August 1988

UILU-ENG-88-2250
ACT-100

(2)

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

DYNAMIC DATA STRUCTURES FOR TWO-DIMENSIONAL SEARCHING

Roberto Tamassia

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA201173

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-88-2250 (ACT-#100)	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 Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 1800 G. St., Washington, D.C. 20552 800 N. Quincy, Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation, JSEP 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ECS-84-10902 N00014-84-C-0149
8c. ADDRESS (City, State, and ZIP Code) 1800 G. St., Washington D.C. 20552 800 N. Quincy, Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Dynamic Data Structures for Two-Dimensional Searching

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 _____	August 1988	142

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) (over)
design and analysis of algorithms, dynamic data structure, on-line algorithm, two-dimensional searching, computational geometry, point location, planar subdivision, planar graph,

19. ABSTRACT

In this report we investigate dynamic data structures and algorithms for searching in a subdivision of the plane. Three specific problems have been addressed in this area. The first problem, *dynamic point location*, considers a *geometric* subdivision of the plane into polygonal regions, and asks for the region that contains a given query point. The second problem, *dynamic planar embedding*, considers a *topological* subdivision of the plane induced by a planar embedding of a graph, and asks for a region that contains two given query vertices on its boundary (if one exists). The third problem, *dynamic transitive closure*, considers a planar acyclic digraph embedded in the plane, and asks for testing the existence of and/or reporting a directed path between two query vertices. In all of the three problems the update operations consist of inserting/deleting vertices and edges. We present several dynamic techniques that improve previously published results in the area. The space requirement ranges from O(n) to O(nlogn), and the query and update times range from O(logn) to O(log^{2}n), where n is the size of the subdivision. In addition to their good theoretical space/time performance, all the data structures and algorithms presented are also practical and easy to implement, and therefore suited for real-world applications. (KP)

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

DYNAMIC DATA STRUCTURES FOR TWO-DIMENSIONAL SEARCHING

BY

ROBERTO TAMASSIA

Laurea, University of Rome, Italy, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

DYNAMIC DATA STRUCTURES FOR TWO-DIMENSIONAL SEARCHING

Roberto Tamassia, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1988

In this thesis we investigate dynamic data structures and algorithms for searching in a subdivision of the plane. Three specific problems have been addressed in this area. The first problem, *dynamic point location*, considers a *geometric* subdivision of the plane into polygonal regions, and asks for the region that contains a given query point. The second problem, *dynamic planar embedding*, considers a *topological* subdivision of the plane induced by a planar embedding of a graph, and asks for a region that contains two given query vertices on its boundary (if one exists). The third problem, *dynamic transitive closure*, considers a planar acyclic digraph embedded in the plane, and asks for testing the existence of and/or reporting a directed path between two query vertices. In all of the three problems the update operations consist of inserting/deleting vertices and edges. We present several dynamic techniques that improve previously published results in the area. The space requirement ranges from $O(n)$ to $O(n \log n)$, and the query and update times range from $O(\log n)$ to $O(\log^2 n)$, where n is the size of the subdivision. In addition to their good theoretical space/time performance, all the data structures and algorithms presented are also practical and easy to implement, and therefore suited for real-world applications.

DEDICATION

Ai miei genitori.

To my parents.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Franco Preparata, for useful discussions and suggestions. I have especially benefited from his insight, enthusiasm, and encouragement.

I am also indebted to the other members of my thesis committee, Herbert Edelsbrunner, Chung L. Liu, Michael Loui, and David Muller for their constructive criticism. In particular, I am grateful to Herbert Edelsbrunner for discussions on point location and planar graph embedding, and to Michael Loui for suggestions to improve the presentation of this thesis.

At various stages of this research work I have had a profitable interaction with my friends and colleagues Scot Hornick, Arkady Kanevsky, David Luginbuhl, Sanjeev Maddila, Marsha Prastein, Majid Sarrafzadeh, Ioannis Tollis, and Dian Zhou.

The Coordinated Science Laboratory of the University of Illinois has provided computer facilities and staff support that have made this work easier. Janet Reese and Lila Rhoades deserve special thanks for their prompt and courteous secretarial assistance.

This work was supported in part by National Science Foundation Grant ECS-84-10902 and by the Joint Services Electronics Program under Contract N00014-84-C-0149.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
1.1. Dynamic Data Structures	1
1.2. Overview	2
2. PLANAR POINT LOCATION	5
2.1. Planar Subdivisions	5
2.2. Point Location Techniques	8
3. DYNAMIC POINT LOCATION: CHAIN METHOD	10
3.1. Introduction	10
3.2 Preliminaries	11
3.3. Ordering the Regions of a Monotone Subdivision	14
3.4. Dynamic Point Location	20
4. DYNAMIC POINT LOCATION: TRAPEZOID METHOD	36
4.1. Introduction	36
4.2. Preliminaries	36
4.3. Trapezoids	42
4.4. Data Structure	46
4.5. Query	50
4.6. Insertion of Edges	51
4.7. Other Update Operations	58
4.8. Extensions	60
5. DYNAMIC POINT LOCATION: RECONSTRUCTION METHOD	62
5.1. Introduction	62
5.2. Preliminaries	62
5.3. The Incremental Reconstruction Method	65
6. DYNAMIC TRANSITIVE CLOSURE	69
6.1. Introduction	69
6.2. Planar *st*-Graphs	71
6.3. On-Line Maintenance of a Planar *st*-Graph	79
6.4. Applications	87

CHAPTER PAGE

7. DYNAMIC PLANAR GRAPH EMBEDDING 94
7.1. Introduction 94
7.2. Preliminaries 96
7.3. Orientations of Planar Graphs 97
7.4. Topological Location 106
7.5. Dynamic Planar Graph Embedding 112
7.6. Applications 125

REFERENCES 128

VITA 135

CHAPTER 1

INTRODUCTION

1.1. Dynamic Data Structures

In recent years, the design of dynamic data structures has received increasing attention, and is one of the most challenging areas of theoretical computer science.

As a typical example, consider search problems where a set S of objects is given. We want to answer *queries* about these objects in an *on-line* fashion, i.e., the queries are not known in advance, and the answer to a query must be given before the next query can be processed. Typically, the set S is represented by means of a data structure D on which the query operations can be efficiently performed. The complexity of such a data structure is usually measured in terms of the *space* required for storing D, the *preprocessing time* needed to construct D, and the *query time*. In several applications the set S is not static, but dynamically evolves in time. This can be modeled by means of *update operations* that modify S and can be arbitrarily interleaved with the queries. Therefore, the problem arises of designing a dynamic data structure that supports on-line both queries and updates on S. In this case, the previous complexity measures are supplemented by the time bounds for the update operations.

The classical *dictionary problem* is the simplest example of a dynamic search problem. Here, S is a subset of an ordered set U, the queries ask whether a given element of U is in S, and updates are insertions and deletions of elements. For this problem, the earliest example of an efficient dynamic data structure is the AVL tree [1], which requires linear space and supports queries, insertions, and deletions in logarithmic time.

Our investigations focus on the problem of searching in a subdivision of the plane, a fundamental issue in computational geometry, for it represents the 2-dimensional extension of the dictionary problem. Three specific problems have been addressed in this area. The first problem, *dynamic point location*, considers a *geometric* subdivision of the plane into polygonal regions,

and asks for the region that contains a given query point. The second problem, *dynamic planar embedding*, considers a *topological* subdivision of the plane induced by a planar embedding of a graph G, and asks for a region that contains two given query vertices of G on its boundary (if one exists). The third problem, *dynamic transitive closure* considers a planar acyclic digraph G embedded in the plane, and asks for testing the existence of or reporting a directed path between two query vertices of G. In all of the three problems the update operations consist of inserting or deleting vertices and edges.

1.2. Overview

This thesis is conceptually subdivided into two parts. In the first part, consisting of Chapters 2, 3, 4, and 5, we consider searching in a geometric structure, and present three dynamic techniques for planar point location. In the second part, consisting of Chapters 6 and 7, we consider searching in a topological structure, and present dynamic techniques for maintaining the planar embedding and the transitive closure of a planar graph.

In Chapter 2, we provide basic geometric definitions and summarize previous research on planar point location, a fundamental geometric search problem which is used as a subroutine in a variety of algorithms. Most of the past research on planar point location has been targeted to the *static* case, where the subdivision is fixed, and point location queries have to be answered online. For this instance of the problem several efficient methods are available. The analogy with one-dimensional search, for which both static and dynamic optimal techniques have long been known, naturally motivates the desire to develop techniques for *dynamic* planar point location, where the planar subdivision can be modified by insertions and deletions of points and segments. Work on dynamic point location is a rather new undertaking, and in a recent paper [50] Sarnak and Tarjan indicate as one of the most challenging problems in computational geometry the development of a fully dynamic point location data structure whose space and query time performance are of the same order as that of the best known static techniques for this problem. As shown in the following, we come very close to this goal.

In Chapter 3, we present a dynamic point location technique for monotone subdivisions. The update operations are the insertion of a vertex on an edge and of a chain of edges between two vertices, and their reverse operations. Let n be the number of vertices of the subdivision.

The data structure uses space $O(n)$. The query time is $O(\log^2 n)$, the time for insertion/deletion of a vertex is $O(\log n)$, and the time for insertion/deletion of a chain with k edges is $O(\log^2 n + k)$, all worst case. This technique is based on the chain method of Lee-Preparata [35]. The emergence of full dynamic capabilities is afforded by a subtle choice of the chain set (separators), which induces a total order on the set of regions of the planar subdivision.

In Chapter 4, we present a dynamic point location technique for a class of monotone subdivisions (which includes convex subdivisions), whose vertices lie on a fixed set of N horizontal lines. The supported update operations are insertion/deletion of vertices and edges, and (horizontal) translation of vertices. This technique achieves query time $O(\log n + \log N)$, space $O(n \log N)$, and insertion/deletion time $O(\log n \log N)$, where n is the number of vertices of the subdivision. Hence, whenever $N = O(n)$, the query time is $O(\log n)$, which is optimal. This technique is based on the *trapezoid method* of Preparata [46], which has been experimentally shown to be the fastest point location method among those with asymptotically optimal query time [12]. It is easily realized that in many significant applications the most frequent operation is point location query, while updates are more rarely executed. Hence, this technique provides the most efficient solution for such applications.

In Chapter 5, we present a dynamic point location technique for triangulations, which is based on the incremental reconstruction method of Bentley-Saxe [3] and Overmars [42] for decomposable searching problems. It achieves $O(\log^2 n / \log\log n)$ query time, $O(n)$ space requirement, and $O(\log^3 n / \log\log n)$ update time, where n is the number of vertices. This technique can be used in conjunction with any static point location algorithm.

In Chapter 6, we present a dynamic technique for maintaining on-line the transitive closure of a planar *st*-graph, i.e., a planar acyclic digraph embedded in the plane with exactly one source, s, and one sink, t, both on the external face. This class of graphs was first introduced in the planarity testing algorithm of Lempel et al. [36], and was fruitfully used in a number of applications, which include planar graph embedding, graph planarization, graph drawing, floor planning, planar point location, visibility representations, motion planning, and VLSI layout compaction. Also, planar *st*-graphs are important in the theory of partially ordered sets, since they are associated with planar lattices [32]. We show that a planar *st*-graph G admits two total orders

(called leftist and rightist, respectively) on the set $V \cup E \cup F$, where V, E, and F are respectively the set of vertices, edges, and faces of G, with $|V| = n$. Assuming that G is to be dynamically modified by means of insertions of edges and expansions of vertices (and their inverses), we exhibit a $O(n)$-space dynamic data structure for the maintenance of these orders, such that an update can be performed in time $O(\log n)$. From this result, we derive a dynamic data structure for the *transitive-closure query* problem, which consists of testing for the existence of (or reporting) a directed path between two vertices u and v of G. The space is $O(n)$, and the time for queries and updates is $O(\log n)$ (worst-case). Also, we show that the discovered dynamic properties of planar *st*-graphs provide the topological underpinning of the planar point location technique for monotone subdivisions presented in Chapter 3, and can be applied to the problem of dynamic contact chain queries.

In Chapter 7, we present a dynamic technique that allows for incrementally constructing a planar embedding of a planar graph with n vertices. This problem, referred to as *dynamic embedding problem*, naturally arises in interactive CAD layout environments. A query operation asks whether a new edge (u,v) can be added to the embedding without introducing crossings, i.e., whether there is a face of the embedding whose boundary contains both u and v. The update operations are the insertion and deletion of vertices and edges. We present a data structure for the dynamic embedding problem that uses $O(n)$ space and supports queries and updates in time $O(\log n)$ (worst-case). These results are obtained by maintaining on-line an orientation of the graph, called *spherical st-orientation*, a notion that generalizes planar *st*-graphs. This technique has applications to circuit layout, graphics, motion planning, and computer-aided design. It also constitutes a first step toward the development of an efficient algorithm for testing on-line the planarity of a graph, under insertions and deletions of vertices and edges.

As a general remark, we would like to underscore that in addition to their good theoretical space/time performance, all the data structures and algorithms presented in this thesis are also practical and easy to implement, and therefore suited for real-world applications.

The material in the thesis is presented so that each chapter can be read independently from the others. Chapter 2 provides background material for Chapters 3, 4, and 5. For the reader's convenience, some definitions are repeated wherever appropriate.

CHAPTER 2

PLANAR POINT LOCATION

2.1. Planar Subdivisions

The motivation for the geometric definitions given below is readily obtained if we view a point of the plane as the central projection of a point of a hemisphere to whose pole this plane is tangent.

A *vertex* v in the plane is either a finite point or a point at infinity (the latter is the projection of a point on the hemisphere equator). An *edge* $e=(u,v)$ is the portion of the straight line between u and v, with the only restriction that u and v are not points at infinity associated with the same direction. Thus, e is either a segment or a straight-line ray, but not a whole straight line. When both u and v are at infinity, then e is an edge at infinity, i.e., a portion of the line at infinity (the projection of an arc of the equator). A (*polygonal*) *chain* γ is a sequence $(e_i : e_i=(v_i,v_{i+1}), i=1, \cdots, p-1)$ of edges; it is *simple* if nonintersecting; it is *monotone* if any line parallel to the x-axis intersects γ in either a point or a segment. In the following, the notion of "left" and "right" refer, when not specified otherwise, to a bottom-up orientation of the entity being considered (a chain, or, later on, a separator, an edge, etc.).

A *simple polygon* r is a region of the plane delimited by a simple chain with $v_p=v_1$, called the *boundary* of r. Note that r could be unbounded; in this case the boundary of r contains one or more edges belonging to the line at infinity. A polygon r is *monotone* if its boundary is partitionable into two monotone chains γ_1 and γ_2, called the *left chain* and *right chain* of r, respectively (see Fig. 2.1). Chains γ_1 and γ_2 share two vertices, referred to as $HIGH(r)$ and $LOW(r)$, respectively, with $y(HIGH(r)) > y(LOW(r))$. In other words, $HIGH(r)$ and $LOW(r)$ are, respectively, points of maximum and minimum ordinates in polygon r; each of them is unique unless it is the extreme of either a horizontal edge or an edge at infinity, in which case the selection between the two edge extremes is arbitrary. A *convex polygon* is a simple polygon r such that for any two vertices u and v of r the segment with endpoints u and v lies entirely inside r. A

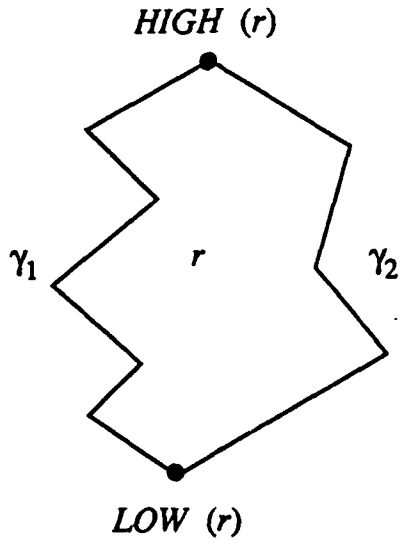


Figure 2.1 Nomenclature for a monotone polygon.

convex polygon is also a monotone polygon.

A *planar subdivision* $\mathfrak{R}$ is a partition of the plane into simple polygons, called the *regions* of $\mathfrak{R}$. It is easily realized that $\mathfrak{R}$ is determined by a planar graph G embedded in the plane whose edges are either segments or rays of straight lines (referred to as a "planar straight-line graph" in [35]): edges, vertices, and chains of G are referred to as edges, vertices, and chains of $\mathfrak{R}$. The vertices of $\mathfrak{R}$ are both the finite ones and those at infinity. This ensures the validity of the well-known Euler's formula and its corollaries:

$$|V| + |R| = |E| + 2; \quad |E| \leq 3|V| - 6; \quad |R| \leq 2|V| - 4,$$

where V, E, and R are respectively the set of vertices, edges, and regions of $\mathfrak{R}$.

A *monotone subdivision* $\mathfrak{R}$ is a planar subdivision whose regions are monotone polygons (see Fig. 2.2(a)). Monotone subdivisions form an important class of planar subdivisions, since they include *convex subdivisions*, whose regions are convex polygons (see Fig. 2(b)), and *triangulations*, whose regions are triangles (see Fig. 2(c)).

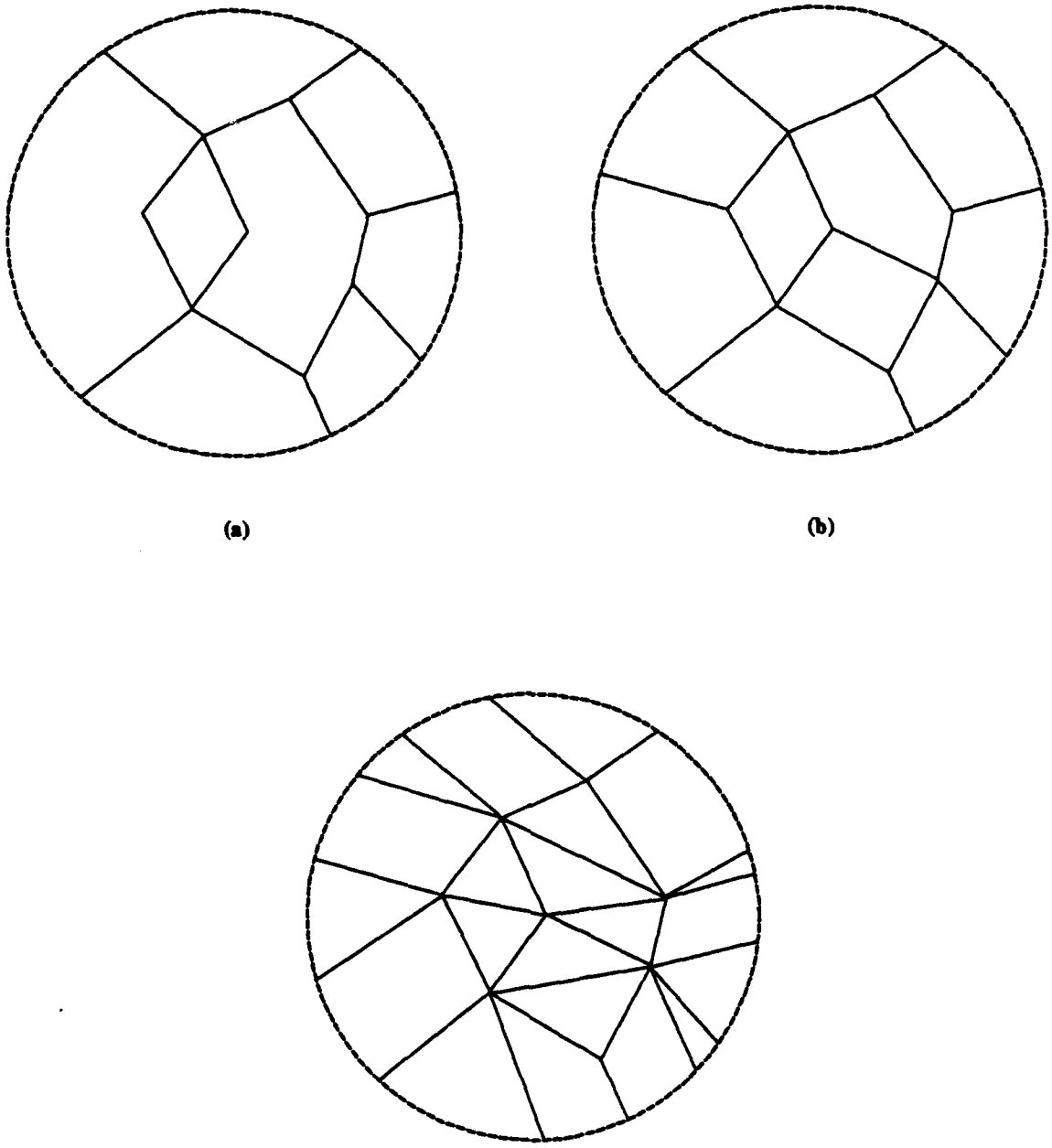


Figure 2.2 (a) A monotone subdivision. The dashed circle represents the line at infinity. (b) A convex subdivision. (c) A triangulation.

2.2. Point Location Techniques

The *point location* problem is formulated as follows: Given a planar subdivision $\Re$ with n vertices, determine to which region of $\Re$ a query point q belongs. If the query point q is on some vertex/edge of $\Re$, then that vertex/edge is reported. The repetitive use of $\Re$ and the on-line requirement on the answers call for a preprocessing of $\Re$ that may ease the query operation, analogous to sorting and binary search in one-dimensional search. The history of planar point location research spans more than a decade and is dense in results: the reader is referred to the extensive literature on this subject [11-13, 34, 35, 37, 46, 47, 50].

Most of the past research on the topic has focused on the *static* case of planar point location, where the subdivision $\Re$ is fixed. For this instance of the problem, several practical techniques are available today (e.g., [12, 13, 35, 46, 50]), some of which have $O(n)$ space requirement and $O(\log n)$ query time [13, 50], and are thus provably optimal in the asymptotic sense.

Work on dynamic point location is a rather recent undertaking. Overmars [43] proposed a technique for the case where the n vertices of $\Re$ are given, the boundary of each region has a bounded number of edges, and only edges can be easily inserted or deleted. The basic entities used in Overmars' method are the edges themselves; each edge currently in $\Re$ is stored in a segment tree defined on the fixed set of vertex abscissae, and the edge fragments assigned to a given node of the segment tree form a totally ordered set and are therefore efficiently searchable. This approach yields $O(n \log n)$ space requirement, and $O(\log^2 n)$ query and update times. Note that these measures are unrelated to the current number of edges in $\Re$. This technique can be extended to support insertions and deletions of vertices in $O(\log^2 n)$ *amortized* time, at the expense of deploying a rather complicated and not very practical data structure.

Another interesting dynamic point location technique, allowing both edge and vertex updates, is presented by Fries and Mehlhorn [18, 19, 39]. Their approach achieves $O(n)$ space requirement, $O(\log^2 n)$ query time, and $O(\log^4 n)$ *amortized* update time. If only insertions are considered, the update time is reduced to $O(\log^2 n)$ (*amortized*) [39, pp. 135-143].

Tables 2.1 and 2.2 summarize the performance of the aforementioned static and dynamic point location techniques. The new results on dynamic point location presented in Chapters 3, 4, and 5 of this thesis are included in Table 2.2.

Table 2.1 Static point location techniques

Space	Query Time	Source	Notes
n^2	$\log n$	Dobkin-Lipton 76	slab method, not practical
n	$\log^2 n$	Lee-Preparata 76	chain method, practical
n	$\log n$	Lipton-Tarjan 77	not practical
$n \log n$	$\log n$	Preparata 81	trapezoid method, practical
n	$\log n$	Kirkpatrick 83	not practical
n	$\log n$	Edelsbrunner et al. 85	chain method with fractional cascading, practical
n	$\log n$	Sarnak-Tarjan 85	slab method with persistent search trees, practical

Table 2.2 Dynamic point location techniques. Notation: $S(n)$: space; $Q(n)$: query time $I_e(n)$: time for edge insertion; $I_v(n)$: time for vertex insertion; $D_e(n)$: time for edge deletion; $D_v(n)$: time for vertex deletion.

$S(n)$	$Q(n)$	$I_e(n)$	$I_v(n)$	$D_e(n)$	$D_v(n)$	Source	Notes
$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	Overmars 85	fixed vertices, regions with $O(1)$ edges, not practical vertex updates
n	$\log^2 n$	$\log^4 n$	$\log^4 n$	$\log^4 n$	$\log^4 n$	Fries-Mehlhorn 85	amortized update times
n	$\log^2 n$	$\log^2 n$	$\log^2 n$	–	–	Fries-Mehlhorn 84	only insertions, amortized update time
n	$\log^2 n$	$\log^2 n + k$	$\log n$	$\log^2 n + k$	$\log n$	This Thesis Chapter 3	monotone subdivision, insertions/deletions of k-edge chains
$n \log N$	$\log n + \log N$	$\log n \log N$	$\log n \log N$	$\log n \log N$	$\log n \log N$	This Thesis Chapter 4	convex subdivision, vertices on N fixed horizontal lines
n	$\frac{\log^2 n}{\log \log n}$	$\frac{\log^3 n}{\log \log n}$	$\frac{\log^3 n}{\log \log n}$	$\frac{\log^3 n}{\log \log n}$	$\frac{\log^3 n}{\log \log n}$	This Thesis Chapter 5	triangulations
n	$\frac{1}{\varepsilon^2} \log n$	$\frac{1}{\varepsilon} n^{\varepsilon} \log n$	$\frac{1}{\varepsilon} n^{\varepsilon} \log n$	$\frac{1}{\varepsilon} n^{\varepsilon} \log n$	$\frac{1}{\varepsilon} n^{\varepsilon} \log n$	This Thesis Chapter 5	triangulations

CHAPTER 3

DYNAMIC POINT LOCATION: CHAIN METHOD

3.1. Introduction

In this chapter we present a fully dynamic point location technique for monotone subdivisions. The central result is expressed by the following theorem:

Theorem 3.1 Let $\mathfrak{R}$ be a monotone planar subdivision with n vertices. There exists a dynamic point location data structure with $O(n)$ space requirement and $O(\log^2 n)$ query time, which allows for insertion/deletion of a vertex in time $O(\log n)$ and insertion/deletion of a chain of k edges in time $O(\log^2 n + k)$, all time bounds being worst-case.

It must be underscored that our method allows for arbitrary insertions and deletions of vertices and edges, the only condition being that monotonicity of the subdivision be preserved. It is also interesting to observe that the dynamic technique is based on the same geometric objects, the separating chains, which yielded the first practical, albeit suboptimal, point location technique of Lee-Preparata [35], and later on the practical and optimal algorithm of Edelsbrunner-Guibas-Stolfi [13]

Our technique represents a reasonably efficient solution of the dynamic point location problem in monotone subdivisions. It requires no new sophisticated or bizarre data structures, and it appears eminently practical.

It remains an open problem whether $O(\log n)$ optimal performance is achievable for query/update times; in particular, whether the technique of fractional cascading, which achieved optimality for its suboptimal static predecessor [13], can also be successfully applied to the technique discussed in this chapter.

The rest of the chapter is organized as follows. In Section 3.2 we review the technical background and formulate the problem. In Section 3.3 we introduce a total ordering of the

regions of $\mathfrak{R}$, which plays a crucial role in the dynamic point location algorithm illustrated in Section 3.4.

3.2. Preliminaries

We recall from the preceding chapter that a monotone subdivision is a partition of the plane into monotone polygons (see Fig. 3.1(a)). Given a monotone subdivision $\mathfrak{R}$, a *separator* σ of $\mathfrak{R}$ is a monotone chain $(v_1, \cdots, v_p)$ of $\mathfrak{R}$ with the property that v_1 and v_p are points at infinity (hence, each horizontal line intersects a separator either in a point or in a segment). A separator of $\mathfrak{R}$ is illustrated with bold line segments in Fig. 3.1(a). Given separators σ_1 and σ_2 of $\mathfrak{R}$, we say that σ_1 *is to the left of* σ_2, denoted $\sigma_1 < \sigma_2$, if, for any horizontal line l intersecting both σ_1 and σ_2 in a single point, the abscissa of the intersection of l with σ_2 is no smaller than the corresponding one with σ_1. A *partial subdivision* is the portion of a monotone subdivision contained between two distinct separators σ_1 and σ_2, with $\sigma_1 < \sigma_2$. A *complete family of separators*

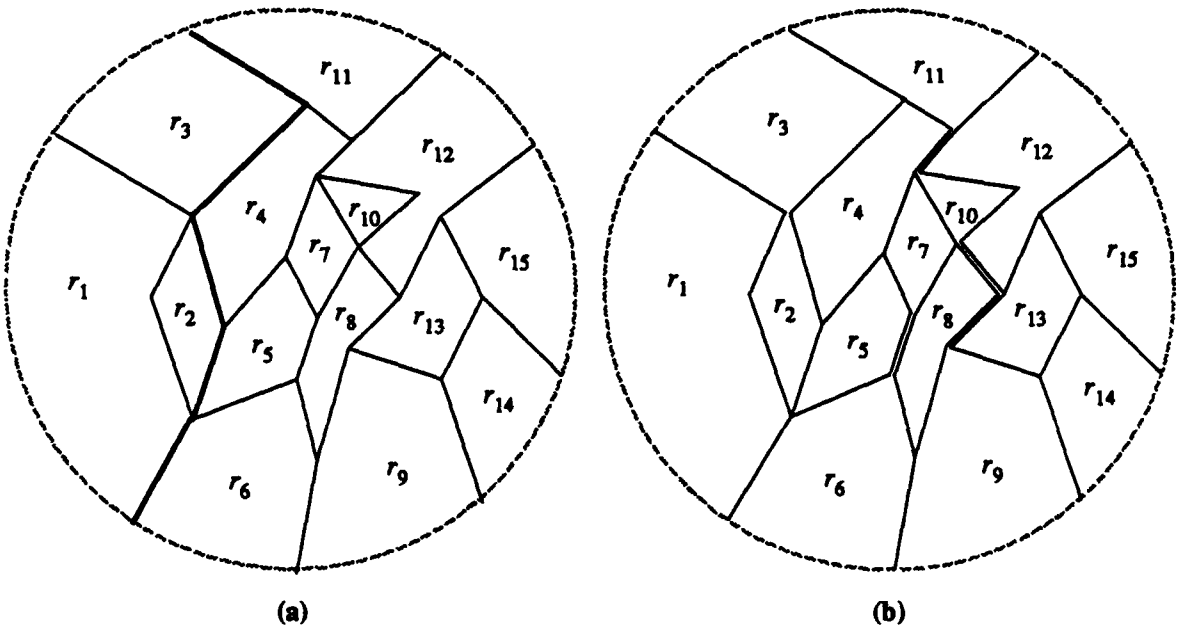


Figure 3.1 (a)) A monotone subdivision $\mathfrak{R}$. The dashed circle represents the line at infinity. A separator is shown with bold line segments. **(b)** The regular subdivision $\mathfrak{R}^*$ obtained from $\mathfrak{R}$ by forming all maximal clusters.

Σ for $\mathfrak{R}$ is a sequence $(\sigma_1,\sigma_2,\cdots,\sigma_t)$ with $\sigma_1 < \sigma_2 < \cdots < \sigma_t$, such that every edge of $\mathfrak{R}$ is contained in at least one separator of Σ. Notice that from Euler's formula $t = O(n)$. As shown in [35], every monotone subdivision admits a complete family of separators.

Given a complete family of separators Σ for $\mathfrak{R}$, it is well-known [35] how to use Σ to perform planar point location in $\mathfrak{R}$. If n is the number of vertices of $\mathfrak{R}$, then in time $O(\log n)$ we can decide on which side of a separator the query point q lies; applying this operation as a primitive, a bisection search on Σ determines in time $O(\log t \cdot \log n)$ two consecutive separators between which q lies. This process can be adapted or supplemented to determine the actual region r to which q belongs.

Since Σ is used in a binary search fashion, each separator is assigned to a node of a binary search tree, called the *separator-tree* $\mathfrak{I}$. With a minor abuse of language, we call "node σ" the node of $\mathfrak{I}$ to which σ has been assigned. An edge e of $\mathfrak{R}$ belongs, in general, to a nonempty interval $(\sigma_i,\sigma_{i+1},\cdots,\sigma_j)$ of separators. Let node σ_k, $i \le k \le j$, be the common ancestor of nodes $\sigma_i,\sigma_{i+1},\cdots,\sigma_j$; then e is called a *proper edge* of σ_k and is stored only once at node σ_k. We denote by $proper(\sigma_k)$ the set of proper edges of σ_k, i.e., the edges of σ_k stored at node σ_k. This yields $O(n)$ storage space while guaranteeing the correctness of the technique (see [13,35]). Note that edges whose extremes are both at infinity need not be stored.

We now illustrate that a planar subdivision $\mathfrak{R}$ can be constructed by an appropriate sequence of the following operations:

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

REMOVEPOINT $(v;e)$: Let v be a vertex of degree 2 whose incident edges, $e_1=(u,v)$ and $e_2=(v,w)$, are on the same straight line. Remove v and replace e_1 and e_2 with edge $e=(u,w)$.

INSERTCHAIN $(\gamma,v_1,v_2,r;r_1,r_2)$: Add the monotone chain $\gamma=(v_1,w_1,\cdots,w_{k-1},v_2)$, with $y(v_1) \le y(v_2)$, to $\mathfrak{R}$ inside region r of $\mathfrak{R}$, which is decomposed into regions r_1 and r_2, with r_1 and r_2, respectively, to the left and to the right of γ, directed from v_1 to v_2.

REMOVECHAIN $(\gamma;r)$: Let γ be a monotone chain whose nonextreme vertices have degree 2. Remove γ and merge the regions r_1 and r_2 formerly on the two sides of γ into region r. [The operation is allowed only if the subdivision $\mathfrak{R}'$ so obtained is monotone.]

With the above repertory of operations, we claim that a monotone subdivision $\mathfrak{R}$ can always be transformed into a monotone subdivision $\mathfrak{R}'$ having either fewer vertices or fewer edges. Then by $O(n)$ such transformations we obtain the trivial subdivision, whose only region is the entire plane (bounded by the line at infinity).

Indeed, let σ be a separator of $\mathfrak{R}$, and imagine traversing σ from $-\infty$ to $+\infty$. Let $\deg^-(v)$ and $\deg^+(v)$, respectively, denote the numbers of edges incident on v and lying in the halfplanes $y < y(v)$ and $y > y(v)$ (assume for simplicity that no edge is horizontal). Let $(v_1, \cdots, v_p)$ be the sequence of finite vertices of σ with $\deg^+(v_i) + \deg^-(v_i) \geq 3$, as encountered in the traversal of σ. If this sequence is empty, the entire chain can be trivially removed. Therefore, assume $p \geq 1$. If there are two consecutive vertices v_i and v_{i+1} such that $\deg^+(v_i) \geq 2$ and $\deg^-(v_{i+1}) \geq 2$, then the chain γ (of degree-2 vertices) between v_i and v_{i+1} can be deleted by *REMOVECHAIN* while preserving monotonicity. Suppose that there are no two such vertices. If $\deg^-(v_1) \geq 2$, then the portion of σ from $-\infty$ to v_1 can be deleted. Otherwise, $\deg^-(v_1) = 1$, and the preceding conditions give raise to the following chain of implications:

$$\left[\deg^-(v_1) = 1\right] \Rightarrow \left[\deg^+(v_1) \geq 2\right] \Rightarrow \left[\deg^+(v_2) \geq 2\right] \Rightarrow \cdots \Rightarrow \left[\deg^+(v_p) \geq 2\right];$$

the latter shows that the portion of σ from v_p to $+\infty$ can be deleted. This establishes our claim.

When all finite vertices have disappeared, the resulting subdivision consists of a closed chain of arcs whose union is the line at infinity. Removal of the vertices at infinity completes the transformation. Since all of the above operations are reversible, this shows that any monotone subdivision $\mathfrak{R}$ with n vertices can be constructed by $O(n)$ operations of the above repertory:

Theorem 3.2 An arbitrary planar subdivision $\mathfrak{R}$ with n vertices can be assembled starting from the empty subdivision by a sequence of $O(n)$ *INSERTPOINT* and *INSERTCHAIN* operations, and can be disassembled to obtain the empty subdivision by a sequence of $O(n)$ *REMOVEPOINT* and *REMOVECHAIN* operations.

Although the above operations are sufficient to assemble and disassemble any monotone subdivision, the following operation is also profitably included in the repertory:

MOVEPOINT $(v;x,y)$: Translate a degree-2 vertex v from its present location to point (x,y). [The operation is allowed only if the subdivision $\mathfrak{R}'$ so obtained is monotone and topologically equivalent to $\mathfrak{R}$.]

3.3. Ordering the Regions of a Monotone Subdivision

Let $\mathfrak{R}$ be a monotone subdivision, and assume for simplicity that none of its finite edges is horizontal. Given two regions r_1 and r_2 of $\mathfrak{R}$, we say that r_1 is *left-adjacent to* r_2, denoted $r_1 \ll r_2$, if r_1 and r_2 share an edge e, and any separator of $\mathfrak{R}$ containing e leaves r_1 to its left and r_2 to its right. Notice that relation $\ll$ is trivially antisymmetric. But we can also show

Lemma 3.1 Relation $\ll$ on the regions of $\mathfrak{R}$ is acyclic.

Proof: Assume $r_1 \ll r_2 \ll \cdots \ll r_k \ll r_1$. Let Σ be a complete family of separators and let $\{\sigma_1, \cdots, \sigma_{k-1}\} \subseteq \Sigma$ be such that σ_i separates r_i and r_{i+1}, so that $\sigma_1 < \sigma_2 < \cdots < \sigma_{k-1}$. If $\sigma \in \Sigma$ leaves r_k to its left and r_1 to its right, we have $\sigma_{k-1} < \sigma$ and $\sigma < \sigma_1$, a contradiction since the separators are ordered. □

Thus, the transitive closure of $\ll$ is a partial order, referred to as *to the left of*, and denoted $\rightarrow$. Specifically, $r_1 \rightarrow r_2$ if there is a path from r_1 to r_2 in the directed graph of the relation $\ll$ on the set of regions. Correspondingly, given two regions r_1 and r_2 of $\mathfrak{R}$, we say that r_1 *is below* r_2, denoted $r_1 \uparrow r_2$, if there is a monotone chain from $HIGH(r_1)$ to $LOW(r_2)$. Obviously, $\uparrow$ is a partial order on the set of regions. The following lemma shows that these two partial orders are complementary.

Lemma 3.2 Let r_1 and r_2 be two regions of $\mathfrak{R}$. Then one and only one of the following holds:

$$r_1 \rightarrow r_2,\ r_2 \rightarrow r_1,\ r_1 \uparrow r_2,\ r_2 \uparrow r_1.$$

Proof: Let σ_L be the leftmost separator that contains the left chain of the boundary of r_1 and, analogously, let σ_R be the rightmost separator containing the right chain of the boundary of r_1. These separators partition $\mathfrak{R}$ into five portions, each a partial subdivision: one of them is r_1 itself, and the others are denoted L, R, B, and T (see Fig. 3.2). Now, we consider four mutually exclusive cases for r_2, one of which must occur:

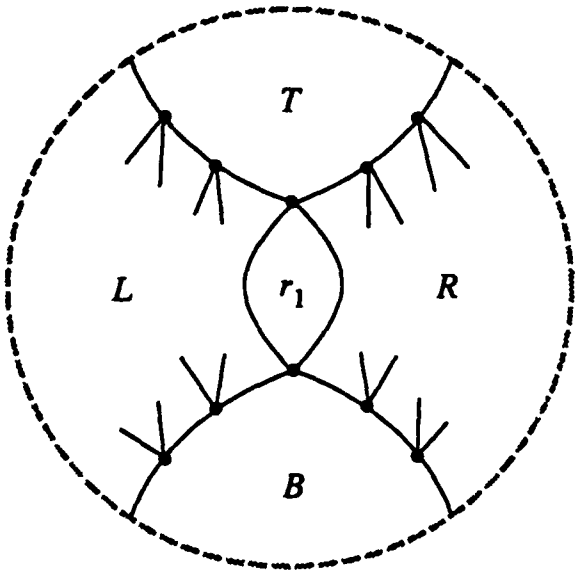


Figure 3.2 For the proof of Lemma 3.2.

(1) $r_2 \in L$. Consider any sequence of regions $(r'_1, \cdots, r'_s)$ such that $r_2 \rightarrow r'_1$, $r'_i \rightarrow r'_{i+1}$ for $i = 1, \cdots, s-1$, and the right chain of r'_s has a nonempty intersection with σ_L. If the right chain of r'_s has an edge that is also on the left boundary of r_1, then $r_2 \rightarrow r_1$. Otherwise, by the definition of σ_L, there is a sequence $r'_{s+1}, \cdots, r'_p$ of regions such that $r'_j \rightarrow r'_{j+1}$ for $j = s+1, \cdots, p-1$, and $r'_p \rightarrow r_1$. Thus, in all cases, $r_2 \rightarrow r_1$.

(2) $r_2 \in R$: Arguing as in (1), we establish $r_1 \rightarrow r_2$.

(3) $r_2 \in B$. Since $LOW(r_1)$ is the highest ordinate vertex in B, there is a monotone chain from any vertex in B to $LOW(r_1)$, and, in particular, from $HIGH(r_2)$ to $LOW(r1)$. Thus $r_2 \uparrow r_1$.

(4) $r_2 \in T$. Arguing as in (3), we establish $r_1 \uparrow r_2$. □

We say that r_1 *precedes* r_2, denoted $r_1 < r_2$, if either $r_1 \rightarrow r_2$ or $r_1 \uparrow r_2$.

Theorem 3.3 The relation $<$ on the regions of $\mathfrak{R}$ is a total order.

As an example, the region subscripts in Fig. 3.1(a) reflect the order $<$.

A *regular subdivision* is a monotone subdivision having no pair (r_1, r_2) of regions such that $r_1 \uparrow r_2$. For example, in Fig. 3.1(a) we have $r_9 \uparrow r_{10}$, which shows that the illustrated monotone subdivision is not regular. An example of regular subdivision is given in Fig. 3.3(a).

The significance of regular subdivisions is expressed by Theorem 3.4 below. It is easily realized that there is a unique, complete family $\Sigma = (\sigma_1, \cdots, \sigma_t)$ of separators for a regular subdivision $\mathfrak{R}$. By the definition of separator, all regions to the left of σ_j precede all those to its right in the order $<$. Let $\mathfrak{I}$ be the separator-tree for the above family Σ. Recalling the rule for storing the edges of separator σ in $\mathfrak{I}$, as reviewed in Section 3.2, we have

Theorem 3.4 In a regular subdivision $\mathfrak{R}$, the edges of *proper*(σ) in $\mathfrak{I}$ form a single chain (see Fig. 3.3(b)).

Proof: Assume for a contradiction that σ contains a chain γ which is the bottom-to-top concatenation of three nonempty chains γ_1, γ_2, and γ_3, where γ_1 and γ_3 consist of proper edges of σ, and γ_2 contains no such edge. Let v_1 and v_2 be the bottom and top vertices, respectively, of γ_2 and let $e' \in \gamma_1$ and $e'' \in \gamma_2$ be the edges of σ incident on v_1. Since $e'' \notin$ *proper*(σ), we must have

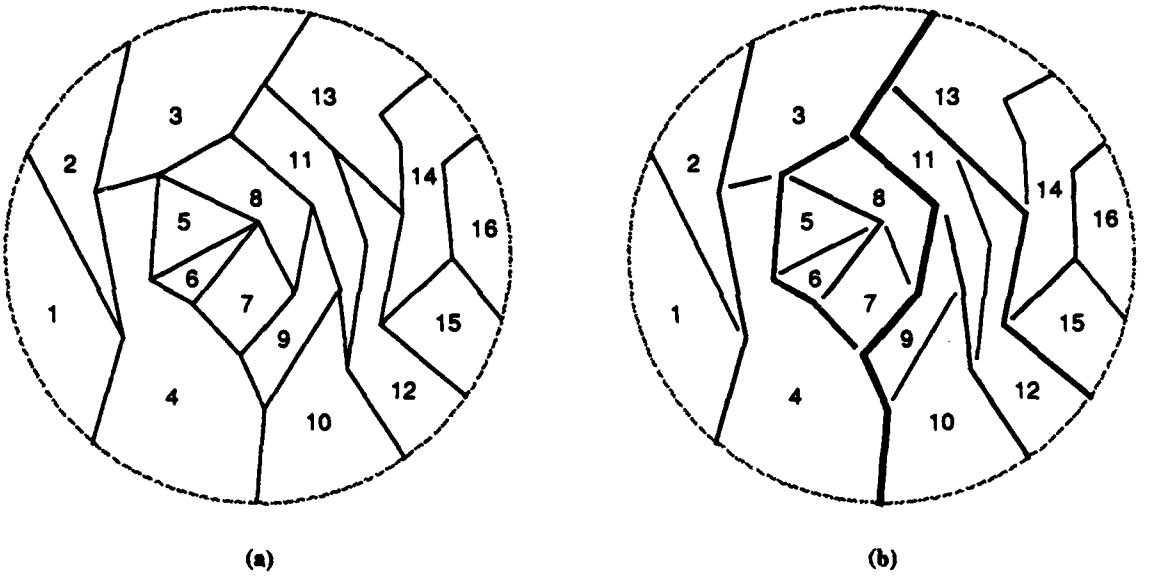


Figure 3.3 (a) A regular subdivision, and (b) its chains $\{proper(\sigma) : \sigma \in \Sigma\}$.

$e'' \in$ *proper* (σ'), where node σ' is an ancestor of node σ in $\mathfrak{I}$. We claim there is a region r_1 such that $v_1 = HIGH(r_1)$. Otherwise, each separator containing e'' – and so σ' – also contains e', contrary to the hypothesis that node σ is closest to the root among the nodes whose separator contains e'. Analogously, we show that there is a region r_2 for which $v_2 = LOW(r_2)$. Since γ_2 is a monotone chain from $HIGH(r_1)$ to $LOW(r_2)$, then $r_1 \uparrow r_2$, whence a contradiction. □

This theorem shows that a regular subdivision has a particularly simple separator-tree. In the next section we shall show that the property expressed by Theorem 3.4 is crucial for the efficient dynamization of the chain method for point location. We now slightly generalize the notion of region in a way that will enable us to show that any monotone subdivision embeds a unique regular subdivision. We say that two regions, r_1 and r_2, consecutive in $<$, with $r_1 \uparrow r_2$, are *vertically consecutive*. We then have

Lemma 3.3 If r_1 and r_2 are two vertically consecutive regions of a monotone subdivision $\mathfrak{R}$, then the chain from $HIGH(r_1)$ to $LOW(r_2)$ is unique.

Proof: The lemma holds trivially if $HIGH(r_1) = LOW(r_2)$. Thus, assume the contrary. Since $r_1 \uparrow r_2$, there is a monotone chain γ from $HIGH(r_1)$ to $LOW(r_2)$. Suppose now, for a contradiction, that there is a monotone chain γ' from $HIGH(r_1)$ to $LOW(r_2)$ distinct from γ. Then $\gamma \cup \gamma'$ defines the boundary of a partial subdivision that contains at least one region of $\mathfrak{R}$. For any region r inside this partial subdivision, there are (possibly empty) chains from $HIGH(r_1)$ to $LOW(r)$ and from $HIGH(r)$ to $LOW(r_2)$, so that $r_1 \uparrow r \uparrow r_2$, contrary to the hypothesis that r_1 and r_2 are consecutive in $<$. □

Given two vertically consecutive regions, r_1 and r_2, in $\mathfrak{R}$, with $r_1 \uparrow r_2$, the unique chain from $HIGH(r_1)$ to $LOW(r_2)$ is called a *channel*.

Lemma 3.4 All channels are pairwise vertex-disjoint.

Proof: Assume, for a contradiction, that there are two channels γ_1 and γ_2 that are not vertex-disjoint, where γ_1 connects regions r_1 and r_2, γ_2 connects regions r_3 and r_4, and $r_1 < r_2 < r_3 < r_4$ (see Fig. 3.4). Since γ_1 and γ_2 share a vertex x, there is a chain from $HIGH(r_3)$ to $LOW(r_2)$, which consists of the portion of γ_2 from $HIGH(r_3)$ to x, and the portion

of γ_1 from x to $LOW(r_2)$. Hence, we have $r_3 < r_2$, a contradiction. □

Given two vertically consecutive regions, r_1 and r_2, with $r_1 \uparrow r_2$, we imagine duplicating the channel from r_1 to r_2 and view the measure-zero region delimited by the two replicas as a degenerate polygon joining r_1 and r_2 and merging them into a new region $r_1 \cup r_2$ (see Fig. 3.5). Clearly, we can merge in this fashion any sequence of vertically consecutive pairs. This is formulated in the following definition:

Clusters are recursively defined as follows:

(1) An individual region r is a cluster;

(2) Given two vertically consecutive clusters χ_1 and χ_2, with $\chi_1 \uparrow \chi_2$, their union is a cluster χ, denoted $\frac{\chi_1}{\chi_2}$ (the horizontal bar denotes the channel).

A *maximal cluster* χ is one which is not properly contained in any other cluster.

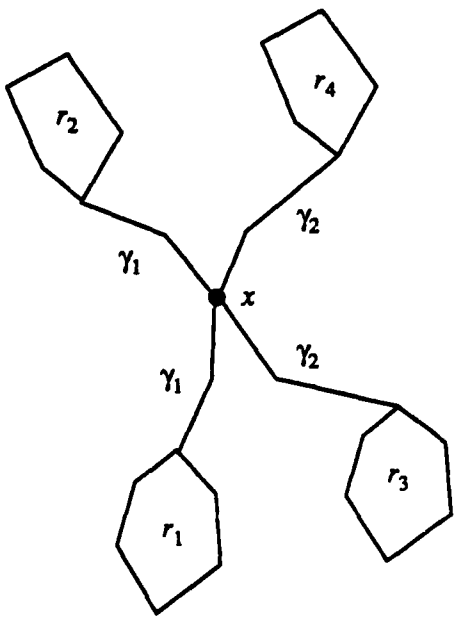


Figure 3.4 Example for the proof of Lemma 3.4.

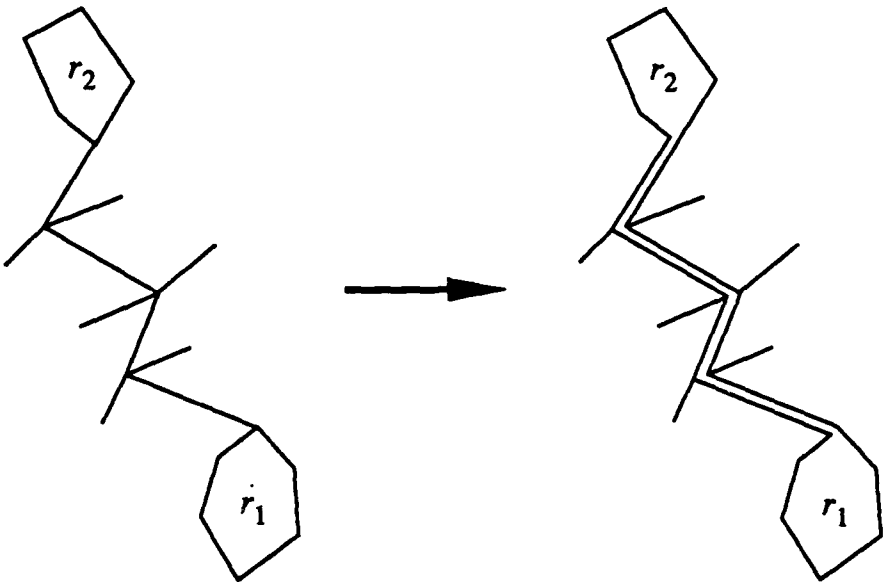


Figure 3.5 Creation of a channel between two vertically consecutive regions.

The unique subdivision resulting by forming all maximal clusters of $\mathfrak{R}$ is denoted $\mathfrak{R}^*$. Figure 3.1(b) illustrates the regular subdivision $\mathfrak{R}^*$ corresponding to the subdivision $\mathfrak{R}$ of Fig. 3.1(a). Notice the clusters r_2–r_3, r_6–r_7, and r_9–r_{10}–r_{11}.

The above definition leads us to a convenient string notation for the order $<$, as well as for the cluster structure where appropriate. Normally, we shall use lower-case roman letters for individual regions, lower-case greek letters for clusters, and upper-case roman letters otherwise (i.e., for collections of consecutive regions not forming a single cluster). Specifically, we have

(1) A cluster (possibly, a region) is a string.

(2) Given two strings A and B, such that the rightmost cluster of A and the leftmost cluster of B are consecutive (contiguity), then AB is a string.

A subdivision may be represented by means of its structural decomposition. For example, the subdivision of Fig. 3.1(b) is described by the string

$$r_1\chi_1 r_4 r_5 \chi_2 r_8 \chi_3 r_{12} r_{13} r_{14} r_{15},$$

where $\chi_1 = r_2 - r_3$, $\chi_2 = r_6 - r_7$, and $\chi_3 = r_9 - r_{10} - r_{11}$.

Later on, we will find it convenient to explicitly indicate that two consecutive regions r_1 and r_2 may or may not form a cluster. We shall denote this with the string notation $r_1 - - r_2$, where "- -" means "potential channel."

We conclude this section with the following straightforward observation:

Theorem 3.5 The subdivision $\mathfrak{R}^*$ obtained by forming all maximal clusters of a monotone subdivision $\mathfrak{R}$ is regular.

Note that in the transformation of $\mathfrak{R}$ to $\mathfrak{R}^*$ only the edges of channels are duplicated. By Lemma 3.4, each edge is duplicated at most once, thereby ensuring that the number of edges remains $O(n)$.

3.4. Dynamic Point Location

3.4.1. Data structure In the following description, we assume that all sorted lists are stored as *red-black trees* [20, 55]. We recall the following properties of red-black trees, which are important in the subsequent time complexity analyses.

(1) Only $O(1)$ rotations are needed to rebalance the tree after an insertion/deletion.

(2) The data structure can be used to implement concatenable queues. Operations *SPLICE* and *SPLIT* of concatenable queues take $O(\log n)$ time and need $O(\log n)$ rotations each for rebalancing.

The search data structure consists of a main component, called the augmented separator tree, and an auxiliary component, called the dictionary. The *augmented separator-tree* $\mathfrak{I}$ has a primary and secondary structure. The *primary structure* is a separator-tree for $\mathfrak{R}^*$, i.e., each of its leaves is associated with a region of $\mathfrak{R}^*$ (a maximal cluster of $\mathfrak{R}$), and each of its internal nodes is associated with a separator of $\mathfrak{R}^*$. (The left-to-right order of the leaves of the primary structure of $\mathfrak{I}$ corresponds to the order $<$ on the regions of $\mathfrak{R}^*$.) The *secondary structure* is a collection of lists, each realized as a search tree. Specifically, node σ points to the list *proper*(σ) sorted from bottom to top, and the leaf associated with cluster χ (briefly called "leaf χ") points to the list *regions*(χ) of the regions that form cluster χ, also sorted from bottom to top.

Given two regions r_1 and r_2 consecutive in $<$, the separator σ between r_1 and r_2 is associated with the least common ancestor of the leaves associated with the respective clusters of r_1 and r_2 in $\Im$. By the definition of separator-tree, the edges of σ are stored in the path from node σ to the root of $\Im$; by Theorem 3.4, in a regular subdivision, each extreme vertex of *proper*(σ) splits *proper*(σ'), for some ancestor node σ' of node σ, into two chains. More precisely, the following simple lemma, stated without proof, makes explicit the allocation of the edges of σ to the nodes of $\Im$.

Lemma 3.5 Let σ be a separator of $\Re^*$, and $\sigma_1, \cdots, \sigma_h$ be the sequence of nodes of $\Im$ on the path from the root ($=\sigma_1$) to σ. Then

$$\sigma = (\alpha_1, \alpha_2, \cdots, \alpha_h, proper(\sigma), \beta_h, \beta_{h-1}, \cdots, \beta_1),$$

where α_i and β_i are (possibly empty) subchains of *proper*(σ_i), $i = 1, \cdots, h$.

In order to dynamically maintain the channels, it is convenient to keep two representatives e' and e'' of each edge e, which are created when e is inserted into $\Re$. If e does not belong to a channel, e' and e'' are joined into a *double edge* and belong to the same *proper*(σ). If instead e is part of a channel, then e' and e'' are *single edges* and belong to distinct *proper* (σ') and *proper* (σ''). In the latter case e' and e'' are on the boundary of the same cluster χ, so that nodes σ' and σ'' are on the path from leaf χ to the root of $\Im$. Therefore, we represent *proper*(σ) by means of two lists, denoted *strand1* (σ) and *strand2* (σ). List *strand1* (s), called *primary strand*, stores a representative for each edge of *proper*(σ), in bottom-to-top order. List *strand2* (σ), called *secondary strand*, stores a representative for each double edge of *proper*(σ), in bottom-to-top order.

Moreover, associated with each chain *proper*(σ) there are two boolean indicators $t(\sigma)$ and $b(\sigma)$, corresponding respectively to the topmost and bottommost vertices of *proper*(σ). Specifically, let σ' be the ancestor of σ such that the topmost vertex of *proper* (σ) is an internal vertex of the chain *proper* (σ') (for the special case where the topmost vertex of σ is at infinity, we let σ' be the father of σ). We define $t(\sigma) = left$ if σ is to the left of σ', and $t(\sigma) = right$ if σ is to the right of σ'. Parameter $b(\sigma)$ is analogously defined.

The *dictionary* contains the sorted lists of the vertices, edges, and regions of $\mathfrak{R}$, each sorted according to the alphabetic order of their names. With each vertex v we store a pointer to the representative of v in the (at most two) chains $proper(\sigma)$ and $proper(\sigma)$ of which v is a nonextreme vertex. With each edge e we store pointers to the two representatives of e in the data structure. Finally, with each region r we store the vertices $HIGH(r)$ and $LOW(r)$, and a pointer to the representative of r in the list $regions(\chi)$ such that χ is the maximal cluster containing r. The dynamic maintenance of the dictionary in the various operations can be trivially performed in $O(\log n)$ time, and will not be explicitly mentioned in the following.

To analyze the storage used by the data structure, we note: the primary structure of $\mathfrak{I}$ has $O(n)$ nodes, since there are $O(n)$ regions (by Euler's formula) and therefore $O(n)$ separators; the secondary structure of $\mathfrak{I}$ also has size $O(n)$, since there are $O(n)$ edges in $\{proper(\sigma) : \sigma \in \mathfrak{I}\}$ and $O(n)$ regions in $\{regions(\chi) : \chi \in \mathfrak{I}\}$ (again, by Euler's formula); the auxiliary component has one record of bounded size per vertex, edge, and region. Therefore, we conclude

Theorem 3.6 The data structure for dynamic point location has storage space $O(n)$.

Note that the above data structure is essentially *identical* with the one originally proposed for the static version of the technique [35]. What is remarkable is that the single-chain structure of the proper edges of any given separator, due to our specific choice of the separator family, is the key for the emergence of full dynamic capabilities.

We now show that the property expressed by Theorem 3.4 allows us to establish an important dynamic feature of the data structure. According to standard terminology, a *rotation at node* v of a binary search tree is the restructuring of the subtree rooted at v so that one of the children of v becomes the root thereof. A rotation is either *left* or *right* depending upon whether the right or left child becomes the new root, respectively. We then have

Lemma 3.6 A rotation at a node of $\mathfrak{I}$ can be performed in $O(\log n)$ time.

Proof: Without loss of generality, we consider a left rotation as illustrated in Fig. 3.6. Clearly, the separators stored at nodes outside the subtree rooted at σ_2 in Fig. 3.6(a) are not affected by the rotation, nor are those in the subtrees rooted at σ_1, σ_3, and σ_5. Thus the only alterations involve separators σ_2 and σ_4. If $\sigma_4 \cap proper(\sigma_2) = \emptyset$, then the modification is trivial. Suppose then that $\sigma_4 \cap proper(\sigma_2) \neq \emptyset$. In this case the set $\sigma_4 \cap proper(\sigma_2)$ forms either the initial or the final segment of the chain $proper(\sigma_2)$, or both, for, otherwise, its removal from $proper(\sigma_2)$ would yield an updated (after the rotation) $proper(\sigma_2)$ violating Theorem 3.4. Thus the update is accomplished by (1) splitting $proper(\sigma_2)$ into $\gamma_2 = \sigma_4 \cap proper(\sigma_2)$ and its relative complement γ_1; (2) splicing γ_2 with $proper(\sigma_4)$ to form the updated $proper(\sigma_4)$; and (3) setting the updated $proper(\sigma_2)$ equal to γ_1. Note that the extreme vertices of $proper(\sigma_4)$ are obtained in time $O(1)$, and the splitting vertices of $proper(\sigma_2)$ are determined in time $O(\log n)$. Since data structures for $proper(\sigma_2)$ and $proper(\sigma_4)$, (i.e., the red-black trees associated with lists *strand1* (σ) and *strand2* (σ)) are also concatenable queues, the splitting and splicing operations are executed in time $O(\log n)$ as well. The parameters $t(\sigma)$ and $b(\sigma)$ for the resulting separators are updated in $O(1)$ time by means of straightforward rules. □

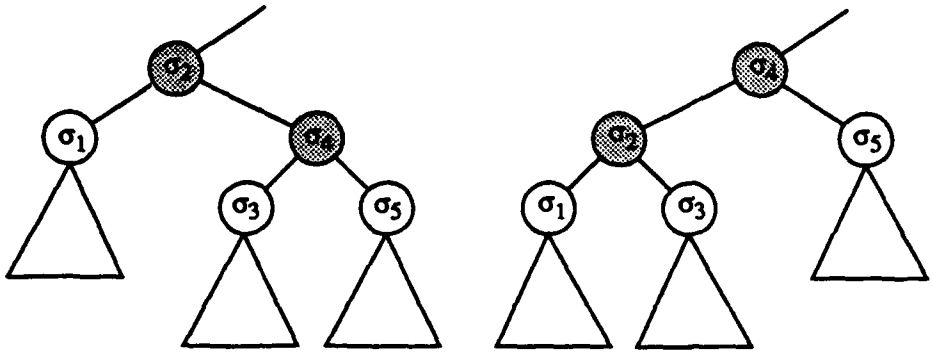
Hereafter, the red-black tree $\mathfrak{I}$ is assumed to be balanced. The rest of this section is devoted to the discussion of the algorithms to perform searches, insertions, and deletions. We have shown in Section 3.2 that the four operations *INSERTCHAIN*, *REMOVECHAIN*, *INSERTPOINT*, and *REMOVEPOINT* are sufficient to generate any planar subdivision. The measures of time complexity ought to be expressed as functions of the form $f(n,k)$, where n is the current size of $\mathfrak{R}$, and k is the size of the chain γ to be inserted or deleted.

3.4.2. Query

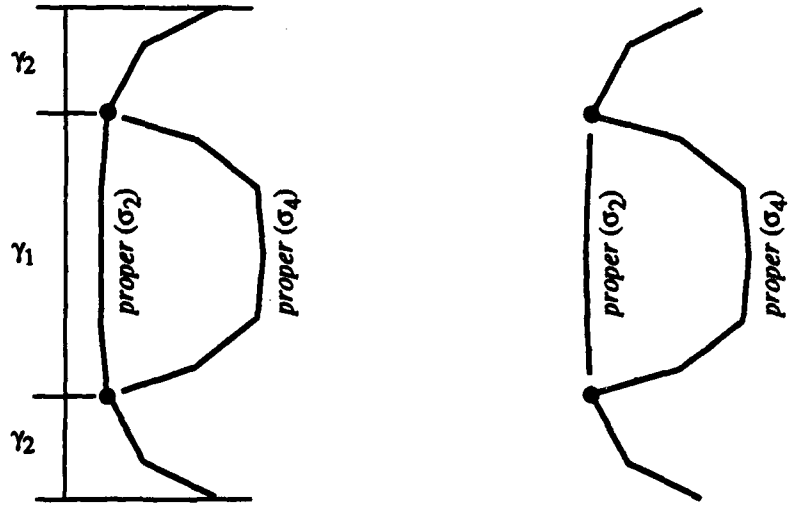
To perform a point location search for a query point q, we use essentially the same method as [35]. The search consists of tracing a path from the root to a leaf χ of $\mathfrak{I}$. At each internal node σ we discriminate q against separator σ. Three cases may occur:

(1) $q \in \sigma$: we return the edge of σ that contains q and stop;

(2) q is to the left of σ: we proceed to the left child of σ;



(a)



(b)

Figure 3.6 Illustration of a (left) rotation. The nodes involved in the update are shaded. **(a)** Separator tree. **(b)** Chains of proper edges.

(3) q is to the right of σ: we proceed to the right child of σ.

Once we reach a leaf χ, we know that q belongs to a region of χ. Since the regions of χ are sorted from bottom to top, such region is determined by searching in the list *regions*(χ). The above technique can be viewed as a "horizontal" binary search in the set of separators of $\mathfrak{R}^*$, followed by a "vertical" binary search in the set of regions of the leaf χ.

Let e be the edge of σ whose vertical span contains $y(q)$. When $e \in$ *proper*(σ), the discrimination of q against σ is a conventional search in *strand1* (σ). When $e \notin$ *proper*(σ) then we use the pair ($t(\sigma)$,$b(\sigma)$): for example, when e is above *proper*(σ), if $t(\sigma)$=*left*, then q is discriminated to the right of σ, and to its left otherwise. (This is a minor variant of the criterion adopted in [35]). The case when e is below *proper*(σ) is treated analogously. This simple analysis confirms that the time spent at each node is $O(\log n)$. We have [35]

Theorem 3.7 The time complexity of the query operation is $O(\log^2 n)$.

3.4.3. Insertion

We shall first show that the effect of operation *INSERTCHAIN* (γ,v_1,v_2,r;r_1,r_2) on the order $<$ of the regions of $\mathfrak{R}$ can be expressed as a syntactical transformation between the strings expressing the order before and after the update. The situation is illustrated in Fig. 3.7.

On the boundary of r there are two distinguished vertices: *HIGH*(r_1) and *LOW*(r_2). Note that *HIGH*(r_1)=*HIGH*(r) if v_2 is on the right chain of the boundary of r (and similarly *LOW*(r_2)=*LOW*(r) if v_1 is on the left chain). Thus, in general, *HIGH*(r_1) is on the left chain of the boundary of r, and *LOW*(r_2) is on the right chain. Using the string notation introduced in Section 3.3, let L and R be the strings corresponding to the regions that respectively precede and follow r in $<$. Thus, the subdivision $\mathfrak{R}^*$ is described by the string LrR.

Let e_1 be the edge of $\mathfrak{R}^*$ on the left boundary of r incident on *HIGH*(r_1) from below, and let χ be the maximal cluster on the left of e_1. In general, this cluster consists of two portions, χ_1 and χ_2 (such that χ_1-χ_2), where χ_2 consists exactly of the regions q' of χ for which $y(LOW(q')) \geq y(HIGH(r_1))$. Thus, we have $L=L'\chi L''$. We now distinguish three cases and define substrings λ_1, λ_2, L_1, and L_2 as follows.

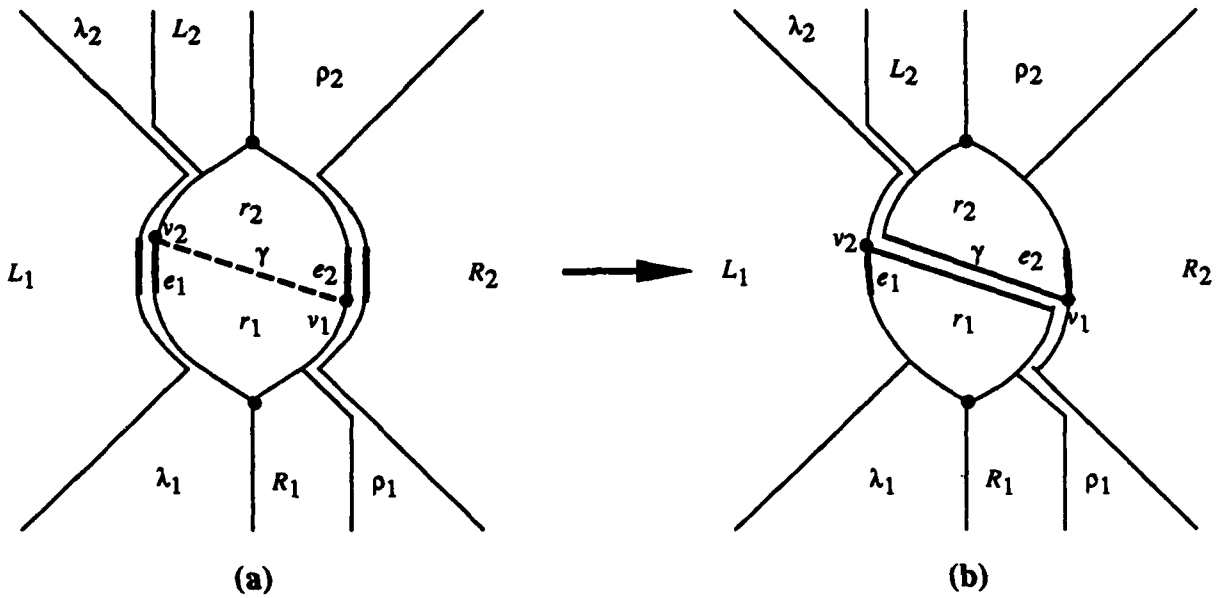


Figure 3.7 (a) Canonical partition of subdivision $\mathfrak{R}^*$ with reference to region r and vertices v_1 and v_2, and (b) the restructured subdivision after the insertion of chain γ between v_1 and v_2.

(1) $\chi_2 \neq \emptyset$. Let $\lambda_1 = \chi_1$, $\lambda_2 = \chi_2$, $L_1 = L'$, $L_2 = L''$, so that $L = L_1\lambda_1 - \lambda_2 L_2$.

(2) $\chi_2 = \emptyset$. Let q be the region preceding r (note q could form a cluster with r). We further distinguish:

(2.1) $y(LOW(q)) \geq y(HIGH(r_1))$. In this case we let $L = L_1\lambda_2 L_2$, where λ_2 is the maximal cluster immediately following χ.

(2.2) $y(LOW(q)) < y(HIGH(r_1))$. In this case we let $L = L_1\lambda_1 - -$, where λ_1 is the rightmost maximal cluster of L (but not necessarily a maximal cluster in $\mathfrak{R}^*$).

The three cases are conveniently summarized by the notation

$$L = L_1\lambda_1 - -\lambda_2 L_2.$$

Analogously, string R can be reformulated as

$$R = R_1\rho_1 - -\rho_2 R_2$$

with straightforward meanings of the symbols.

Thus, in general, for any given region r and choice of v_1 and v_2 on its boundary, we have the following canonical string decomposition of $\mathfrak{R}^*$:

$$L_1\lambda_1\text{--}\lambda_2L_2rR_1\rho_1\text{--}\rho_2R_2.$$

The corresponding partition of the subdivision is illustrated in Fig. 3.7(a). Examples of configurations corresponding to cases (2.1) and (2.2) are shown in Fig. 3.8. Namely, part (a) shows case (2.1) for L and case (1) for R, part (b) shows case (2.2) for both L and R, and part (c) shows case (2.2) for L and case (1) for R.

We now investigate the rearrangement of this order caused by the insertion of chain γ into r. Referring to Fig. 3.7(b), it is immediately observed that the order after the update is as follows:

$$L_1 < \lambda_1 < r_1 < R_1 < \rho_1 < \lambda_2 < L_2 < r_2 < \rho_2 < R_2.$$

To obtain the string description of the updated subdivision we must determine whether any new channel has been created. Any such channel can only arise in correspondence with a new adjacency caused by the update, specifically for the following pairs: (λ_1,r_1), (ρ_1,λ_2), and (r_2,ρ_2). The channel from λ_1 to r_1 exists only if $y(HIGH(\lambda_1)) \le y(LOW(r_1))$, and analogously for the channel from r_2 to ρ_2. Instead, since $y(HIGH(\rho_1)) < y(LOW(\lambda_2))$, the cluster ρ_1–λ_2 always exists. Therefore, the order caused by the insertion of γ is represented by the string

$$L_1\lambda_1\text{--}r_1R_1\rho_1\text{–}\lambda_2L_2r_2\text{--}\rho_2R_2.$$

(In purely syntactic terms, this transformation corresponds to rewriting r as r_2–r_1 and then exchanging substrings $r_1R_1\rho_1$ and $\lambda_2L_2r_2$.) This is summarized as follows:

Theorem 3.8 Let $L_1\lambda_1\text{--}\lambda_2L_2rR_1\rho_1\text{--}\rho_2R_2$ be the string description of the order of $\mathfrak{R}^*$, where L_1, r, and R_2 are nonempty. After operation *INSERTCHAIN* $(\gamma,v_1,v_2,r;r_1,r_2)$ the new order is described by

$$L_1\lambda_1\text{--}r_1R_1\rho_1\text{–}\lambda_2L_2r_2\text{--}\rho_2R_2.$$

The algorithm for the *INSERTCHAIN* $(\gamma,v_1,v_2,r;r_1,r_2)$ operation implements the syntactical transformation of the string description by decomposing the subdivision $\mathfrak{R}$ into its components L_1, λ_1, λ_2, L_2, r, R_1, ρ_1, ρ_2, and R_2, which are subsequently reassembled according to

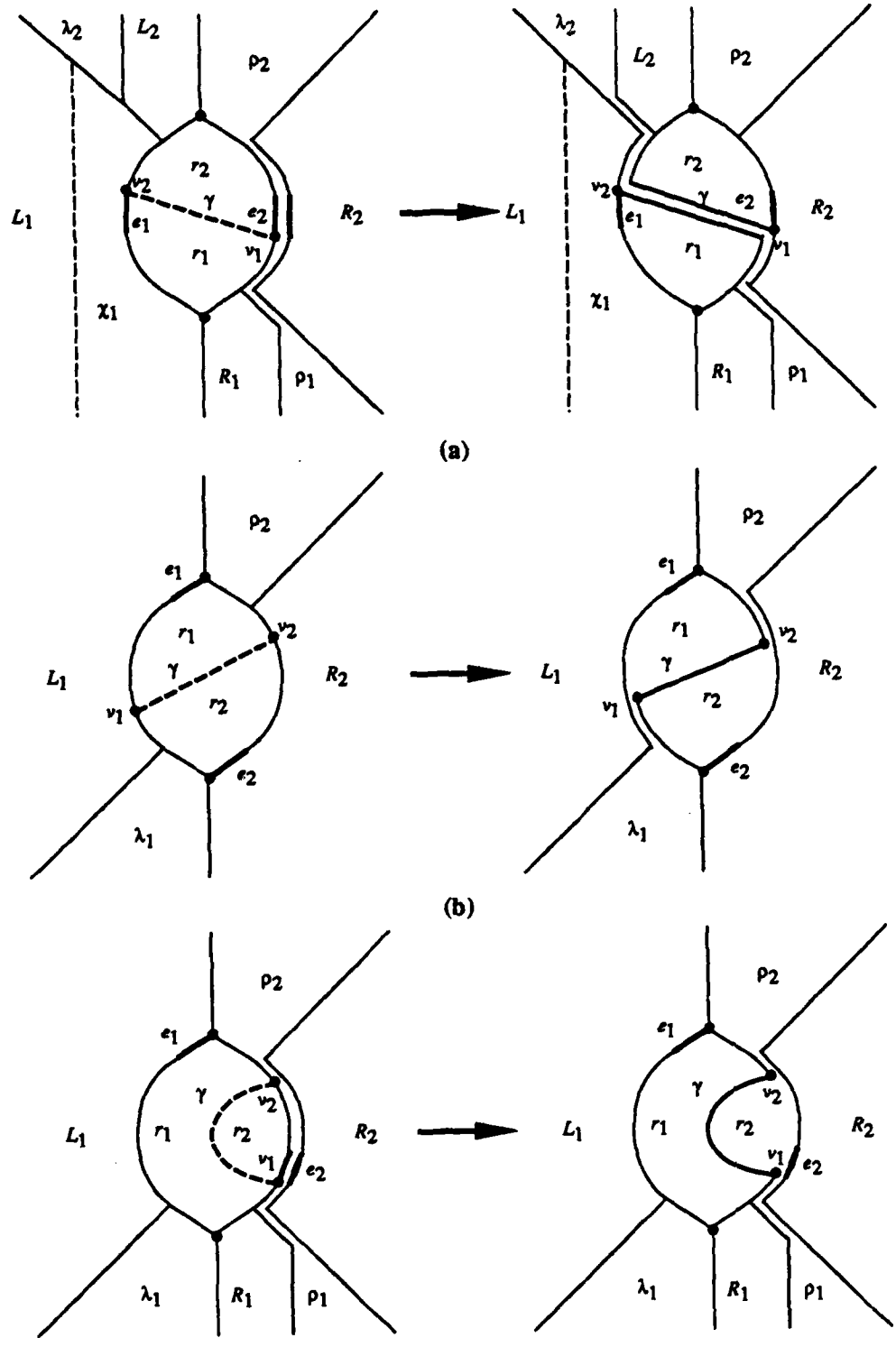


Figure 3.8 Special cases of the structural partition of Fig. 3.7. (a) Case (2.1) for L and case (1) for R. (b) Case (2.2) for both L and R. (c) Case (2.2) for L and case (1) for R.

the new order given by Theorem 3.8.

To formally describe the algorithm, we denote by $\Re(S)$ the partial subdivision associated with a string S of consecutive regions of $\Re$. We can represent $\Re(S)$ with the same data structure described in Section 3.4.1, and we denote with $\Im(S)$ the augmented separator-tree for S. Partial subdivisions can be cut and merged with the same rules as for the decomposition and concatenation of the corresponding strings.

Let $\Re(S_1)$, $\Re(S_2)$, and $\Re(S)$ be partial subdivisions such that $S=S_1S_2$. We show in the following how to *merge* $\Im(S_1)$ and $\Im(S_2)$ into $\Im(S)$, and how to *cut* $\Im(S)$ to produce $\Im(S_1)$ and $\Im(S_2)$. The merge operation needs also the separator σ at the boundary between $\Re(S_1)$ and $\Re(S_2)$, represented by its primary and secondary strands. The cut operation returns the separator σ. These operations can be implemented by means of the following six primitives:

Procedure *MERGE1*$(S_1,\sigma,S_2;S)$

[It merges partial subdivisions $\Re(S_1)$ and $\Re(S_2)$, with $S_1 \rightarrow S_2$; σ is the separator between $\Re(S_1)$ and $\Re(S_2)$.]

(1) Construct a separator-tree $\Im(S)$ for $\Re(S)$, by placing σ at the root, and making $\Im(S_1)$ and $\Im(S_2)$ the left and right subtrees of σ, respectively.
[$\Im(S)$ is a legal separator-tree for $\Re(S)$, but might be unbalanced.]

(2) Rebalance $\Im(S)$ by means of rotations.

Procedure *MERGE2*$(\chi_1,\alpha,\chi_2;\chi)$

[It merges partial subdivisions $\Re(\chi_1)$ and $\Re(\chi_2)$ such that $\chi_1 \uparrow \chi_2$ into $\Re(\chi)$, where $\chi=\chi_1-\chi_2$, and α is the channel between γ_1 and χ_2.]

(1) Separate the two strands of α, and make the secondary strand become a new primary strand.

(2) Splice *regions*(χ_1) and *regions*(χ_2) to form *regions*(χ).

Lemma 3.7 Operations *MERGE1*$(S_1,\sigma,S_2;S)$ and *MERGE2*$(\chi_1,\alpha,\chi_2;\chi)$ have time complexity $O(\log^2 n)$ and $O(\log n)$, respectively.

Proof: The time bound for operation *MERGE2* follows immediately from the properties of concatenable queues. With regard to *MERGE1*, Step 1 consists of forming $\Im(S)$ by joining the primary structures of $\Im(S_1)$ and $\Im(S_2)$ through node σ, which takes $O(1)$ time. Since we use red-black trees, we can rebalance $\Im(S)$ with $O(\log n)$ rotations [55, pp. 52-53]. By Lemma 3.6, each such rotation takes $O(\log n)$ time, so that the total time complexity is $O(\log^2 n)$. □

Procedure *CUT1* $(S,\chi_1,\chi_2;S_1,\sigma,S_2)$

[It cuts partial subdivision $\Re(S)$ into $\Re(S_1)$ with rightmost cluster χ_1 and $\Re(S_2)$ with leftmost cluster χ_2, such that $\chi_1 \rightarrow \chi_2$, and also returns the separator σ between $\Re(S_1)$ and $\Re(S_2)$.]

(1) Find the node σ of $\Im(S)$ that is the lowest common ancestor of leaves χ_1 and χ_2.

(2) Perform a sequence of rotations to bring σ to the root of $\Im(S)$, where after each rotation we rebalance the subtree of σ involved in the rotation, namely, the left subtree for a left rotation and the right subtree for a right rotation (see Fig. 3.9).

(3) Set $\Im(S_1)$ as the left subtree of σ and $\Im(S_2)$ as the right subtree of σ. Return the chain *proper*(σ).

Procedure *CUT2* $(\chi;\chi_1,\alpha,\chi_2)$

[It cuts partial subdivision $\Re(\chi)$ into $\Re(\chi_1)$ and $\Re(\chi_2)$, where $\chi=\chi_1-\chi_2$ and α is the channel between χ_1 and χ_2.]

(1) Join the two previously separated strands of α, so that the rightmost one becomes the secondary strand of the other.

(2) Split *regions*(χ) into *regions*(χ_1) and *regions*(χ_2).

Lemma 3.8 Operations *CUT1* $(S,\chi_1,\chi_2;S_1,\sigma,S_2)$ and *CUT2* $(\chi;\chi_1,\gamma,\chi_2)$ have time complexity $O(\log^2 n)$ and $O(\log n)$, respectively.

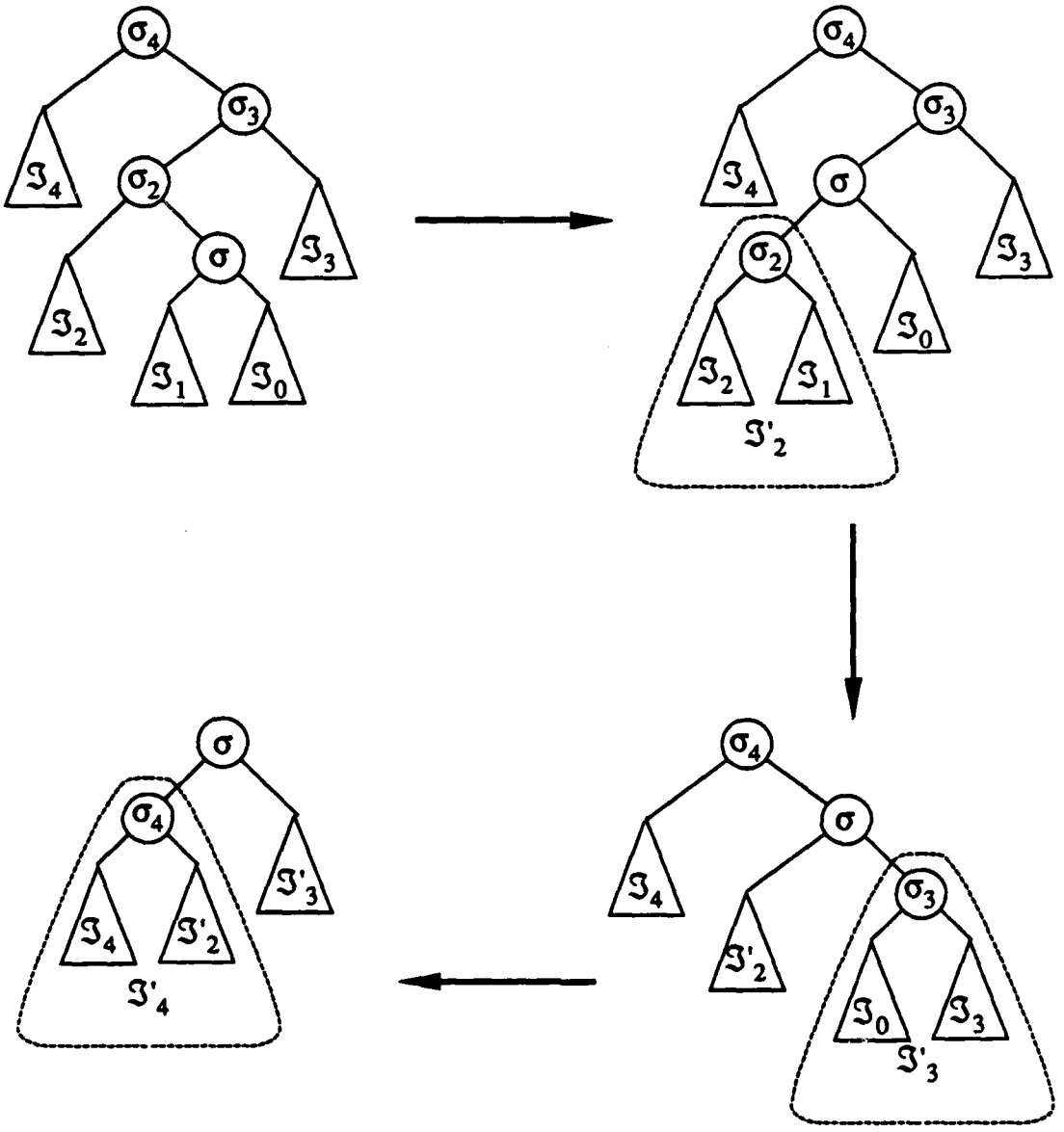


Figure 3.9 Example for Step 2 of Procedure *CUT1*.

Proof: The time bound for operation *CUT2* is immediate. With regard to operation *CUT1*, Step 1 takes $O(height(\Im(S))) = O(\log n)$ time. In Step 2, we perform no more than $height(\Im(S))$ rotations to bring σ to the root. After each such rotation, we have to rebalance a subtree $\Im'$ whose left and right subtrees are already balanced, so that the number of rotations required for rebalancing is proportional to the difference of height of the subtrees of $\Im'$. Such differences form a sequence whose sum is proportional to $height(\Im(S))$ [55, p. 53]. To see this, consider the path of $\Im(S)$ from node σ to the root, and let $X_1, \cdots, X_p$ be the sequence of left subtrees of the nodes of this path whose right child also belongs to the path, in bottom-to-top order. Also, define $X'_1 = X_1$. The i-th left-rotation performed in bringing σ to the root splices trees X_{i+1} and X'_i to form a new tree, which after rebalancing will be denoted X'_{i+1}. We recall that each node μ of a red-black tree is associated with an integer $rank(\mu)$ such that

$$rank(\mu) \le height(\mu) \le 2\, rank(\mu).$$

Also, the rank of a subtree is equal to the rank of its root. From the theory of red-black trees [55, p. 53] we have

(1) $rank(X_i) \le rank(X_{i+1})$;

(2) $rank(X_{i)} \le rank(X'_i) \le rank(X_i) + 1$;

(3) the number t_i of rotations needed to obtain the balanced tree X'_{i+1} is at most

$$t_i \le |\, rank(X_{i+1}) - rank(X'_i)\,|.$$

The total number t of rebalancing rotations is

$$t = \sum_{i=1}^{p-1} t_i = \sum_{i=1}^{p-1} |\, rank(X_{i+1}) - rank(X'_i)\,|.$$

Now, if $rank(X'_i) = rank(X_{i)}$ or $rank(X_i) < rank(X_{i+1})$, we have

$$|\, rank(X_{i+1}) - rank(X'_i)\,| = rank(X_{i+1}) - rank(X'_i) \le rank(X_{i+1}) - rank(X_i).$$

Otherwise (i.e., $rank(X'_i) = rank(X_i) + 1$ and $rank(X_i) = rank(X_{i+1})$), we have

$$|\, rank(X_{i+1}) - rank(X'_i)\,| = |\, rank(X_{i+1}) - rank(X_i) - 1\,| = 1.$$

Hence, in both cases the following inequality holds:

$$|rank(X_{i+1}) - rank(X'_i)| \leq rank(X_{i+1}) - rank(X_i) + 1,$$

from which we obtain

$$t \leq \sum_{i=1}^{p-1} (rank(X_{i+1}) - rank(X_i) + 1) \leq rank(X_p) + (p-1) \leq 2\, height(\Im(S)) = O(\log n).$$

By Lemma 3.6, each rotation takes $O(\log n)$ time, so that the total time complexity is $O(\log^2 n)$. □

Procedure *FINDLEFT* $(e;\chi)$

[It finds the cluster χ to the left of edge e. If e is part of a channel, then χ is the cluster that contains such a channel.]

(1) Perform a point location search for (any point of) edge e. The search will stop at a node σ of $\Im$ that stores (a representative of) e.

(2) If e is a double edge of σ [i.e., e does not belong to a channel], resume the point location search in the left subtree of σ and return the leaf χ where the search terminates. [This corresponds to searching for a point p^- immediately to the left of edge e.]

(3) Otherwise [i.e., e is a single edge of σ and belongs to a channel] resume the point location search in both subtrees of σ. [This corresponds to searching for points p^- and p^+ immediately to the left and right of edge e, respectively.] One of the two searches, say the left one, will terminate in a leaf, while the other search, say the right one, will stop at a node σ' that stores the other representative of e. [Recall that the two nodes storing e are on the path from leaf χ to the root of $\Im$.] We resume the search in the left subtree of σ' and return the leaf χ where the search terminates. [The case where the right search out of σ terminates in a leaf is analogous.]

Procedure *FINDRIGHT* $(e;\chi)$

[It finds the cluster χ to the right of edge e. If e is part of a channel, then χ is the cluster that contains such a channel]

[Analogous to *FINDLEFT*]

Lemma 3.9 Operations *FINDLEFT* $(e;\chi)$ and *FINDRIGHT* $(e;\chi)$ have each time complexity $O(\log^2 n)$.

Proof: Since each edge has two representatives, there are at most two nodes of $\Im$ where we proceed to both children. Hence, we visit a total of $O(\log n)$ nodes, spending $O(\log n)$ time at each node. □

The complete algorithm for operation *INSERTCHAIN* $(\gamma, v_1, v_2, r; r_1, r_2)$ is as follows:

Algorithm *INSERTCHAIN* $(\gamma, v_1, v_2, r; r_1, r_2)$

(1) Find regions q and s immediately preceding and following r, respectively; also, find clusters χ_L and χ_R by means of *FINDLEFT* $(e_1;\chi_L)$ and *FINDRIGHT* $(e_2;\chi_R)$. From these obtain λ_1, λ_2, ρ_1, and ρ_2.

(2) Perform a sequence of *CUT1* and *CUT2* operations to decompose $\Re = \Re(L_1\lambda_1 - \lambda_2 L_2 r R_1 \rho_1 - \rho_2 R_2)$ into $\Re(L_1)$, $\Re(\lambda_1)$, $\Re(\lambda_2)$, $\Re(L_2)$, $\Re(r)$, $\Re(R_1)$, $\Re(\rho_1)$, $\Re(\rho_2)$, and $\Re(R_2)$. The primary and secondary strands returned by each such operation, which form the boundaries of the above partial subdivisions, are collected into a list $\wp$.

(3) Construct the primary and secondary strands of chain γ and add them to $\wp$.

(4) Destroy $\Re(r)$ and create $\Re(r_1)$ and $\Re(r_2)$.

(5) Test for channels $\lambda_1 - r_1$ and $r_2 - \rho_2$, and perform a sequence of *MERGE1* and *MERGE2* operations to construct the updated subdivision $\Re(L_1\lambda_1 - r_1 R_1 \rho_1 - \lambda_2 L_2 r_2 - \rho_2 R_2)$. The separators and channels needed to perform each such merge are obtained by splitting and splicing the appropriate strands of $\wp$.

Theorem 3.9 The time complexity of operation *INSERTCHAIN* $(\gamma, v_1, v_2, r; r_1, r_2)$, where γ consists of k edges, is $O(\log^2 n + k)$.

Proof: In Step 1, finding q and s takes $O(\log n)$ time. In fact, q is either in the cluster of r or in the cluster immediately preceding the one of r, and analogously for s. By Lemma 3.9, finding χ_L and χ_R takes $O(\log^2 n)$ time. The remaining computation of λ_1, λ_2, ρ_1, and ρ_2 can be done in $O(\log n)$ time. By Lemma 3.8, Step 2 takes $O(\log^2 n)$ time. Notice that the list $\wp$ has $O(1)$

elements. Step 3 can be clearly performed in time $O(k)$. Step 4 takes $O(1)$ time since r, r_1 and r_2 are single-region structures. In Step 5, testing for channels λ_1--r_1 and r_2--ρ_2 takes $O(1)$ time. Since the list $\wp$ has $O(1)$ elements, we can construct in $O(\log n)$ time the separators and channels needed for each merge operation of Step 5. By Lemma 3.7 the total time for such merges is $O(\log^2 n)$. □

With regard to the *INSERTPOINT* operation, we locate the edge e in the dictionary, and replace each of the two representatives of e in the data structure with the chain (e_1, v, e_2). This corresponds to performing two insertions into sorted lists, so that we have

Theorem 3.10 The time complexity of operation *INSERTPOINT* $(v, e; e_1, e_2)$ is $O(\log n)$.

A similar argument shows that

Theorem 3.11 The time complexity of operation *MOVEPOINT* $(v; x, y)$ is $O(\log n)$.

3.4.4. Deletion

The transformations involved in a *REMOVECHAIN* operation are exactly the reverse of the ones for the *INSERTCHAIN* operation. We observe that all the updates performed in the latter case are totally reversible, which establishes

Theorem 3.12 The time complexity of operation *REMOVECHAIN* $(\gamma; r)$, where γ consists of k edges, is $O(\log^2 n + k)$.

The same situation arises with respect to the *INSERTPOINT* and *REMOVEPOINT* operations, so that we have

Theorem 3.13 The time complexity of operation *REMOVEPOINT* $(v; e)$ is $O(\log n)$.

Theorem 3.1 stated in Section 3.1 results from the combination of the above Theorems 3.6, 3.7, 3.9, 3.10, 3.12, and 3.13.

CHAPTER 4

DYNAMIC POINT LOCATION: TRAPEZOID METHOD

4.1. Introduction

In this chapter we present a dynamic point location technique with optimal query time. This technique considers convex subdivisions whose vertices lie on a fixed set of N horizontal lines. The supported update operations are insertion/deletion of vertices and edges, and (horizontal) translation of vertices. The method achieves query time $O(\log n + \log N)$, space $O(n \log N)$, and insertion/deletion time $O(\log n \log N)$. Hence, whenever $N = O(n)$, the query time is $O(\log n)$, which is optimal. In addition to its good theoretical performance, this technique is also easy to implement and very efficient in practice. It is based on the *trapezoid method* of Preparata [46], which has been experimentally shown to be the fastest point location method among those with asymptotically optimal query time [12]. It is easily realized that in many significant applications the most frequent operation is the point location query, while updates are more rarely executed. Hence, this technique provides the most efficient solution for such applications.

The rest of the chapter is subdivided into seven sections. In Section 4.2 we formally state the problem. Section 4.3 describes the geometric foundations of the technique. Sections 4.4 and 4.5 present the dynamic data structure and the query algorithm. The update algorithms are discussed in Sections 4.6 and 4.7. Finally, Section 4.8 discusses extensions of the technique and directions of further research.

4.2. Preliminaries

Let $L = \{l_0, l_1, \cdots, l_N\}$ be a set of horizontal lines, in this order from bottom to top. The lines of L partition the plane into horizontal strips, called *elementary slabs*, two of which are (the bottom and top) half-planes. A *slab* is either an elementary slab or the union of two contiguous slabs. We consider a convex subdivision $\mathfrak{R}$ whose n vertices are on the lines of L (see Fig. 4.1).

This implies that all finite edges of $\mathfrak{R}$ occur in the plane strip between l_0 and l_N, and that all infinite edges are horizontal rays.

The following update operations on $\mathfrak{R}$ are defined:

INSERTPOINT $(v,e;e_1,e_2)$:

Split the edge $e=(u,w)$ into two edges $e_1=(u,v)$ and $e_2=(v,w)$, by inserting vertex v, which lies on some line of L.

REMOVEPOINT $(v,e_1,e_2;e)$:

Let v be a vertex of degree 2 whose incident edges, $e_1=(u,v)$ and $e_2=(v,w)$, are on the same straight line. Remove v and replace e_1 and e_2 with edge $e=(u,w)$.

INSERTSEGMENT $(e,v_1,v_2,r;r_1,r_2)$:

Add the edge $e=(v_1,v_2)$, with $y(v_1)\leq y(v_2)$, to $\mathfrak{R}$ inside region r of $\mathfrak{R}$, which is decomposed into regions r_1 and r_2, with r_1 and r_2, respectively, to the left and to the right of e, directed from v_1 to v_2.

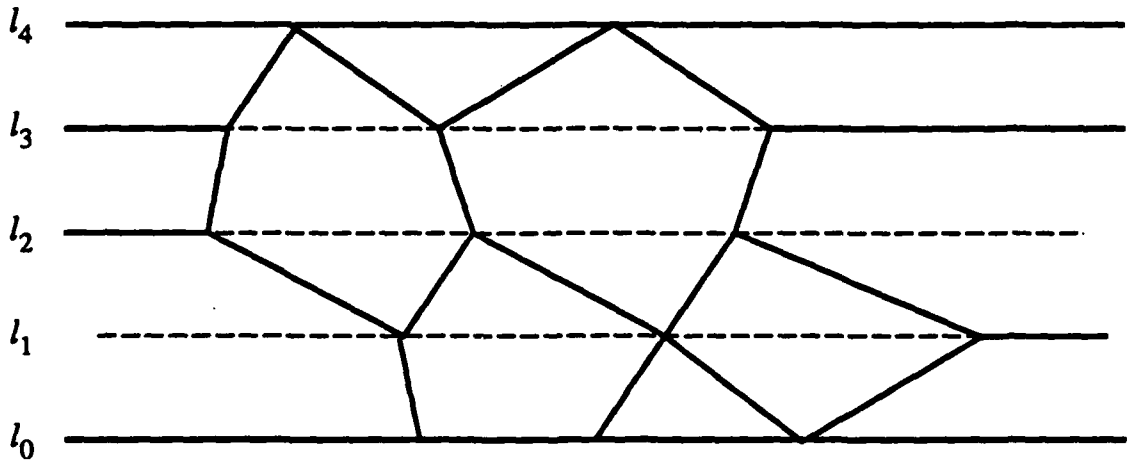


Figure 4.1 Example of convex subdivision with vertices on the set L of horizontal lines.

REMOVESEGMENT $(e,v_1,v_2,r_1,r_2;r)$:

Remove edge e and merge the regions r_1 and r_2 formerly on the two sides of e into region r. [The operation is allowed only if the subdivision $\Re'$ so obtained is convex.]

MOVEPOINT $(v;x)$:

Horizontally translate a vertex v from its present location to abscissa x. [The operation is allowed only if the subdivision $\Re'$ so obtained is convex and topologically equivalent to $\Re$.]

The following theorem demonstrates the adequacy of the above repertory:

Theorem 4.1 An arbitrary convex subdivision $\Re$ with n vertices can be assembled starting from the empty subdivision, and disassembled to obtain the empty subdivision, by a sequence of $O(n)$ operations from the above repertory.

Proof: It suffices to constructively demonstrate the assembly of $\Re$, since its disassembly is obtainable by replacing each operation with its inverse and by reversing their order. First, we recall the definition of the acyclic relation "$\ll$" between the regions of $\Re$ (see Section 3.3 of Chapter 3). Given r_1 and r_2 in $\Re$, r_1 is *left-adjacent to* r_2 ($r_1 \ll r_2$) if their boundaries share an edge e, and r_1 and r_2 are respectively to the left and to the right of e. (If r_1 and r_2 share a horizontal edge, then $r_1 \ll r_2$ if r_1 is below e.) Consider an order $(r_1,r_2,\cdots,r_f)$ of the regions of $\Re$ which is consistent with "$\ll$". We build $\Re$ by adding its regions one by one, according to the chosen order. At the generic step of the construction, after adding region r_i we call $\Re_i$ the resulting subdivision. (Obviously, $\Re_f=\Re$.) In $\Re_i$ there is a monotone polygonal line σ_i, which forms the boundary of the set of regions $\{r_1,r_2,\cdots,r_i\}$ (refer to Fig. 4.2). Consider the (external) angles to the right of σ_i at each nonextreme vertex of σ_i: each angle $>\pi$ is bisected by a horizontal ray to ensure the property of convexity for $\Re_i$. Consider now region r_{i+1}: obviously, the left chain of r_{i+1} is already present in $\Re_i$, and let the right chain of r_{i+1} be $v'_1,\cdots,v'_s$. By means of *INSERTSEGMENT* we insert the edge $e=(LOW(r_{i+1}),HIGH(r_{i+1}))$. (This operation is performed as many times as there are regions.) Next, with the sequence *INSERTPOINT*, *INSERTSEGMENT*, *MOVEPOINT* we insert vertex v'_2 and edge (v'_1,v'_2) into their final position, and partition the region to the right by a horizontal ray to preserve convexity. This process

is performed for $v'_3, \cdots, v'_{s-1}$. (This sequence of operations is performed at most once for each vertex of $\mathfrak{R}$.) Finally, by *REMOVESEGMENT* we remove any horizontal ray originating at v'_1 and v'_s if the corresponding external angles have become $\leq \pi$. (This operation is performed at most twice per region.) The resulting subdivision is $\mathfrak{R}_{i+1}$ and exhibits the same invariant as regards the region to the right of σ_{i+1}. The comments about the frequency of each executed operation complete the proof. □

Let r be a convex polygon. Polygon r is said to be *1-supported* if it has an edge e such that any horizontal line intersecting r intersects also e, i.e., r contains the edge (*LOW* (r),*HIGH* (r)) (see Fig. 4.3(a)). Polygon r is said *2-supported* if it has two edges, e_1 and e_2, such that any horizontal line intersecting r intersects at least one of e_1 and e_2 (see Fig. 4.3(b)). (Notice that a 1-supported polygon is also 2-supported.) A convex subdivision is said to be 1-supported (respectively 2-supported) if all its regions are 1-supported (respectively 2-supported).

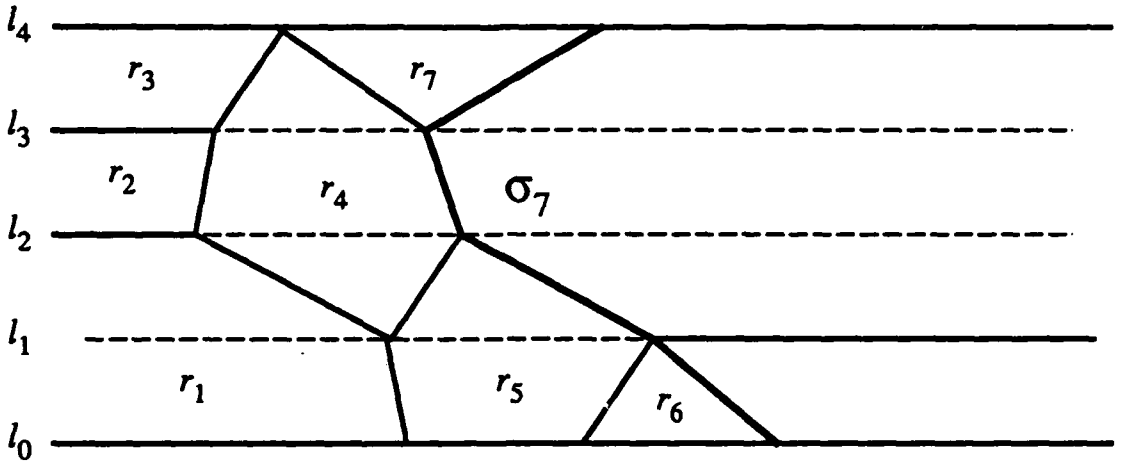
An important property of i-supportedness ($i = 1,2$) is expressed by the following straightforward lemma:

Lemma 4.1 Let convex polygon r be cut by a horizontal line into two convex polygons r_1 and r_2. We have:

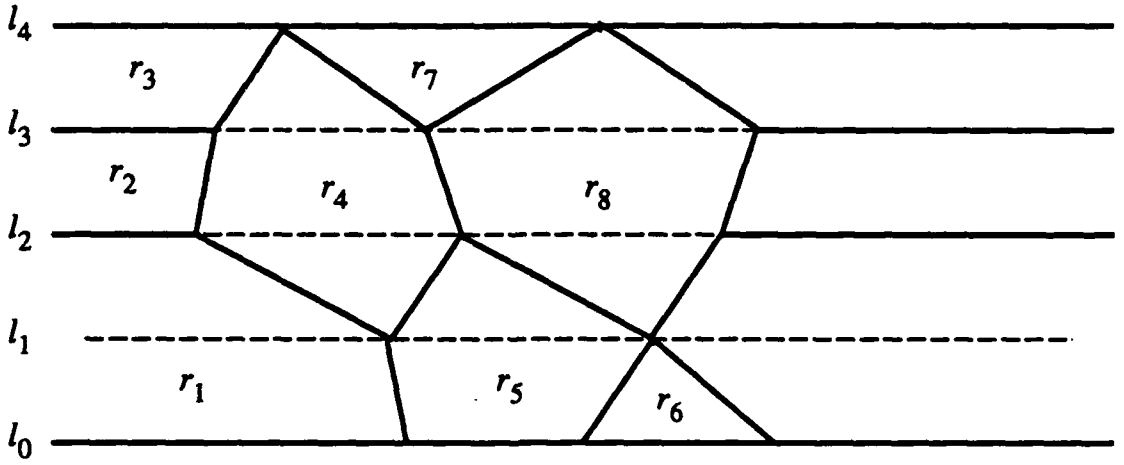
(*i*) If r is 1-supported, then both r_1 and r_2 are 1-supported.

(*ii*) If r is 2-supported, then at least one of r_1 and r_2 is 1-supported.

In the description of the data structure for dynamic point location we will assume that the subdivision $\mathfrak{R}$ is 2-supported. As shown in the following, this is not restrictive. Let $\mathfrak{R}^*$ be the subdivision obtained from $\mathfrak{R}$ by adding to each region r which is not 1-supported the edge from *LOW* (r) to *HIGH* (r). (Notice that $\mathfrak{R}$ may have degenerate regions of measure zero.) Clearly $\mathfrak{R}^*$ is 1-supported. The answer to a point location query in $\mathfrak{R}^*$ can be immediately converted to an answer in $\mathfrak{R}$. Regarding updates, we have



(a)



(b)

Figure 4.2 Example for the proof of Theorem 4.1. **(a)** Subdivision $\mathfrak{R}_7$. **(b)** Subdivision $\mathfrak{R}_8$.

Lemma 4.2 For each update operation performed on $\mathfrak{R}$ there exists a sequence of at most four operations which update $\mathfrak{R}^*$ so that after each intermediate operation the resulting subdivision is 2-supported (possibly 1-supported).

Proof: Consider operation *INSERTSEGMENT* $(e,v_1,v_2,r;r_1,r_2)$ performed on $\mathfrak{R}$. The sequence of operations in $\mathfrak{R}^*$ is as follows (see Fig. 4.4):

(1) insert edge $s_1 = (LOW(r_1), HIGH(r_1))$;

(2) insert edge $s_2 = (LOW(r_2), HIGH(r_2))$;

(3) remove edge $s = (LOW(r), HIGH(r))$, unless $s = s_1$ or $s = s_2$;

(4) insert edge e.

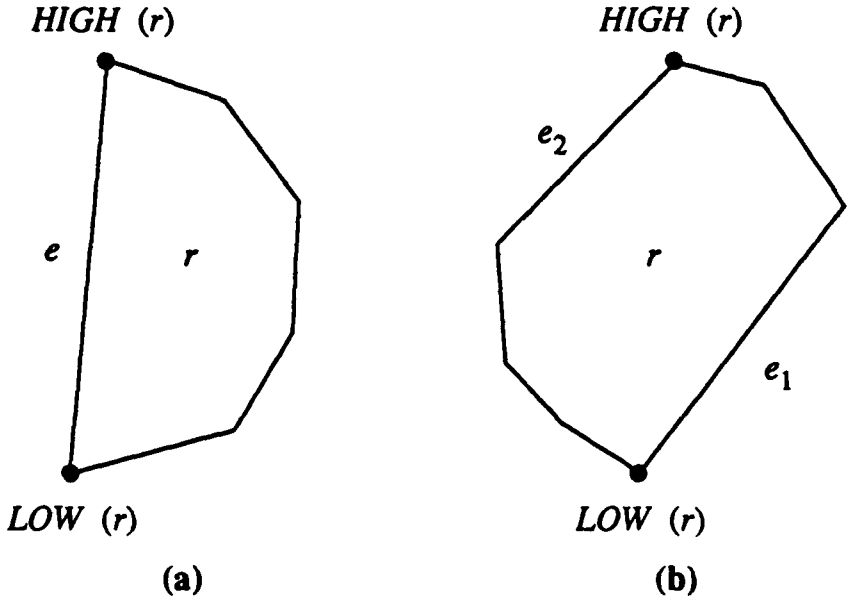


Figure 4.3 (a) A 1-supported polygon. (b) A 2-supported polygon.

(We assume that an insertion is not performed if the edge to be inserted already exists.) Similar arguments hold for operations *REMOVESEGMENT*.

Regarding operation *INSERTPOINT* $(v,e:e_1,e_2)$. let r_1 and r_2 be the regions to the left and right of r, respectively. The sequence of operations in $\mathfrak{R}^*$ is as follows:

(1) insert vertex v;

(2) insert edge $s_1 = (LOW(r_1), HIGH(r_1))$;

(3) insert edge $s_2 = (LOW(r_2), HIGH(r_2))$.

Again, the case of operation *REMOVEPOINT* is similar.

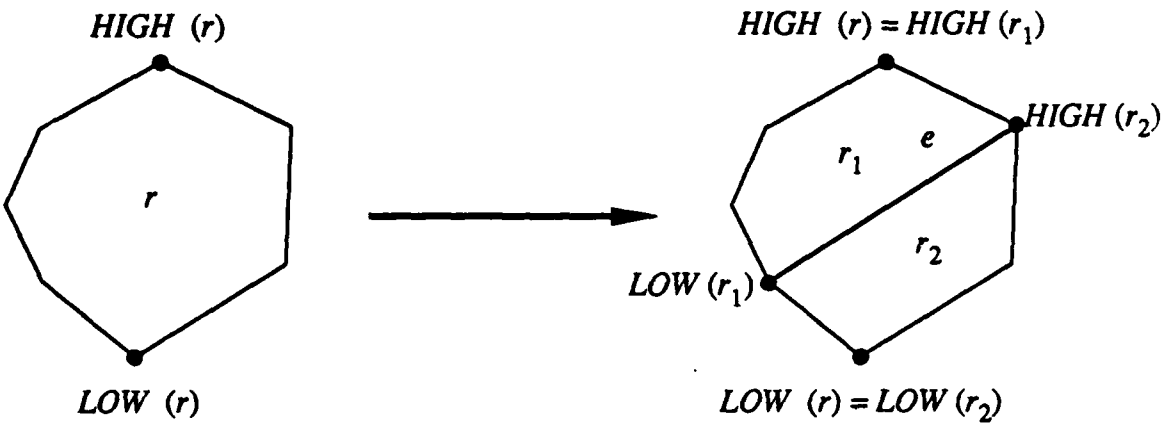
Finally, operation *MOVEPOINT* does not affect the supportedness of any region of $\mathfrak{R}$, so that no further operation is required in $\mathfrak{R}^*$. □

4.3. Trapezoids

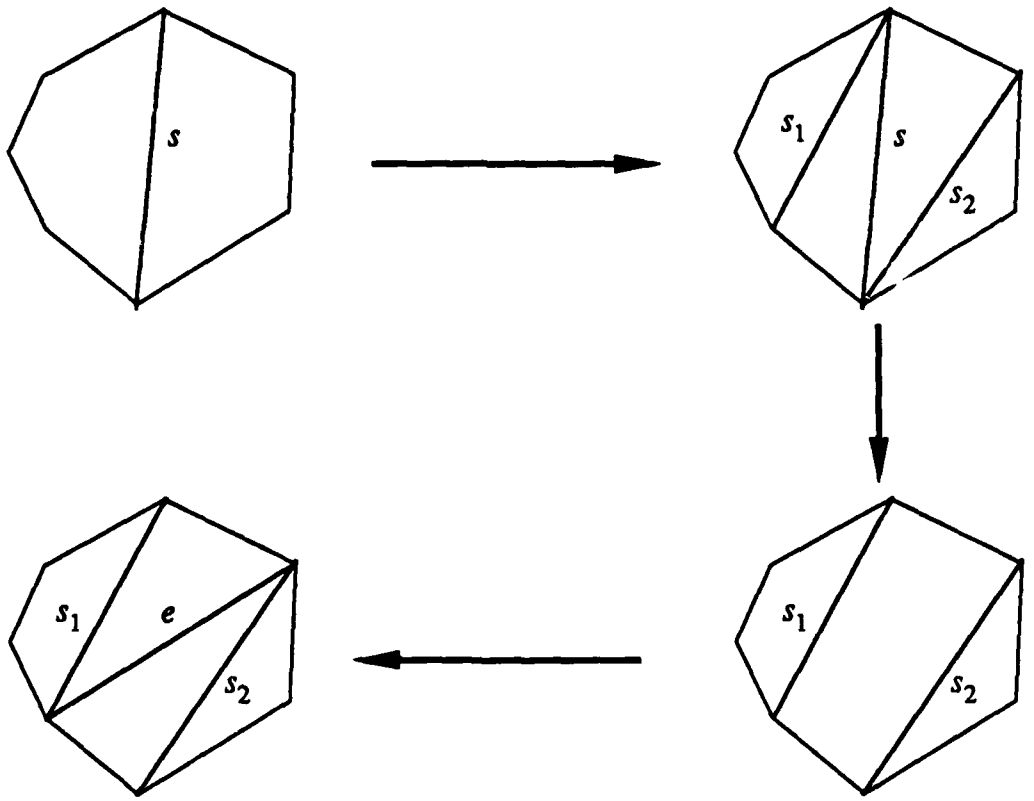
Throughout this section we denote by $\mathfrak{R}$ a 2-supported convex subdivision whose n vertices are on the lines of L. We show that the *trapezoid* method for point location [46] can be modified to perform efficiently updates on $\mathfrak{R}$.

The trapezoid method is based on a recursive decomposition of the subdivision $\mathfrak{R}$ into components, called trapezoids. A *trapezoid* τ is a convex subdivision whose external boundary is a quadrilateral with two (not necessarily bounded) horizontal sides, which belong to the lines of L. We denote the four sides of τ by $LEFT(\tau)$, $RIGHT(\tau)$, $BOT(\tau)$, and $TOP(\tau)$. Let $BOT(\tau)$ be on line l_i, and $TOP(\tau)$ on l_j. The *median line* of τ, denoted $MEDIAN(\tau)$, is the line l_k with $k = \lfloor (i+j)/2 \rfloor$. The subdivision $\mathfrak{R}$ is the trapezoid with $BOT(\mathfrak{R}) = l_0$, $TOP(\mathfrak{R}) = l_N$, and having $LEFT(\tau)$ and $RIGHT(\tau)$ at infinity.

A *spanning edge* of trapezoid τ is an edge of τ with endpoints on the top and bottom sides of τ. A spanning edge s partitions trapezoid τ into two trapezoids, denoted τ_L and τ_R, with $s = RIGHT(\tau_L) = LEFT(\tau_R)$. This is called a *vertical cut* (see Fig. 4.5(a)). If trapezoid τ does not have spanning edges, we decompose it by means of a horizontal line through its median line, and obtain trapezoids τ_B and τ_T, with $MEDIAN(\tau) = TOP(\tau_B) = BOT(\tau_T)$. This is called a *horizontal cut* (see Fig. 4.5(b)). A trapezoid can always be decomposed unless it is empty. We



(a)



(b)

Figure 4.4 Example for the proof of Lemma 4.2. **(a)** Operation *INSERTSEGMENT* on $\Re$. **(b)** Sequence of operations on $\Re^*$.

assume that a vertical cut takes precedence over a horizontal cut, so that the decomposition process is uniquely determined to within the specification of the sequence of consecutive vertical cuts of the same trapezoid.

The above decomposition process can be conveniently represented by means of a string notation. Namely, a vertical cut can be represented by the transformation $\tau \rightarrow \tau_L\, s\, \tau_R$, where s is the spanning edge; and a horizontal cut by the transformation $\tau \rightarrow (\tau_B)\, l\, (\tau_T)$, where l is the median line of τ. Finally, an empty trapezoid τ is eliminated using the transformation $\tau \rightarrow \epsilon$, where ϵ is the null string. Hence, the recursive decomposition of a trapezoid τ can be viewed as a sequence of rewritings using one of the above rules. In this framework, trapezoids correspond to nonterminal symbols (variables), while spanning edges and lines correspond to terminal symbols. The initial string is the nonterminal symbol τ. The final string is a sequence of spanning

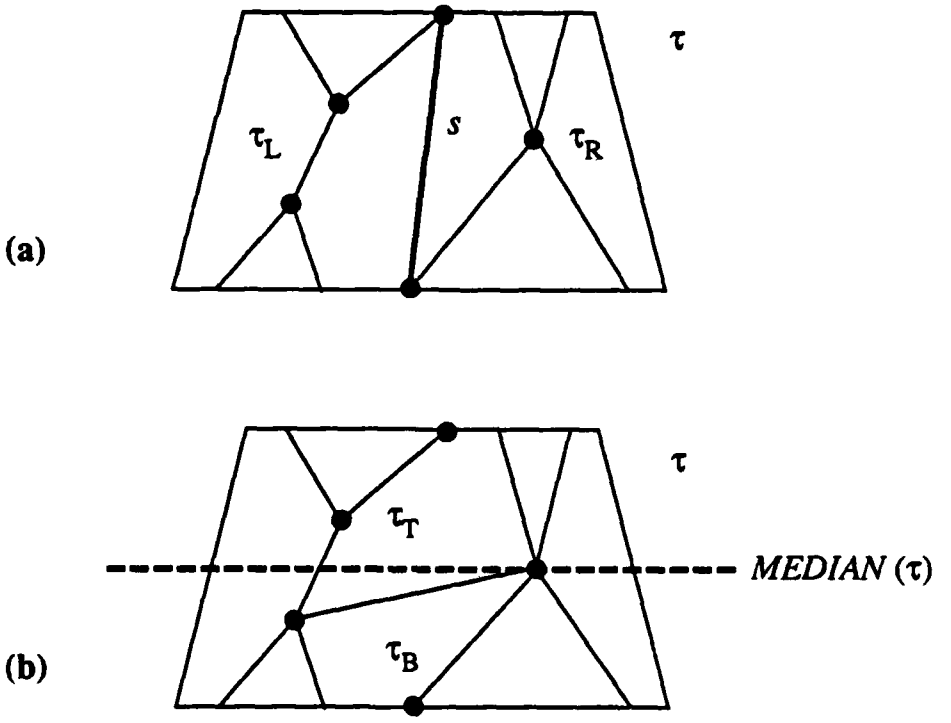


Figure 4.5 (a) Example of vertical cut. (b) Example of horizontal cut.

edges, lines, and parentheses. For example, the recursive decomposition of the trapezoid shown in Fig. 4.6 is described by the string

$$((e_1\, e_3)\, l_1\, (e_2\, e_4\, e_7)\, e_8\, e_9)\, l_2\, (e_2\, (e_4)\, l_3\, (e_5\, e_6)\, e_{10}\, e_{11}).$$

Notice that the parentheses have been introduced in the horizontal decomposition to eliminate ambiguities in the parsing of such strings.

The *canonical decomposition* of a trapezoid τ with spanning edges is defined by (see Fig. 4.7):

$$\tau \rightarrow \tau_0 \sigma_1 \tau_1 \sigma_2 \cdots \sigma_k \tau_k$$

where

(*i*) each τ_i, $i = 1, \cdots, k-1$, is a nonempty trapezoid without spanning edges;

(*ii*) each of τ_0 and τ_k either is empty or is a nonempty trapezoid without spanning edges; and

(*iii*) each σ_j, $j = 1, \cdots, k$, is a maximal sequence of spanning edges that delimit empty trapezoids.

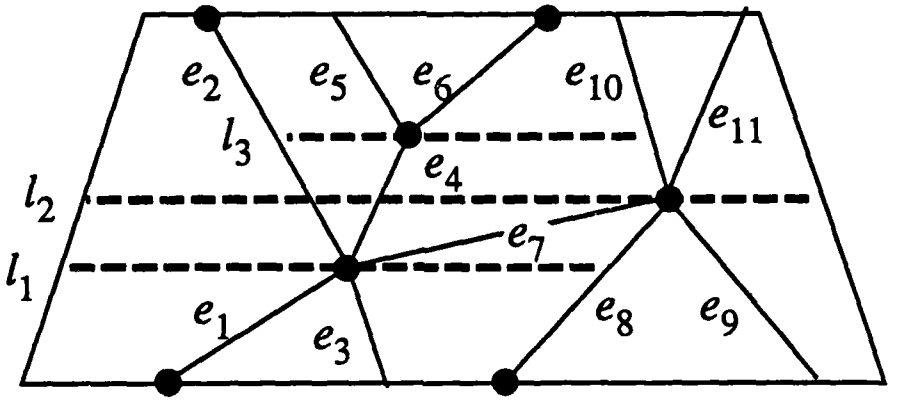


Figure 4.6 Recursive decomposition of a trapezoid.

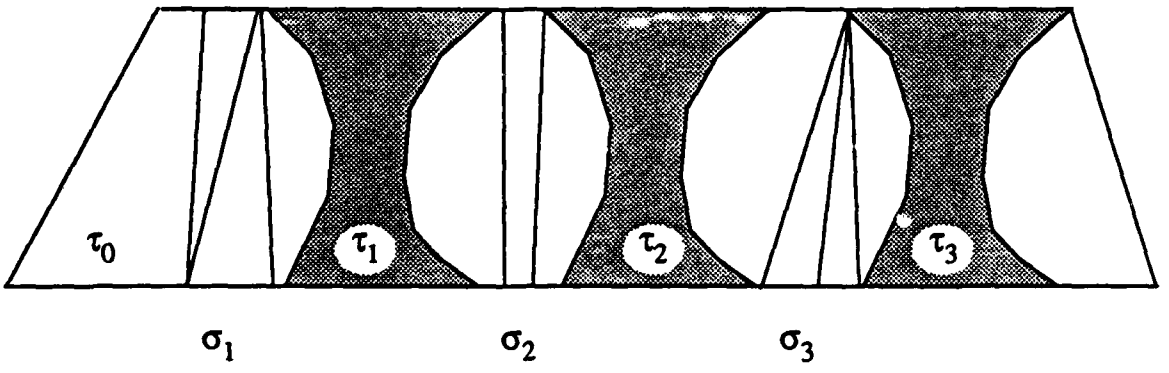


Figure 4.7 Canonical decomposition of a trapezoid.

4.4. Data Structure

The data structure for point location makes use of the so-called *biased binary tree* [2], which is a dynamic binary search tree that represents a sorted list of weighted items. Specifically, the i-th item in the sorted list, with weight w_i, is represented by the i-th leaf of the tree. Let $w = \sum_{i=1}^{k} w_i$ be the total weight of the items in the tree. Some important properties of biased binary trees, which are relevant to this work, are listed below:

(1) The depth of the i-th leaf is at most $\log(w/w_i) + 2$.

(2) Dynamic operations such as insertion/deletion of an item, change of weight of an item, and split/splice of biased trees, can be performed in time $O(\log w)$. (For the splice operation, w is the total weight of the trees to be spliced).

The point location data structure consists of a *main component* and of an *auxiliary component*. The main component is a binary tree $\mathfrak{I}(\mathfrak{R})$, called *trapezoid tree*, which represents the recursive decomposition of the subdivision $\mathfrak{R}$.

The nodes of the trapezoid tree $\mathfrak{I}(\mathfrak{R})$ are of three types (see Fig. 4.8):

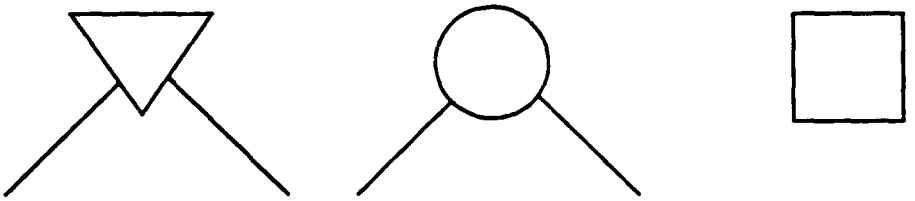


Figure 4.8 Nodes of a trapezoid tree: (a) ∇-node; (b) *O*-node; (c) ε-node.

(1) ∇-*node:* each such node is associated with a nonempty trapezoid τ which is partitioned by a horizontal cut; it stores the line *MEDIAN* (τ).

(2) *O-node:* each such node is associated with a nonempty trapezoid τ which is partitioned by vertical cuts; it stores a maximal sequence σ of spanning edges in the canonical decomposition of τ, represented by a balanced tree.

(3) ε-*node:* each such node is associated with an empty trapezoid, and is a leaf of the trapezoid tree.

In addition, each node μ stores an integer *weight*(μ), which denotes the number of vertices in the interior of the trapezoid associated with μ.

Each node of the trapezoid tree is uniquely associated with a trapezoid, so that we will sometimes use the same name for a node of the tree and for the associated trapezoid.

The trapezoid tree $\Im(\tau)$ for a nonempty trapezoid τ is recursively defined as follows (see Fig. 4.9):

(1) If τ has no spanning edges, the root of $\Im(\tau)$ is a ∇-node, and the left and right subtrees of $\Im(\tau)$ are trapezoid trees for τ_B and τ_T, respectively.

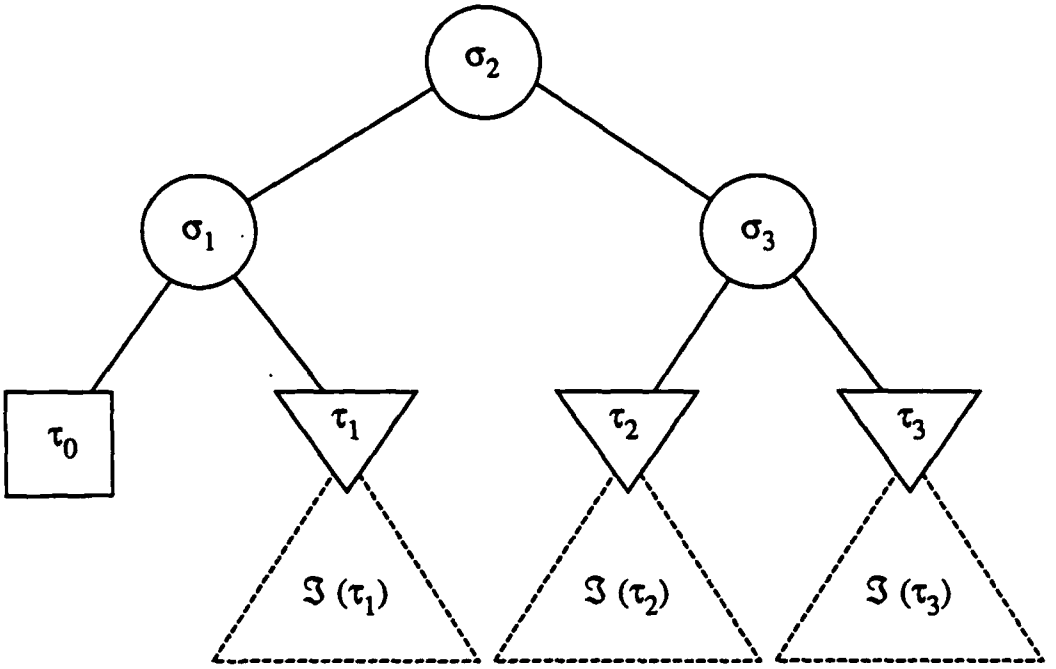


Figure 4.9 Trapezoid tree for the trapezoid of Fig. 4.7.

(2) Otherwise (τ has spanning edges), we consider the canonical decomposition of τ, $\tau_0\sigma_1\tau_1\sigma_2 \cdots \sigma_k\tau_k$, and define $T(\tau)$ as a biased binary tree for the items $\tau_0, \cdots, \tau_k$. Tree $T(\tau)$ is called the *decomposition tree* of τ. The i-th leaf of $T(\tau)$ is a ∇-node for trapezoid τ_i. For the special case when $i = 0$ or $i = k$ and τ_i is empty, leaf i is a ε-node, and has nominal unit weight. The i-th internal node of T, denoted μ_i, stores the sequence σ_i of spanning edges, and is associated with the trapezoid formed by the union of the trapezoids associated with the leaves of the subtree of $T(\tau)$ rooted at μ_i. Each node of $T(\tau)$ is also equipped with pointers *pred* and *succ*, which form a doubly-linked list of the nodes of $T(\tau)$ in their left-to-right order. Finally, the trapezoid tree $\Im(\tau)$ is obtained by replacing each leaf τ_i of $T(\tau)$ with the trapezoid tree $\Im(\tau_i)$.

It is interesting to observe that each edge e of the subdivision $\mathfrak{R}$ may be stored in the sequences of spanning edges of several O-nodes of tree $\mathfrak{I}(\mathfrak{R})$. Each such representative is called a *fragment* of edge e. The decomposition of e into fragments is performed according to the well-known segment-tree scheme [47, pp. 13-15], so that the number of fragments of each edge is at most $2\log N - 1$.

Theorem 4.2 The space requirement of the trapezoid tree $\mathfrak{I}(\mathfrak{R})$ is $O(n \log N)$.

Proof: Let the *order* of a ∇-node μ be the number of ∇-nodes on the path from μ to the root of $\mathfrak{I}(\mathfrak{R})$, including μ. Because of the median line decomposition rule, the order of μ is at most $\log N$. Also, for any $p \leq \log N$, there are at most n nodes of order p because each such node corresponds to a trapezoid with at least one vertex. Hence, the total number of ∇-nodes in $\mathfrak{I}(\mathfrak{R})$ is at most $n \log N$. Regarding O-nodes, we observe that each edge is decomposed according to the well-known segment-tree scheme, so that the total space requirement for the O-nodes and their sequences of spanning edges is $O(n \log N)$. Finally, we can charge at most two ε-nodes to each ∇-node, so that the total space for the trapezoid tree $\mathfrak{I}(\mathfrak{R})$ is $O(n \log N)$. □

Theorem 4.3 The depth of the trapezoid tree $\mathfrak{I}(\mathfrak{R})$ is $O(\log n + \log N)$.

Proof: Without significant loss of generality, assume that N is a power of 2. Let $height(\tau)$ be the number of elementary slabs spanned by a trapezoid τ. Clearly, $height(\tau) \leq N$. We show that for a ∇-node of $\mathfrak{I}(\mathfrak{R})$ with associated trapezoid τ, the depth of $\mathfrak{I}(\tau)$ is at most $\log weight(\tau) + 3 \log height(\tau) + 1$. (For simplicity of notation, we assume that $\log 0 = 1$.) The proof is by induction on $height(\tau)$. The base of the induction, $height(\tau) = 2$, is immediate, since the left and right subtrees of $\mathfrak{I}(\tau)$ consist each of a O-node and two ε-nodes. For the inductive step, define $w = weight(\tau)$ and $h = height(\tau)$. Consider the deeper of the two subtrees of τ, say, the one associated with trapezoid τ_B. Let τ_i be a leaf of the decomposition tree $T(\tau_B)$, and define $w_i = weight(\tau_i)$ and $h_i = height(\tau_i)$. Since $T(\tau_B)$ is a biased binary tree, leaf τ_i has depth bounded by $\log(w/w_i) + 2$. By the inductive hypothesis, the depth of the trapezoid tree $\mathfrak{I}(\tau_i)$ is at most $\log w_i + 3 \log h_i + 1$. Hence, the total depth of the trapezoid tree $\mathfrak{I}(\tau)$ is at most

$$(\log (w/w_i)+2)+1+(\log w_i+3\log h_i+1).$$

Since $h_i = h/2$, this is exactly $\log w + 3\log h + 1$. □

The auxiliary component of the data structure, called *dictionary*, contains a record for each vertex, region, and edge of $\Re$. The record for a vertex v stores the name of v and its coordinates. The record for a region r stores the name of r, pointers to the records of vertices *LOW* (r) and *HIGH* (r), and pointers to two balanced search trees, *lchain*(r) and *rchain*(r), respectively, representing the edges of the left and right chain of r in bottom-to-top order. The record for an edge e stores the name of e, pointers to the records of the endpoints of e, denoted *LOW* (e) and *HIGH* (e), and pointers *lface*(e) and *rface*(e) to the representatives of e in the trees *rchain*(r_1) and *lchain*(r_2) such that r_1 and r_2 are the regions on the left and right sides of e, respectively. Clearly, the dictionary takes $O(n)$ space. The dynamic maintenance of the dictionary in the various update operations can be easily performed in $O(\log n)$ time, and will not be further described.

4.5. Query

The query algorithm is essentially the same as in the static case [46]. It consists of tracing a path in $\Im(\Re)$. The actions taken at the current node μ are as follows:

(1) If μ is a ε-node, we stop.

(2) If μ is a ∇-node, we discriminate the query point q with respect to the median line *MEDIAN* (τ) of the trapezoid τ associated with μ. We proceed to the left or right child of μ depending upon whether q is below or above *MEDIAN* (τ). (If q is on *MEDIAN* (τ) we proceed to the left child.)

(3) If μ is a O-node, we discriminate q with respect to the sequence of spanning edges associated with μ, $\sigma(\mu) = s_1, \cdots, s_p$. If q lies to the left of edge s_1, we set $s_R := s_1$ and proceed to the left child of μ. If q lies to the right of s_p, we set $s_L := s_p$ and proceed to the right child of μ. If q lies between two edges of $\sigma(\mu)$, say s_i and s_{i+1}, we set $s_L := s_i$ and $s_R := s_{i+1}$, and stop.

The region r reported is the one which lies on the right side of edge s_L and on the left side of edge s_R.

The time spent is $O(1)$ for the intermediate nodes on the search path, and $O(\log n)$ for the last node. The final computation of r from s_L and s_R can be done by accessing the representatives of s_L and s_R in the trees *lchain*(r) and *rchain*(r) pointed by *rface*(s_L) and *lface*(s_R), and walking up to the root of such trees, which takes $O(\log n)$ time. Since the height of tree $\mathfrak{I}$ ($\mathfrak{R}$) is logarithmic, we conclude

Theorem 4.4 The complexity of a point location query is $O(\log n + \log N)$.

4.6. Insertion of Edges

First, we consider operation *INSERTSEGMENT* $(e, v_1, v_2, r; r_1, r_2)$. The insertion algorithm is described by means of a recursive procedure, whose actions, depending on the type of the current node μ, are as follows.

1. If μ is a O-node associated with the string $\sigma(\mu) = s_1 \cdots s_p$, we locate e in $\sigma(\mu)$.

1.1 If e is to the left of s_1, we recursively call the procedure on the left child of μ.

1.2 If e is to the right of s_p, we recursively call the procedure on the right child of μ.

1.3 Otherwise, e lies between two edges of $\sigma(\mu)$, say s_i and s_{i+1}. In this case, we insert e into $\sigma(\mu)$ between s_i and s_{i+1}, corresponding to the string transformation

$$s_1 \cdots s_i\, s_{i+1} \cdots s_p \rightarrow s_1 \cdots s_i\, e\, s_{i+1} \cdots s_p.$$

Note that this case can only occur when e spans the slab of $\sigma(\mu)$, for, in the interior of the corresponding region of the plane, there are no vertices of $\mathfrak{R}$.

2. If μ is a ε-node, we know that it is either the leftmost or rightmost leaf of a decomposition tree. In the former case, edge e is added to the left of the sequence of spanning edges of the O-node *next*(μ) immediately following μ. In the latter case, edge e is added to the right of the sequence of spanning edges of the O-node *pred*(μ) immediately preceding μ.

3. If μ is a ∇-node associated with trapezoid τ, we determine first whether e spans τ or not.

3.1 If e does not span τ, then if *LOW* (e) is below *MEDIAN* (τ) we recursively call the procedure on the left child of μ, and if *HIGH* (e) is above *MEDIAN* (τ) we recursively call the procedure on the right child of μ. (Note: here is where the edge may be subjected to the segment-tree fragmentation.)

3.2 If e spans τ, we determine the leftmost and rightmost regions of τ, i.e., the region r_L to the right of edge *LEFT* (τ) and the region r_R to the left of *RIGHT* (τ). Such regions can be computed by observing that *LEFT* (τ) is the last spanning edge in the sequence $\sigma(pred(\mu))$, and *RIGHT* (τ) is the first spanning edge in the sequence $\sigma(next(\mu))$. We have three further subcases:

3.2.1 If $r = r_L$, i.e., edge e is to the left of each vertex of τ, then edge e is added to the right of the sequence of spanning edges of the O-node $pred(\mu)$ immediately preceding μ, and no further action is required. (If $pred(\mu) = nil$, a new O-node and a new ε-node are created, and the decomposition tree containing leaf μ is rebalanced.) This case corresponds to the syntactic transformation $\tau \rightarrow e\ \tau$.

3.2.2 If $r = r_R$, i.e., e is to the right of each vertex of τ, then edge e is added to the left of the sequence of spanning edges of the O-node $next(\mu)$ immediately following μ, and no further action is required. (If $next(\mu) = nil$, a new O-node and a new ε-node are created, and the decomposition tree containing leaf μ is rebalanced.) This case corresponds to the syntactic transformation $\tau \rightarrow \tau\, e$, and is symmetric to case 3.2.1.

3.2.3 If $r \neq r_L$ and $r \neq r_R$, we have that e lies inside τ, which will cause a more substantial update of the data structure (see Fig. 4.10). In fact, while τ was formerly horizontally decomposed, it has now to be vertically decomposed by means of a vertical cut along edge e. First, we call recursively the procedure on both the left and right children of μ. At this point, both trapezoids τ_B and τ_T have a vertical cut along edge e, corresponding to the string

$$(\tau_{11}\sigma_{11}e\sigma_{12}\tau_{12})\,I\,(\tau_{21}\sigma_{21}e\sigma_{22}\tau_{22}).$$

To comply with the rule that vertical cuts take precedence over horizontal cuts, we have to restructure the tree according to the string

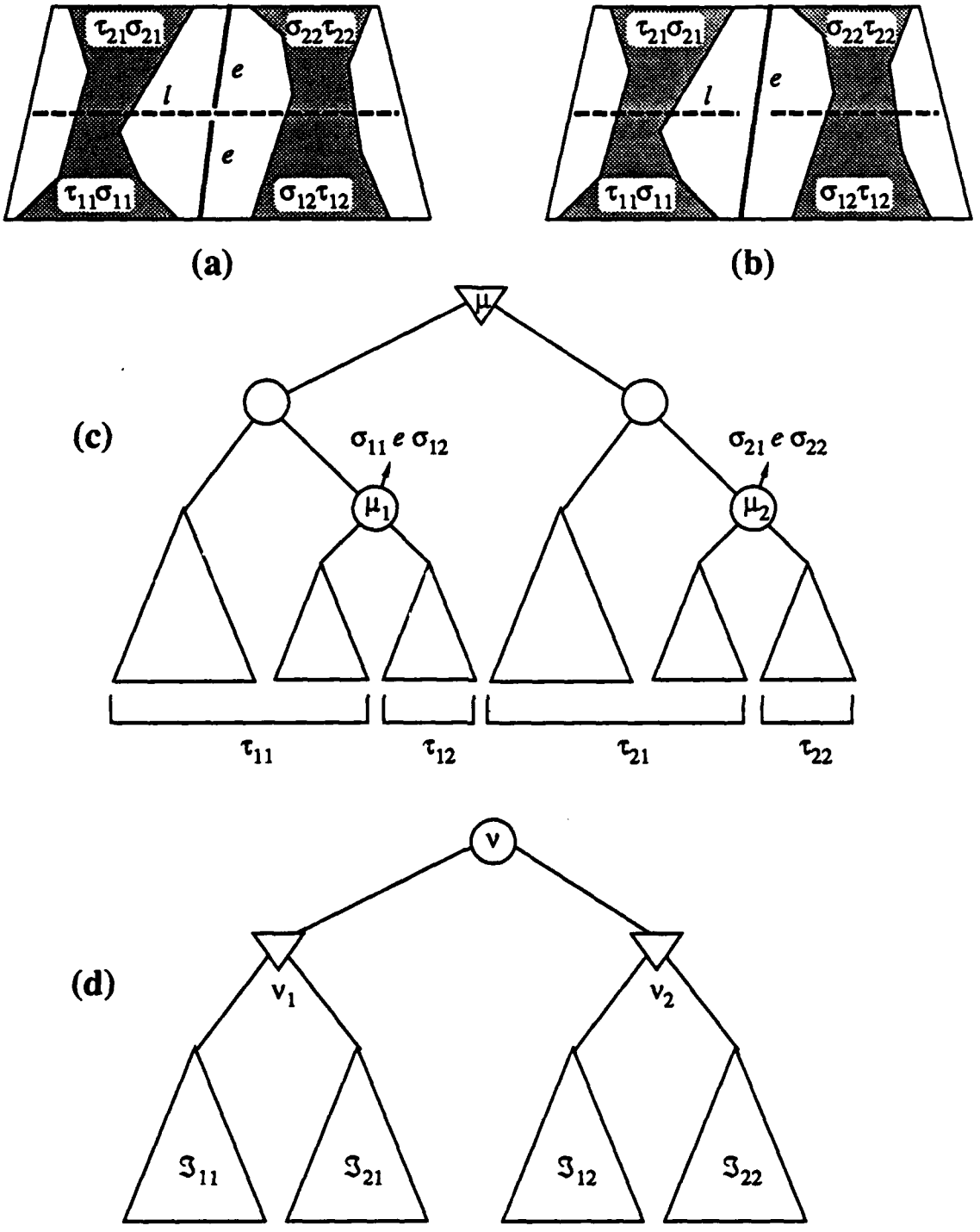


Figure 4.10 Update of the data structure in Case 3.2.3 of *INSERTSEGMENT*. (a,c) Preliminary restructuring (edge e is inserted in both children of μ). (b,d) Final restructuring.

$$(\tau_{11}\sigma_{11})\, l\, (\tau_{21}\sigma_{21})\, e\, (\sigma_{12}\tau_{12})\, l\, (\sigma_{22}\tau_{22}).$$

This is done as follows:

(1) Let μ_L and μ_R be the left and right child of μ, respectively. Find node μ_1 of $T(\mu_L)$ associated with the sequence $\sigma_1 = \sigma_{11} e \sigma_{12}$. Similarly, find node μ_2 of $T(\mu_R)$ associated with the sequence $\sigma_2 = \sigma_{21} e \sigma_{22}$.

(2) Split $T(\mu_L)$ and $T(\mu_R)$ at nodes μ_1 and μ_2, respectively. This yields trees $\Im_{11}$, $\Im_{12}$, $\Im_{21}$, and $\Im_{22}$.

(3) Split the sequences $\sigma(\mu_1)$ and $\sigma(\mu_2)$ at edge e. This yields the sequences σ_{11}, σ_{12}, σ_{21}, and σ_{22}.

(4) If $\sigma_{11} \neq \emptyset$, create a O-node μ_{11} for it, and add μ_{11} and a ε-node to the right of $\Im_{11}$. If $\sigma_{21} \neq \emptyset$, create a O-node μ_{21} for it, and add μ_{21} and a ε-node to the right of $\Im_{21}$. If $\sigma_{12} \neq \emptyset$, create a O-node μ_{12} for it, and add μ_{12} and a ε-node to the left of $\Im_{12}$. If $\sigma_{22} \neq \emptyset$, create a O-node μ_{22} for it, and add μ_{22} and a ε-node to the left of $\Im_{22}$.

(5) Create a ∇-node ν_1 (associated with the portion of τ to the left of e), with left subtree $\Im_{11}$ and right subtree $\Im_{21}$.

(6) Create a ∇-node ν_2 (associated with the portion of τ to the right of e), with left subtree $\Im_{12}$ and right subtree $\Im_{22}$.

(7) Replace μ with a O-node ν having $\sigma(\nu) = e$, left child ν_1, and right child ν_2. Finally, rebalance the decomposition tree containing node ν as a leaf.

This completes the description of the algorithm for operation *INSERTSEGMENT*.

We now analyze the time complexity of operation *INSERTSEGMENT*. Let r^* be the intersection of region r and the horizontal slab spanned by segment e (see Fig. 4.11). Notice that, since r is 2-supported, so is r^* . The insertion of e into the trapezoid tree can be viewed as a visit of a subgraph of $\Im(\Re)$ (see Fig. 4.12). Specifically, this subgraph consists of an initial path originating at the root and consisting of O-nodes such that e is always external to the sequence of spanning edges (Case 1.1 or 1.2), and of ∇-nodes such that e never crosses the median of the corresponding slab (Case 3.1). For either type of node, $O(1)$ time is spent at each node. The visit of this initial path terminates at a ∇-node (referred to as the *fork*), where, for the first time, e

crosses the median.

At the fork, e is passed on to both children, i.e., the visit continues with two distinct paths, called the *left spine* and *right spine*. Let us consider the left spine (the right spine is similarly analyzed). It is easily realized that this path contains a set of distinguished ∇-nodes such that, for trapezoid τ associated with any of them, $y(TOP(\tau)) \leq y(HIGH(e))$ and $y(BOT(\tau)) < y(LOW(e)) \leq y(MEDIAN(\tau))$. This implies that e spans the trapezoid τ_T associated with the right subtree of the distinguished ∇-node τ. The decomposition tree $T(\tau_T)$ is called an *allocation subtree* for edge e. The slab spanned by the roots of all allocation subtrees of e form a partition of r^* into a set of convex polygons, of which it is easily realized that at most one is 2-supported while all the others are 1-supported (see Fig. 4.11).

Each distinguished node of the left spine is a branching point for the visit, since the procedure is called recursively on both children. The left child is on the left spine, while the right child is the root of an allocation subtree for e. Within the allocation subtree the visit continues along a path, called *allocation trail* until we reach a node, called *allocation target* for which one of the Cases 1.3, 2, or 3.2 occurs. The time spent at each node of the left spine and at each node

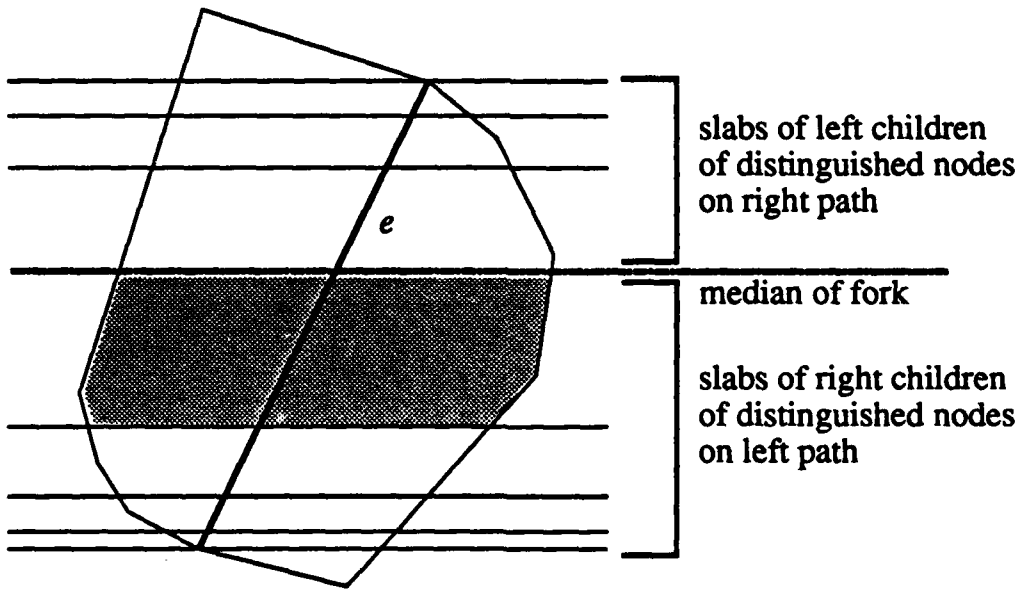


Figure 4.11 Region r^* in the analysis of operation *INSERTSEGMENT*.

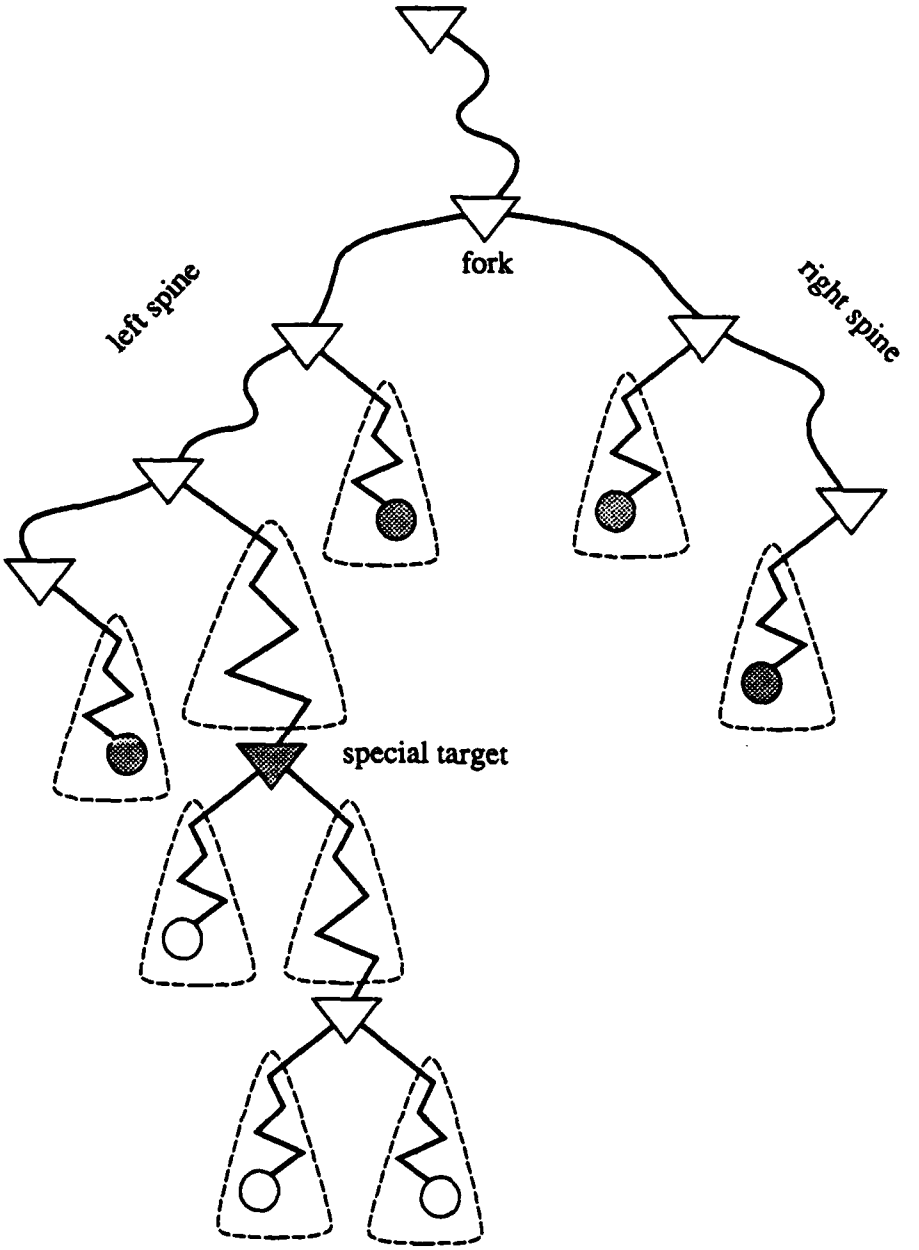


Figure 4.12 Subgraph of the trapezoid tree visited by the *INSERTSEGMENT* algorithm. The target nodes are shaded.

of the allocation trail, target excluded, is $O(1)$.

Consider now an allocation target μ of e. If μ is an ε-node, then its handling takes $O(\log n)$ time (Case 2). If μ is an O-node, then in $O(\log n)$ time e is inserted into the corresponding $\sigma(\mu)$ (Case 1.3). Finally, if μ is a ∇-node, we are in Case 3.2. If the intersection of r with the slab of μ is a 1-supported polygon, then we have either Case 3.2.1 or Case 3.2.2, since e is adjacent to a spanning edge of this slab. For such cases, the processing of μ takes time $O(\log n)$. Else (when the intersection of r with the slab of μ is a 2-supported polygon), we are in Case 3.2.3. From Lemma 4.1, we have that this case can occur for at most one target node, called henceforth *special target.*

Let μ be the special target, and let τ be the associated trapezoid. In the processing of μ, we recursively call the procedure on both children of μ, so that the visit branches again into two paths. Since edge e spans both τ_B and τ_T, each path will encounter a node for which one of the Cases 1.3, 2, or 3.2 occurs. However, by Lemma 4.1, at most one of the polygons obtained by intersecting region r with τ_B and τ_T is 2-supported, so that Case 3.2.3 can occur again for at most one such node. Finally, after the recursive calls have been executed, we perform the restructuring specified in the description of Case 3.2.3, which takes $O(\log n)$ time. Let p be the order of μ, i.e., the number of nodes on the path from μ to the root. The work done for completing the visit originating at μ requires in the worst case time $O(\log n)$ plus the time for executing one recursive call to the procedure on a node of order $p-1$. Since the order of a node is at most $\log N$, this sums up to $O(\log n \log N)$.

We summarize the preceding discussion as follows:

(*i*) The total time spent at the nodes of the path from the root to the fork and at the nodes of the left and right spine is $O(\log n + \log N)$.

(*ii*) The total time spent in traversing each allocation trail is $O(\log n)$, for a total time $O(\log n \log N)$.

(*iii*) The time spent for processing each nonspecial target node is $O(\log n)$, for a total time $O(\log n \log N)$.

(*iv*) The time spent for processing the special target and the subsequent visit is $O(\log n \log N)$.

We therefore conclude

Theorem 4.5 The complexity of operation *INSERTSEGMENT* $(e,v_1,v_2,r;r_1,r_2)$ is $O(\log n \log N)$.

Remark. The underlying segment-tree structure of the method effects the fragmentation into a logarithmic number of subsegments of an edge to be inserted. Each of these subsegments, however, may not be insertable as a single entity, if it does not occur next to existing spanning edges of the corresponding slab. Note, however, that such a spanning edge would always exist if each region of $\mathfrak{R}$ were a 1-supported polygon. Relaxing such strong constraint to 2-supportedness, we effectively create a "channel" (a sort of "virtual diagonal") between *LOW* (r) and *HIGH* (r) of each polygon r of $\mathfrak{R}$, which guarantees that at most one segment fragment (the one reaching the special target) is further subdivided into a number of subsegments at most logarithmic in N, before being restored to its original length as specified by the string transformation given for Case 3.2.3.

4.7. Other Update Operations

The algorithm for operation *REMOVESEGMENT* $(e,v_1,v_2,r_1,r_2;r)$ is similar to the one for the *INSERTSEGMENT* operation, and for brevity we omit its description.

Theorem 4.6 The complexity of operation *REMOVESEGMENT* $(e,v_1,v_2,r_1,r_2;r)$ is $O(\log n \log N)$.

We consider now operation *INSERTPOINT* $(v,e;e_1,e_2)$. The algorithm consists of two phases. The first phase is performed by a recursive procedure consisting of the following steps:

(1) Using the query algorithm, we search for a node μ of the trapezoid tree such that v lies in the trapezoid τ of μ, and e is a spanning edge of $\sigma(\mu)=s_1,\cdots,s_p$, i.e., $e=s_i$ for some i.

(2) We perform some restructurings at node μ (to be specified later), which correspond to inserting v onto line *MEDIAN* (τ).

(3) We recursively call the procedure if v is not on line *MEDIAN* (τ).

The restructurings performed in Step 2 depend on the position of edge e in the sequence $\sigma(\mu)$. We distinguish three cases:

1. If $e = s_i$ is a nonextreme element of the sequence $\sigma(\mu)$, i.e. $i \neq 1, p$, the restructuring is dictated by the string transformation

$$s_{i-1}\, e\, s_{i+1} \rightarrow s_{i-1}\, (e)\, l\, (e)\, s_{i+1}.$$

Omitting the details, node μ is replaced in the decomposition tree by nodes μ_1, μ', and μ_2, where μ_1 and μ_2 are O-nodes with sequences $\sigma(\mu_1) = s_1, \cdots, s_{i-1}$ and $\sigma(\mu_2) = s_{i+1}, \cdots, s_p$, and μ' is a ∇-node. In turn, μ' is the root of an elementary decomposition tree, whose left and right subtree consist each of a O-node with sequence e, and of two ε-nodes.

2. If $e = s_1$ is the leftmost element of $\sigma(\mu)$ and $p \geq 2$, the restructuring is dictated by the string transformation

$$(\tau_1)\, l\, (\tau_2)\, e\, s_2 \rightarrow (\tau_1\, e)\, l\, (\tau_2\, e)\, s_2.$$

Let $\mu' = pred(\mu)$. Node μ' is associated with trapezoid τ' to the left of e. Edge e is inserted into the left and right children of μ' as in the algorithm for *INSERTSEGMENT*.

3. If $e = s_p$ is the rightmost element of $\sigma(\mu)$ and $p \geq 2$, the restructuring is dictated by the string transformation

$$s_{p-1}\, e\, (\tau_1)\, l\, (\tau_2) \rightarrow s_{p-1}\, (e\, \tau_1)\, l\, (e\, \tau_2).$$

This case is symmetric to Case 2 above.

4. If $e = s_1 = s_p$ is the unique element of $\sigma(\mu)$, the restructuring is dictated by the string transformation

$$(\tau_{11})\, l\, (\tau_{21})\, e\, (\tau_{12})\, l\, (\tau_{22}) \rightarrow (\tau_{11}\, e\, \tau_{12})\, l\, (\tau_{21}\, e\, \tau_{22}).$$

This restructuring merges the two trapezoids associated with the ∇-nodes $\mu_1 = pred(\mu)$ and $\mu_2 = next(\mu)$ into a unique trapezoid, represented by a new node μ'. The tasks to be performed are analogous to the ones for Case 3.2.3 of the edge insertion algorithm, and omitted for brevity.

This completes the description of the restructurings performed in the three cases. In Step 3, if v is below *MEDIAN* (τ) we recursively call the procedure on the left subtree of μ', while if v is

above *MEDIAN* (τ) we recursively call the procedure on the right subtree of μ'.

A detailed time complexity analysis of the first phase can be conducted using techniques similar to the ones presented in the preceding section. In summary, the recursion depth is at most $O(\log N)$, and the time for Step 2 is $O(\log n)$ for all but (possibly) one node, for which it is $O(\log n \log N)$. Hence the first phase takes time $O(\log n \log N)$.

The second phase rebalances the decomposition trees embedded in $\Im(\Re)$ for which the weight of some leaf has increased by one. There are at most $O(\log N)$ such trees, so that the second phase also takes time $O(\log n \log N)$.

We therefore conclude

Theorem 4.7 The complexity of operation *INSERTPOINT* $(v,e;e_1,e_2)$ is $O(\log n \log N)$.

The algorithm for operation *REMOVEPOINT* $(v,e_1,e_2;e)$ is similar to the one for operation *INSERTPOINT*, and we have

Theorem 4.8 The complexity of operation *REMOVEPOINT* $(v,e_1,e_2;e)$ is $O(\log n \log N)$.

Finally, as regards operation *MOVEPOINT* $(v;x)$, we observe that it does not affect the trapezoid tree. Hence, it can be simply carried out by updating the dictionary in time $O(\log n)$.

In conclusion, we have proved the following theorem:

Theorem 4.9 Let $\Re$ be a convex subdivision whose n vertices lie on a fixed set L of N horizontal lines. There exists a dynamic point location data structure for $\Re$ with space $O(n \log N)$ and query time $O(\log n + \log N)$, which allows for insertion/deletion of vertices and edges in time $O(\log n \log N)$, all time bounds being worst-case.

4.8. Extensions

The analysis of the query and update operations shows that the essential requirement for the method is the monotonicity and 2-supportedness of the regions of the subdivision, while the convexity requirement can be relaxed. Hence, we obtain the following corollary:

Corollary 4.1 Let $\mathfrak{R}$ be a monotone 2-supported subdivision whose n vertices lie on a fixed set L of N horizontal lines. There exists a dynamic point location data structure for $\mathfrak{R}$ with space $O(n \log N)$ and query time $O(\log n + \log N)$, which allows for insertion/deletion of vertices and edges in time $O(\log n \log N)$, all time bounds being worst-case.

CHAPTER 5

DYNAMIC POINT LOCATION: RECONSTRUCTION METHOD

5.1. Introduction

In this section, we present a fully dynamic technique for point location in a triangulation with n vertices. This method features a simple and versatile $O(n)$-space data structure, which can be used in conjunction with any of the known static point location data structures, and allows a tradeoff between query and insertion time. Namely, we show that for any $2 \le b \le \sqrt{n}$, there exists a dynamic point location data structure with $O(\log_b n \log n)$ query time and $O((\log_b n)^2 b \log b)$ update time.

By setting $b = 2$, we obtain $O(\log^2 n)$ query and update times, which match the result of Chapter 3. By setting $b = \log n$, we obtain $O(\log^2 n / \log\log n)$ query time and $O(\log^3 n / \log\log n)$ update time. Finally, by setting $b = \sqrt{n}$, we obtain $O(\log n)$ query time and $O(\sqrt{n} \log n)$ update time. This method is based on the dynamization techniques for decomposable search problems described in [3, 42],

The rest of the chapter is organized as follows. Section 5.2 contains preliminary definitions. The dynamic technique is described in Section 5.3.

5.2. Preliminaries

Let $\mathfrak{R}$ be a triangulation. We consider the following update operations on $\mathfrak{R}$:

INSERTSTAR $(v, r; r_1, r_2, r_3)$: Add vertex v inside region r and edges between v and the vertices of r, which is decomposed into new regions r_1, r_2, and r_3 (see Fig. 5.1(a)).

REMOVESTAR $(v, r_1, r_2, r_3; r)$: Remove degree-3 vertex v and its incident edges, which merges the three regions formerly containing v into a new region r (see Fig. 5.1(a)).

SWAPDIAGONAL $(e; r_1, r_2)$: Let e be an edge such that the union of the two regions on the left and right of e is a convex quadrilateral. Remove e and reinsert it as the other diagonal of

the quadrilateral, thus creating two new regions r_1 and r_2. (see Fig. 5.1(b)).

The problem of performing point location in $\Re$ under the above set of dynamic updates will be called problem *PL-1*. The following theorem demonstrates the adequacy of the above repertory.

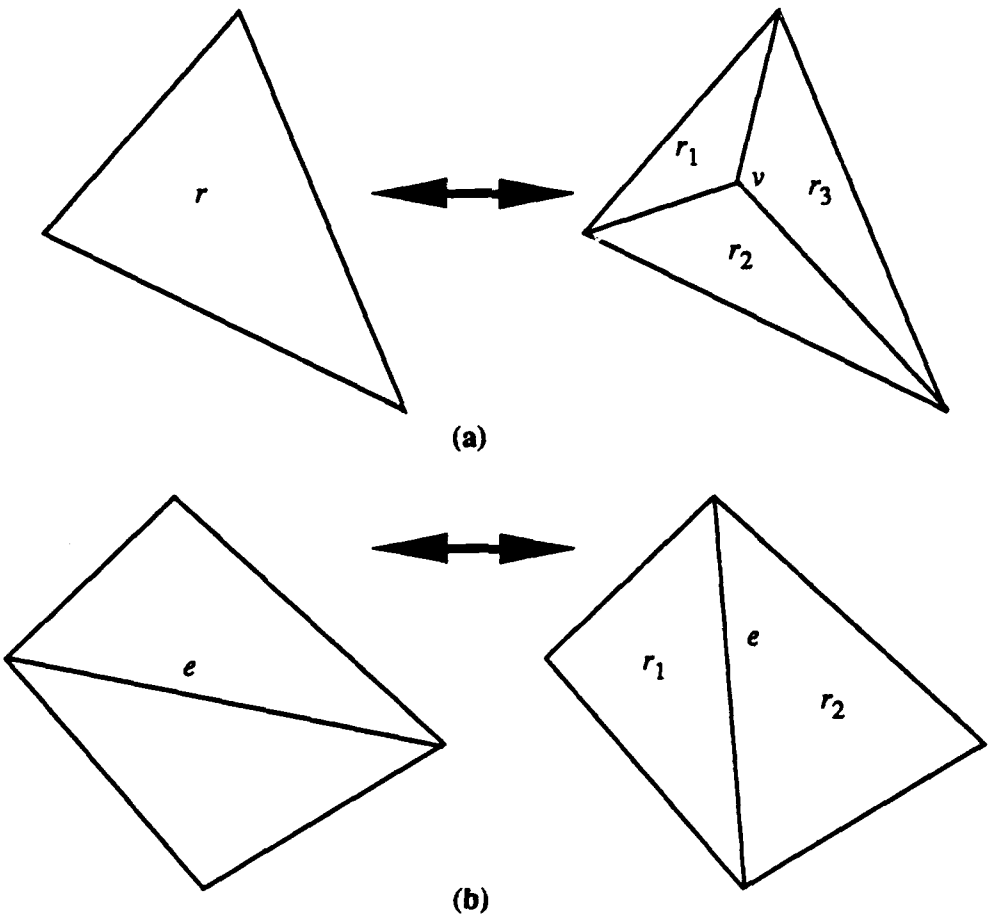


Figure 5.1 (a) Example of operation *INSERTSTAR/ REMOVESTAR*; (b) Example of operation *SWAPDIAGONAL*.

Theorem 5.1 An arbitrary triangulation $\Re$ with n vertices can be assembled starting from the empty subdivision, and disassembled to obtain the empty subdivision, by a sequence of $O(n)$ operations from the above repertory.

Proof: We constructively demonstrate only the disassembly of $\Re$, since the assembly is obtainable by replacing each operation with its inverse and by reversing their order. If $\Re$ has a vertex u of degree 3, we remove u by means of a *REMOVESTAR* operation. Otherwise, we know that $\Re$ has at least one vertex v of degree $d \in \{4,5\}$ (an easy consequence of Euler's formula). Consider the neighbors $v_0, v_1, \cdots, v_{d-1}$ of vertex v, in clockwise order around v. Each pair of two consecutive neighbors forms with v a triangle $t_i = (v, v_i, v_{i+1})$ (sum over the indices is modulo d). Since the sum of the internal angles of the polygon $(v_0, v_1, \cdots, v_{d-1})$ is equal to $(d-2)\pi$, it is easy to verify that for at least one index i, the union of triangles t_i and t_{i+1} is a convex quadrilateral. Hence, by a sequence of at most two *SWAPDIAGONAL* and one *REMOVESTAR* operation, we can remove vertex v. A simple inductive argument completes the proof. □

First, we consider another variant of the dynamic point location problem, denoted as problem *PL-2*, which is described as follows: Let T be a set of triangles such that the interiors of any two triangles of T are disjoint. We want to perform the following operations on T:

FIND (q,T): Return the triangle t of T that contains the query point q; if q is on some edge (or vertex) of t, the edge (or vertex) itself is returned; if no triangle contains q, a null value is returned.

ADD (t,T): Add triangle t to T; the operation is allowed only if t does not intersect the interior of any other other triangle of T.

DELETE (t,T): Remove triangle t from T.

We denote with *Find*(m), *Add*(m), and *Delete*(m) the time to perform operations *FIND*, *ADD*, and *DELETE*, respectively, where m is number of triangles of T.

Since a triangulation $\Re$ with n vertices has $m = 2n-4$ regions, and each of the update operations for problem *PL-1* corresponds to at most four *ADD/DELETE* operations in problem *PL-2*, we have

Lemma 5.1 Given a data structure for problem *PL-2*, there exists a data structure for problem *PL-1* with the same (order of) space requirement and query time, and such that each update operation takes time $O(Add(n)+Delete(n))$.

5.3. The Incremental Reconstruction Method

In this section, we show an efficient solution to problem *PL-2* based on the following straightforward decomposition property:

Lemma 5.2 Denoting by (T_1,T_2) a partition of T, we have

$$FIND(p,T)=FIND(p,T_1) \cup FIND(p,T_2). \qquad \square$$

The main idea is to maintain a partition of the set T into $O(\log m)$ subsets, where each subset has a separate static data structure for point location. Using the decomposition property, the answer to $FIND(p,T)$ can be obtained by performing a separate *FIND* in each subset, and then combining the answers. Since each subset has no more than m elements, each subquery takes $O(\log m)$ time, so that the global time for operation *FIND* is $O(\log^2 m)$. Also, since the space requirement for each subset is linear, the overall space requirement is $O(m)$. In order to show that operation *ADD* can also be efficiently performed, we describe the data structure in more detail.

Let $a_\nu, a_{\nu-1}, \cdots, a_0$ be the binary representation of m (the number of triangles of T), where $\nu=\lfloor \log m \rfloor$. The dynamic data structure D for T consists of $\nu+1$ static point location data structures $S_0, \cdots, S_\nu$, where S_i represents a subdivision induced by exactly $a_i 2^i$ triangles. After the addition of a new triangle, m increases to $m+1$, so that in order to maintain the data structure D, we have to discard the static structures S_i corresponding to the bits that change from one to zero, and create new static structures for the bits that change from zero to one. In the worst case, namely when $m+1$ is a power of two, this takes $O(m)$ time, since we have to discard all the existing static structures and build from scratch a new data structure for $m+1$ triangles. However, the average time complexity over a sequence of m *ADD* operations to an initially empty data structure, denoted $\overline{Add}(m)$, is $\overline{Add}(m)=O(\log^2 m)$. To see this, assume for simplicity that $m=2^\nu-1$. From elementary properties of binary counting, the static structure S_i is

built exactly 2^{v-i-1} times, so that the total insertion cost over a sequence of m insertions is

$$m\,\overline{Add}(m)=\sum_{i=0}^{v-1}2^{v-i-1}P(2^i)$$

where $P(k)=O(k\log k)$ is the (preprocessing) cost for building an optimal static point location structure for k triangles [13]. With simple manipulations, we obtain

$$m\,\overline{Add}(m)=O\left[\sum_{i=0}^{v-1}2^{v-i-1}2^i i\right]=O\left[2^{v-1}\sum_{i=0}^{v-1}i\right]=O(m\log^2 m).$$

At this point, we have an $O(m)$ space semidynamic data structure with $O(m)$ space requirement that supports operation *FIND* in time $O(\log^2 m)$ (worst-case) and operation *ADD* in time $O(\log^2 m)$ (amortized). The amortized time bound for *ADD* can be turned into a "worst-case" bound by spreading the work for building each static structure S_i over several consecutive steps, so that after each *ADD* operation $O(\log^2 m)$ work is performed [3, 42]. This data structure has a special type of worst-case time bound, since the restructuring operations are distributed over time. We will refer to such time bound as a *distributed worst-case* time.

The above discussion can be summarized as follows:

Lemma 5.3 There is a data structure for problem *PL-2* with $O(m)$ space requirement that supports operations *FIND* and *ADD* in time $O(\log^2 m)$ (distributed worst-case).

In order to perform efficiently operation *DELETE* also, we use a "lazy" deletion scheme. First, we modify the aforementioned data structure by adding a *mark* field to every record that describes a triangle of T, so that *DELETE* (t,T) can be simply performed by setting the mark of t. Also, we modify the procedure for constructing the static point location structures, so that only the unmarked triangles are considered in search operations, while the marked ones serve the only purpose of maintaining the nominal size of the structure.

This modified structure D allows for efficient queries and updates as long as the number k of marked triangles does not exceed a fixed fraction of the total number m of triangles, say one half. If after a deletion we have $k > m/2$, we start building a new structure D' of the same type for the unmarked triangles only. We spread the construction of D' over $m/4$ steps, where a step

corresponds to an update operation. At each step we add two "old" triangles to D' and we perform the current update operation (*ADD* or *DELETE*) on both D and D'. After the completion of all the steps, D' is ready and replaces D. Notice that the total number m' of triangles in D' and the number of marked triangles k' satisfy the relations

$$\frac{m}{2} \le m' \le \frac{m}{2} + \frac{m}{4} = \frac{3}{4}m \qquad 0 \le k' \le \frac{m}{4}$$

so that we have $k'/m' \le 1/2$, which means that also D' can support queries and updates efficiently. During the construction of D', we perform the *FIND* operations on D, for which the number of marked triangles does not exceed one fourth of the total number of triangles.

Therefore, by using at most two data structures that allow only for insertions and lazy deletions, and maintaining the invariant that the number of marked triangles does not exceed a fixed fraction of the total number of triangles, we obtain a complete dynamic solution for problem *PL-2* with $O(\log^2 m)$ query and update time:

Theorem 5.2 There is a data structure for problem *PL-2* with $O(m)$ space requirement that supports operations *FIND*, *ADD*, and *DELETE* in time $O(\log^2 m)$ (distributed worst-case).

Recalling Lemma 5.1, we obtain the following result for the original problem *PL-1*.

Corollary 5.1 There is a data structure for the dynamic point location problem *PL-1* with $O(n)$ space requirement and $O(\log^2 n)$ (distributed worst-case) query and update times.

The aforementioned technique for problem *PL-2* makes use of the properties of counting in base 2. By using a larger base, we can reduce the number of static data structures used. In fact, using a base $b \ge 2$ the above data structure can be modified to work with a partition of T into subsets S_{ij}, $i = 0, \cdots, \nu$; $j = 1, \cdots, (b-1)$; where $\nu = \log_b m$, and each subset S_{ij} has size b^i.

Using arguments similar to the ones already given for the case $b = 2$, we can show that this data structure has $O(m)$ space requirement and

$$Find(m) = O(\log_b m \log m); \quad Add(m) = Delete(m) = O((\log_b m)^2\, b \log b).$$

To obtain an improvement in the search time, we can let the base b grow with m. Applying the

results described in [38], we obtain

Theorem 5.3 Let $\mathfrak{R}$ be a triangulation with n vertices, and let $b = b(n)$ be a smooth nondecreasing integer function with $2 \le b \le \sqrt{n}$. There exists a data structure for point location problem *PL-1* with $O(n)$ space requirement, $O(\log_b n \log n)$ query time and $O\left((\log_b n)^2 b \log b\right)$ update time.

CHAPTER 6

DYNAMIC TRANSITIVE CLOSURE

6.1. Introduction

The notion of a planar *st*-graph – i.e., a planar acyclic digraph embedded in the plane with exactly one source, s, and one sink, t, both on the external face – was first introduced in the planarity testing algorithm of Lempel et al. [36], and was fruitfully used in a number of applications, which include planar graph embedding [9, 30, 54], graph planarization [29, 44], graph drawing algorithms [10, 53, 59], floor planning [4, 57] planar point location [13, 35], visibility representations [41, 49, 51, 58], motion planning [21, 48], and VLSI layout compaction [25, 57]. Also, planar *st*-graphs are important in the theory of partially ordered sets since they are associated with planar lattices [32].

In this chapter we further the investigation of these structures, and show that any planar *st*-graph G admits two total orders (referred to as leftist and rightist orders) on the set $V \cup E \cup F$, where V, E, and F are respectively the set of vertices, edges, and faces of G. Each of these two orders yields a unique representation of G as a string whose terms are symbols representing all the topological constituents of G. Graph G can be dynamically modified by means of insertion of edges and expansions of vertices, and of their inverses. These operations form a complete set, since any n-vertex planar *st*-graph can be assembled or disassembled by an appropriate sequence of $O(n)$ such operations.

The central result of this chapter is that the string representation of the graph resulting from one of the postulated updating operations is obtained as a syntactic transformation of the original string representation. This transformation consists of the execution of $O(1)$ primitives, such as insertions, deletions, and swaps of substrings.

This general framework provides the theoretical underpinning and unifying viewpoint for three significant applications: point location in a planar monotone subdivision, transitive-closure

query in planar *st*-graphs, and contact-chain query in convex subdivisions. In this chapter we shall only briefly illustrate (in Section 6.4) the connection between planar *st*-graphs and monotone subdivisions, since the point location problem in the latter has been treated earlier in Chapter 3 in purely geometric terms.

A *transitive-closure query* for a planar *st*-graph G consists of testing for the existence of (and/or reporting) a directed path between two vertices u and v of G. We are interested in a graph G that can be dynamically modified.

The previous best results concern semi-dynamic versions of the transitive-closure query problem in digraphs (namely, edge insertions in general digraphs and edge deletions in acyclic digraphs), and have $O(1)$ query time, $O(n)$ *amortized* update time, and $O(n^2)$ storage [27,28,45]. In this chapter we establish the following result:

Theorem 6.1 Let G be a planar *st*-graph with n vertices. There exists an $O(n)$-space dynamic data structure for the transitive-closure query problem on G, which supports queries and updates in time $O(\log n)$ (worst-case).

Finally, we consider the problem of contact-chain query in convex subdivisions, which arises in motion planning and computer graphics, and is described as follows [8,21,48]. Given a *convex subdivision* $\Re$ of the plane and an (oriented) direction θ, we say that region r_1 *pushes* region r_2 if r_1 and r_2 are adjacent and there exists a line in direction θ which intersects r_1 and r_2 in that order. A *contact chain* in $\Re$ is a sequence of regions $r_1, r_2, \cdots, r_k$ such that r_i pushes r_{i+1} for $i = 1, \cdots, k-1$ (see Fig. 6.1). Assume that the regions of $\Re$ are rigid objects, and we want to translate them one at a time in direction θ avoiding collisions. Then the existence of a contact chain from r_1 to r_2 implies that r_2 obstructs r_1, i.e., r_2 must be translated before r_1.

A *contact-chain query* consists of testing the existence of (and/or reporting) a contact chain between two regions of $\Re$. We are interested in answering contact-chain queries in a very general dynamic environment, where $\Re$ can be updated by means of insertions/deletions of vertices and edges, and the direction θ can be changed by *elementary* increments/decrements. (An elementary increment/decrement of direction is such that the push relation is inverted for exactly one pair of adjacent regions.) Casting this problem in the planar *st*-graph framework, we

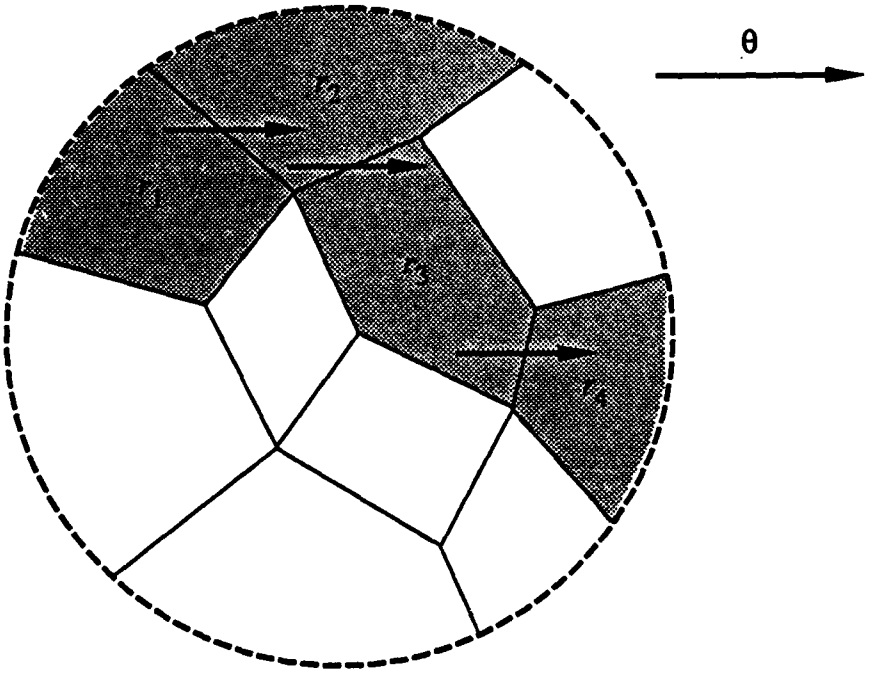


Figure 6.1 Example of contact chain.

establish the following result:

Theorem 6.2 Let $\Re$ be a convex subdivision with n vertices. There exists an $O(n)$-space dynamic data structure for the contact-chain query problem in $\Re$, which supports queries and updates in time $O(\log n)$ (worst-case).

The rest of this chapter is organized as follows. Section 6.2 provides preliminary definitions and properties of planar *st*-graphs. Section 6.3 presents the technique for the dynamic maintenance of planar *st*-graphs. Applications to planar point location, transitive closure, and contact chains are described in Section 6.4.

6.2. Planar *st*-Graphs

Basic definitions on graphs and posets can be found in textbooks such as [5, 15].

Let G be a directed graph, for brevity digraph, and v a vertex of G. As usual, we denote by $\deg^-(v)$ the *indegree* of v, i.e., the number of incoming edges of v, and by $\deg^+(v)$ the *outdegree* of v, i.e., the number of outgoing edges of v. A *source* of G is vertex s with $\deg^-(s)=0$. A *sink* of G is vertex t with $\deg^+(t)=0$. A *transitive* edge of G is an edge $e=(u,v)$ such that there exists another directed path from u to v consisting of at least two edges.

A *planar st-graph* is a planar acyclic digraph G with exactly one source, s, and exactly one sink, t, which is embedded in the plane so that s and t are on the boundary of the external face (see Fig. 6.2).

These graphs were first introduced in the planarity testing algorithm of Lempel et al. [36]. Several important properties of planar st-graphs are expressed by the following lemmas:

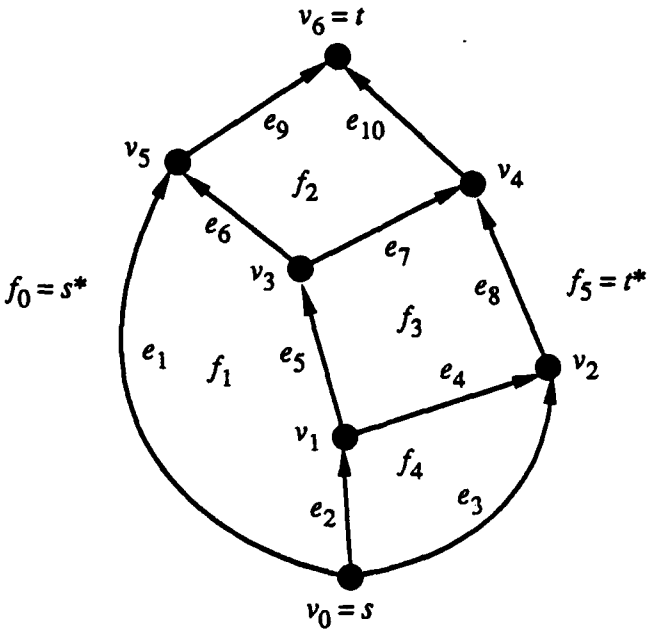


Figure 6.2 Example of planar st-graph.

Lemma 6.1 [36] Every vertex of G is on some directed path from s to t.

Lemma 6.2 [51] For every vertex v of G, the incoming (outgoing) edges appear consecutively around v (See Fig. 6.3(a)).

Lemma 6.3 [51] For every face f of G, the boundary of f consists of two directed paths with common origin and destination. (See Fig. 6.3(b)).

Lemma 6.4 [10, 33] G admits a planar *upward* drawing, i.e., a planar drawing such that every edge (u,v) is a curve monotonically increasing in the vertical direction.

Let P be a poset (partially ordered set), where $\ll$ denotes the partial order on the elements of P. The *Hasse diagram* (also called *covering digraph*) of P is a digraph G whose vertices are the elements of P, and such that (u,v) is an edge of G if and only if $u \ll v$ and there is no other element x of P such that $u \ll x \ll v$. G is acyclic and has no transitive edges, i.e., G is the

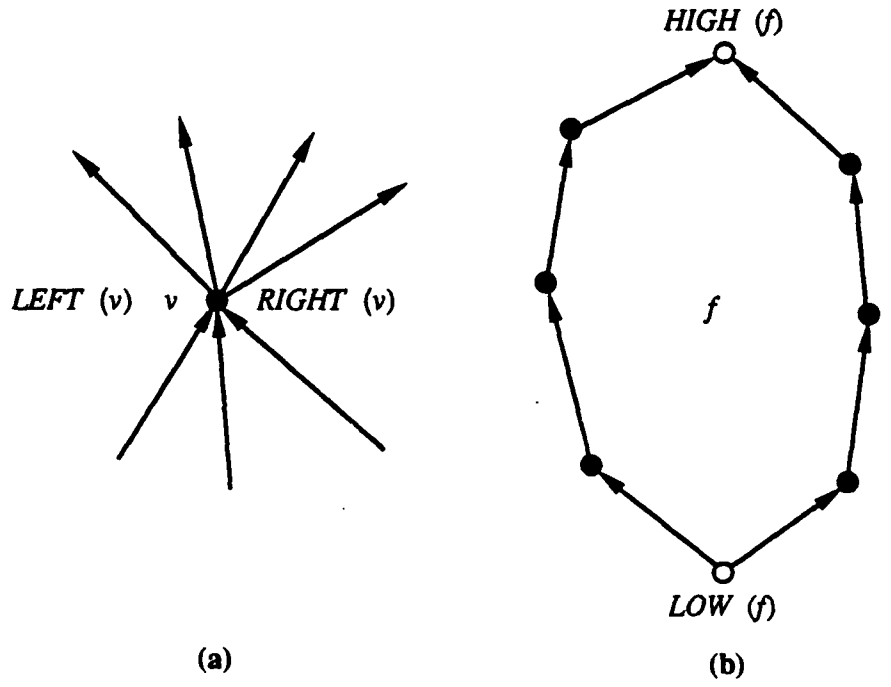


Figure 6.3 (a) Example for Lemma 6.2. (b) Example for Lemma 6.3.

transitive reduction of the graph of $\ll$ on vertex set P. Hasse diagrams are usually represented by straight-line drawings such that for each edge (u,v) the ordinate of vertex u is smaller that that of vertex v.

A *planar lattice* is a poset whose Hasse diagram is a planar *st*-graph. Also, every plane *st*-graph without transitive arcs is the Hasse diagram of some planar lattice. Several properties of planar lattices are described in [32].

A *linear extension* of a poset P is a total order $<$ on the elements of P such that, for any two elements u and v of P, $u \ll v$ implies $u < v$. A linear extension corresponds to a topological sorting of the vertices of the Hasse diagram of P. We say that P has *dimension* k if G admits k linear extensions $<_1, <_2, \cdots, <_k$, such that $u \ll v$ if and only if $u <_1 v, u <_2 v, \cdots, u <_k v$, and k is minimum.

It is known that planar lattices have dimension 2 [5, p. 32, ex. 7(c)] [31, 32], which implies the following lemma:

Lemma 6.5 [5, 31, 32] Let G be a planar *st*-graph with n vertices. There exist two total orders on the vertices of G, denoted $<_L$ and $<_R$, such that there is a directed path from u to v if and only if $u <_L v$ and $u <_R v$. Furthermore, orders $<_L$ and $<_R$ can be computed in $O(n)$ time.

Lemma 6.5 is based on the fact that the underlying partial order of a planar lattice admits a "complementary" partial order (see [32]). Figure 6.4(a) shows a planar *st*-graph where each vertex is labeled by its ranks in the orders $<_L$ and $<_R$.

In the following definitions, the concepts of *left* and *right* refer to the orientation of the edges. For example, the face to the left of an edge (u,v) is the face containing edge e which appears on the left side when traversing edge (u,v) from vertex u to vertex v. Also, the reader will find it convenient to visualize the planar *st*-graph G as being drawn in the plane with edges monotonically increasing in the vertical direction (see Lemma 6.4).

Given vertices u and v of G such that there exists a path from u to v, the set of paths from u to v defines a planar *st*-graph with source u and sink v which is an induced subgraph of G. The two paths that form the external boundary of this subgraph will be called the *leftmost path* and *rightmost path* from u to v, respectively. For example, the external boundary of G consists of the

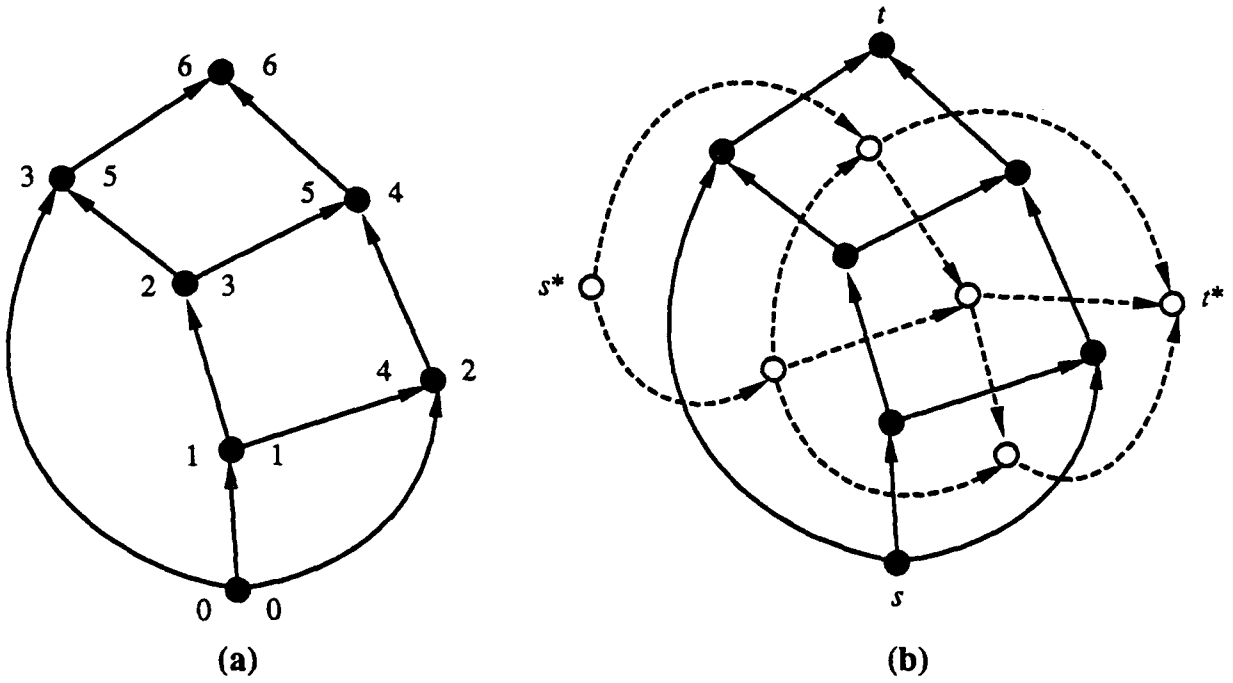


Figure 6.4 (a) Orders $<_L$ and $<_R$ on the vertices of a planar *st*-graph; (b) A planar *st*-graph G and its dual G^*.

leftmost and rightmost paths from s to t.

Let G^* be the digraph obtained from the dual graph of G as follows (see Fig. 6.4(b)): (1) the dual edge e^* of an edge e is directed from the face to the left of e to the face to the right of e; (2) the external face of G is dualized to two vertices of G^*, denoted s^* and t^*, which are incident with the duals of the edges on the leftmost and rightmost paths from s to t, respectively. Vertices s^* and t^* can be thought of as being the "left" and "right external face" of G, respectively. It is simple to verify that G^* is a planar *st*-graph with source s^* and sink t^* [41, 51]. Notice that G^* might have multiple arcs.

Let V, E, and F denote the set of vertices, edges, and faces of G, respectively, where F has elements s^* and t^* representing the external face. We will show that the orders $<_L$ and $<_R$ can be extended to the set $V \cup E \cup F$, thereby giving a unique total order of all topological constituents of G.

First, 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 6.2 and Fig. 6.3(a), 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. For $v = s$ or $v = t$, we conventionally define $LEFT(v) = s^*$ and $RIGHT(v) = t^*$.

(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.

(3) If $x = f \in F$ and $f \neq s^*, t^*$, 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 6.3 and Fig. 6.3(b)). Vertices $LOW(f)$ and $HIGH(f)$ are called the *extreme* vertices of face f. For $f = s^*$ or $f = t^*$, $LOW(f)$ and $HIGH(f)$ are undefined. 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 say that x is *below* y, denoted $x \uparrow y$, if there is a path in G from $HIGH(x)$ to $LOW(y)$. Also, we say that x is *to the left of* y, denoted $x \rightarrow y$, if there is a path in G^* from $RIGHT(x)$ to $LEFT(y)$.

For example, in the planar st-graph shown in Fig. 6.2, we have $e_2 \uparrow v_4$, $f_4 \uparrow v_4$, $v_5 \rightarrow f_4$, and $e_1 \rightarrow f_2$.

Lemma 6.6 Relations $\uparrow$ and $\rightarrow$ are partial orders on $V \cup E \cup F$.

Proof: A consequence of the fact the graphs G and G^* are acyclic. □

The following lemma shows that $\uparrow$ and $\rightarrow$ are complementary partial orders.

Lemma 6.7 Let x and y be any two elements of $V \cup E \cup F$. Then one and only one of the following holds:

$$x \uparrow y, \quad y \uparrow x, \quad x \rightarrow y, \quad y \rightarrow x.$$

Proof: We prove the theorem for the case when $y = v$ is a vertex of G. The other two cases can be proved using similar arguments.

Let π_1 and π_2 be the leftmost and rightmost paths from s to v, respectively. Also, let π_3 and π_4 be the leftmost and rightmost paths from v to t, respectively. These paths partition $V \cup E \cup F$ into five subsets, one of which is v, and the others are defined as follows (see Fig. 6.5):

(1) A contains the vertices, edges, and faces enclosed by paths π_1 and π_2, including the vertices and edges of these paths, but excluding v;

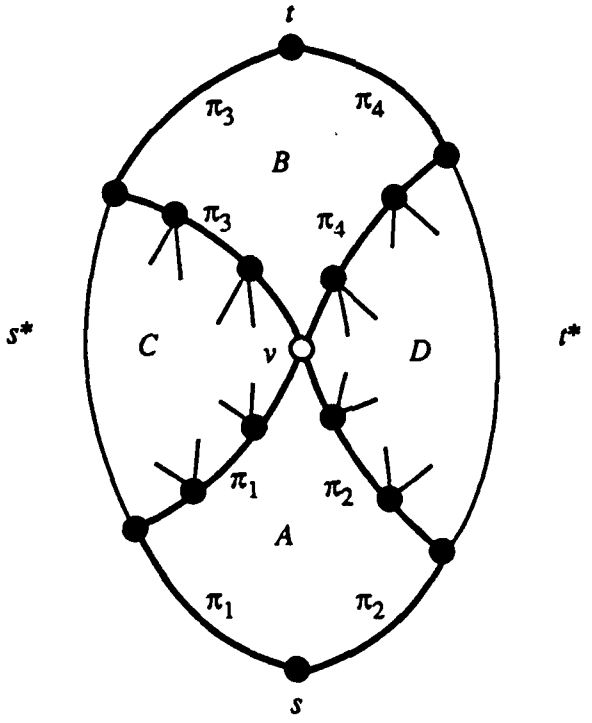


Figure 6.5 Partiton of $V \cup E \cup F$ with respect to vertex v.

(2) B contains the vertices, edges, and faces enclosed by paths π_3 and π_4, including the vertices and edges of these paths, but excluding v;

(3) C contains the vertices, edges, and faces to the left of paths π_1 and π_3, excluding the vertices and edges of these paths;

(4) D contains the vertices, edges, and faces to the right of paths π_2 and π_4, excluding the vertices and edges of these paths.

It is easy to verify that the edges of A are those of a planar st-graph with source s and sink v, which is an induced subgraph of G, and, similarly, the edges of B are those of a planar st-graph with source v and sink t. Notice that the vertices w of A are exactly those such that there is a directed path in G from w to v, and analogously for the vertices of B.

Using simple duality arguments, we can show that the duals of the edges of C are those of a planar st-graph with source s^* and sink $LEFT(v)$, which is an induced subgraph of G^*. Similarly, the duals of the edges of D are those of a planar st-graph with source $RIGHT(v)$ and sink t^*. Notice that the faces f of C are exactly those such that there is a directed path in G^* from f to $LEFT(v)$, and analogously for the faces of D.

By Lemma 6.1, there are directed paths from every vertex of A to v, and from v to every vertex of B. Since for every edge or face x of A (B), both $LOW(x)$ and $HIGH(x)$ are in A (B), we conclude that $x \in A$ implies $x \uparrow v$ and $x \in B$ implies $v \uparrow x$. With similar arguments, we conclude that $x \in C$ implies $x \rightarrow v$ and $x \in D$ implies $v \rightarrow x$.

It remains to be shown that relations $\uparrow$ and $\rightarrow$ are mutually exclusive. Let $x \in A \cup B$, i.e., either $x \uparrow v$ or $v \uparrow x$. Suppose $x \uparrow v$; if $x \rightarrow v$, then there is a path in G^* from $RIGHT(x)$ to $LEFT(v)$. This implies that $RIGHT(x) \in C$, a contradiction. An analogous contradiction is reached if we assume that $x \uparrow v$ and $v \rightarrow x$ jointly hold. Finally, let $x \in C \cup D$, i.e., either $x \rightarrow v$ or $v \rightarrow x$. Suppose $x \rightarrow v$; if $x \uparrow v$,then there is a path in G from $HIGH(x)$ to v. This implies that $HIGH(x) \in A$, a contradiction. An analogous contradiction is reached if we assume that $x \rightarrow v$ and $v \uparrow x$ jointly hold. □

We define relations $<_L$ and $<_R$ on $V \cup E \cup F$, as follows:

$$x <_L y \Leftrightarrow x \uparrow y \text{ or } x \rightarrow y; \quad x <_R y \Leftrightarrow x \uparrow y \text{ or } y \rightarrow x.$$

As a consequence of Lemma 6.7, we obtain

Theorem 6.3 The relations $<_L$ and $<_R$ on $V \cup E \cup F$ are total orders.

We also note that there is a path in G from vertex u to vertex v if and only if $u <_L v$ and $u <_R v$, since such path exists if and only if $u \uparrow v$.

We define the *left-sequence* of G as the sequence of elements of $V \cup E \cup F$, sorted according to $<_L$ (leftist order). The *right-sequence* of G is defined similarly with respect to $<_R$ (rightist order).

For example, the right-sequence of the graph of Fig. 6.2 is:

$$f_5 v_0 e_3 f_4 e_2 v_1 e_4 v_2 e_8 f_3 e_5 v_3 e_7 v_4 e_{10} f_2 e_6 f_1 e_1 v_5 e_9 v_6 f_0.$$

We will use a convenient string notation for such sequences. Namely, we use terminal symbols (lower-case letters) for the elements of $V \cup E \cup F$, and variables (upper-case letters) for substrings of the left- or right-sequence. For example, the left-sequence of the graph of Fig. 6.2 can be represented by the string

$$f_0 v_0 e_1 A v_3 e_6 v_5 e_9 f_2 B$$

where $A = f_1 e_2 v_1 e_5$ and $B = e_7 f_3 e_4 f_4 e_3 v_2 e_8 v_4 e_{10} v_6 f_5$.

6.3. On-Line Maintenance of a Planar *st*-Graph

In this section we define a complete set of update operations on a planar *st*-graph, and show that the restructuring of the orders $<_L$ and $<_R$ resulting from any such update operation can be expressed by means of a simple string transformation. From this result, we derive an efficient data structure for the on-line maintenance of the two orders of a planar *st*-graph.

The update operations on a planar *st*-graph are defined as follows:

INSERTEDGE $(e,u,v,f;f_1,f_2)$: Add edge $e=(u,v)$ inside face f, which is decomposed into faces f_1 and f_2, with f_1 to the left of e and f_2 to the right (see Fig. 6.6(a)).

REMOVEEDGE $(e,u,v,f_1,f_2;f)$: Delete edge $e=(u,v)$ and merge the two faces f_1 and f_2 formerly on the two sides of e into a new face f (see Fig. 6.6(a)).

EXPANDVERTEX $(e,f,g,v;v_1,v_2)$: Expand vertex v into vertices v_1 and v_2, which are connected by a new edge e with face f to its left and face g to its right (see Fig. 6.6(b)).

CONTRACTVERTEX $(e,f,g,v_1,v_2;v)$: Contract edge $e=(v_1,v_2)$, and merge its endpoints into a new vertex v. Faces f and g are to the left and right of e, respectively (see Fig. 6.6(b)). Parallel edges resulting from the contraction are merged into a simple edge.

Each operation is allowed if the resulting graph is itself a planar *st*-graph. It is interesting to observe that operations *EXPANDVERTEX* and *CONTRACTVERTEX* are dual of *INSERTEDGE* and *REMOVEEDGE*, respectively, since performing one on G corresponds to performing the other on G^*.

We say that an edge e of G is *removable*, if operation *REMOVEEDGE* $(e,u,v,f_1,f_2;f)$ on G yields a planar *st*-graph. We say that e is *contractible* if operation *CONTRACTVERTEX* $(e,f,g,v_1,v_2;v)$ on G yields a planar *st*-graph.

Lemma 6.8 Let e be an edge of G.

(1) If e is not removable then it is contractible.

(2) If e is not contractible then it is removable.

Proof: From the definition of planar *st*-graph, it is easy to see that an edge $e=(u,v)$ is removable if and only if $\deg^+(u)\geq 2$ and $\deg^-(v)\geq 2$, and it is contractible if and only if it is not a transitive edge. (1) Assume that edge $e=(u,v)$ is not removable. Then we have $\deg^+(u)=1$ and/or $\deg^-(v)=1$. This implies that there is no other path in G from u to v, so that e cannot be a transitive edge. Hence, edge e is contractible. (2) Conversely, assume that edge $e=(u,v)$ is not contractible. Then e is a transitive edge, which implies $\deg^+(u)\geq 2$ and $\deg^-(v)\geq 2$, so that e is removable. □

A simple induction based on Lemma 6.8 yields

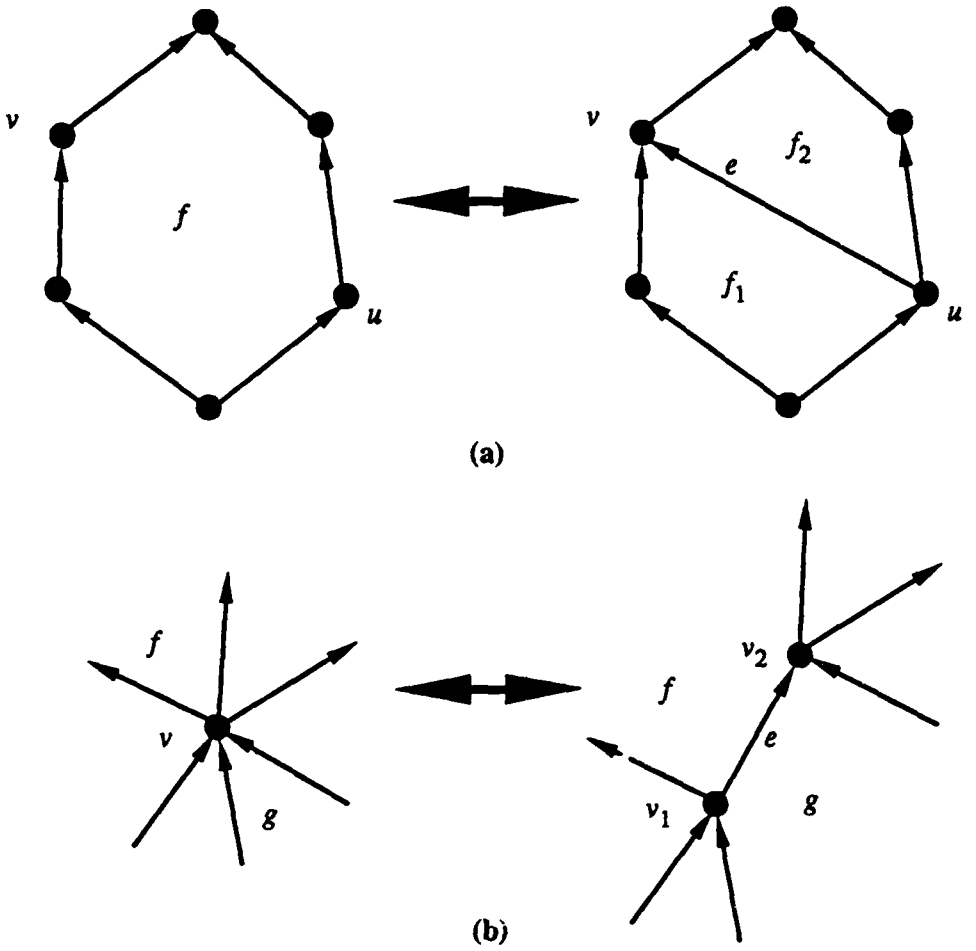


Figure 6.6 (a) Operations *INSERTEDGE* and *REMOVEEDGE*. (b) Operations *EXPANDVERTEX* and *CONTRACTVERTEX*.

Lemma 6.9 Let G_0 be the trivial planar *st*-graph consisting of a single vertex. Any planar *st*-graph with n vertices can be assembled starting form G_0 by means of $O(n)$ *INSERTEDGE* and *EXPANDVERTEX* operations, and can be disassembled to yield G_0 by means of $O(n)$ *REMOVEEDGE* and *CONTRACTVERTEX* operations.

Now, we describe the transformation of the leftist order $<_L$ as a consequence of operations *INSERTEDGE* $(e,u,v,f;f_1,f_2)$. Similar arguments hold for the order $<_R$ and for operation *EXPANDVERTEX* $(e,f,g,v;v_1,v_2)$.

Theorem 6.4 Let G be a planar *st*-graph, and G' be the graph obtained from G after the execution of operation *INSERTEDGE* $(e,u,v,f;f_1,f_2)$. Depending on the relative orders of u, v, and f we have the following transformations (*left-sequence of* G) $\Rightarrow$ (*left-sequence of* G'):

(1) $u <_L v <_L f$: $A\,u\,B\,v\,C\,f\,D \Rightarrow A\,u\,B\,f_1\,e\,v\,C\,f_2\,D$;

(2) $f <_L u <_L v$: $A\,f\,B\,u\,C\,v\,D \Rightarrow A\,f_1\,B\,u\,e\,f_2\,C\,v\,D$;

(3) $u <_L f <_L v$: $A\,u\,B\,f\,C\,v\,D \Rightarrow A\,u\,B\,f_1\,e\,f_2\,C\,v\,D$;

(4) $v <_L f <_L u$: $A\,v\,B\,f\,C\,u\,D \Rightarrow A\,f_1\,C\,u\,e\,v\,B\,f_2\,D$.

Proof: The four cases are illustrated in Fig. 6.7. First, we observe that the union of the elements of $V \cup E \cup F$ associated with any one of the substrings A, B, C, and D, is a topologically connected region of the plane. The above regions, together with u, v, and f, form a partition of the entire plane, which is determined by four paths, and specifically

(*i*) the leftmost path from *HIGH* (f) to t;

(*ii*) the rightmost path from s to *LOW* (f); and

(*iii*) depending respectively on each of the four cases, the following two paths:

(1) the leftmost paths from u to t and from v to t (see Fig. 6.7(a-b));

(2) the rightmost paths from s to u and from s to v (see Fig. 6.7(c-d));

(3) the leftmost path from u to t and the rightmost path from s to v (see Fig. 6.7(e-f));

(4) the leftmost path from v to t and the rightmost path from s to u (see Fig. 6.7(g-h)).

We discuss in detail Case 4 (see Fig. 6.7(g-h)). The proof for the other cases can be derived with similar arguments. The insertion of edge e causes every vertex in C to be connected with a directed path to every vertex of B. At the same time, the insertion of e breaks all the paths of G^* from the faces of B to the faces of C. Hence, we have the following relations:

$$A <_L f_1,\ f_1 \rightarrow C,\ C \uparrow u,\ u \uparrow e,\ e \uparrow v,\ v \uparrow B,\ B \rightarrow f_2,\ f_2 <_L D,$$

where a substring represents compactly all of its elements. These relations yield immediately the updated left-sequence. □

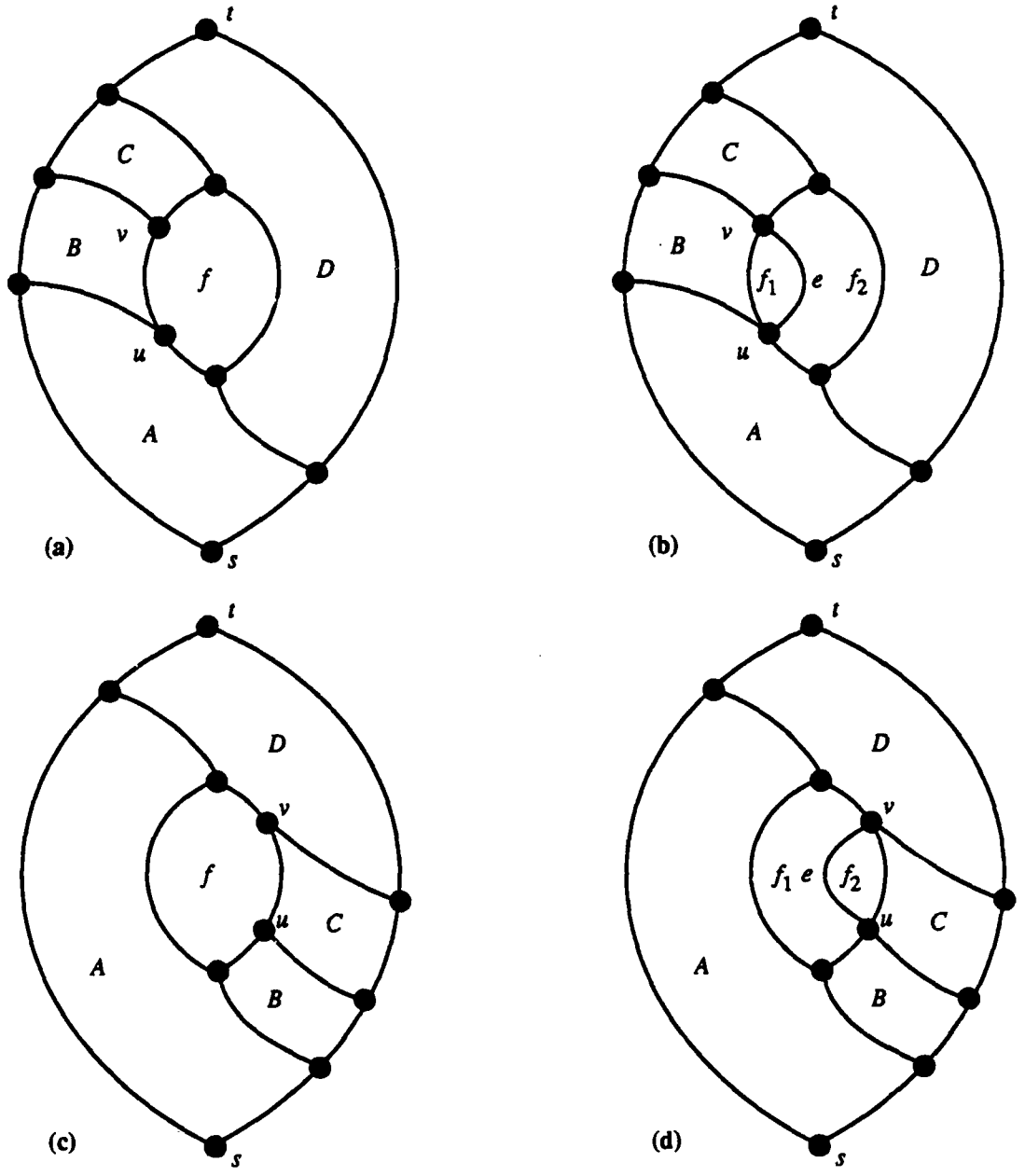


Figure 6.7 Example for Theorem 6.4. (a) Case (1) before insertion. (b) Case (1) after insertion. (c) Case (2) before insertion. (d) Case (2) after insertion.

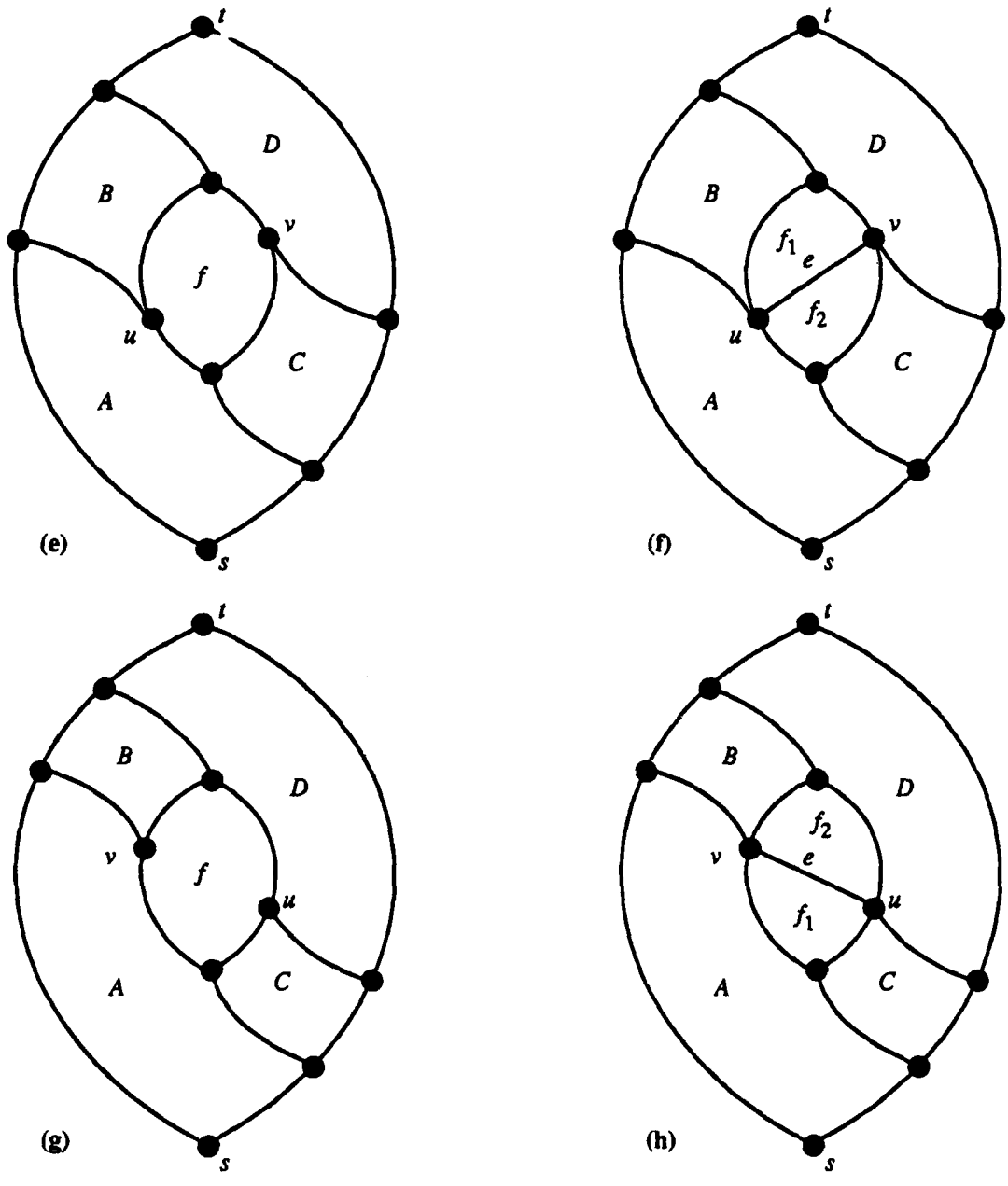


Figure 6.7 (Continued) Example for Theorem 6.4. **(e)** Case (3) before insertion. **(f)** Case (3) after insertion. **(g)** Case (4) before insertion. **(h)** Case (4) after insertion.

Theorem 6.4 shows that the update of the order $<_L$ is a simple syntactic transformation of the left-sequence, consisting of at most four insertions/deletions of elements, and at most one swap of substrings. Since operation *REMOVEEDGE* is the inverse of operation *INSERTEDGE*, the order before and after the deletion can be obtained by reversing the transformations given in Theorem 6.4. The same situation arises with respect to operations *EXPANDVERTEX* and *CONTRACTVERTEX*. We can summarize these results as follows:

Theorem 6.5 Let G be a planar *st*-graph, and G' be the graph obtained from G after update Π, where Π is one of *INSERTEDGE*, *REMOVEEDGE*, *EXPANDVERTEX*, or *CONTRACTVERTEX* operations. Then the left-sequence of G' can be obtained from the left-sequence of G by means of at most four insertions/deletions of elements, and at most one swap of substrings.

Theorem 6.5 allows us to design a simple and yet efficient data structure for maintaining on-line the orders of a planar *st*-graph G. We represent orders $<_L$ and $<_R$ by means of two balanced binary trees (such as *red-black trees* [55, pp. 52-53]), denoted T_L and T_R, where the left-to-right order of the leaves of T_L gives the left-sequence of G, and the left-to-right order of the leaves of T_R gives the right-sequence of G. From Euler's formula, trees T_L and T_R have $O(n)$ nodes, so that their depth is $O(\log n)$.

An *order-query* on a planar *st*-graph G consists of determining, given elements x and y of $V \cup E \cup F$, whether $x <_L y$ or $y <_L x$, and similarly with respect to order $<_R$.

Lemma 6.10 Let T be a binary tree, λ_1 and λ_2 two leaves of T, and μ the lowest common ancestor of λ_1 and λ_2. Then λ_1 precedes λ_2 in the left-to-right order if and only if λ_1 is in the left subtree of μ (and λ_1 is in the right subtree of μ).

Proof: The theorem is an immediate consequence of the following recursive definition of the left-to-right order of the leaves of T:

(1) If leaf λ_1 is in the left subtree of T and λ_2 is the right subtree, then λ_1 precedes λ_2.

(2) If λ_1 and λ_2 are in the same subtree T' of T, say the left subtree, then λ_1 precedes λ_2 in T if and only if λ_1 precedes λ_2 in T'. □

Theorem 6.6 An order-query can be executed in $O(\log n)$ time.

Proof: The order-query algorithm is as follows. We access the leaves of tree T_L associated with elements x and y, and we trace the paths p_x and p_y from these leaves to the root of T_L. This allows us to find the lowest common ancestor μ of leaves x and y. From Lemma 6.10, we have that $x <_L y$ if and only if the node of p_x immediately preceding μ is the left child of μ. Since paths p_x and p_y have length $O(\log n)$, we obtain the stated time bound. □

Let T be a balanced binary tree. The left-to-right sequence of the leaves of T will be denoted by $\Lambda(T)$. Two basic operations on balanced binary trees are defined as follows:

SPLIT $(T,\lambda;T_1,T_2)$: Construct from tree T two balanced binary trees T_1 and T_2, such that $\Lambda(T_1)$ is the portion of $\Lambda(T)$ from its leftmost leaf to λ, and $\Lambda(T_2)$ is the remaining portion of $\Lambda(T)$. Tree T is destroyed by the operation.

SPLICE $(T_1,T_2;T)$: Construct from the balanced binary trees T_1 and T_2 a new balanced binary tree T such that $\Lambda(T)$ is the concatenation of $\Lambda(T_1)$ and $\Lambda(T_2)$, with $\Lambda(T_1)$ occurring to the left of $\Lambda(T_2)$. Trees T_1 and T_2 are destroyed by the operation.

Let m be the number of leaves of tree T. Standard techniques allow the performance of each of the above operations in $O(\log m)$ time [55, pp. 52-53].

Regarding the update operations on the planar *st*-graph G, the syntactic transformations on the left- and right-sequence of G correspond to performing $O(1)$ insertions/deletions and *SPLIT/SPLICE* operations on the trees T_L and T_R. Notice that the elements of $V \cup E \cup F$ involved in the update identify the elements of the left-sequence that are inserted, deleted, or are at the boundary of substrings to be swapped. For example, the algorithm for operation *INSERT-EDGE* is as follows:

Algorithm *INSERTEDGE* $(e,u,v,f;f_1,f_2)$

(1) Determine the relative order of u, v, and f in the left-sequence of G by applying the order-query algorithm outlined in the proof of Theorem 6.6. This determines which of the four cases of Theorem 6.4 applies.

(2) Access leaves u, v, and f in tree T_L and remove them. Also, by means of at most three *SPLIT* operations, construct from T_L four trees associated with substrings A, B, C, and D.

(3) Destroy leaf f and create new leaves f_1 and f_2.

(4) Assemble the updated tree T_L from the leaves u, v, f_1, and f_2, and from the trees associated with A, B, C, and D by a sequence of *SPLICE* operations and insertions. The correct left-to-right order of these constituents is selected according to the specifications of Theorem 6.4.

(5) Perform the above Steps 1-4 on the right-sequence and tree T_R.

Analogous algorithms can be formulated for the other update operations, and we have

Theorem 6.7 The restructuring of trees T_L and T_R after any one of the update operations *INSERTEDGE*, *REMOVEEDGE*, *EXPANDVERTEX*, and *CONTRACTVERTEX* can be performed in $O(\log n)$ time.

6.4. Applications

The general framework for the maintenance of orders $<_L$ and $<_R$ in a planar *st*-graph can be profitably used in three interesting applications: (*i*) dynamic point location in monotone subdivisions, (*ii*) dynamic transitive-closure query in planar *st*-graphs, and (*iii*) dynamic contact-chain query in convex subdivisions.

6.4.1. Dynamic planar point location revisited A monotone subdivision $\Re$ is associated with a planar *st*-graph G such that (see Fig. 6.8):

(1) the vertices of G are the vertices of $\Re$, plus two special vertices s and t, associated with vertices at infinity in the vertical direction;

(2) the arcs of G are associated with the edges of $\Re$, and oriented from the lower to the upper endpoint (horizontal edges are oriented from left to right); also G contains arcs connecting consecutive vertices of $\Re$ at infinity.

Note that the vertices on the external boundary of G are the vertices of $\Re$ at infinitity, plus s and t. Since monotone chains in $\Re$ correspond to directed paths in G, Theorem 3.2 of Chapter 3 can

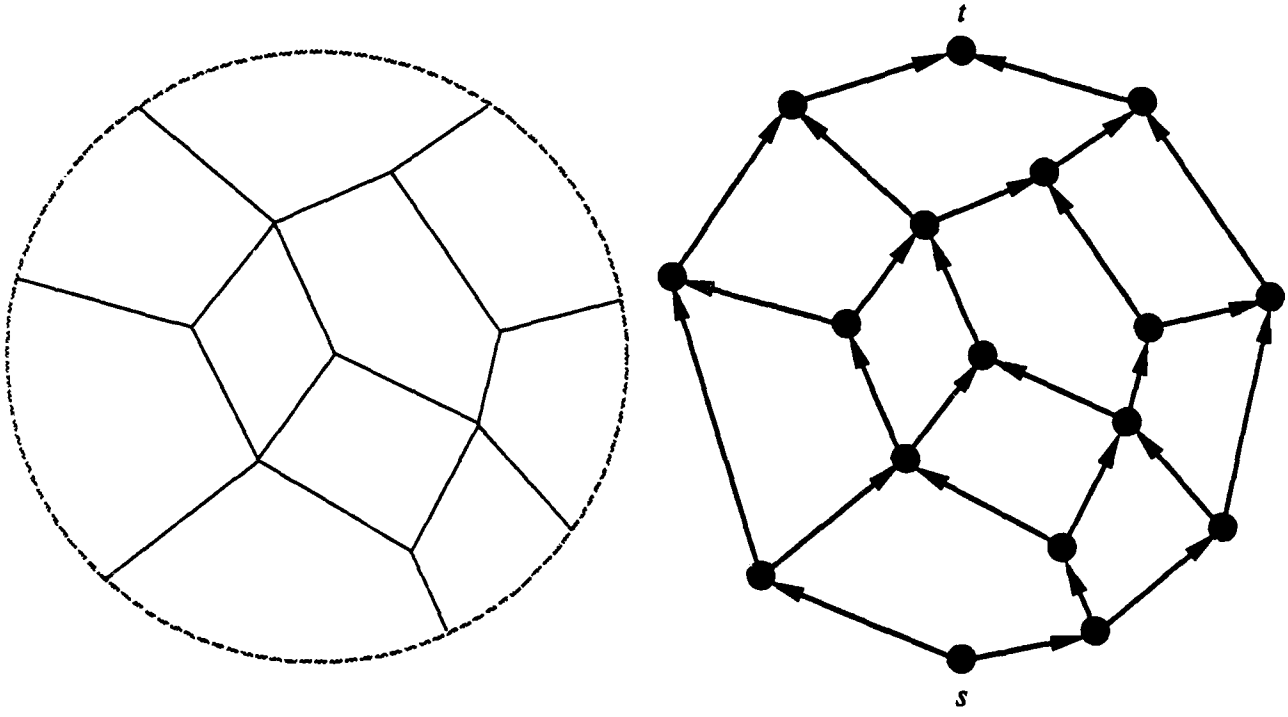


Figure 6.8 (a) Monotone subdivision; (b) The planar *st*-graph associated with the monotone subdivision of part (a).

be viewed as the geometric counterpart of Lemma 6.9.

Let $r_1, r_2, \cdots, r_f$ be the regions of $\mathfrak{R}$, sorted according to some total order $<$ compatible with relation $\rightarrow$, i.e., $r_i \rightarrow r_j$ implies $i < j$. The common boundary of the regions with index less than or equal to i and of the ones with index greater than i is a separator of $\mathfrak{R}$ denoted σ_i. Clearly, σ_i is to the left of σ_j for $i < j$. Hence, the order $<$ defines a complete family of separators $\Sigma = \{\sigma_1, \cdots \sigma_{f-1}\}$. Notice that region r_i is the portion of the plane between separators σ_{i-1} and σ_i. Conversely, a complete family $\Sigma = \{\sigma_1, \cdots \sigma_{f-1}\}$ of $f-1$ separators defines a total order $r_1, r_2, \cdots, r_f$ on the regions compatible with relation $\rightarrow$, where r_i is the unique region contained between separators σ_{i-1} and σ_i.

In the point location technique of Lee-Preparata [35] the leaves of the separator-tree $\mathfrak{I}$ are the regions of $\mathfrak{R}$, sorted from left to right according to $<$, and the sequence of the nodes of $\mathfrak{I}$ in

symmetric order is $r_1\sigma_1 r_2 \cdots \sigma_{f-1} r_f$. A *REMOVECHAIN* operation might cause the order of the regions given by $\Im$ to become inconsistent with relation $\rightarrow$ (see Fig. 6.9), and the consequent rearrangement of this order might require the reconstruction of large portions of $\Im$. Our dynamic point location technique overcomes the above obstacle using the leftist order $<_L$ on the regions of $\Re$ to define the family of separators Σ, so that the rearrangement of the separator tree after an update can be performed by a simple transformation.

With regard to channels, we observe that the channel between two vertically consecutive regions r_1 and r_2 consists exactly of the vertices and edges between $HIGH(r_1)$ and $LOW(r_2)$ in the left-sequence of G. For example, the left-sequence of the subdivision of Fig. 6.10 (omitting

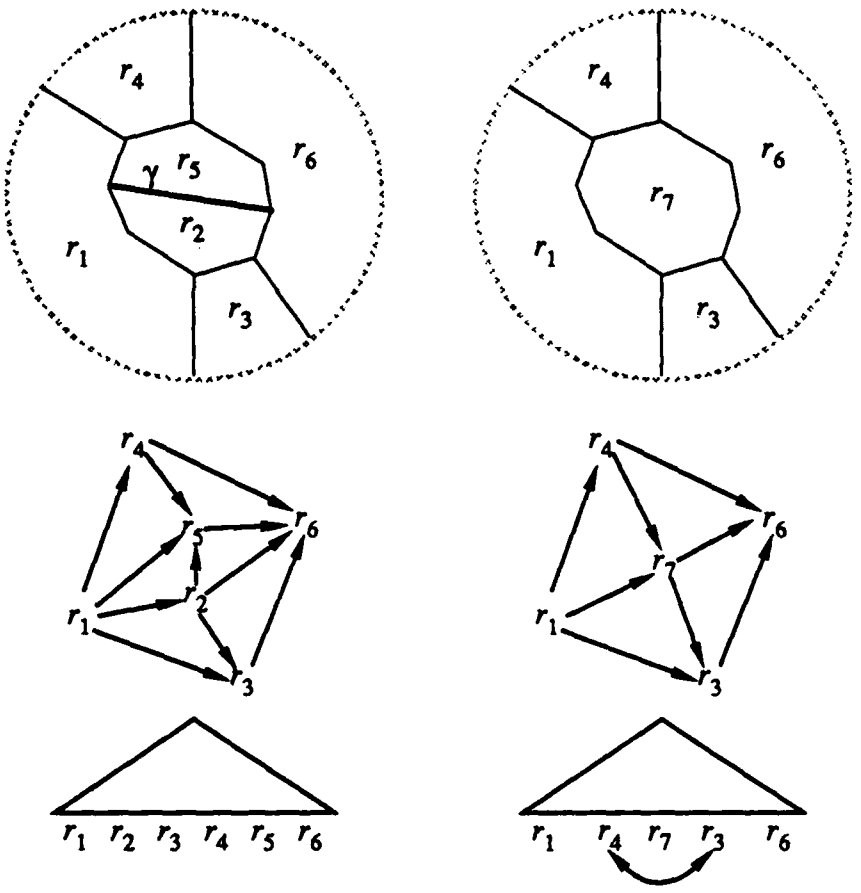


Figure 6.9 Example of *REMOVECHAIN* operation which causes the order of the regions given by the separator-tree to become inconsistent with relation $\rightarrow$.

vertices and edges at infinity) is given by the string $r_1 e_1 r_2 e_2 A e_5 r_3 e_6 r_4$, where the substring $A = v_1 e_3 v_2 e_4 v_3$ is associated with the channel between r_1 and r_2. With regard to the *INSERTCHAIN* operation, we observe that Theorem 3.8 of Chapter 3 is the geometric counterpart of Theorem 6.4.

6.4.2. Transitive-closure query Recall that a transitive-closure query on a planar *st*-graph G consists of determining the existence of a directed path between vertices u and v of G. Such query is equivalent to testing whether both $u <_L v$ and $u <_R v$ so that, by Theorem 6.6, it takes $O(\log n)$ time. This establishes Theorem 6.1 of Section 6.1.

A variant of query reports a path between u and v, and can be executed in time $O(\log n + k)$, where k is the number of path edges. First, we query (in $O(\log n)$ time) the existence of a path between u and v. Suppose that such path exists and, say, $u \uparrow v$. We know that the leftmost path from u to t and the leftmost path from s to v have at least one vertex in common. Resorting to a standard DCEL representation of the planar *st*-graph (see [47, pp. 15-17]), we can trace each of these two paths. Alternating between them one edge at a time, we trace the

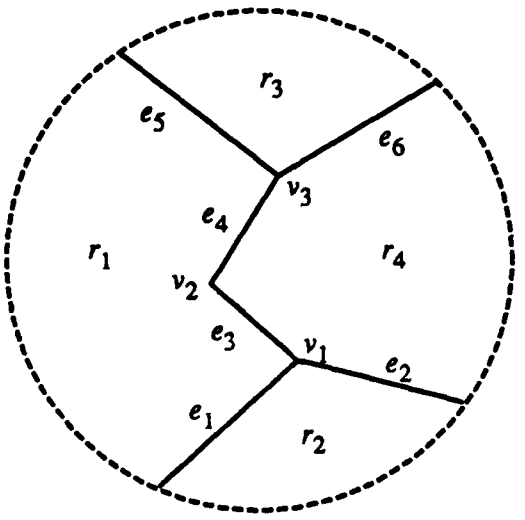


Figure 6.10 Example of channel.

path between u and t forward from u, and the path between s and v backward from v. In this traversal we mark each visited vertex. The process terminates when we reach a vertex for the second time. If k is the length of the path to be reported, clearly at most $2k$ vertices have been visited by the process. This establishes that the report-type query is executed in time $O(\log n + k)$.

6.4.3. Contact-chain query We can reformulate the problem of contact chains by assuming without any sacrifice of generality that the reference direction θ is always the x-axis. In this setting, we have that region r_1 pushes region r_2 if and only if r_1 is to the left of r_2. Hence, the transitive closure of the "push" relation is the same as relation $\rightarrow$, and variations of θ correspond to rotations of the subdivision.

We assume, with negligible loss of generality, that the slopes of the edges are all distinct. (In the case of parallel edges, a virtual perturbation of their slopes achieves this simplifying condition.) Thus, if we continuously rotate the subdivision, only one edge at a time becomes horizontal. An *elementary clockwise rotation* from a given position of $\mathfrak{R}$ is the minimal nonzero clockwise rotation such that an edge becomes horizontal. An *elementary counterclockwise rotation* is correspondingly defined. Thus, a full 2π-rotation of $\mathfrak{R}$ is a sequence of elementary rotations.

Since a convex subdivision $\mathfrak{R}$ is also a monotone subdivision, we consider the planar *st*-graph G associated with $\mathfrak{R}$, and its dual G^*. It is easy to see that contact chains of $\mathfrak{R}$ are in one-to-one correspondence with paths in the graph G^*.

We consider the following update operations on $\mathfrak{R}$:

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

REMOVEPOINT $(v,e_1,e_2;e)$: Let v be a vertex of degree 2 whose incident edges, $e_1=(u,v)$ and $e_2=(v,w)$, are on the same straight line. Remove v and replace e_1 and e_2 with edge $e=(u,w)$.

INSERTSEGMENT $(e,u,v,r;r_1,r_2)$: Add edge $e=(u,v)$ inside region r, which is decomposed into regions r_1 and r_2, with r_1 to the left of e and r_2 to the right

REMOVESEGMENT $(e,u,v,r_1,r_2;r)$: Remove edge $e=(u,v)$ and merge the regions r_1 and r_2 formerly on the two sides of e into region r. [The operation is allowed only if the subdivision $\mathfrak{R}'$ so obtained is convex.]

ROTATE (δ): Perform an elementary rotation of the subdivision $\mathfrak{R}$. The binary parameter δ indicates whether the rotation is clockwise or counterclockwise.

To maintain information on the paths of G^*, we use the theoretical framework developed in Sections 6.2 and 6.3, and exchange the roles of G and G^*. Operations *INSERTPOINT* and *REMOVEPOINT* on $\mathfrak{R}$ correspond to performing operations *INSERTEDGE* and *REMOVEEDGE* on G^*. Operations *INSERTSEGMENT* and *REMOVESEGMENT* on $\mathfrak{R}$ correspond to performing operations *EXPANDVERTEX* and *CONTRACTVERTEX* on G^*. These correspondences yield algorithms for performing contact-chain queries and insertions/deletions of vertices and edges in time $O(\log n)$.

With regard to the operation *ROTATE*, let e be the edge of $\mathfrak{R}$ that becomes horizontal at some time during the rotation. The effect of such rotation on G^* is to invert the direction of the dual edge e^* of e (see Fig. 6.11). Hence, operation *ROTATE* on $\mathfrak{R}$ corresponds to performing a *REMOVEEDGE* operation on G^*, followed by an *INSERTEDGE* operation of the same edge in the reverse orientation.

Let the azimuth of a directed edge be defined counterclockwise with respect to the x-axis, so that it lies in the range $[0,\pi]$. The edge e involved in the rotation can be identified by maintaining a list of the edges of $\mathfrak{R}$ sorted by increasing azimuth. Specifically, the edge involved in a clockwise (counterclockwise) elementary rotation is the first (last) edge of this list, and is moved to the end (front) of the list after the rotation. The list is implemented as a balanced binary tree, so that edges can be efficiently inserted/deleted as specified by the operations *INSERTPOINT*, *REMOVEPOINT*, *INSERTSEGMENT*, and *REMOVESEGMENT*.

In conclusion, all the update operations have $O(\log n)$ time complexity, which establishes Theorem 6.2 of Section 6.1.

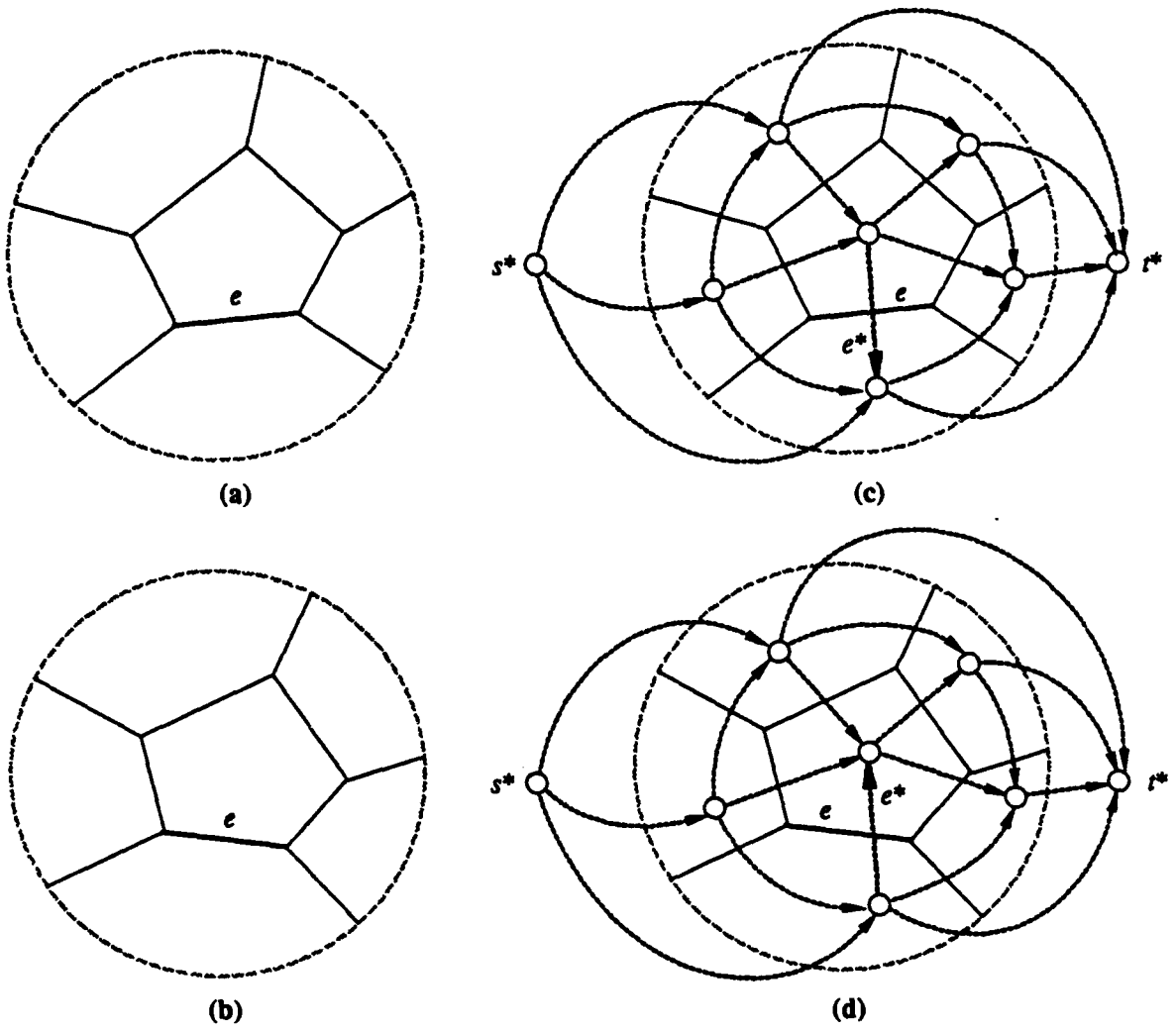


Figure 6.11 (a) Convex subdivision $\Re$. (b) Subdivision $\Re$ after an elementary clockwise rotation (edge e becomes horizontal at some time during the rotation). (c) Graph G^* before the rotation. (d) Graph G^* after the rotation (the orientation of edge e^* is reversed).

CHAPTER 7

DYNAMIC PLANAR GRAPH EMBEDDING

7.1. 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 and of constructing a planar embedding of a graph 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 [7, 24]. Nevertheless, as confirmed by recent results [9, 16, 17], graph planarity is still a vital area of research, rich in interesting issues to be explored.

In this chapter 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.

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 $e=(u,w)$.

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 defined in Section 7.2.

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 [49, 51, 52].

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 chapter is organized as follows: Section 7.2 contains definitions and preliminary results. Section 7.3 deals with orientations of planar graphs. In Section 7.4, we study the *topological location problem*, which consists of performing operations *TEST* and *LIST*. Section 7.5 describes the full-fledged data structure for the dynamic embedding problem. Finally, further applications are discussed in Section 7.6.

7.2. 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 [6, 15]. Unless otherwise specified, paths and cycles of a directed graph are assumed to be directed.

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. 7.1). A graph G is *st-2-connectible* if adding the edge (s,t) to G makes G 2-connected [36]. 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 [36].

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.

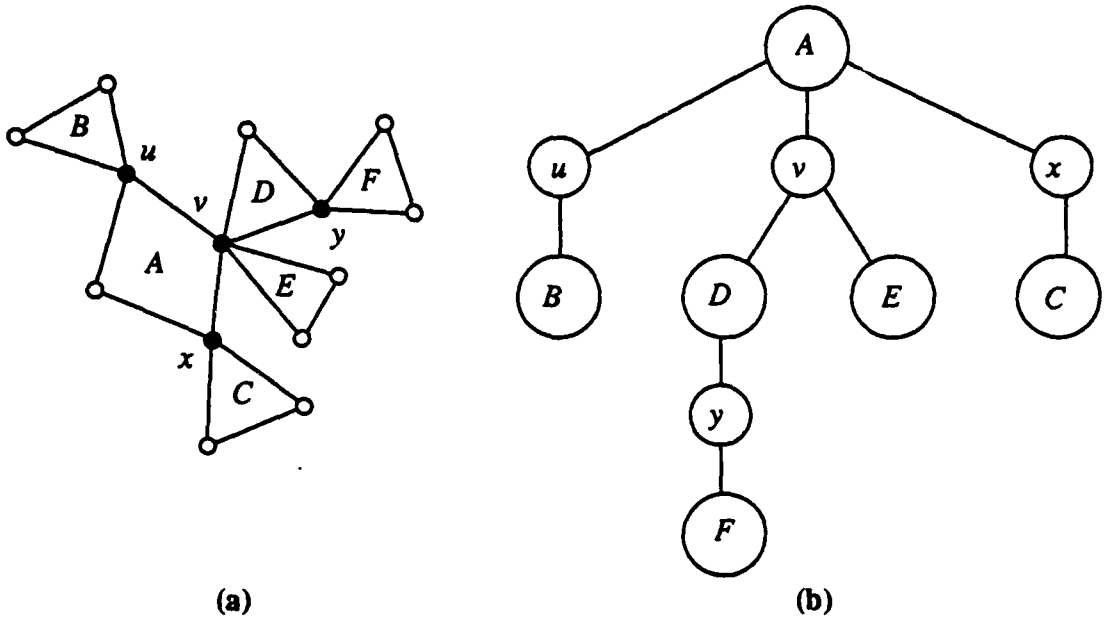


Figure 7.1 (a) A 1-connected graph, and (b) its block-cutvertex tree.

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 chapter, $\log x$ means $\max\{1, \log_2 x\}$.

7.3. 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. 7.2).

The concept of spherical *st*-graph extends the one of *planar st-graph* introduced in [36], which has important applications in the test of graph planarity [36] and the construction of planar drawings [10, 49, 51] (see Chapter 6 for the definition and properties of planar *st*-graphs). 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 (Lemma 6.2 of Chapter 6), and that form the boundary of a face (Lemma 6.3 of Chapter 6).

Lemma 7.1 For every vertex v of G, the incoming (outgoing) edges appear consecutively around v (See Fig. 7.2(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. 7.3). 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 are in A, then π_4 intersects γ at some vertex (see Fig. 7.3(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. 7.3(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. □

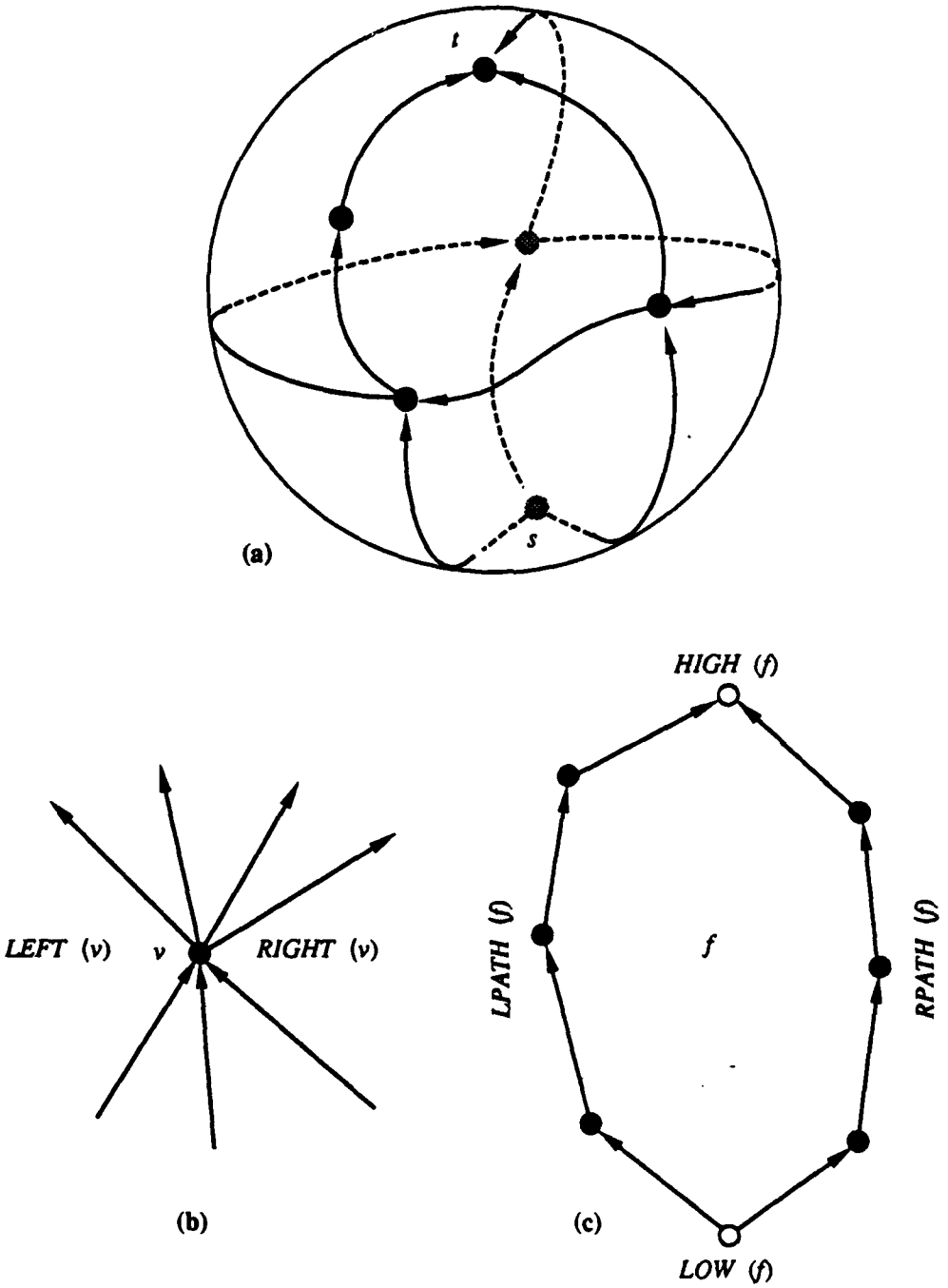


Figure 7.2 (a) Example of spherical *st*-graph. (b) Example for Lemma 7.1. (c) Example for Lemma 7.2.

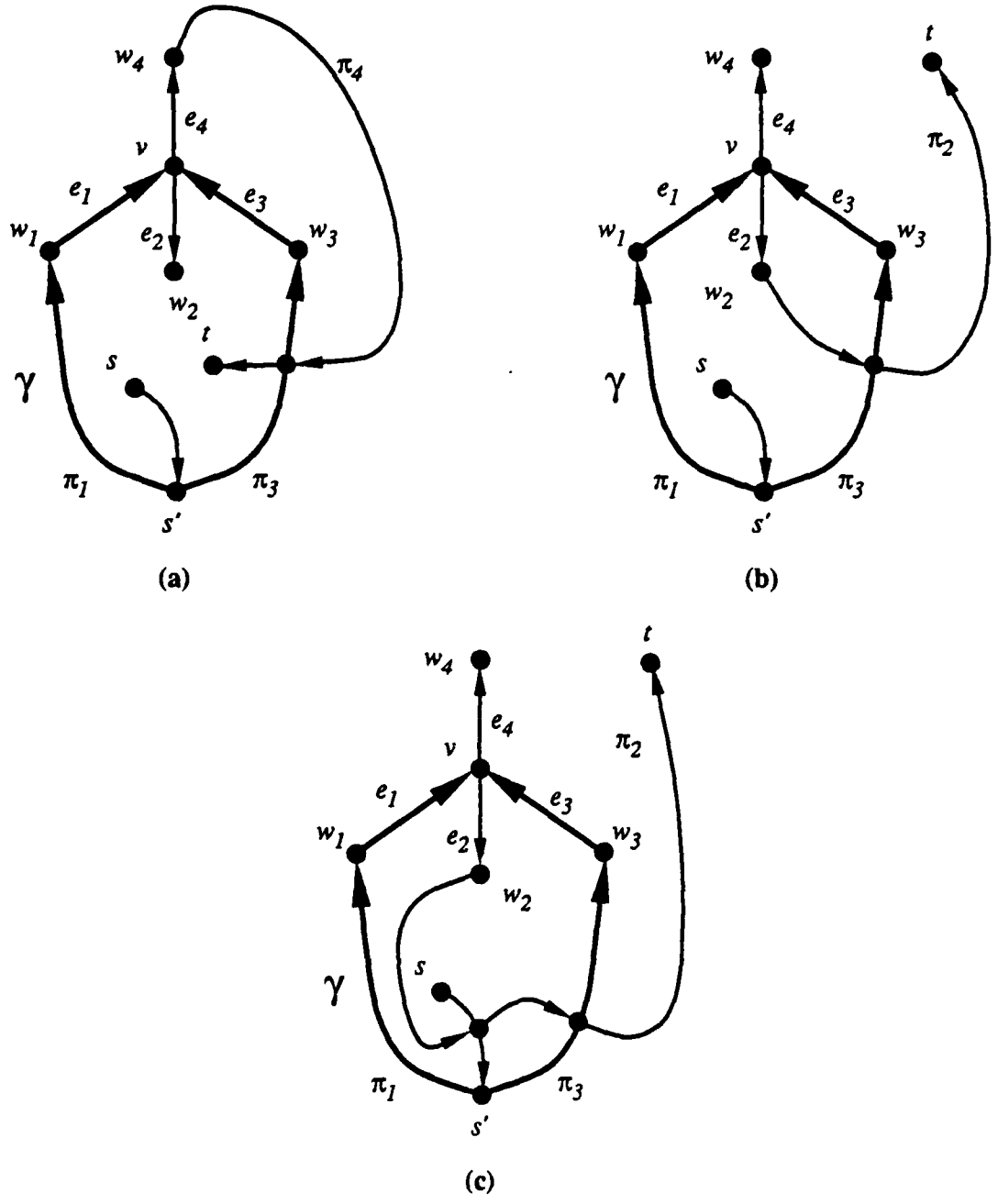


Figure 7.3 Examples for the proof of Lemma 7.1 **(a)** s and t in A; **(b-c)** s in A and t in B.

Lemma 7.2 For every face f of G, the boundary of f consists of two directed paths with common origin and destination (See Fig. 7.2(c)).

Proof: Assume, for a contradiction, that there is a face f for which the lemma is not true (see Fig. 7.4). 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. 7.4(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. 7.4(b-c)).

(*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. 7.4(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. 7.4(e)). The last subcase to examine is when π_2 intersects first π_v. Let r be the first

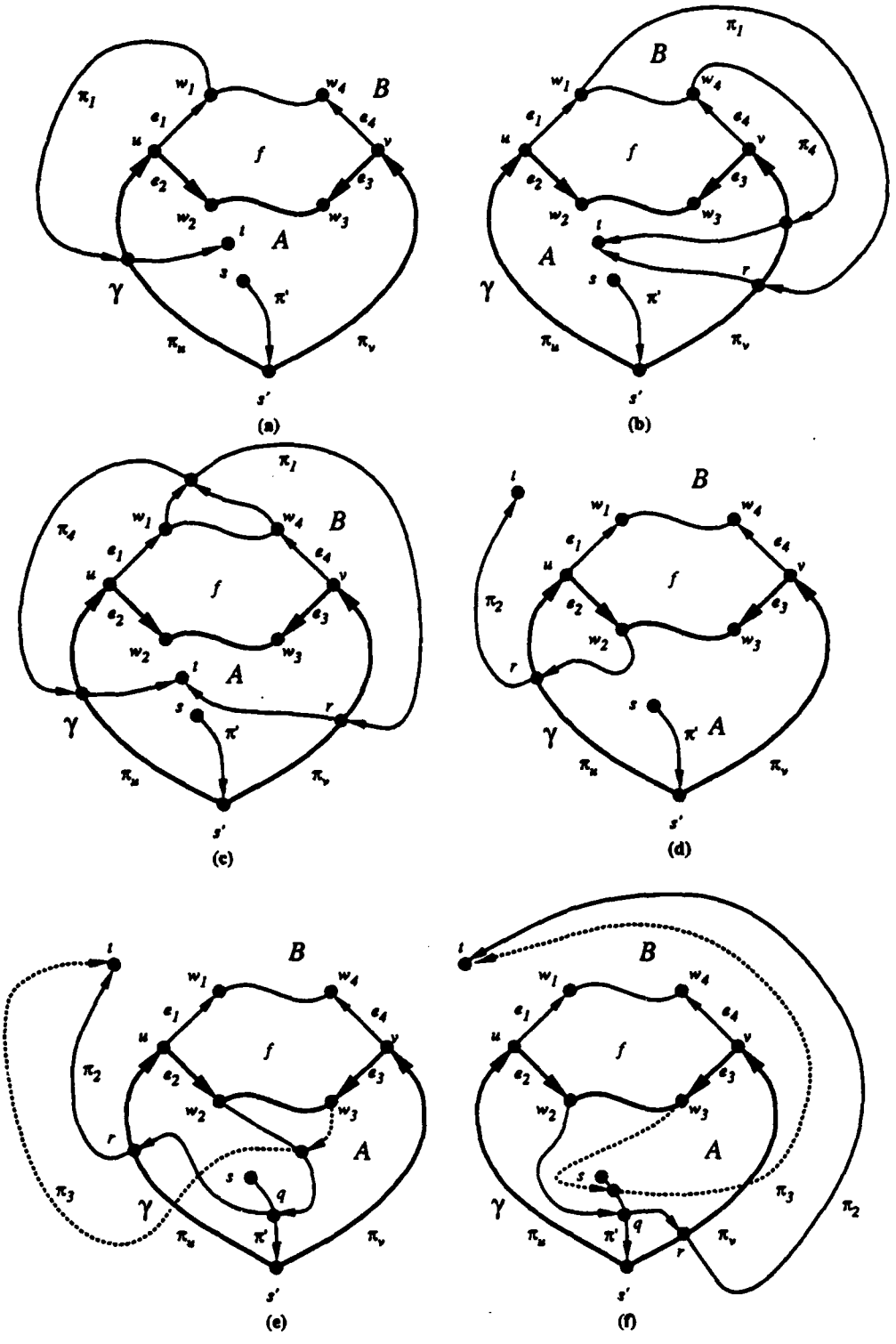


Figure 7.4 Examples for the proof of Lemma 7.2 **(a-c)** Case (*i*). **(d-f)** Case (*ii*).

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. 7.4(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 spherical *st*-graphs the same terminology as for planar *st*-graphs, given in Section 6.2 of Chapter 6, which is repeated here for the reader's convenience. 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 7.1 and Fig. 7.2(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.

(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 7.2 and Fig. 7.2(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 [36].

Theorem 7.1 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 [36] 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 [14], 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 7.3 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 $\deg^-(v) \geq 2$ (where, as usual, $\deg^+(w)$ and $\deg^-(w)$ denote the *outdegree* and *indegree* of a vertex w, respectively);

(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. 7.5. 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 7.1 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.

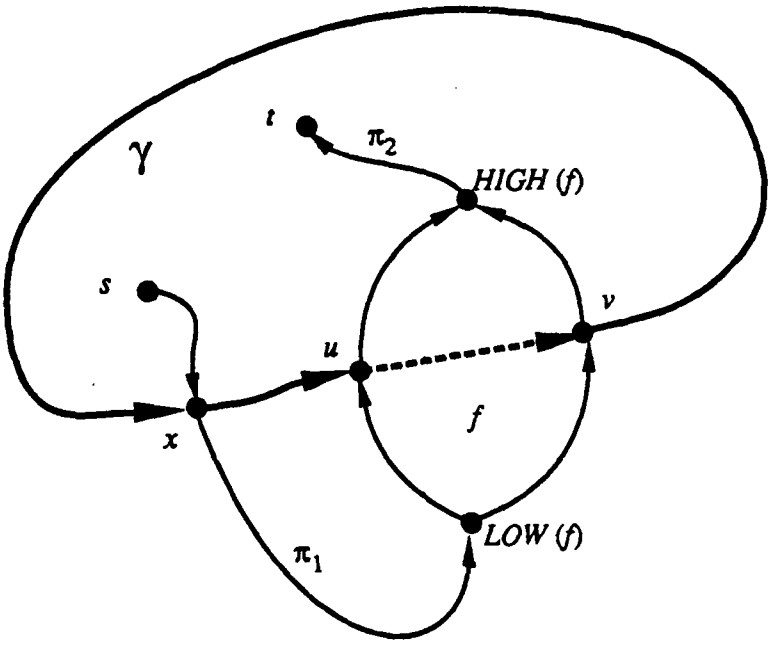


Figure 7.5 Example for the proof of Lemma 7.3

Proof: By Lemma 7.3, 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. □

7.4. 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.

7.4.1. *st*-orientable graphs Let G be a spherical st-graph. From Lemma 7.2, 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. 7.6):

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

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

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

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

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

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

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

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

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

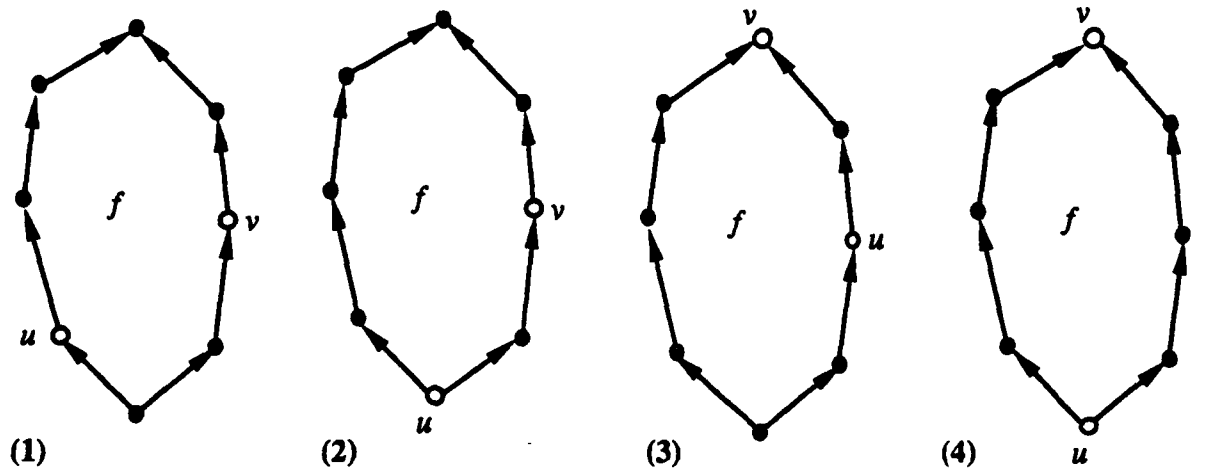


Figure 7.6 The four cases for two vertices on the same face

proves the following theorem:

Theorem 7.2 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 7.2 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 7.2. □

7.4.2. 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. 7.1, 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 7.4 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 7.5 [22, 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 7.6 The graph of relation *OUTER* is a directed source tree, whose root is the external face.

Figure 7.7 shows a 1-connected graph and the corresponding *OUTER* relation for the blocks, cutvertices, and cutfaces.

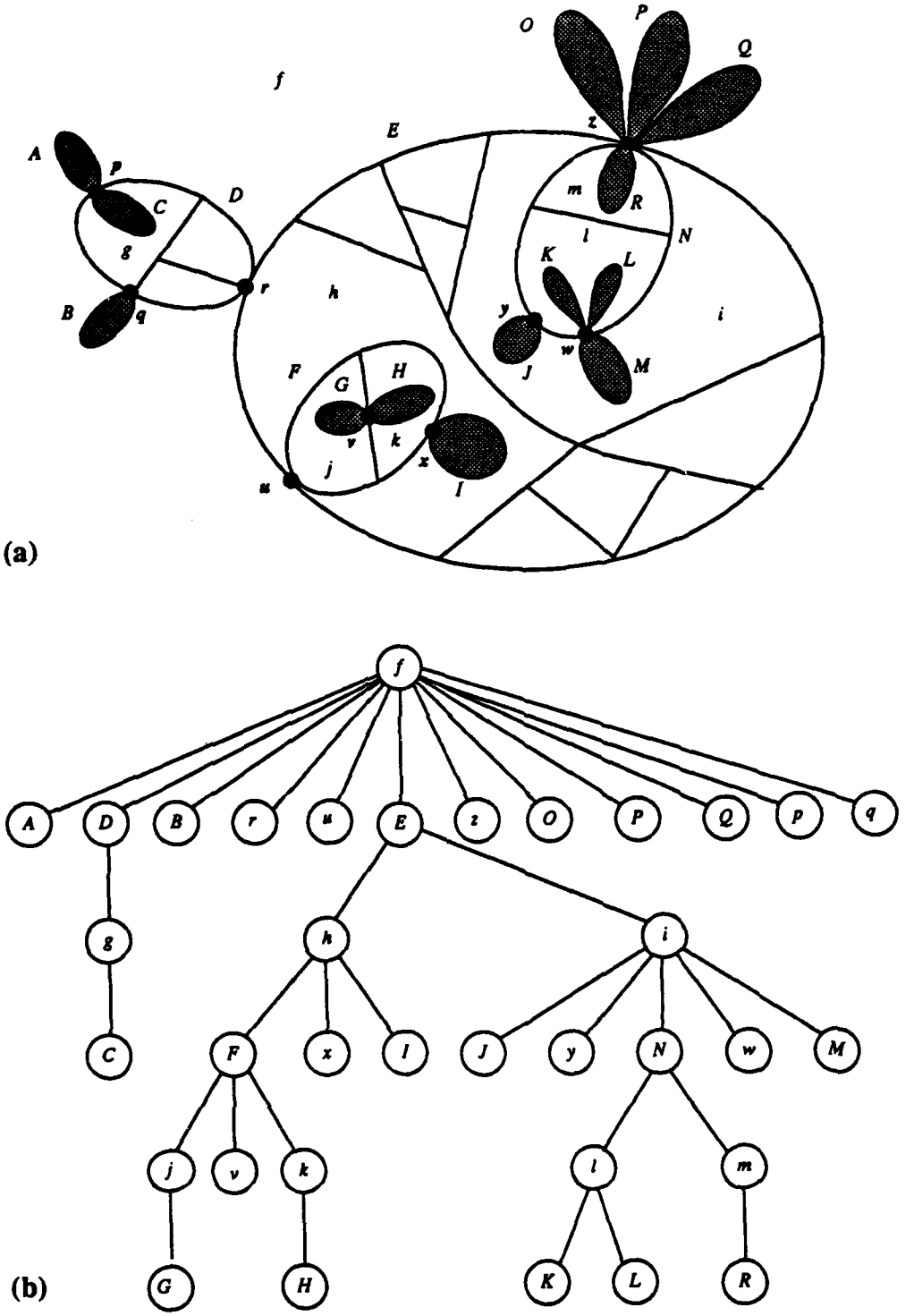


Figure 7.7 (a) A 1-connected graph and (b) its *OUTER* relation.

Theorem 7.3 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 $OUTER(f)$.

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 7.4.

(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 7.3, we conclude

Theorem 7.4 There exists a data structure that solves the topological location problem for general plane graphs with the following performance:

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

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

Theorem 7.5 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 $\overline{Q}(n,m) = O\left(\log \frac{m}{n}\right)$. 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 $\sum_v \deg^-(v)=m$. □

7.5. 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.

7.5.1. 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. 7.8 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. 7.9):

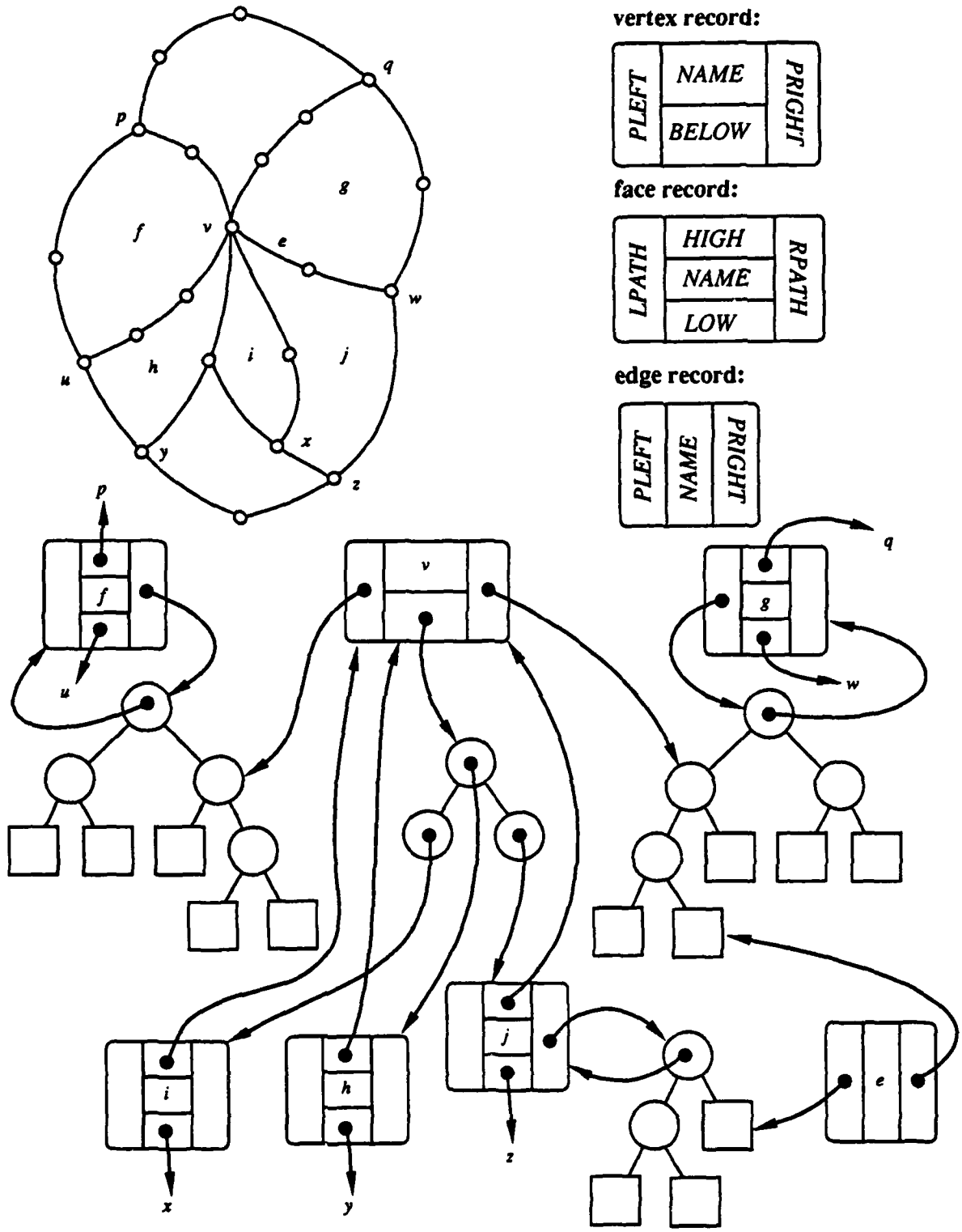


Figure 7.8 A spherical *st*-graph and a fragment of the dynamic data structure for it. The edges are oriented upward.

$$LPATH(f_1)=L_2;\quad RPATH(f_1)=R_1eR_3;\quad LPATH(f_2)=L_1eL_3;\quad RPATH(f_2)=R_2.$$

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 [39, pp. 213-216]).

The remaining updates of the data structure are as follows:

(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.

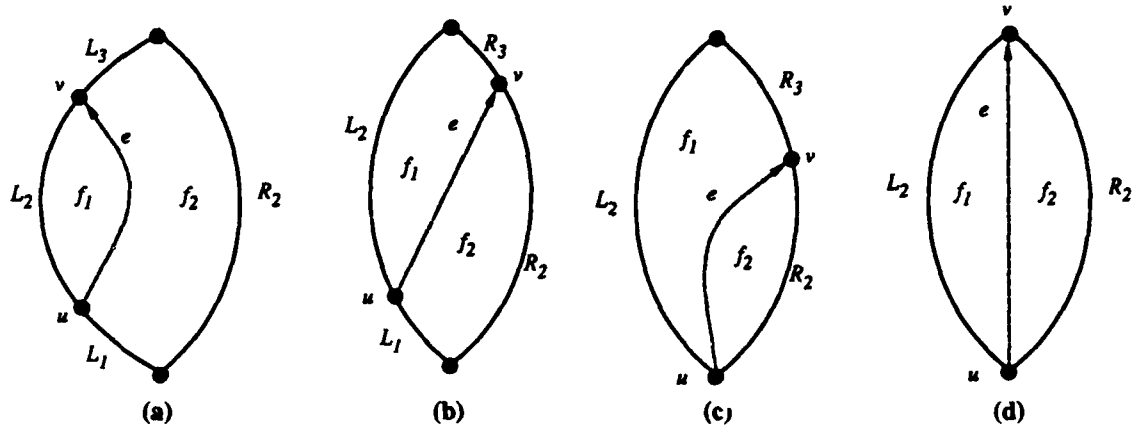


Figure 7.9 Restructuring of the face boundaries after an *INSERTEDGE* operation. **(a)** u and v on left path; **(b)** u on left path and v on right path; **(c)** u bottommost and v on right path; **(d)** u bottommost and v topmost.

(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 $LPATH(g)$.

(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 7.3, 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 7.6 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 [56].

Theorem 7.7 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 7.6.

(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* [20]) to realize trees T_F, T_V, T_E, and *BELOW*(v). Such trees have $O(1)$ amortized rebalancing time for insertions and deletions [26]. 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 [23]. 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 [23] 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 $O(1)$. □

7.5.2. *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 7.1, this can be done easily by reversing the direction whenever the *INSERTEDGE* algorithm rejects the operation. This proves

Theorem 7.8 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 7.9 The data structure of Theorem 7.8 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.

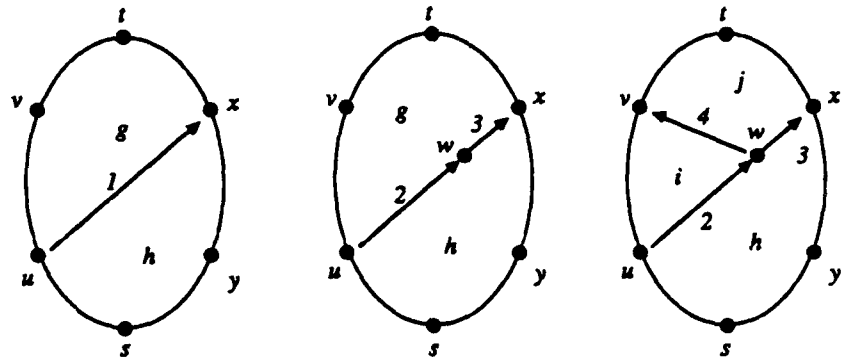
In the following, we show that the class of free edges and vertices is sufficiently large to support a hierarchical deletion scheme that allows us to "undo" any *INSERTEDGE* and *INSERTVERTEX* operation performed in the past (not just the last operation). For instance, we define the *hierarchical embedding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

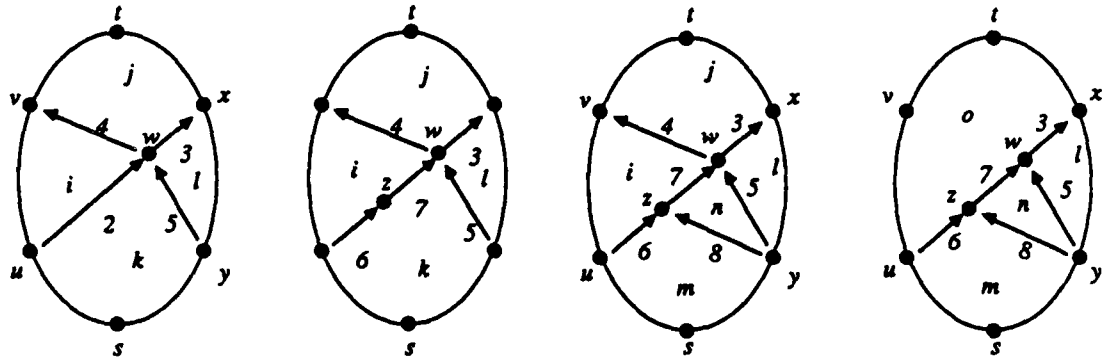
(1) An edge can be deleted by a *REMOVEEDGE* operation only if it was created (at any time in the past) by means of an *INSERTEDGE* operation.

(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. 7.10 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. 7.10, a flag in the "set" condition is indicated by showing the corresponding endpoint unconnected.





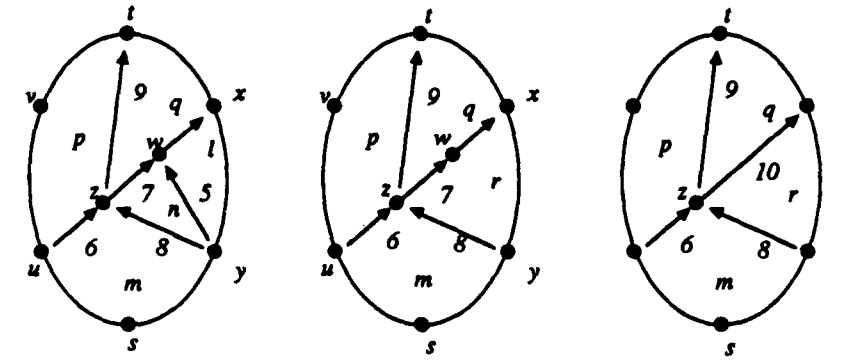


Figure 7.10 Sequence of hierarchical update operations on a plane graph. Flags in the "set" condition are shown by disconnecting the corresponding endpoint.

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

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

7.5.3. 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 [56], 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 7.7 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. 7.11(a) we have

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

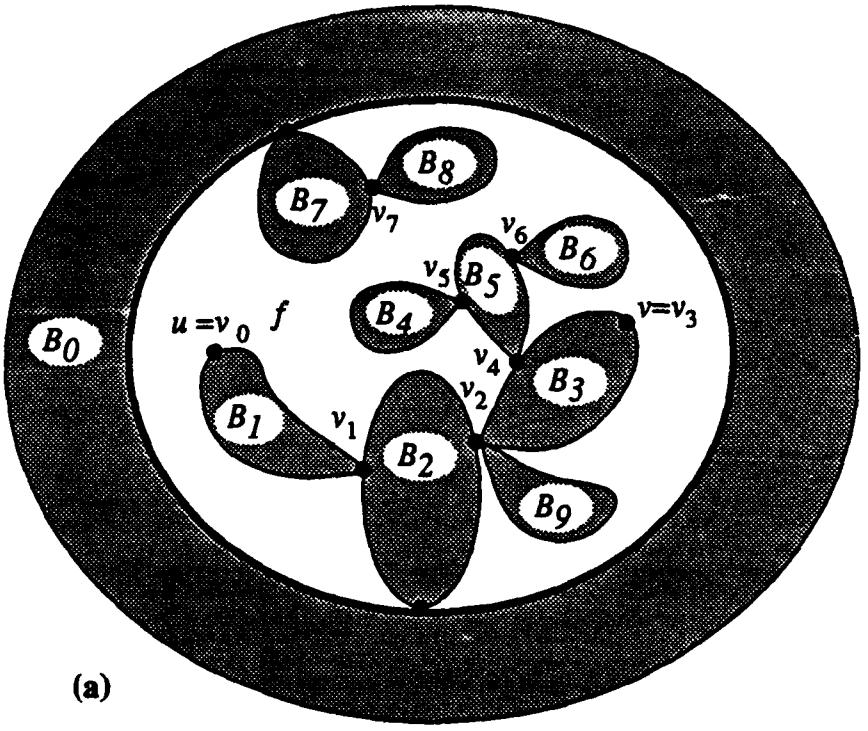
$$EXT(B_2)=(v_1,v_2);\ EXT(B_3)=(v_4);\ EXT(B_5)=(v_5,v_6);\ EXT(B_7)=(v_7);$$

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

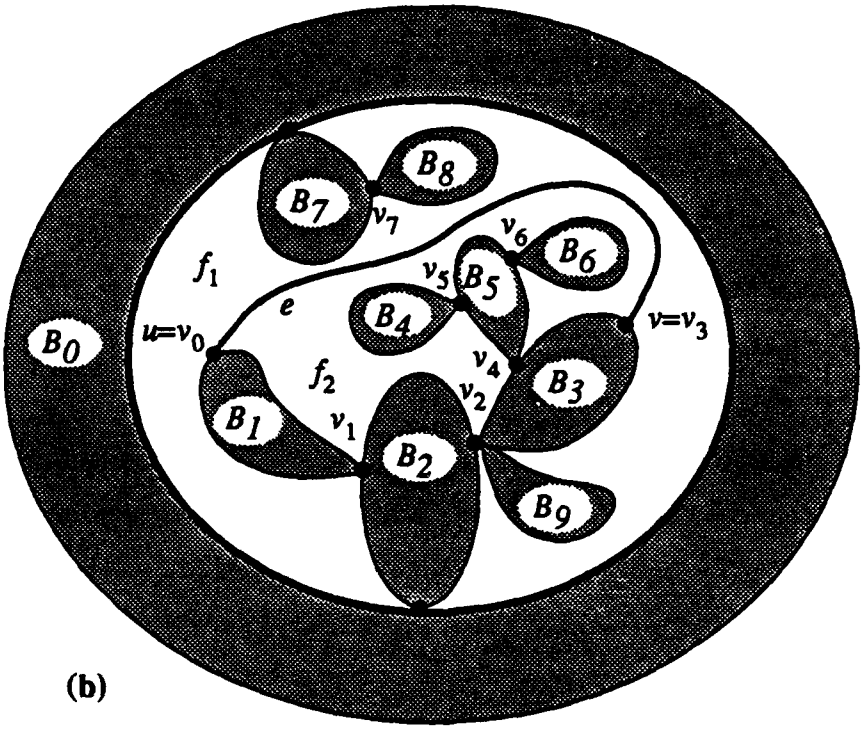
Implementing the above lists by means of balanced search trees, $OUTER(\cdot)$ can be easily retrieved in $O(\log m)$ time, so that the complexity of the *TEST* and *LIST* operations is the same as in the static case.

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. 7.11(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



(a)



(b)

Figure 7.11 Example of *INSERTEDGE* operation in a cutface (a) Before insertion. (b) After insertion.

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. 7.11(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}), 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. 7.11, 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 7.11 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 $-m \log m \le \Phi \le 0$. 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. 7.12 a plot of $H(x_1,1-x_1)$ in the interval $x_1 \in [0,1]$. It is easy to see that

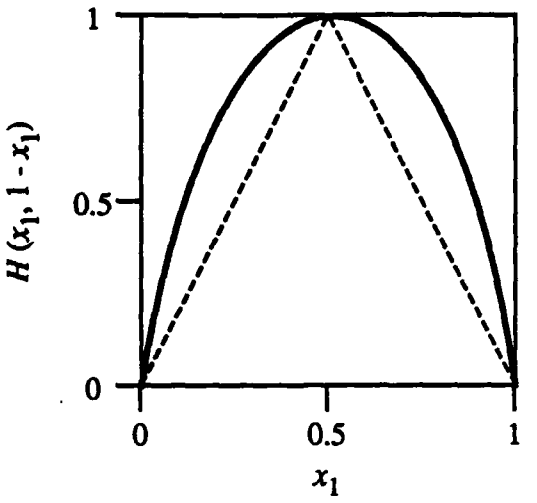


Figure 7.12 Plot of the binary entropy function $H(x_1, 1-x_1)$.

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

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 7.3 The data structure of Theorem 7.11 also supports operations *REMOVEEDGE*, *REMOVEVERTEX*, and *DETACH* on free edges and vertices in $O(\log m)$ time (worst-case).

7.6. Applications

7.6.1. 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. 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 7.4 and duality arguments, we obtain

Theorem 7.12 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 7.13 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:

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

7.6.2. Computing separating pairs 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. The identification of separating pairs is important in problems of fault-tolerance of networks.

Lemma 7.8 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 7.4 The data structure of Theorem 7.10 allows us to test whether two given vertices form a separating pair in $O(\log m)$ time.

REFERENCES

[1] G.M. Adel'son-Vel'skii and E.M. Landis, "An Information Organization Algorithm," *Doklady Akad. Nauk SSSR*, vol. 146, pp. 263-266, 1962.

[2] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing*, vol. 14, no. 3, pp. 545-568, 1985.

[3] J. Bentley and J. Saxe, "Decomposable Searching Problems I: Static to Dynamic Transformations," *J. Algorithms*, vol. 1, pp. 301-358, 1980.

[4] J. Bhasker and S. Sahni, "A Linear Algorithm to Find a Rectangular Dual of a Planar Triangulated Graph," *Algorithmica*, vol. 3, no. 2, pp. 247-278, 1988.

[5] G. Birkhoff, "Lattice Theory," *American Mathematical Society Colloquium Publications*, vol. 25, 1979.

[6] J. Bondy and U. Murty, *Graph Theory with Applications*, North Holland, Amsterdam, 1976.

[7] 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.

[8] B. Chazelle, H. Edelsbrunner, and L.J. Guibas, "The Complexity of Cutting Convex Polytopes," *Proc. 19th ACM Symp. on Theory of Computing*, pp. 66-76, 1987.

[9] 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.

[10] G. Di Battista and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs," *Theoretical Computer Science*, vol. 61, no. 3, 1988 (to appear).

[11] D.P. Dobkin and R.J. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing*, vol. 5, no. 2, pp. 181-186, 1976.

[12] M. Edahiro, I. Kokubo, and T. Asano, "A New Point-Location Algorithm and its Practical Efficiency – Comparison with Existing Algorithms," *ACM Trans. on Graphics*, vol. 3, no. 2, pp. 86-109, 1984.

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

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

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

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

[17] 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.

[18] O. Fries, "Zerlegung einer planaren Unterteilung der Ebene und ihre Anwendungen," M.S. thesis, Inst. Angew. Math. and Inform., Univ. Saarlandes, Saarbrücken, Germany, 1985.

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

[20] 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.

[21] L.J. Guibas and F.F. Yao, "On Translating a Set of Rectangles," in *Advances in Computing Research*, vol. 1, F.P. Preparata (Ed.), JAI Press Inc., pp. 61-77, 1983.

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

[23] 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.

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

[25] M.Y. Hsueh and D.O. Pederson, "Computer-Aided Layout of LSI Circuit Building-Blocks," *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 474-477, 1979.

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

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

[28] G.F. Italiano, "Finding Paths and Deleting Edges in Directed Acyclic Graphs," *Information Processing Letters*, 1988 (to appear).

[29] R. Jayakumar, K. Thulasiraman, and M.N.S Swamy, "An Optimal Algorithm for Maximal Planarization of Nonplanar Graphs," *Proc. 1986 IEEE Int. Symp. on Circuits and Systems*, pp. 1237-1240, 1986.

[30] R. Jayakumar, K. Thulasiraman, and M.N.S Swamy, "Planar Embedding: Linear-Time Algorithms for Vertex Placement and Edge Ordering," *IEEE Trans. on Circuits and Systems*, vol. 35, no. 3, pp. 334-344, 1988.

[31] T. Kameda, "On the Vector Representation of the Reachability in Planar Directed Graphs," *Information Processing Letters*, vol. 3, no. 3, pp. 75-77, 1975.

[32] D. Kelly and I. Rival, "Planar Lattices," *Canadian J. Mathematics*, vol. 27, no. 3, pp. 636-665, 1975.

[33] D. Kelly, "Fundamentals of Planar Ordered Sets," *Discrete Mathematics*, vol. 63, pp. 197-216, 1987.

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

[35] 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.

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

[37] R.J. Lipton and R.E. Tarjan, "Applications of a Planar Separator Theorem," *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pp. 162-170, 1977.

[38] K. Mehlhorn and M. Overmars, "Optimal Dynamization of Decomposable Searching Problems," *Information Processing Letters*, vol. 12, pp. 93-98, 1981.

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

[40] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, New York, 1984.

[41] R.M. Otten and J.G. van Wijk, "Graph Representations in Interactive Layout Design," *Proc. IEEE Int. Symp. on Circuits and Systems*, New York, pp. 914-918, 1978.

[42] M. Overmars, "The Design of Dynamic Data Structures," *Lecture Notes in Computer Science*, vol. 156, Springer Verlag, 1983.

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

[44] T. Ozawa and H. Takahashi, "A Graph-planarization Algorithm and its Applications to Random Graphs," *Lecture Notes in Computer Science* (*Graph Theory and Algorithms*), vol. 108, Springer-Verlag, pp. 95-107, 1981.

[45] 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.

[46] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing*, vol. 10, no. 3, pp. 473-483, 1981.

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

[48] I. Rival and J. Urrutia, "Representing Orders on the Plane by Translating Convex Figures," Technical Report TR-87-05, Univ. of Ottawa, 1987.

[49] 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.

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

[51] 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.

[52] 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.

[53] R. Tamassia and I.G. Tollis, "Efficient Embedding of Planar Graphs in Linear Time," *Proc. IEEE Int. Symp. on Circuits and Systems*, Philadelphia, pp. 495-498, 1987.

[54] R. Tamassia, "A Dynamic Data Structure for Planar Graph Embedding," *Automata, Languages and Programming* (Proc. 15th ICALP, Tampere, Finland, 1988), T. Lepisto and A. Salomaa (Eds.), *Lecture Notes in Computer Science*, vol. 317, Springer-Verlag, pp. 576-590, 1988.

[55] R.E. Tarjan, "Data Structures and Network Algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 44, Society for Industrial Applied Mathematics, 1983.

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

[57] S. Wimer, I. Koren, and I. Cederbaum, "Floorplans, Planar Graphs, and Layouts," *IEEE Trans. on Circuits and Systems*, vol. 35, no. 3, pp. 267-278, 1988.

[58] S. Wismath, "Characterizing bar line-of-sight graphs," *Proc. ACM Symp. on Computational Geometry*, Baltimore, Maryland, pp. 147-152, 1985.

[59] D. Woods, "Drawing Planar Graphs," Ph.D. dissertation (Technical Report STAN-CS-82-943), Computer Science Dept., Stanford Univ., 1982.

VITA

Roberto Tamassia [redacted] He received the Laurea in Electrical Engineering from the University of Rome, in March 1984, and the Ph.D. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign in August 1988. He is currently an Assistant Professor in the Department of Computer Science of Brown University. Previously, he was a research associate in the Dipartimento di Informatica e Sistemistica of the University of Rome, and a research assistant in the Coordinated Science Laboratory of the Unversity of Illinois. He was awarded a Fulbright grant in 1985, and has been a consultant for several computer companies. His research interests include design and analysis of algorithms, computational geometry, graph layout, VLSI, and computer-aided design for information systems.