October 1987

UILU-ENG-87-2265
ACT-83

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

A DYNAMIC DATA STRUCTURE FOR PLANAR GRAPH EMBEDDING

Roberto Tamassia

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-87-2265 ACT #83	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (If applicable) N/A
7a. NAME OF MONITORING ORGANIZATION National Science Foundation	
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 1800 G Street, N.W. Washington, D.C. 20550
8a. NAME OF FUNDING / SPONSORING ORGANIZATION National Science Foundation	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ECS-84-10902	
8c. ADDRESS (City, State, and ZIP Code) 1800 G Street, N.W. Washington, D.C. 20550	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A Dynamic Data Structure for Planar Graph Embedding

12. PERSONAL AUTHOR(S)
Tamassia, Roberto

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	1987, October 26	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
planar graph, planar embedding, dynamic data structure, on-line algorithm, analysis of algorithms

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present a dynamic data structure that allows for incrementally constructing a planar embedding of a planar graph. 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 $O(\log n)$, where n is the number of vertices of the graph. The space used and the preprocessing time are $O(n)$. This work finds applications in circuit layout, graphics, motion planning, and computer-aided design.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

A DYNAMIC DATA STRUCTURE FOR PLANAR GRAPH EMBEDDING*

Roberto Tamassia

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

We present a dynamic data structure that allows for incrementally constructing a planar embedding of a planar graph. 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 $O(\log n)$, where n is the number of vertices of the graph. The space used and the preprocessing time are $O(n)$. This work finds applications in circuit layout, graphics, motion planning, and computer-aided design.

* Research partially supported by the National Science Foundation under grant ECS-84-10902.

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 problem of testing the planarity of a graph and of constructing a planar embedding has been extensively studied in the past years, and the development of linear time algorithms for it has brought significant advances in algorithm design and analysis [2, 14]. Nevertheless, as confirmed by recent results [3, 10], graph planarity is a still 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 preliminary nature, and limited to connectivity [9], minimum spanning tree [11], transitive closure [16, 17], and shortest path [24]. In fact, in several of the above data structures the capability of handling update operations is limited, and the space/time performance appears far form 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 parallel 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 if there is a face f that has both vertices u and v on its boundary. In case such a face f exists, output its name.

LIST (u,v):	List all the faces that have both vertices u and v on their boundary.
ADD (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.
INSERT (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.
REMOVE (e,f):	Remove the edge e, and merge faces f_1 and f_2 formerly on the two sides of e into face f.
JOIN (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. Notice that if G is simple, i.e. it has no parallel edges, $m=O(n)$, so that the above bounds become $O(n)$ space requirement and preprocessing time, and $O(\log n)$ query and update times. In addition to the good theoretical space/time performance, our data structure is also practical and easy to implement, and therefore suited for real-life 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 such orientation. Besides the applications 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 [25,27,28].

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 [6,18,20,26]. 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. Preliminary results on dynamic point-location are given in [12, 23].

This work constitutes also a first step in the direction 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 on orientations of planar graphs. In Section 3, we study the *topological location problem*, which consists of performing operations *TEST* and *LIST*. Section 4 describes the fully fledged data structure for the dynamic embedding problem. Finally, applications to a dual problem are discussed in Section 5.

2. ORIENTATIONS OF PLANAR GRAPHS

We consider only planar finite undirected and directed graphs without self-loops and without isolated vertices. We allow parallel edges between two vertices. For the basic terminology about graphs and planarity, see [1, 8].

Before introducing the following definitions, we recall that a *source* (*sink*) of a digraph is a vertex that has no incoming (outgoing) edges. A *spherical st-graph* is a plane digraph G such that:

(1) G has exactly one source, s, and exactly one sink t;

(2) every vertex v of G is on some directed simple path from s to t; and

(3) every directed cycle separates s from t.

We can visualize a spherical st-graph as embedded in a sphere, with s and t at the South and North pole, respectively (see Fig. 1).

The concept of spherical st-graph extends the one of *planar st-graph* introduced in [21], which has important applications in the test of graph planarity [21] and the construction of

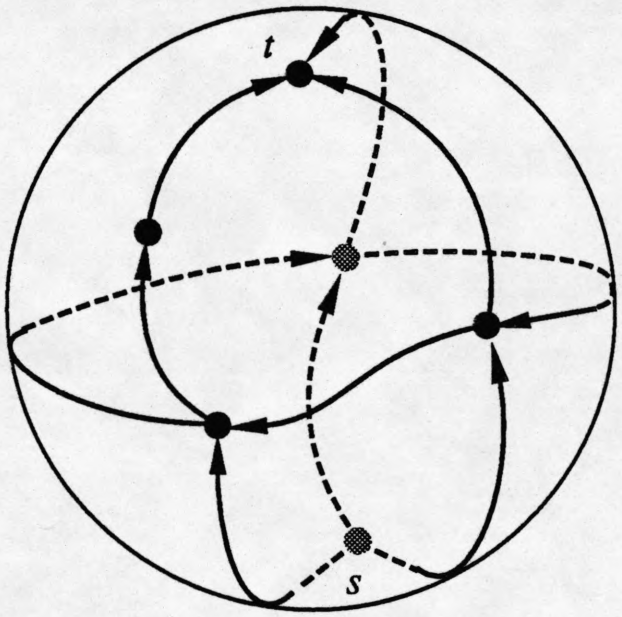
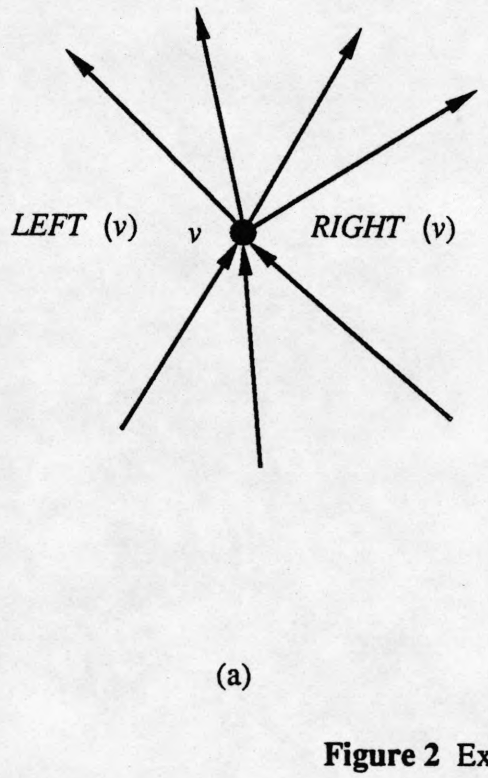


Figure 1 Example of spherical st-graph

planar drawings [5,25,27].

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

Proof: Assume, for a contradiction, that there is a vertex v, $v \neq s,t$, for which the lemma is not true. 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. Vertices w_1, w_2, w_3, and w_4 must be distinct, otherwise there is a face consisting of two edges whose boundary is a cycle that does not separate s from t, which would violate Property (3). From 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



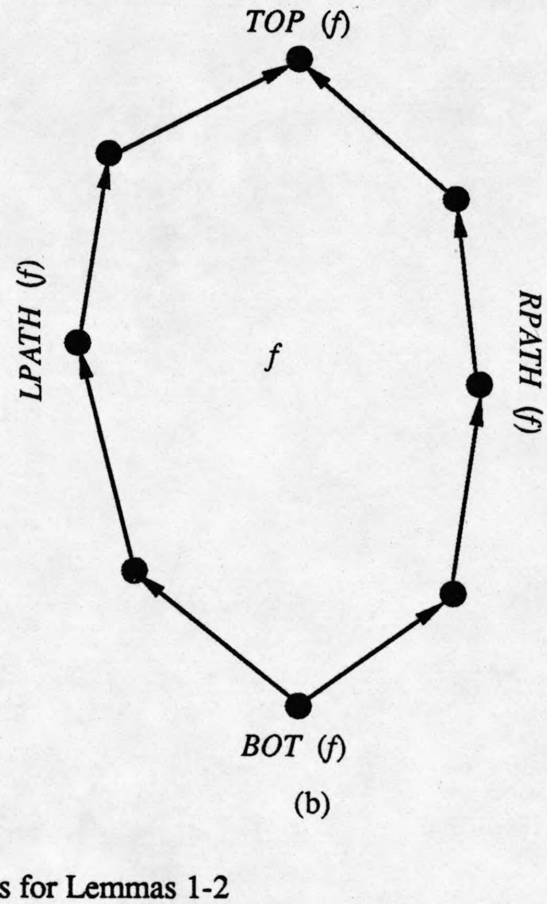


Figure 2 Examples for Lemmas 1-2

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. From 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. 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 similar. 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.b).

Proof: Assume, for a contradiction, that there is a face f for which the lemma is not true. 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 $e_1(u, w_1)$, $e_2(v, w_4)$, $e_3(v, w_3)$,

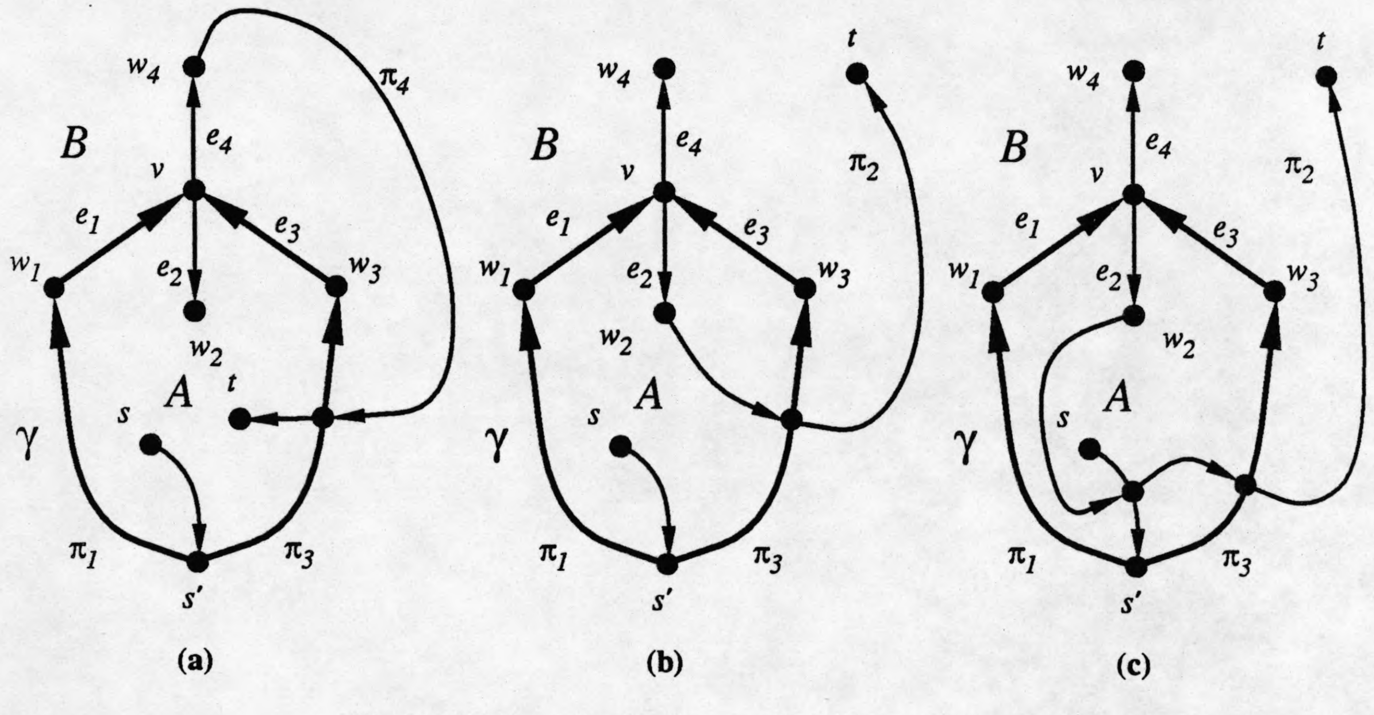


Figure 3 Examples for the proof of Lemma 1

and $e_4(u,w_2)$, 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. 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.

First, suppose that both s and t are 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. The 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).

Now, consider the case when s is in A and t is in B. 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 $e_4(u,y)$, 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, let π' be a path from s to s'. π_2 must intersect π' at some vertex q, and 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, also in this case, we have a cycle that does not separate s from t (see Fig. 4.e).

The cases when both s and t are in B, or s is in B and t is is in A are similar, and 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 further terminology: Let $e=(u,v)$ be an edge of G. First, we denote with $LEFT(e)$ and $RIGHT(e)$ the faces that appear on the left and right side of e when traversed from u to v. With reference to Lemma 1, we denote with $LEFT(v)$ and $RIGHT(v)$ the faces that separate the incoming and outgoing edges of a vertex v, 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 (see Fig. 2.a). With reference

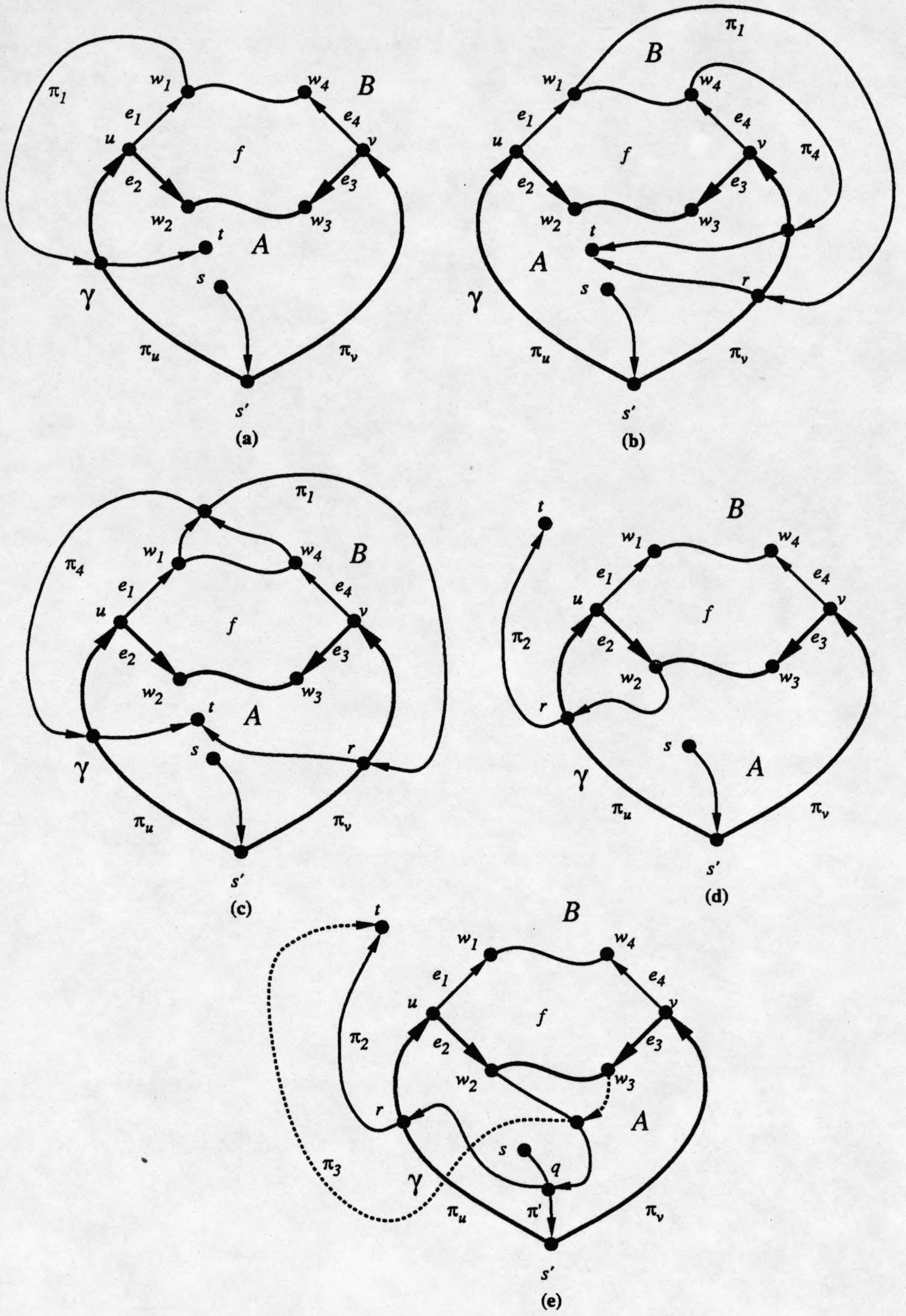


Figure 4 Examples for the proof of Lemma 2

to Lemma 2, wė call the two paths on the boundary of a face f the *left path* and *right path* of f, respectively. Also, we call *bottommost* and *topmost* vertex of f, denoted $BOT(f)$ and $TOP(f)$, the common origin and destination of these paths, respectively. (see Fig. 2.b). Notice that if vertex v in on the left (respectively, right) path of face f and is distinct from $BOT(f)$ and $TOP(f)$, then $RIGHT(v)=f$ (respectively, $LEFT(v)=f$).

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 [21]. We recall that a graph G is *st-2-connectible* if there are vertices s and t such that adding the edge (s,t) to G makes G 2-connected [21]. Clearly, a 2-connected graph is also st-2-connectible for every pair of vertices s and t.

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 [21] 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 G, denoted by C, does not contain neither s nor 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 [7], we have also 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 *ADD*, *INSERT*, *REMOVE*, and *JOIN*, 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 graph obtained by performing operation Π on G. Depending on Π, G' is a spherical *st*-graph if and only if:

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

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

(3) for $\Pi = REMOVE\,(e,f)$, $e=(u,v)$ must be an edge such that $\deg^+(v) \geq 2$ and $\deg^-(v) \geq 2$, where $\deg^+(v)$ and $\deg^-(v)$ are the *outdegree* and *indegree* of v, respectively;

(4) for $\Pi = JOIN\,(v,e)$, v must be a vertex distinct from s and t.

Proof: The proof of (2), (3), and (4) is straightforward. For operation *ADD*, 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_B and π_T be paths from s to $BOT(f)$ and from $TOP(f)$ to t, respectively. One of these two paths must intersect γ, which implies that G already had a nonseparating cycle, again a contradiction. □

Lemma 4 Let G be a spherical *st*-graph, and G_1 and G_2 be the graphs obtained by performing operations $ADD\,(e,u,v,f,f_1,f_2)$ and $ADD\,(e,v,u,f,f_1,f_2)$ 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.

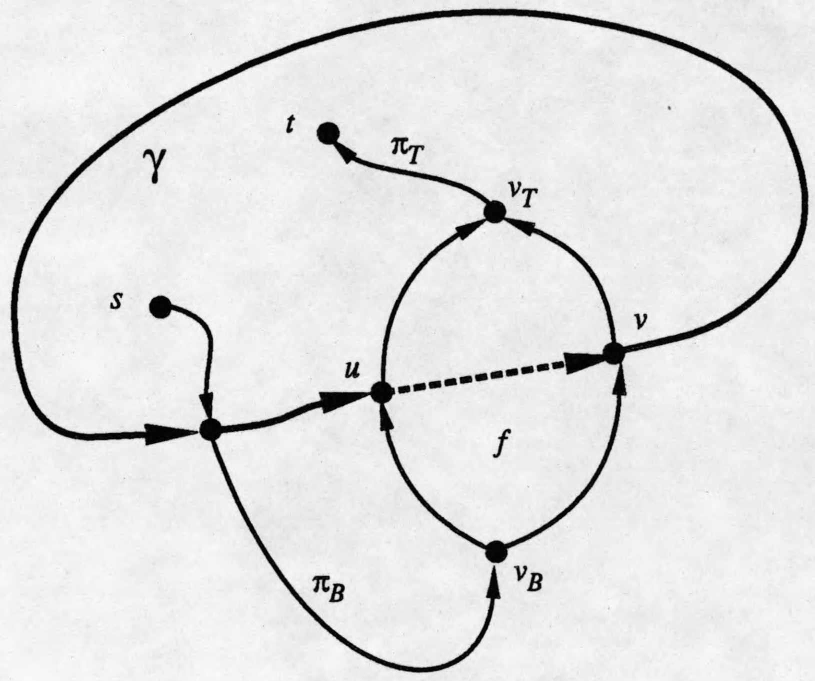


Figure 5 Example for the proof of Lemma 3

3. 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. We assume that vertices, edges, and faces are identified by *names*, which are elements of a sorted set. For example, names can be integers, alphanumeric strings, or pairs of coordinates. The total order among names will be referred to as *lexicographic order*. Regarding the complexity analysis, we assume that a name uses $O(1)$ space, and that the lexicographic 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. Throughout this paper, $\log x$ means $\max\{1, \log_2 x\}$.

3.1. st-Orientable Graphs

Let G be a spherical *st*-graph. If vertices u and v of G are on the same face f, one of the following must be true (see Fig. 6):

Case 1: f is either to the left or right of u and, also, either to the left or right of v, i.e.:

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

Case 2: u is either at the top or bottom of f, and f is either to the left or right of v, i.e.:

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

Case 3: v is either at the top or bottom of f, and f is either to the left or right of u, i.e.:

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

Case 4: one of u and v is at the top of f, and the other is at the bottom, i.e.:

$$(u = BOT(f) \textbf{ and } v = TOP(f)) \textbf{ or } (u = TOP(f) \textbf{ and } v = BOT(f)).$$

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 $TOP(f)$ and $BOT(f)$. 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 $BELOW(v)$ that contains the faces f such that $TOP(f) = v$. The faces in this table are sorted according to the lexicographic order of the name of vertex $BOT(f)$, so that the test for this case takes $O(\log m)$ time. This proves the following theorem:

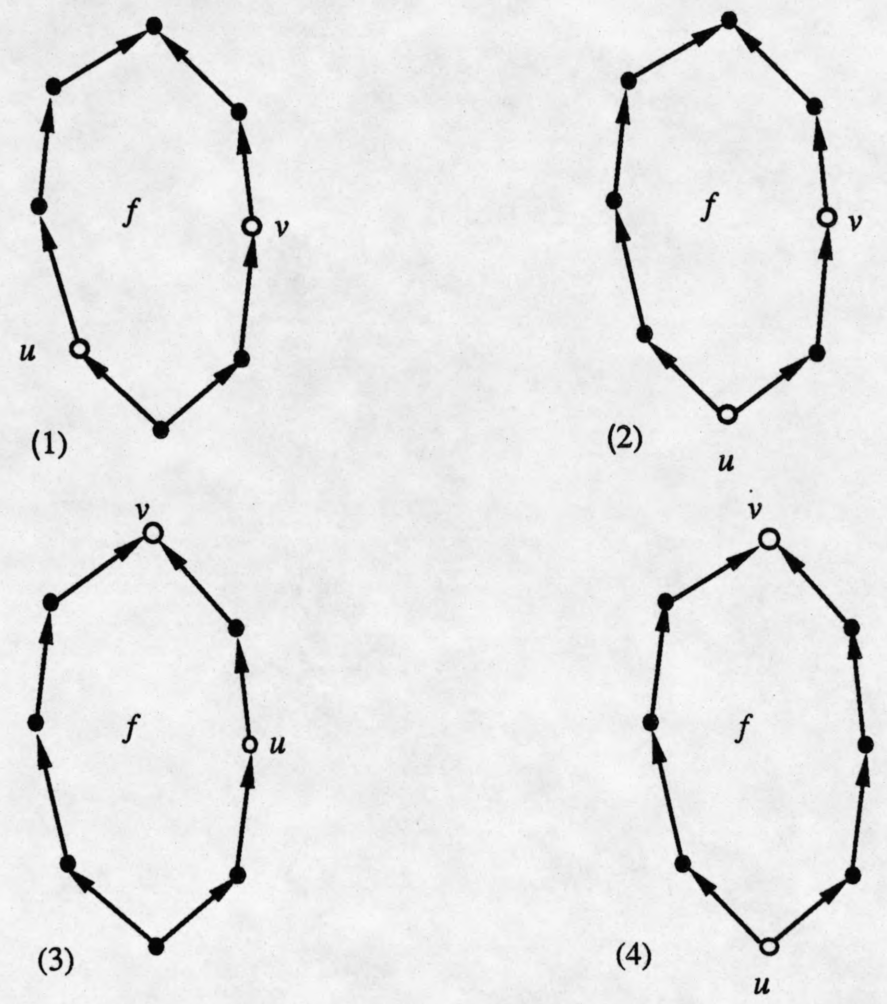


Figure 6 The four cases for two vertices on the same face

Theorem 2 There exists a data structure for the topological location problem in spherical *st*-graphs that uses $O(m)$ space, can be constructed in $O(m)$ preprocessing time, and supports operations *TEST* and *LIST* in time $O(\log m)$ and $O(\log m + k)$, respectively, where k is the number of retrieved faces.

Corollary 1 There exists a data structure that solves the topological location problem for *st*-orientable graphs with the following performance:

(1) the space requirement and preprocessing time are both $O(m)$;

(2) operations *TEST* and *LIST* take time $O(\log m)$ and $O(\log m + k)$, respectively, where k is the number of retrieved faces.

Proof: Construct a spherical *st*-orientation for G and apply Theorem 2. $\square$

3.2. General Plane Graphs

For plane graphs that are not *st*-orientable, the algorithm of the previous 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.

First, we recall some definitions on graph connectivity. A graph that is not connected will be called *0-connected.* A *cutvertex* of a graph G is a vertex whose removal disconnects G [1, p. 31]. A connected graph G is said to be *2-connected* if it has no cutvertices, and *1-connected* otherwise. The *block-cutvertex tree* of a 1-connected graph 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. For 0-connected graphs, the *block-cutvertex forest* T of G is defined as the set of the block-cutvertex trees of the connected components of G. 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 each tree of T so that it becomes a directed source tree with any block at the root, and marking each vertex v with a label $BLOCK(v)$, where:

(1) if v is not a cutvertex, $BLOCK(v)$ is the (unique) block that contains v;

(2) if v is a cutvertex, $BLOCK(v)$ is the father of v in T.

See an example in Fig. 7.

Lemma 5 Using the above data structure, testing if u and v belong to the same block can be done in $O(1)$ time and $O(m)$ space.

Proof: The algorithm consists of testing the following condition:

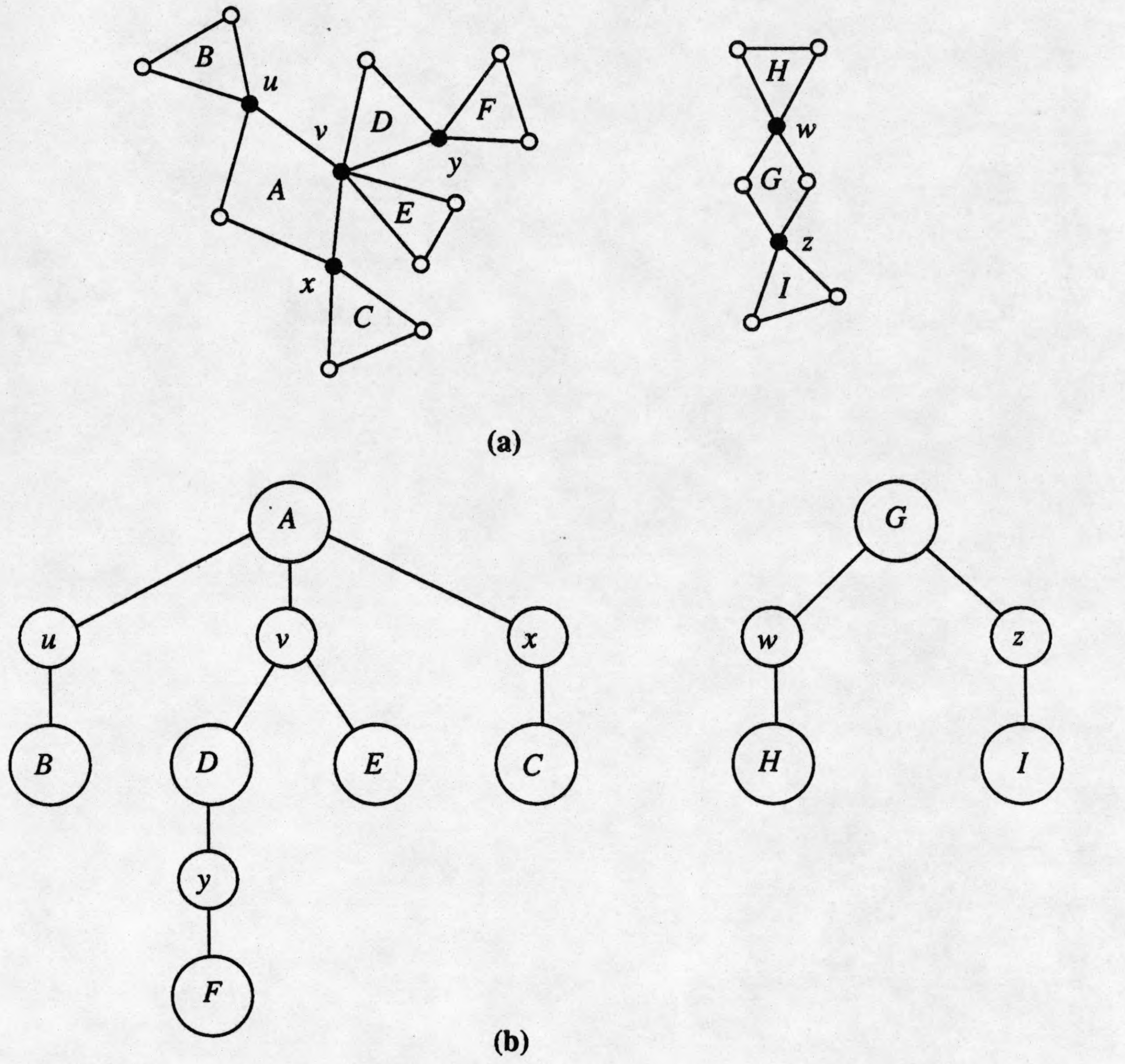


Figure 7 Example of orientation of the block-cutvertex forest of a graph.

$\mathit{BLOCK}(u)=\mathit{BLOCK}(v)$ **or** $u=\mathit{FATHER}(\mathit{BLOCK}(v))$ **or** $\mathit{BLOCK}(v)=\mathit{FATHER}(u)$ **or** $v=\mathit{FATHER}(\mathit{BLOCK}(u))$ **or** $\mathit{BLOCK}(u)=\mathit{FATHER}(v)$. □

Now, let G^* be the dual graph of a plane graph G. There is a bijection between the blocks of G and the ones of G^*, expressed by the following lemma:

Lemma 6 A subset of edges of G forms a block of G if and only if their duals form a block in G^*.

We construct the block-cutvertex forest of G^*, and denote it with T^*. Notice that, since G^* is always connected, T^* is actually a tree. In the following, the faces that are cutvertices of G^* will be called *cutfaces*, and T^* will be called the *block-cutface tree* of G. We orient each edge $e^* = (f,B)$ of T^* from f to B if the boundary of f is *external* to B, i.e. all the edges of the external boundary of B are on f, and from B to f otherwise (see Fig. 8).

Lemma 7 The above orientation transforms T^* into a directed source tree, whose root is either the external face or the outermost block of G.

For uniformity, if the root of T^* is a block, we augment T^* with a new root representing the external face. This corresponds to considering the external face as being always a cutface. From now on, T and T^* are assumed to be oriented and augmented as explained above.

Lemma 8 Let v be a cutvertex of G. The set of blocks and cutfaces that contain v is a subgraph of T^* with a unique source node, which is a cutface.

According to the previous lemma, we call $HIGH(v)$ the source of the subgraph of T^* associated with cutvertex v, i.e. the outermost cutface that contains v. For example, in Fig. 8 the cutvertex w_4 is contained in cutfaces f, i, and m, and in blocks E, O, P, Q, and R, and we have $HIGH(v)=f$. We extend this definition also to the vertices that are not cutvertices, but are on the external boundary of their block, by setting $HIGH(v)$ as the external face of $BLOCK(v)$, i.e. $HIGH(v)=FATHER(BLOCK(v))$. Finally, for the remaining vertices $HIGH(v)$ is set to *nil*.

Let f be a cutface, and B_0 be the block which is the father of f in T^*. With reference to Fig. 9, the boundary of f consists of the boundary β_0 of an internal face of B_0, plus the external boundaries of the blocks which are the sons of f in T^*. We call β_0 the *main cycle* of f. For the special case when f is the external face, the main cycle of f is the empty cycle. Using the definition of T^* and simple topological considerations, one can show that if a vertex w_3 is on the

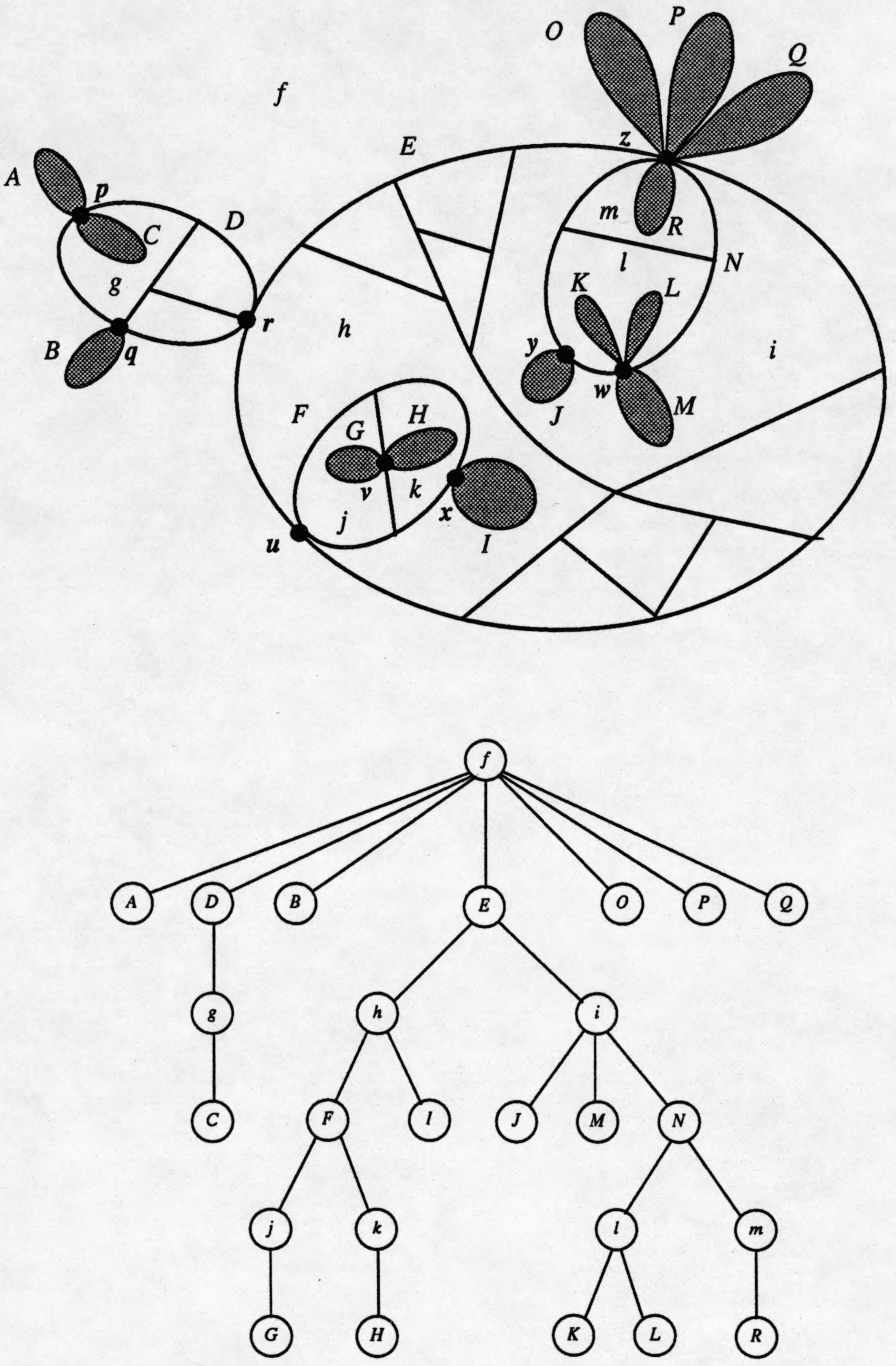


Figure 8 Example of block-cutface tree

boundary of f but not on the main cycle of f, then $HIGH(w)=f$.

Theorem 3 Let u and v be vertices of G that are on the boundary of the same face f:

(1) If u and v belong to the same block B, then they are also on the boundary of face f in B.

(2) Otherwise, f is the outermost face for at least one of u and v; i.e. $f=HIGH(u)$ or $f=HIGH(v)$. Also, if f is the outermost face for only one of u and v, then the other vertex is on the main cycle of f.

The data structure for the topological location problem in general plane graphs is as follows:

(1) A data structure to test whether two vertices belong to the same block, see Lemma 5.

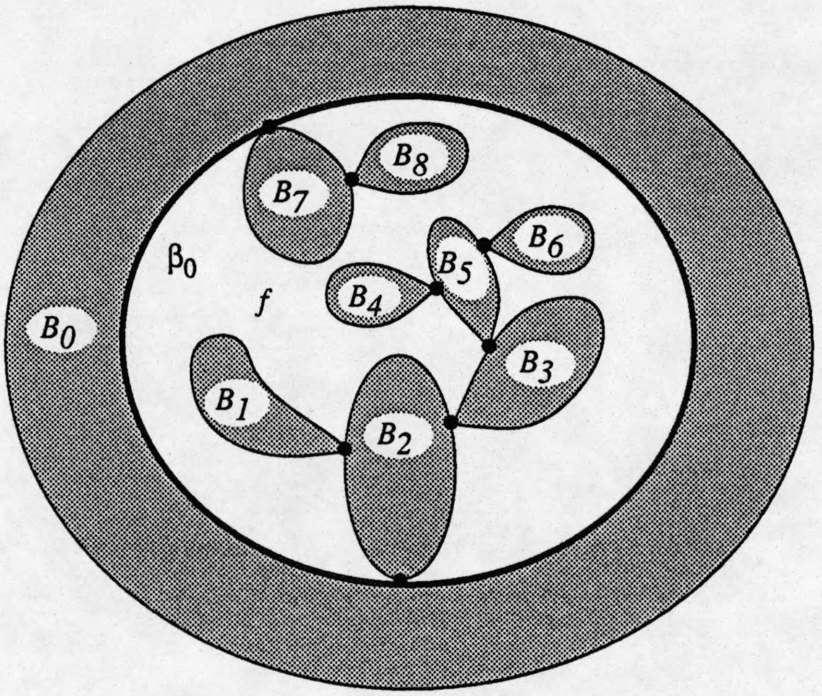


Figure 9 Structure of the boundary of a cutface

(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) A pointer $MAIN(f)$ to the representative of the cutface f in the block B_0, father of f in T^*, for each cutface f. Notice that the boundary of $MAIN(f)$ is the main cycle of f.

(4) A pointer $HIGH(v)$, for each vertex v.

A detailed description of the algorithm for the *TEST* operation is given in Fig. 10. The algorithm for the *LIST* operation is similar and omitted for brevity.

Theorem 4 There exists a data structure that solves the topological location problem for plane graphs with the following performance:

(1) the space requirement and preprocessing time are both $O(m)$;

(2) operations *TEST* and *LIST* can be performed in time $O(\log m)$ and $O(\log m + k)$, respectively.

Proof: The correctness of the data structure and of the algorithms follows from Theorem 3 and the results of the previous section. The time complexity of the *TEST* and *LIST* operations follows from Corollary 1 and Lemma 5. With reference to the space requirement and preprocessing time we note: Constructing the dual and the block-cutvertex forest of a plane graph can be done in $O(m)$ time using standard algorithms [8]. From Lemma 5, the data structure for testing whether two vertices belong to the same block uses $O(m)$ space. If G has b blocks and the i-th block has m_i edges, $i = 1, \cdots, b$, we have that $\sum_{i=1}^{b} m_i = m$, so that the total space requirement and preprocessing time for the data structures of the blocks of G is $O(m)$. Orienting T^* can be done in $O(m)$ time by performing a depth-first-search in G^* starting at the external face. Finally, setting up the *MAIN* and *HIGH* pointers can be also done in $O(m)$ time by maintaining bidirectional pointers between the blocks of T and the corresponding ones of T^*. □

```
procedure TEST(u,v);
{ Test if there is a face f that has both vertices u and v on its boundary. }
begin
  if u and v are in the same block B
    then perform the test on the data structure for B
    else begin
      let $f := HIGH(u)$; $g := HIGH(v)$; $f_0 := MAIN(f)$; $g_0 := MAIN(g)$;
      if $f = g$
        then return (f)
        else if $u = TOP(g_0)$ or $u = BOT(g_0)$ or $LEFT(u) = g_0$ or $RIGHT(u) = g_0$
        { test whether u is on $g_0$ }
          then return ($g_0$);
          else if $v = TOP(f_0)$ or $v = BOT(f_0)$ or $LEFT(v) = f_0$ or $RIGHT(v) = f_0$
          { test whether v is on $f_0$ }
            then return ($f_0$);
            else return ( "u and v are not on the same face" )
          end
      end
```

Figure 10 Algorithm for the *TEST* operation in general planar graphs (static data structure)

3.3. 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:

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

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, we have:

$$\overline{Q}(n,m) = \frac{1}{\binom{n}{2}} O\left(\sum_{\substack{u,v \\ u \neq v}} (\log \deg^-(u) + \log \deg^-(v)) \right) = \frac{1}{\binom{n}{2}} O\left(n \sum_{v} \log \deg^-(v) \right)$$

Since $\sum_{v} \deg^-(v) = m$, we obtain the following theorem:

Theorem 5 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 $\overline{Q}(n,m) = O\left(\log \frac{m}{n} \right)$.

Notice that for simple plane graphs the average query time is $\overline{Q}(n,m) = O(1)$.

4. 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 the results to undirected graphs using spherical *st*-orientations.

4.1. Directed 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 lexicographic order of the names of the vertices. Similarly, the records of the edges and faces are arranged in lexicographic 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) $TOP(f)$: pointer to the record of the topmost vertex of f;

(3) $BOT(f)$: 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. Specifically, let

$$BOT(f)=v_0,e_1,v_1,e_2,\cdots,v_{k-1},e_k,v_k=TOP(f)$$

be, say, the left path of f. The nodes of $LPATH(f)$ represent the edges and vertices $e_1,v_1,e_2,\cdots,v_{k-1},e_k$, in this order. Notice that for all $i=1,\cdots,k-1$, $f=RIGHT(v_i)$, and for all $j=1,\cdots,k$ $f=RIGHT(e_j)$. The tree $RPATH(f)$ is similarly defined. The roots of $LPATH(f)$ and $RPATH(f)$ point back to face 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) $IN(v)$: pointer to a tree whose nodes represent the incoming edges of v, sorted according to the lexicographic order of the name of the other endpoint vertex.

(4) $OUT(v)$: pointer to a tree whose nodes represent the outgoing edges of v, sorted according to the lexicographic order of the name of the other endpoint vertex.

(5) $PLEFT(v)$: pointer to the representative of v in the tree $RPATH(f)$, where $f = LEFT(v)$;

(6) $PRIGHT(v)$: pointer to the representative of v in the tree $LPATH(g)$, where $g = RIGHT(v)$;

(7) $BELOW(v)$: pointer to a balanced search tree whose nodes represent the faces f whose topmost vertex is v. The nodes in $BELOW(v)$ are sorted according to the lexicographic order of the name of vertex $BOT(f)$.

The record for an edge e stores the following information:

(1) e: name of the edge;

(2) the head and tail vertices of e;

(3) pointers to the representatives of e into the IN and OUT trees of the endpoint vertices of e; and

(4) 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. 11 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 *ADD* operation, we assume that f_1 is to the left of e and f_2 is to the right of e. The *LPATH* and *RPATH* trees of the new faces will be obtained by *splitting* $LPATH(f)$ and $RPATH(f)$ at vertices u and v, and *splicing* appropriately the resulting trees. Formally, we will perform on these trees the following operations:

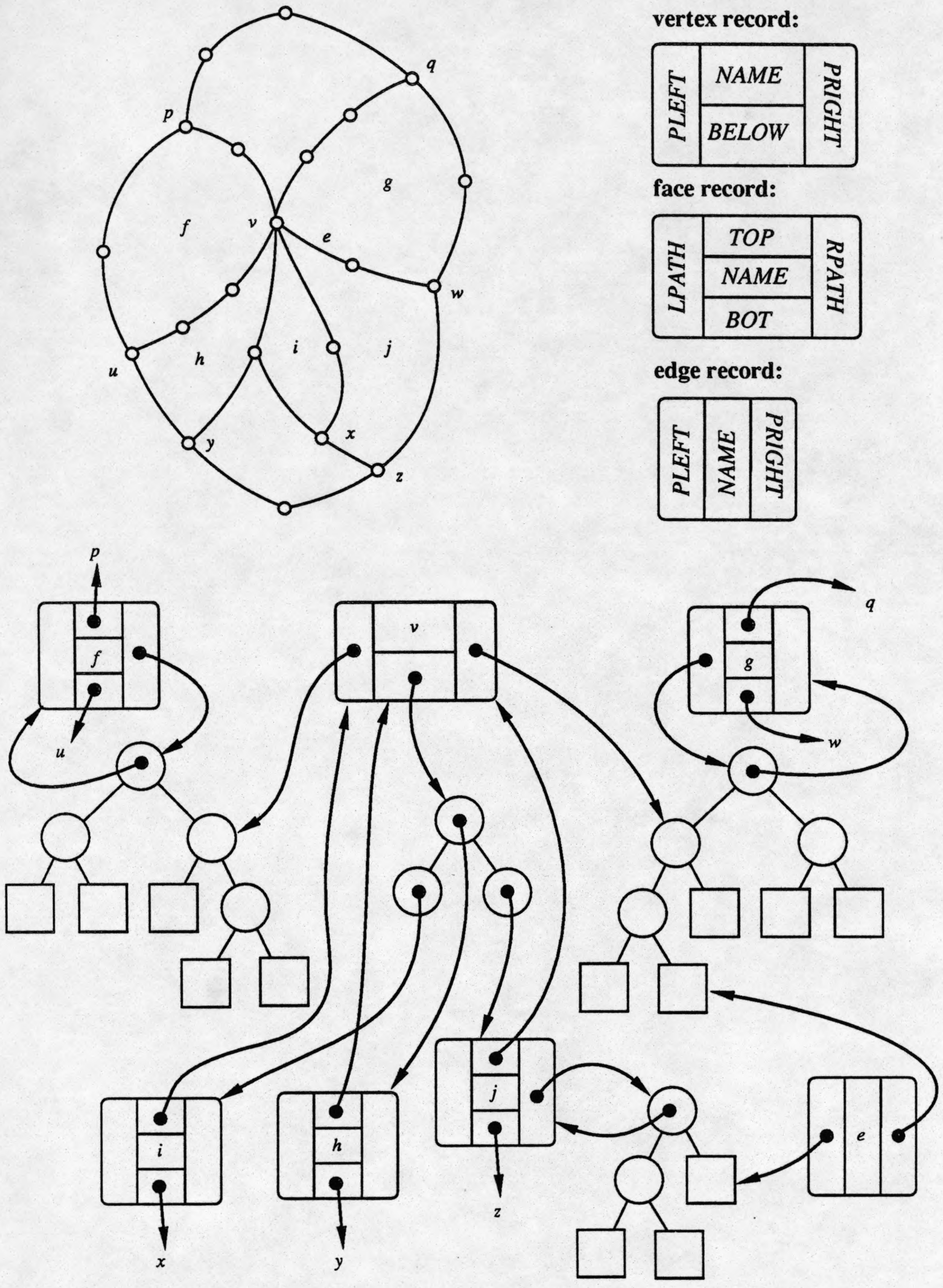


Figure 11 A spherical *st*-graph and a fragment of the dynamic data structure for it. The edges are oriented upward.

(1) *SPLIT*, denoted by $(T_1,v,T_2) := SPLIT(T)$, returns node v and two balanced trees, T_1 and T_2, that contains the nodes of T preceding and following v, respectively.

(2) *SPLICE*, denoted by $T := SPLICE(T_1,T_2)$, takes as input two trees T_1 and T_2 such that the maximum node of T_1 precedes the minimum node of T_2, and returns a new tree T that contains the union of the nodes of T_1 and T_2.

Standard techniques allow us to perform the above *INSERT* and *SPLICE* operations in $O(\log |T|)$ time, see for example the implementation with AVL trees described in [4, 19], and the one with (2,4)-trees given in [22].

Detailed pseudo-code descriptions of the algorithms designed to perform operations *TEST*, *ADD* and *INSERT* are given in Figs. 12, 13, and 14, respectively.

The above data structure supports also operations *REMOVE* and *JOIN*. In fact, the *REMOVE* operation can be performed by reversing the transformations on the data structure realized by the algorithm for the *ADD* operation. Similarly, the *JOIN* operation is the inverse of the *INSERT* operation. Notice also that, from Lemma 3, the counters $\deg^+(v)$ and $\deg^-(v)$ allow us to test the feasibility of each such operation in $O(1)$ time. This completes the proof of:

Theorem 6 The above data structure correctly solves the dynamic embedding problem for spherical *st*-graphs and has the following performance:

(1) the space requirement and the preprocessing time are $O(m)$;

(2) operations *TEST*, *ADD*, *INSERT*, *REMOVE*, and *JOIN* are each executed in $O(\log m)$ time;

(3) operation *LIST* is executed in $O(\log m + k)$ time, where k is the number of retrieved faces.

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 *ADD* and insert operations is optimal. For the definition of amortized time complexity, see [29].

Theorem 7 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 6.

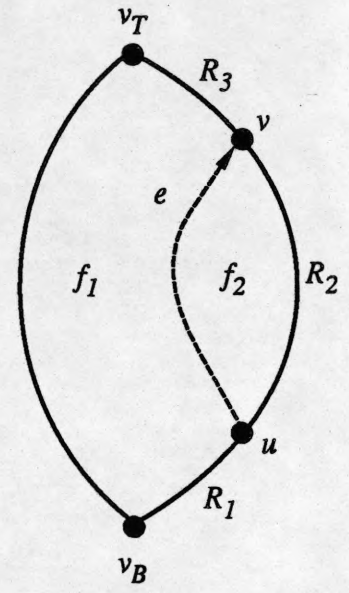
(2) In a sequence of *ADD* and *INSERT* operations the amortized restructuring time complexity of each such operation is $O(1)$.

Proof: Use 2-4 trees for the trees T_F, T_V, T_E, $IN(v)$, $OUT(v)$, and $BELOW(v)$. Such trees have $O(1)$ amortized rebalancing time for insertions and deletions [15]. With regars to the *LPATH* and *RPATH* trees, their manipulation in a sequence of *ADD* and *INSERT* operations involves insertions and the kind of generalized splittings considered in [13]. 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 [13] we can show that using circular level-linked 2-4 trees for the *LPATH* and *RPATH* trees, the amortized rebalancing time for them is also $O(1)$. □

```
procedure TEST(u,v);
{ Test if there is a face f that has both vertices u and v on its boundary. }
begin
  find in $T_V$ the nodes associated with u and v;

  walk up to the roots of the trees that contain PLEFT(u), PRIGHT(u), PLEFT(v),
    and PRIGHT(v) to find faces $f_1$ =LEFT(u), $f_2$ =RIGHT(u),
    $g_1$ =LEFT(v), and $g_2$ =RIGHT(v), respectively.

  if ($f_1 = g_1$) or ($f_1 = g_2$)
    then return ($f_1$)
    else if ($f_2 = g_1$) or ($f_2 = g_2$)
      then return ($f_2$)
      else if TOP($f_1$)=v or BOT($f_1$)=v
        then return ($f_1$)
        else if TOP($f_2$)=v or BOT($f_2$)=v
          then return ($f_2$)
          else if TOP($g_1$)=u or BOT($g_1$)=u
            then return ($g_1$)
            else if TOP($g_2$)= or BOT($g_2$)=u
              then return ($g_2$)
              else begin
                search in tree BELOW(v) for a face f with BOT(f)=u;

                if such a face f exists
                  then return (f)
                  else begin
                    search in tree BELOW(u) for a face f with BOT(f)=v;

                    if such a face f exists
                      then return (f)
                      else return ( ``u and v are not on the same face'' )
                  end
              end
end
```

Figure 12 Algorithm for the *TEST* operation in spherical *st*-graphs (dynamic data structure)

```
procedure $ADD(e,u,v,f,f_1,f_2)$;
{ Add an edge $e=(u,v)$ to $G$ inside face $f$, which is split into faces $f_1$ and $f_2$.
  The operation is rejected if vertices $u$ and $v$ are not both be on the boundary of face $f$. }
begin
  create a new node for edge $e$ and insert it into $T_E$;
  insert edge $e$ into $IN(v)$ and $OUT(u)$;
  delete from $T_F$ the node for face $f$;
  create new nodes for faces $f_1$ and $f_2$ and insert them in $T_F$;
  let $v_B := BOT(f)$ and $v_T := TOP(f)$;
  if $u$ and $v$ are both on $RPATH(f)$ and $u$ precedes $v$
    then begin
      $BOT(f_1) := v_B$;   $TOP(f_1) := v_T$;
      $BOT(f_2) := u$;   $TOP(f_2) := v$;
      $(R_1,u,R_2,v,R_3) := SPLIT(RPATH(f))$;
      $LPATH(f_1) := LPATH(f)$;
      $RPATH(f_1) := SPLICE(R_1,u,e,v,R_3)$;
      $LPATH(f_2) := \{e\}$;
      $RPATH(f_2) := R_2$;
      rename node $f$ into $f_1$ in $BELOW(v_T)$;
      insert $f_2$ into $BELOW(v)$;
    end
  else if $u$ and $v$ are both on $LPATH(f)$ and $u$ precedes $v$
    then { similar to the previous case }

  else if $u$ is on $LPATH(f)$ and $v$ is on $RPATH(f)$
    then begin
      $BOT(f_1) := u$;   $TOP(f_1) := v_T$;
      $BOT(f_2) := v_B$;   $TOP(f_2) := v$;
      $(L_1,u,L_2) := SPLIT(LPATH(f))$;
      $(R_1,v,R_2) := SPLIT(RPATH(f))$;
      $LPATH(f_1) := L_2$;
      $RPATH(f_1) := SPLICE(e,v,R_2)$;
      $LPATH(f_2) := SPLICE(L_1,u,e)$;
      $RPATH(f_2) := R_1$;
      delete $f$ from $BELOW(v_T)$;
      insert $f_1$ into $BELOW(v_T)$;
      insert $f_2$ into $BELOW(v)$
    end

  else if $u$ is on $RPATH(f)$ and $v$ is on $LPATH(f)$
    then { similar to the previous case }
```



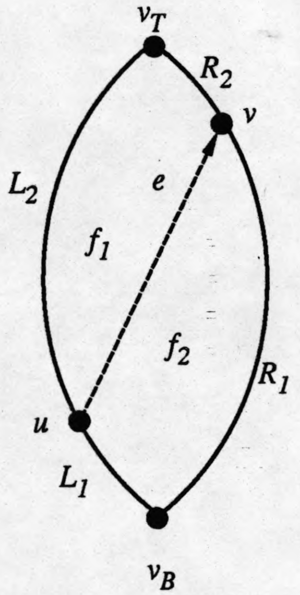
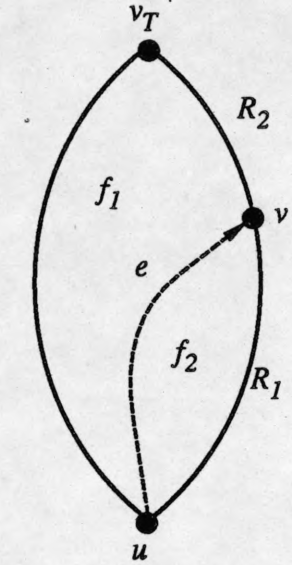
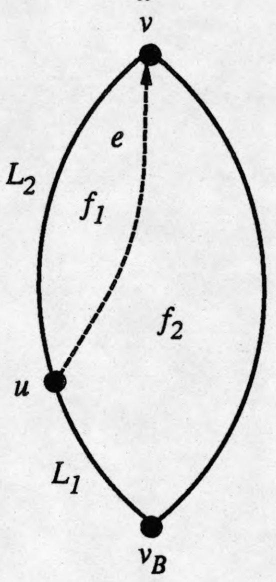


Figure 13 Algorithm for the *ADD* operation in spherical *st*-graphs (continues on the next page)

```
    else if $u = BOT(f)$ and $v$ is on $RPATH(f)$
      then begin
        $BOT(f_1) := u$;   $TOP(f_1) := v_T$;
        $BOT(f_2) := u$;   $TOP(f_2) := v$;
        $(R_1, v, R_2) := SPLIT(RPATH(f))$;
        $LPATH(f_1) := LPATH(f)$;
        $RPATH(f_1) := SPLICE(e, v, R_2)$;
        $LPATH(f_2) := \{e\}$;
        $RPATH(f_2) := R_1$;
        rename node $f$ into $f_1$ in $BELOW(v_T)$;
        insert $f_2$ into $BELOW(v)$
      end
    else if $u = BOT(f)$ and $v$ is on $LPATH(f)$
      then { similar the previous case }

    else if $v = TOP(f)$ and $u$ is on $LPATH(f)$
      then begin
        $BOT(f_1) := u$;   $TOP(f_1) := v$;
        $BOT(f_2) := v_B$;   $TOP(f_2) := v$;
        $(L_1, u, L_2) := SPLIT(LPATH(f))$;
        $LPATH(f_1) := L_2$;
        $RPATH(f_1) := \{e\}$;
        $LPATH(f_2) := SPLICE(L_1, u, e)$;
        $RPATH(f_2) := RPATH(f)$;
        insert $f_1$ into $BELOW(v)$
        rename node $f$ into $f_2$ in $BELOW(v)$;
      end
    else if $v = TOP(f)$ and $u$ is on $RPATH(f)$
      then { similar the previous case }

    else if $u = BOT(f)$ and $v = TOP(f)$
      then begin
        $BOT(f_1) := u$;   $TOP(f_1) := v$;
        $BOT(f_2) := u$;   $TOP(f_2) := v$;
        $LPATH(f_1) := LPATH(f)$;
        $RPATH(f_1) := \{e\}$;
        $LPATH(f_2) := \{e\}$;
        $RPATH(f_2) := RPATH(f)$;
        rename node $f$ into $f_1$ in $BELOW(v)$;
        insert $f_2$ into $BELOW(v)$
      end
    else return ( "illegal operation" )
end
```





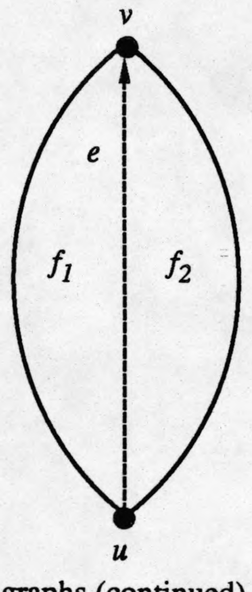


Figure 13 Algorithm for the *ADD* operation in spherical *st*-graphs (continued)

```
procedure INSERT($e, v, e_1, e_2$)
{ Split the edge $(u,w)$ into two edges $e_1 = (u,v)$ and
  $e_2 = (v,w)$, by adding vertex $v$. }
begin
   create a new node for vertex $v$ and insert it into $T_V$
      immediately after $u$;
   delete the node of $e$ from $T_E$;
   delete $e$ from $IN(w)$ and $OUT(u)$;
   create nodes for $e_1$ and $e_2$ and insert them in $T_E$;
   insert $e_1$ into $IN(v)$ and $OUT(u)$;
   insert $e_2$ into $IN(w)$ and $OUT(v)$;

   walk up to the roots of the trees that contain $PLEFT(e)$ and
      $PRIGHT(e)$ to find faces $f = LEFT(e)$ and $g = RIGHT(e)$;

   delete the representative of $e$ in $RPATH(f)$;
   insert representatives for $e_1$, $e_2$, and $v$ into $RPATH(f)$,
      and set the PLEFT pointers accordingly;

   delete the representative of $e$ in $LPATH(g)$;
   insert representatives for $e_1$, $e_2$, and $v$ into $LPATH(g)$,
      and set the PRIGHT pointers accordingly;

   $BELOW(v) := \emptyset$;
end
```

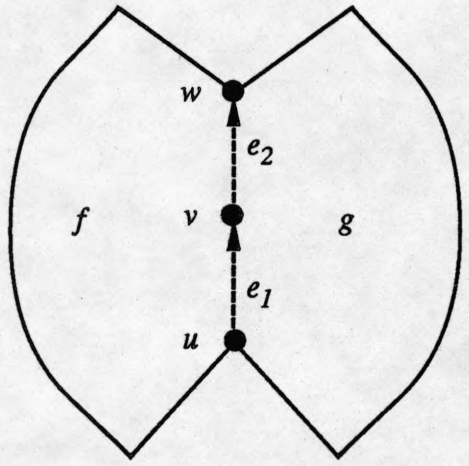


Figure 14 Algorithm for the *INSERT* operation in spherical *st*-graphs

4.2. Undirected Graphs

For undirected plane graphs, we maintain on-line a spherical *st*-orientation and use the data structure previously described. Operations *TEST*, *LIST*, and *INSERT* do not require any modifications. In connection with operation *ADD* (e,u,v,f,f_1,f_2) we have to decide the orientation of edge e so that it does not introduce cycles internal to face f. By Lemma 4, this can be done easily by reversing the direction of the orientation whenever the algorithm of Fig. 13 rejects the operation. This proves:

Theorem 8 There is a data structure that supports operations *TEST*, *LIST*, *ADD*, and *INSERT* in a *st*-orientable plane graph with the following performance:

(1) the space requirement and the preprocessing time are both $O(m)$;

(2) operations *TEST*, *ADD*, and *INSERT* are each executed in $O(\log m)$ time;

(3) operation *LIST* is executed in $O(\log m + k)$ time, where k is the number of retrieved faces.

As regards the *REMOVE* and *JOIN* 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 *JOIN* (*REMOVE*) 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 the vertices and edges that are free. We have thus:

Theorem 9 The data structure of Theorem 8 supports also operations *REMOVE* and *JOIN* 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 hierarchic design strategies for VLSI circuits suggest to build the layout in 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 allow the implementation of a hierarchic deletion scheme that allows to "undo" any *ADD* and

INSERT operation performed in the past (not only the last operation). For instance, we define the *hierarchic embedding problem* as a variation of the previously discussed dynamic embedding problem where the following restrictions are placed on the *REMOVE* and *JOIN* operations:

(1) an edge can be deleted by a *REMOVE* operation only if it was created (at any time in the past) by means of an *ADD* operation;

(2) a vertex can be deleted by a *JOIN* operation only if it was created (at any time in the past) by a *INSERT* operation.

In Fig. 15 we show a sequence of update operations in an instance of the hierarchic embedding problem.

The aforementioned restrictions on the *REMOVE* and *JOIN* operations can be enforced by storing with each edge e two flags, denoted $FREE\text{-}TAIL(e)$ and $FREE\text{-}HEAD(e)$, 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 $FREE\text{-}TAIL(e)$ and $FREE\text{-}HEAD(e)$ are true. This can be done by manipulating the flags in the various operations as follows:

$ADD\,(e,u,v,f,f_1,f_2)$: $FREE\text{-}TAIL(e) := true;\ FREE\text{-}HEAD(e_2) := true.$

$INSERT\,(e,v,e_1,e_2)$: $FREE\text{-}TAIL(e_1) := FREE\text{-}TAIL(e);\ FREE\text{-}HEAD(e_1) := false;$
$FREE\text{-}TAIL(e_2) := false;\ FREE\text{-}HEAD(e_2) := FREE\text{-}HEAD(e).$

$REMOVE\,(e,f)$: Accept the operation only if:

$$(FREE\text{-}TAIL(e) = true) \textbf{ and } (FREE\text{-}HEAD(e_2) = true).$$

$JOIN\,(v,e)$: Let $e_1 = (u,v)$ and $e_2 = (v,w)$ be the edges formerly incident to v.
$FREE\text{-}TAIL(e) := FREE\text{-}TAIL(e_1);$
$FREE\text{-}HEAD(e) := FREE\text{-}HEAD(e_2).$

In the example of Fig. 15 a flag is set whenever the corresponding endpoint is left unconnected.

It is not difficult to show that the edges (vertices) that can be deleted in an instance of the hierarchic embedding problem are free. We have thus the following theorem:

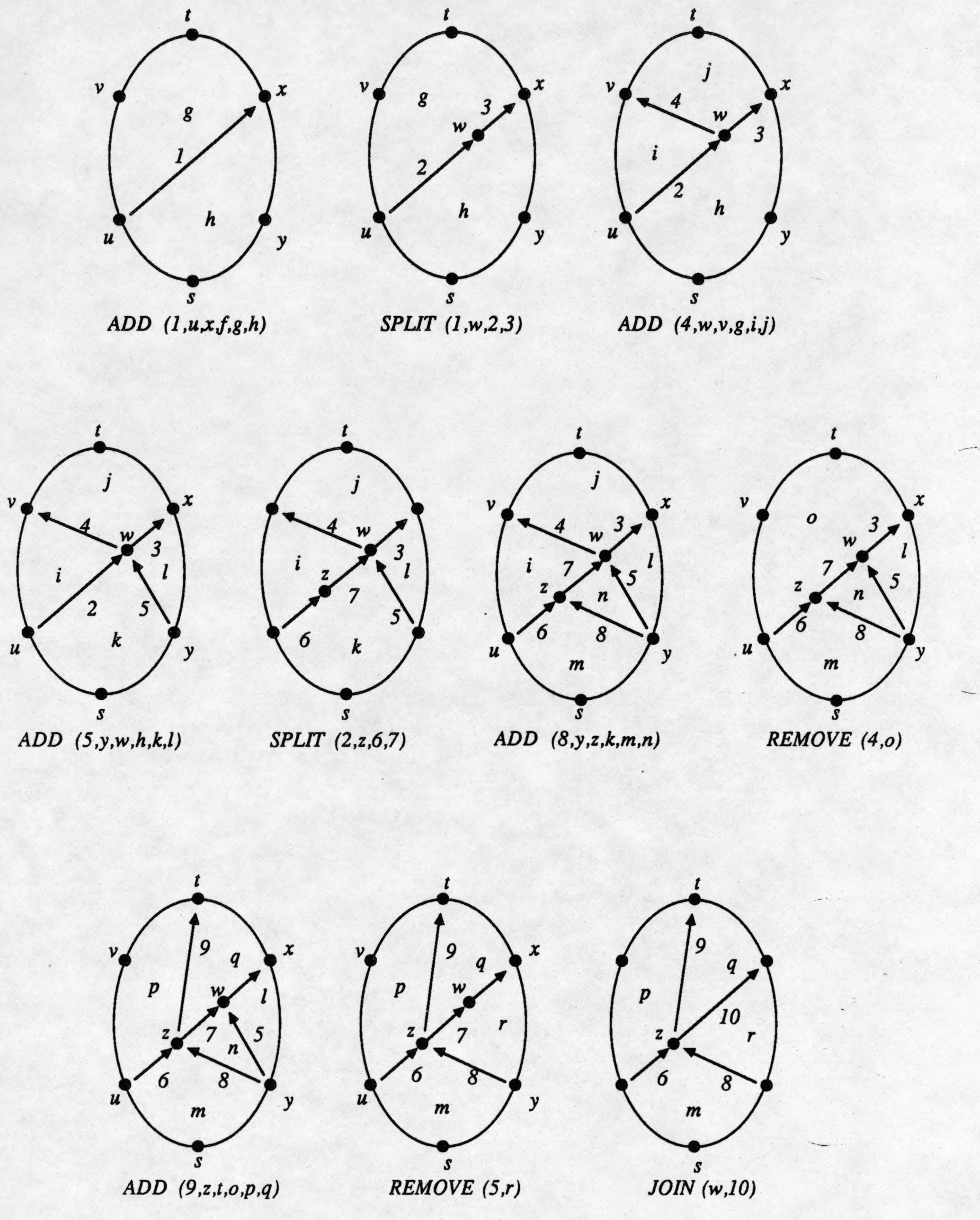


Figure 15 Sequence of hierarchic update operations on a plane graph

Theorem 10 There exists a data structure that allows us to solve the hierarchic embedding problem for *st*-orientable plane graphs with the following performance:

(1) the space requirement is $O(m)$;

(2) operations *TEST*, *ADD*, *INSERT*, *REMOVE*, and *JOIN* are each executed in $O(\log m)$ time;

(3) operation *LIST* is executed in $O(\log m + k)$ time, where k is the number of retrieved faces.

5. A DUAL EMBEDDING PROBLEM

In this section we extend the above results by providing a solution to an embedding problem that is the dual of the previously discussed dynamic embedding problem. We also develop further concepts on orientations of plane graphs, extending the definition of *cylindric orientation* given in [28].

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.
EXPAND (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.
DUPLICATE (e,f,e_1,e_2):	Replace the edge e with two parallel edges, e_1 and e_2, with the same endpoints, and call f the resulting face between them.
CONTRACT (e,v):	Contract the edge $e=(u,w)$, and call v the vertex resulting from the contraction of u and w_3.
MERGE (f,e):	Let f be a face whose boundary consists of two parallel edges e_1 and e_2. Remove f by merging e_1 and e_2 into a new edge e.

We will show that this problem is the *dual* of the dynamic embedding problem in the sense of duality of plane graphs. To this extent, we will extend the notion of duality to spherical *st*-graphs.

The *dual graph* G^* of a directed plane graph G is the directed graph obtained by orienting each edge of the dual of the undirected version of G from the left to the right. This means that, if e is an edge of G with faces f and g on its left and right, respectively, then the dual edge e^* is directed from f to g. Let s^* and t^* be two distinct faces of G. An *s^*t^*-slit* is a simple directed path of G^* from s^* to t^*. A *directed cut* is a (simple) directed cycle of G^*.

A *cylindrical s^*t^*-graph* is a digraph G embedded in the plane such that:

(1) the boundary of only two faces, s^* and t^*, are directed cycles, where s^* is clockwise and t^* is counterclockwise;

(2) every face f of G is on some s^*t^*-slit; and

(3) every directed cut separates s^* from t^*.

We can visualize a cylindrical s^*t^*-graph as embedded in a cylinder, with external faces s^* and t^* (see Fig. 16).

Theorem 11 G is a spherical st-graph if and only if G^* is a cylindrical s^*t^*-graph.

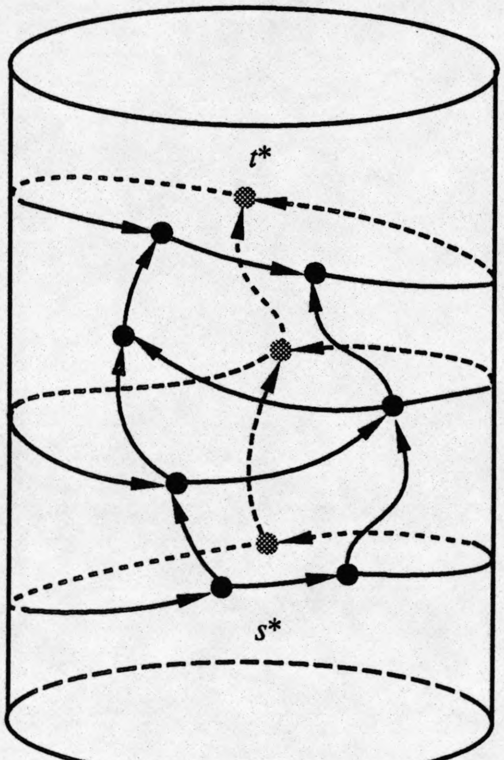


Figure 16 Example of cylindrical s^*t^*-graph

We observe that Lemma 1 of Section 2 expresses a *separation* property of the incoming and outgoing edges of each vertex of a spherical *st*-graph. Similarly, Lemma 2 expresses a separation property for the clockwise and counterclockwise edges on the boundary of each face. The latter property is the *dual* of the former in the sense that if a digraph G has the separation property for vertices, then its dual has the separation property for the faces, and vice-versa. Therefore, by Theorem 11, the separation properties expressed by Lemmas 1-2 hold also for cylindrical s^*t^*-graphs.

A *spherical s^*t^*-orientation* for an undirected plane graph G is a cylindrical s^*t^*-graph whose undirected version is isomorphic to G. G is said to be *s^*t^*-orientable* if it admits a spherical s^*t^*-orientation. From Theorems 1 and 11, we obtain a characterization for s^*t^*-orientable graphs, and a linear-time algorithm for testing this property. Notice that every 2-connected graph is s^*t^*-orientable.

As in the case of the dynamic embedding problem, we can define the same operations on a cylindrical s^*t^*-graph, where the restrictions on the operations are as follows:

Theorem 12 Let G be a cylindrical s^*t^*-graph, and G' be the graph obtained by performing operation Π on G. Depending on Π, G' is a cylindrical s^*t^*-graph if and only if:

(1) for $\Pi = EXPAND\ (e,f,g,v,v_1,v_2)$, edge e must be oriented in such a way that neither v_1 nor v_2 becomes a source or a sink;

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

(3) for $\Pi = CONTRACT\ (e,v)$, $e=(u,v)$ must be an edge such that the right path of face $LEFT(e)$ and the left path of face $RIGHT(e)$ have length at least 2.

(4) for $\Pi = MERGE\ (f,e)$, f must be a face distinct from s^* and t^*.

Using Theorem 11 and the results of Section 4, we obtain:

Theorem 13 There is a data structure that allows us to solve the dual dynamic embedding problem for cylindrical s^*t^*-graphs with the following performance:

(1) the space requirement is $O(m)$;

(2) operations *TEST**, *EXPAND*, *DUPLICATE*, *CONTRACT*, and *MERGE* are each executed in $O(\log m)$ time;

(3) operation *LIST** is executed in $O(\log m + k)$ time, where k is the number of retrieved vertices.

The *hierarchic dual embedding problem* is defined as a variation of the dynamic embedding problem where the *CONTRACT* and *MERGE* operations can performed only on edges (respectively, faces) previously inserted by the *EXPAND* (respectively, *DUPLICATE*) operation. We have:

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

(1) the space requirement is $O(m)$;

(2) operations *TEST**, *EXPAND*, *DUPLICATE*, *CONTRACT*, and *MERGE* are each executed in $O(\log m)$ time;

(3) operation *LIST** is executed in $O(\log m + k)$ time, where k is the number of retrieved vertices.

ACKNOWLEDGMENTS

I am grateful to Franco Preparata for suggesting improvements on the presentation of this paper and for his encouragement. I would also like to thank Herbert Edelsbrunner for pointing out the application of the results of [13] to the amortized complexity analysis given in Theorem 7, and Ioannis Tollis for comments and useful discussions.

REFERENCES

[1] J. Bondy and U. Murty, *Graph Theory with Applications,* North Holland, 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] C. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," Technical Report STAN-CS-72-259 (Ph.D. Dissertation), Computer Science Dept., Stanford Univ., 1972.

[5] G. Di Battista and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs," *Theoretical Computer Science*, (to appear).

[6] H. Edelsbrunner, L. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing*, vol. 15, no. 2, pp. 317-340, 1986.

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

[8] S. Even, *Graph Algorithms,* Computer Science Press, 1979.

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

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

[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] O. Fries, K. Mehlhorn, and S. Naher, "Dynamization of Geometric Data Structures," *Proc. ACM Symposium on Computational Geometry*, pp. 168-176, 1985.

[13] 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.

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

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

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

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

[18] D. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM J. Computing*, vol. 12, no. 1, pp. 28-35, 1983.

[19] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison Wesley, 1973. (pp. 466-468)

[20] D.T. Lee and F.P. Preparata, ''Location of a Point in a Planar Subdivision and its Applications,'' *SIAM J. Computing*, vol. 6, no. 3, pp. 594-606, 1977.

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

[22] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching,* Springer-Verlag, 1984. (pp. 213-216)

[23] M. Overmars, ''Range Searching in a Set of Line Segments,'' *Proc. ACM Symposium on Computational Geometry*, pp. 177-185, 1985.

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

[25] 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.

[26] N. Sarnak and R.E. Tarjan, ''Planar Point Location Using Persistent Search Trees,'' *Communications ACM*, vol. 29, no. 7, pp. 669-679, 1986.

[27] 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.

[28] R. Tamassia and I.G. Tollis, ''Centipede Graphs and Visibility on a Cylinder,'' pp. 252-263 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, 1987.

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