-graphs with the following performance:

$$Space(m) = Preprocess(m) = O(m);$$
$$Test(m) = O(\log m); \quad List(m,k) = O(\log m + k);$$
$$InsertEdge(m) = InsertVertex(m) = RemoveEdge(m) = RemoveVertex(m) = O(\log m).$$

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 *INSERTEDGE* and *INSERTVERTEX* operations is optimal. For the definition of amortized time complexity, see [35].

Theorem 9 There exists a data structure for the dynamic embedding problem in spherical *st*-graphs such that:

(1) The space occupation and time complexity of the various operations are the same as in Theorem 8.

(2) In a sequence of *INSERTEDGE* and *INSERTVERTEX* 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* [12]) to realize trees T_F, T_V, T_E, and $BELOW(v)$. Such trees have $O(1)$ amortized rebalancing time for insertions and deletions [16]. With regard to the *LPATH* and *RPATH* trees, their manipulation in a sequence of *INSERTEDGE* and *INSERTVERTEX* operations involves insertions and the kind of generalized

splittings considered in [14]. It is shown there that *circular level-linked 2-4 trees* support efficiently a sequence of insertions and generalized splittings. With arguments similar to the ones developed in [14] 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*, *LIST*, and *INSERTVERTEX* do not require any modifications. In connection with operation *INSERTEDGE* $(e, u, v, f; f_1, f_2)$, 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 *INSERTEDGE* algorithm rejects the operation. This proves

Theorem 10 There is a data structure that supports operations *TEST*, *LIST*, *INSERTEDGE*, and *INSERTVERTEX* in a *st*-orientable plane graph with the following performance:

$$Space(m) = Preprocess(m) = O(m);$$
$$Test(m) = O(\log m); \quad List(m, k) = O(\log m + k);$$
$$InsertEdge(m) = InsertVertex(m) = O(\log m).$$

In regard to the *REMOVEEDGE* and *REMOVEVERTEX* operations, we are faced with the difficulty that the data structure acts on a spherical *st*-orientation of the graph, therefore permitting only deletions that preserve the *st*-structure of the orientation. We say that a vertex (edge) is *free* if operation *REMOVEVERTEX* (*REMOVEEDGE*) can be performed on it in the spherical *st*-orientation; we say that it is *locked* otherwise. At any time the vertices and edges of the graph are partitioned into free and locked, and we are allowed to delete only vertices and edges that are free. We thus have

Theorem 11 The data structure of Theorem 10 supports also operations *REMOVEEDGE* and *REMOVEVERTEX* on free edges and vertices in $O(\log m)$ time.

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 edges and vertices is sufficiently large to support a hierarchical deletion scheme that allows us to "undo" any *INSERTEDGE* and *INSERTVERTEX* operation performed in the past (not just the last operation). For instance, we define the *hierarchical embedding problem* as a variation of the previously discussed dynamic embedding problem, where the following restrictions are placed on the *REMOVEEDGE* and *REMOVEVERTEX* operations:

(1) An edge can be deleted by a *REMOVEEDGE* operation only if it was created (at any time in the past) by means of an *INSERTEDGE* operation.

(2) A vertex can be deleted by a *REMOVEVERTEX* operation only if it was created (at any time in the past) by an *INSERTVERTEX* operation.

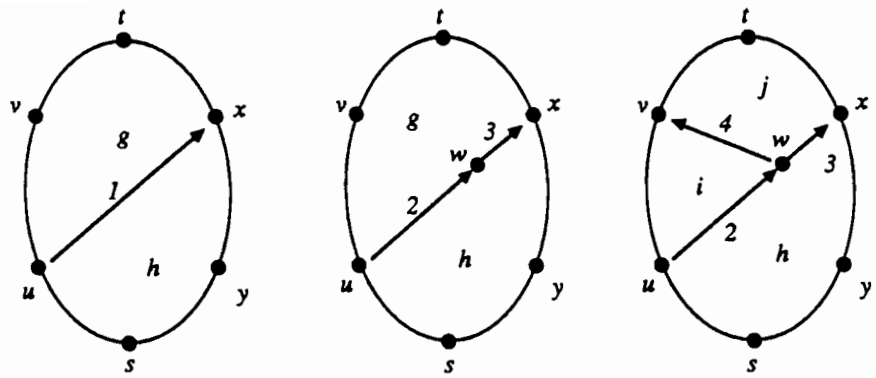
In Fig. 11 we show a sequence of update operations in an instance of the hierarchical embedding problem.

The aforementioned restrictions on the *REMOVEEDGE* and *REMOVEVERTEX* operations can be enforced by storing with each edge e two flags, denoted *FREE–TAIL* (e) and *FREE–HEAD* (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. 11, a flag in the "set" condition is indicated by showing the corresponding endpoint unconnected.

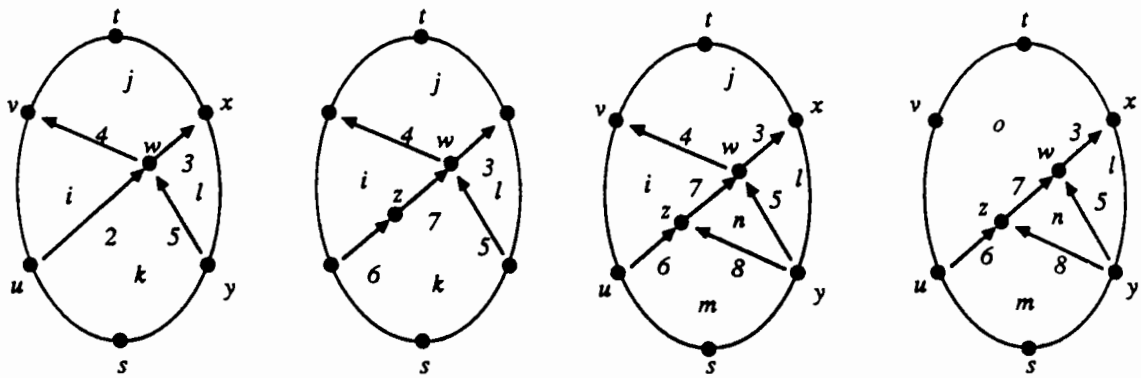
It is not difficult to show that the free edges and vertices include the ones that can be deleted in an instance of the hierarchical embedding problem. We have thus the following theorem:

Theorem 12 There exists a data structure that allows us to solve the hierarchical embedding problem for *st*-orientable plane graphs with the following performance:

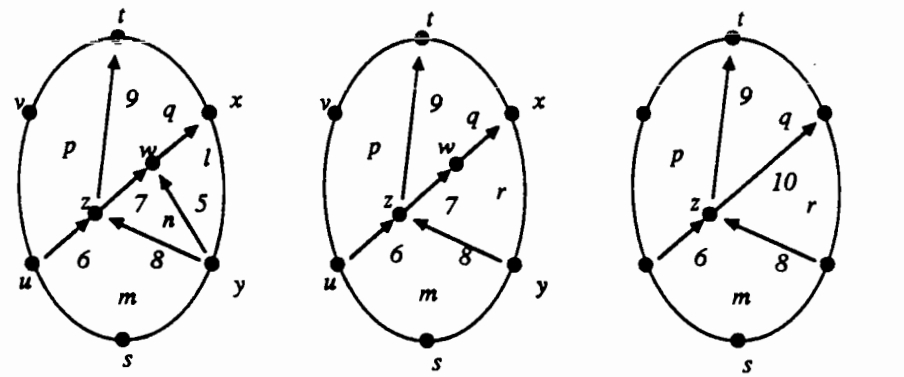
$$Space\ (m) = Preprocess\ (m) = O(m);$$
$$Test\ (m) = O(\log m); \quad List\ (m,k) = O(\log m + k);$$
$$InsertEdge\ (m) = InsertVertex\ (m) = RemoveEdge\ (m) = RemoveVertex\ (m) = O(\log m).$$



INSERTEDGE (1,u,x,f,g,h) *INSERTVERTEX (1,w,2,3)* *INSERTEDGE (4,w,v,g,i,j)*



INSERTEDGE (5,y,w,h,k,l) *INSERTVERTEX (2,z,6,7)* *INSERTEDGE (8,y,z,k,m,n)* *REMOVEEDGE (4,o)*



INSERTEDGE (9,z,t,o,p,q) *REMOVEEDGE (5,r)* *REMOVEVERTEX(w,10)*

Figure 11 Sequence of hierarchical update operations on a plane graph. Flags in the "set" condition are shown by disconnecting the corresponding endpoint.

5.3. General plane graphs For general plane graphs we add to the repertory the following operations, which allow respectively to add and remove an elementary block to the graph:

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

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

In the following, we will provide an *amortized* $O(\log m)$ time bound for operation *INSERTEDGE*, and a worst-case $O(\log m)$ time bound for the remaining operations. According to standard conventions in amortized complexity analysis [35], this means that a sequence of m operations starting from an initial graph consisting of a single edge takes $O(m \log m)$ time.

First, we dynamize the data structure that tests whether two vertices belong to the same block of a graph G. We represent the oriented block-cutvertex tree T of G by means of a balanced search tree for the 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). We have

Lemma 7 There exists a data structure with $O(m)$ space and preprocessing time that allows for testing whether two vertices belong to the same block in $O(\log m)$ time (worst-case), and supports operations *INSERTVERTEX* and *ATTACH* in $O(\log m)$ worst-case time, and operation *INSERTEDGE* in $O(\log m)$ amortized time.

We now show how to dynamically maintain the *OUTER* relation. For each block B, we define *INT* (B) as the list of the cutvertices of B such that *OUTER* $(v) = B$. Also, we define a list *EXT* (B) for each block B and a circular list *INNER* (f) for each cutface f by means of the following constructive procedure.

Consider a cutface f whose outer block is B_0. Initially, all cutvertices of f in B_0 are marked, while the remaining cutvertices of f are unmarked. Also, all the inner blocks of f are initially unmarked. Suppose now that we traverse the boundary of f counterclockwise, starting at a cutvertex. Whenever we traverse an edge of an unmarked block B, we add (a representative of) B into *INNER* (f) and mark B. Also, whenever we encounter an unmarked cutvertex v, we mark v and add (a representative of) v into the list *EXT* (B) of the first block B in the current

$INNER(f)$ that contains v. At the end of the traversal, $INNER(v)$ will contain all the inner blocks of f, and the lists $EXT(B)$ will contain all the cutvertices v of f such that $f = OUTER(v)$. Notice that each block B is contained in exactly one $INNER(f)$ list, and each cutvertex v is contained in exactly one $EXT(B)$ or $INT(B)$ list.

In the example of Fig. 12(a) we have

$$INNER(f) = (B_2, B_1, B_3, B_5, B_4, B_6, B_9, B_7, B_8);$$

$$EXT(B_2) = (v_1, v_2);\ EXT(B_3) = (v_4);\ EXT(B_5) = (v_5, v_6);\ EXT(B_7) = (v_7);$$

$$EXT(B_1) = EXT(B_4) = EXT(B_6) = EXT(B_8) = EXT(B_9) = \emptyset.$$

Implementing the above lists by means of balanced search trees, $OUTER(\cdot)$ can be easily retrieved in $O(\log m)$ time, so that the complexity of the *TEST* and *LIST* operations is the same as in the static case.

Now, we describe the transformations of the data structure due to the execution of operation *INSERTEDGE* $(e, u, v, f; f_1, f_2)$. 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 a sequence of blocks of G, called *old-blocks* are merged into a new block, called *new-block*. In the example of Fig. 12(b), 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 $e = (u, v)$.

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 the ones 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.

In conclusion, the modification of the block structure caused by the *INSERTEDGE* operation requires that we set up a new data structure for the new-block B, and new *INNER* lists for f_1 and f_2.

Now, we show that the data structure for B can be set up 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 $B_1, \cdots, B_l$, with $u \in B_1$ and $v \in B_l$, and let v_i, $i = 1, \cdots, l-1$, be the cutvertex between B_i and B_{i+1}. Also, let $v_0 = u$ and $v_l = v$. (In the example of Fig. 12(b) $l = 3$.) 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}),

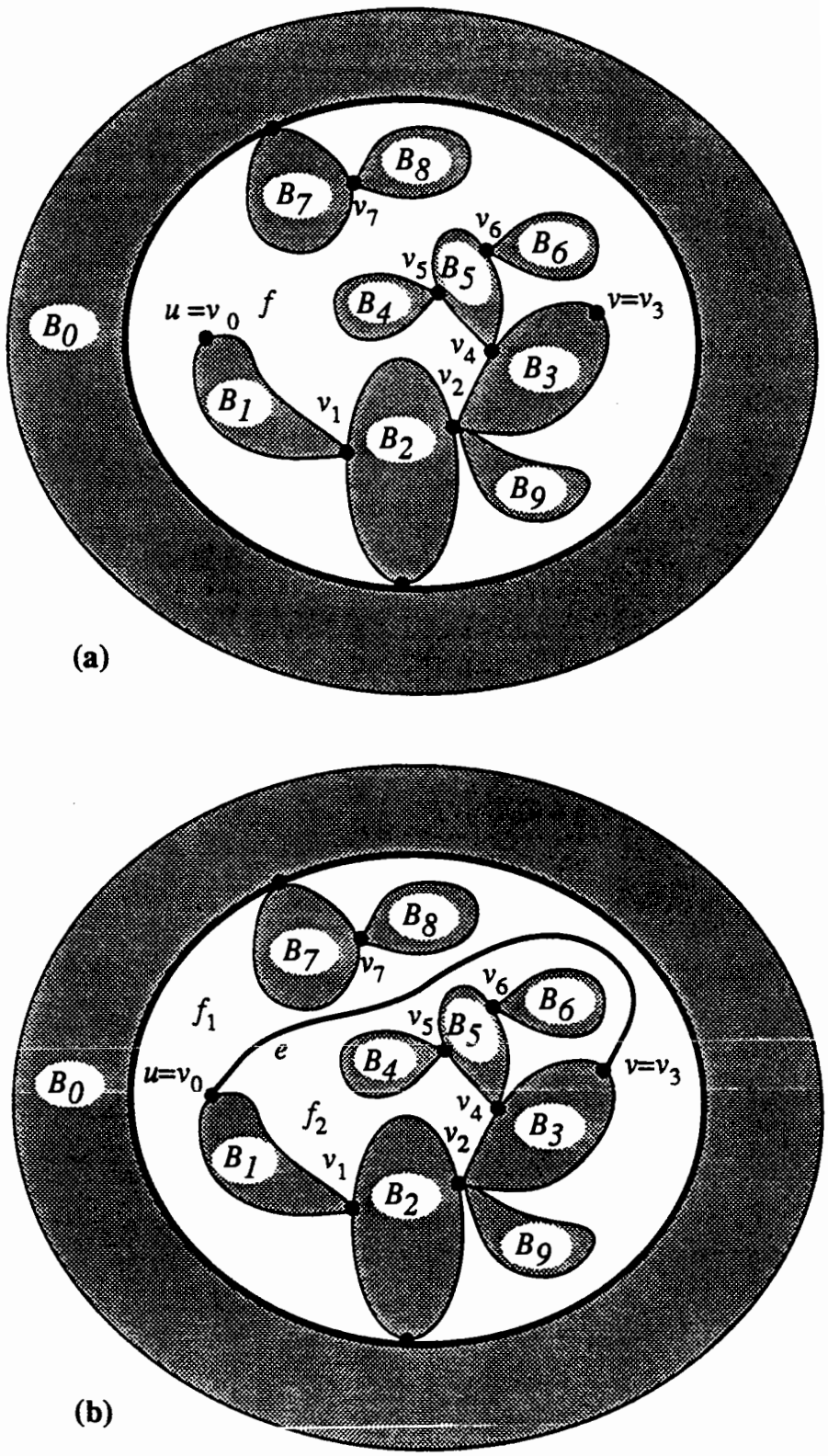


Figure 12 Example of *INSERTEDGE* operation in a cut-face **(a)** Before insertion. **(b)** After insertion.

say (v_j, v_{j-1}), can be added to B_j while preserving the spherical *st*-orientation. 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, 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.) The resulting orientation of the new-block B is a spherical *st*-orientation, from which the data structure for B can be constructed in time:

$$O(\log m + \sum_{i=1}^{l} m_i - \max_{i=1,\cdots,l} m_i),$$

where m_i is the number of edges of B_i.

Lists $INNER(f_1)$ and $INNER(f_2)$ are obtained from $INNER(f)$ by

(1) removing the (representatives of the) old-blocks;

(2) splitting the resulting list into $INNER(f_1)$ and $INNER(f_2)$ with an appropriate cut; and

(3) possibly adding to one of these lists (a representative of) the new-block B.

With regard to the *EXT* lists, we form $INT(B)$ and $EXT(B)$ by a sequence of $O(l)$ split and splice operations on the *EXT* lists of the old-blocks.

In the example of Fig. 12, we have

$$INNER(f_1) = (B, B_9, B_7, B_8);\ INNER(f_2) = (B_5, B_4, B_6);\ EXT(B) = (v_2);\ INT(B) = (v_1, v_4).$$

Theorem 13 There is a data structure that supports operations *TEST*, *LIST*, *INSERTEDGE*, *INSERTVERTEX*, and *ATTACH* with the following performance:

$$Test(m) = O(\log m);\quad List(m, k) = O(\log m + k);$$
$$InsertEdge(m) = O(\log m) \text{ (amortized)};\quad InsertVertex(m) = Attach(m) = O(\log m).$$

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

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

where $B_1, \cdots, B_b$ are the current blocks of G, and m_i is the number of edges of B_i ($\sum_{i=1}^{b} m_i = m$). Notice that $-m \log m \leq \Phi \leq 0$. It is interesting to observe that the function Φ 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, Φ 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 blocks of B^*. By an appropriate choice of the time unit, we have that operation *INSERTEDGE* is executed in time

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

Now, 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 $x_1 = \frac{m_1}{m^*}$ and $x_2 = \frac{m_2}{m^*}$. Clearly, $x_1 + x_2 = 1$. We can write

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

$$= - \left[x_1 \log \frac{1}{x_1} + x_2 \log \frac{1}{x_2} \right] = -H(x_1, x_2).$$

where $H(x_1, x_2)$ is the *binary entropy function* of information theory. We show in Fig. 13 a plot of $H(x_1, 1 - x_1)$ in the interval $x_1 \in [0,1]$. It is easy to see that

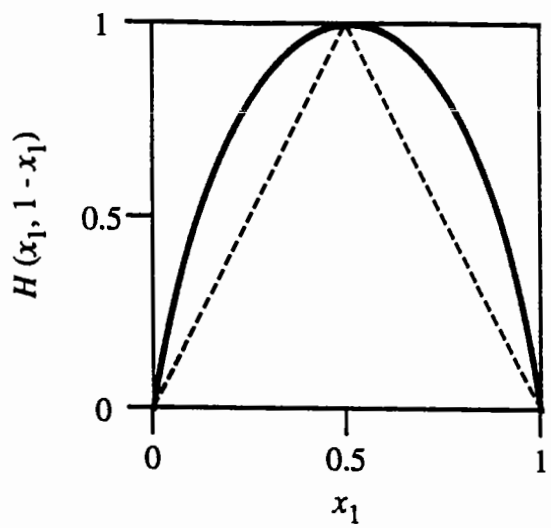


Figure 13 Plot of the binary entropy function $H(x_1, 1 - x_1)$.

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

Therefore, we obtain

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

Hence, the amortized time complexity of operation *INSERTEDGE* is

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

With regard to operations *REMOVEEDGE*, *REMOVEVERTEX*, and *DETACH*, again we can partition the edges of the graph into free and locked, and we have the following:

Corollary 3 The data structure of Theorem 13 also supports operations *REMOVEEDGE*, *REMOVEVERTEX*, and *DETACH* on free edges and vertices in $O(\log m)$ time (worst-case).

6. Applications

6.1. A dual embedding problem The *dual dynamic embedding problem* 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.

*LIST** (f, g): List all the vertices that are on the boundaries of both faces f and g.

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

CONTRACTVERTEX $(e, f, g, v_1, v_2; v)$: 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.

DUPLICATEEDGE $(e, f; e_1, e_2)$: 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.

MERGEEDGE $(e_1, e2, f; e)$: 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. Notice that every 2-connected graph is s^*t^*-orientable. Using the results of Section 4 and duality arguments, we obtain

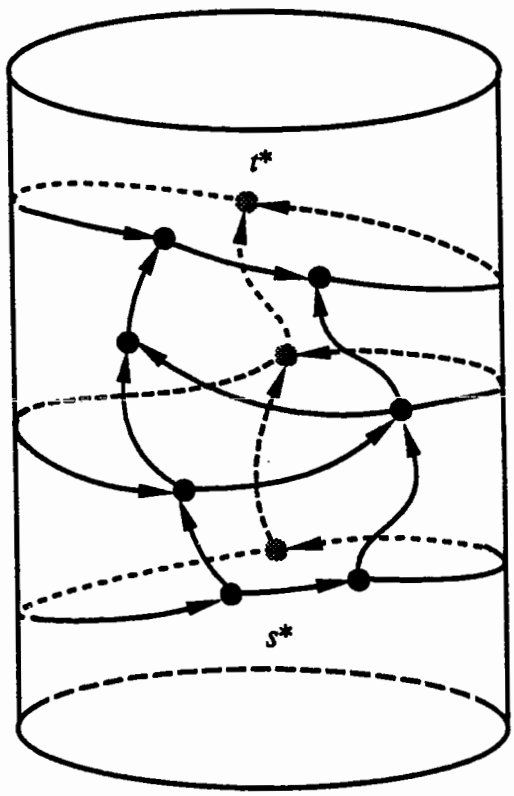


Figure 14 Example of cylindrical s^*t^*-graph

Theorem 14 There is a data structure that allows us to solve the dual dynamic embedding problem for cylindrical s^*t^*-graphs with the following performance:

$$Space\ (m) = Preprocess\ (m) = O(m);$$
$$Test^*(m) = O(\log m); \quad List^*(m,k) = O(\log m + k);$$
$$ExpandVertex(m) = DuplicateEdge(m) = ContractVertex(m) = MergeEdge(m) = O(\log m).$$

Further results can be obtained by dualizing the remaining theorems of the previous section.

The *hierarchical dual embedding problem* is defined as a variation of the dynamic embedding problem where the *CONTRACTVERTEX* and *MERGEEDGE* operations can performed only on edges (respectively, faces) previously inserted by the *EXPANDVERTEX* (respectively, *DUPLICATEEDGE*) operation. We have

Theorem 15 There is a data structure that allows us to solve the hierarchical dual embedding problem for s^*t^*-orientable plane graphs with the following performance:

$$Space\ (m) = Preprocess\ (m) = O(m);$$
$$Test^*(m) = O(\log m); \quad List^*(m,k) = O(\log m + k);$$
$$ExpandVertex(m) = DuplicateEdge(m) = ContractVertex(m) = MergeEdge(m) = O(\log m).$$

6.2. Computing separating pairs The identification of separating pairs is important in problems of fault-tolerance of networks. Our data structure allows to efficiently detect separating pairs.

Lemma 8 Let G be a 2-connected plane graph. Vertices u and v form a separating pair of G if and only if

(1) u and v are adjacent and there are at least three faces whose boundary contains both u and v; or

(2) u and v are not adjacent and there are at least two faces whose boundary contains both u and v.

Hence, we can test whether two vertices form a separating pair by a simple modification of the algorithm for operation *LIST* (u,v), which halts as soon as two or three faces have been reported. We obtain

Corollary 4 The data structure of Theorem 12 allows us to test whether two given vertices form a separating pair in $O(\log m)$ time.

Acknowledgment

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

References

[1] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North Holland, Amsterdam, 1976.

[2] K. Booth and G. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms," *J. of Computer and System Sciences*, vol. 13, pp. 335-379, 1976.

[3] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees," *J. of Computer and System Sciences*, vol. 30, no. 1, pp. 54-76, 1985.

[4] N. Chiba, K. Onoguchi, and T. Nishizeki, "Drawing Planar Graphs Nicely," *Acta Informatica*, vol. 22, pp. 187-201, 1985.

[5] G. Di Battista and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs," *Theoretical Computer Science*, vol. 61, pp. 175-198, 1988.

[6] S. Even and R.E. Tarjan, "Computing an st-Numbering," *Theoretical Computer Science*, vol. 2, pp. 339-344, 1976.

[7] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[8] S. Even and Y. Shiloach, "An On-Line Edge Deletion Problem," *J. ACM*, vol. 28, pp. 1-4, 1981.

[9] H. de Fraysseix and P. Rosenstiehl, "A Depth-First-Search Characterization of Planarity," *Annals of Discrete Mathematics*, vol. 13, pp. 75-80, 1982.

[10] H. de Fraysseix, J. Pach, and R. Pollack, "Small Sets Supporting Fary Embeddings of Planar Graphs," *Proc. 20th ACM Symp. on Theory of Computing*, pp. 426-433, 1988.

[11] G. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications," *SIAM J. Computing*, vol. 14, no. 4, pp. 781-798, 1985.

[12] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pp. 8-21, 1978.

[13] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[14] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R.E. Tarjan, "Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees," *Information and Control*, vol. 68, pp.

170-184, 1986.

[15] J. Hopcroft and R.E. Tarjan, "Efficient Planarity Testing," *J. ACM*, vol. 21, no. 4, pp. 549-568, 1974.

[16] S. Huddleston and K. Mehlhorn, "A New Data Structure for Representing Sorted Lists," *Acta Informatica*, vol. 17, pp. 157-184, 1982.

[17] T. Ibaraki and N. Katoh, "On-Line Computation of Transitive Closure of Graphs," *Information Processing Letters*, vol. 16, pp. 95-97, 1983.

[18] G.F. Italiano, "Amortized Efficiency of a Path Retrieval Data Structure," *Theoretical Computer Science*, vol. 48, pp. 273-281, 1986.

[19] P.N. Klein and J.H. Reif, "An Efficient Parallel Algorithm for Planarity," *J. Computer and System Sciences*, vol. 37, no. 2, pp. 190-246, 1988.

[20] A. Lempel, S. Even, and I. Cederbaum, "An Algorithm for Planarity Testing of Graphs," *Theory of Graphs, Int. Symposium*, Rome, pp. 215-232, 1966.

[21] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, New York, 1984.

[22] J.A. La Poutre' and J. van Leeuwen, "Maintenance of Transitive Closures and Transitive Reductions of Graphs," *Graph-Theoretic Concepts in Computer Science* (Proc. Int. Workshop WG '87, Kloster Banz, June 1987), H. Gottler and H.J. Schneider (Eds.), *Lecture Notes in Computer Science*, vol. 314, Springer-Verlag, pp. 106-120, 1988.

[23] F.P. Preparata and R. Tamassia, "Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures," *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pp. 558-567, 1988.

[24] F.P. Preparata and R. Tamassia, "Fully Dynamic Point Location in a Monotone Subdivision," *SIAM J. Computing*, vol. 18, no. 4, 1989 (to appear).

[25] F.P. Preparata and R. Tamassia, "Dynamic Planar Point Location with Optimal Query Time," *Lecture Notes in Computer Science* (Proc. STACS '89)), Springer-Verlag, 1989 (to appear).

[26] F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[27] J.H. Reif, "A Topological Approach to Dynamic Graph Connectivity," *Information Processing Letters*, vol. 25, pp. 65-70, 1987.

[28] H. Rohnert, "A Dynamization of the All-Pairs Least Cost Problem," *Lecture Notes in Computer Science* (Proc. STACS '85), vol. 182, Springer Verlag, pp. 279-286, 1985.

[29] P. Rosenstiehl and R.E. Tarjan, "Rectilinear Planar Layouts of Planar Graphs and Bipolar Orientations," *Discrete & Computational Geometry*, vol. 1, no. 4, pp. 342-351, 1986.

[30] W. Schnyder, "Embedding Planar Graphs on the Grid," *Congressun Numerantium*, 1989 (to appear).

[31] E. Steinitz and H. Rademacher, *Vorlesung uber die Theorie der Polyeder*, Springer, Berlin, 1934.

[32] R. Tamassia and I.G. Tollis, "A Unified Approach to Visibility Representations of Planar Graphs," *Discrete & Computational Geometry*, vol. 1, no. 4, pp. 321-341, 1986.

[33] R. Tamassia and I.G. Tollis, "Centipede Graphs and Visibility on a Cylinder," in *Graph-Theoretic Concepts in Computer Science*, (Proc. Int. Workshop WG '86, Bernierd, June 1986), G. Tinhofer and G. Schmidt (Eds.), Lecture Notes in Computer Science, vol. 246, Springer-Verlag, pp. 252-263, 1987.

[34] R. Tamassia and F.P. Preparata, "Dynamic Maintenance of Planar Digraphs, with Applications," *Algorithmica*, 1989 (to appear).

[35] R.E. Tarjan, "Amortized Computational Complexity," *SIAM J. Algebraic Discrete Methods*, vol. 6, pp. 306-318, 1985.

[36] W.T. Tutte, "Convex Representations of Graphs," *Proc. London Math Soc.*, vol. 10, pp. 304-320, 1960.

[37] W.T. Tutte, "How to Draw a Graph," *Proc. London Math Soc.*, vol. 3, no. 13, pp. 743-768, 1963.