Algorithms and Theory of Computation Handbook

Second Edition

Special Topics and Techniques

Edited by

Mikhail J. Atallah

Marina Blanton

CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the

Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number: 978-1-58488-820-8 (Hardback)

Library of Congress Cataloging-in-Publication Data

Algorithms and theory of computation handbook. Special topics and techniques / editors, Mikhail J. Atallah and Marina Blanton. -- 2nd ed.
p. cm. -- (Chapman & Hall/CRC applied algorithms and data structures series)
Includes bibliographical references and index.
ISBN 978-1-58488-820-8 (alk. paper)
1. Computer algorithms. 2. Computer science. 3. Computational complexity. I. Atallah, Mikhail J. II. Blanton, Marina. III. Title. IV. Series.

QA76.9.A43A433 2009
005.1--dc22 2009017978

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

1

Computational Geometry I

D.T. Lee
Academia Sinica

1.1 Introduction

Computational geometry, since its inception [66] in 1975,has received a great deal of attention from researchers in the area of design and analysis of algorithms. It has evolved into a discipline of its own. It is concerned with the computational complexity of geometric problems that arise in various disciplines such as pattern recognition, computer graphics, geographical information system, computer vision, **CAD/CAM**, robotics, VLSI layout, operations research, and statistics. In contrast with the classical approach to proving mathematical theorems about geometry-related problems, this discipline emphasizes the computational aspect of these problems and attempts to exploit the underlying geometric properties possible, e.g., the metric space, to derive efficient algorithmic solutions.

An objective of this discipline in the theoretical context is to study the computational complexity (giving lower bounds) of geometric problems, and to devise efficient algorithms (giving upper bounds) whose complexity preferably matches the lower bounds. That is, not only are we interested in the intrinsic difficulty of geometric computational problems under a certain computation model, but we are also concerned with the algorithmic solutions that are efficient or provably optimal in the worst or average case. In this regard, the **asymptotic time** (or **space**) **complexity** of an algorithm, i.e., the behavior of an algorithm, as the input size approaches infinity, is of interest. Due to its applications to various science and engineering related disciplines, researchers in this field have begun to address the *efficacy* of the algorithms, the issues concerning *robustness* and *numerical stability* [33,82], and the actual running times of their implementations. In order to value and get better understanding of the geometric algorithms in action, a computational problem solving environment

has been developed at the Institute of Information Science and the Research Center for Information Technology Innovation, Academia Sinica, Taiwan. Actual implementations of several geometric algorithms have been incorporated into a Java-based algorithm visualization and debugging software system, dubbed *GeoBuilder* (http://webcollab.iis.sinica.edu.tw/Components/GeoBuilder/), which supports remote compilation, visualization of intermediate execution results, and other runtime features, e.g., visual debugging, etc. This system facilitates geometric algorithmic researchers in testing their ideas and demonstrating their findings in computational geometry. GeoBuilder system is embedded into a knowledge portal [51], called OpenCPS (Open Computational Problem Solving), (http://www.opencps.org/) and possesses three important features. First, it is a platform-independent software system based on Java's promise of portability, and can be invoked by Sun's Java Web Start technology in any browser-enabled environment. Second, it has the collaboration capability for multiple users to concurrently develop programs, manipulate geometric objects, and control the camera. Finally, its three-dimensional (3D) geometric drawing bean provides an optional function that can automatically position the camera to track 3D objects during algorithm visualization [79]. GeoBuilder develops its rich client platform based on Eclipse RCP and has already built in certain functionalities such as remote addition, deletion, and saving of files as well as remote compiling, and execution of LEDA C/C++ programs, etc., based on a multipage editor. Other notable geometric software projects include, among others, CGAL (http://www.cgal.org/) [32] and LEDA (http://www.algorithmic-solutions.com/leda/about/index.htm.) [60].

In this and the following chapter (Chapter 2) we concentrate mostly on the theoretical development of this field in the context of sequential computation, and discuss a number of typical topics and the algorithmic approaches. We will adopt the *real* RAM (random access machine) model of computation in which all arithmetic operations, comparisons, kth root, exponential, or logarithmic functions take unit time.

1.2 Convex Hull

The convex hull of a set of points in $\Re^k$ is the most fundamental problem in computational geometry. Given is a set of points in $\Re^k$, and we are interested in computing its convex hull, which is defined to be the smallest convex set containing these points. There are two ways to represent a convex hull. An implicit representation is to list all the **extreme points**, whereas an explicit representation is to list all the extreme d-faces of dimensions $d = 0, 1, \ldots, k-1$. Thus, the complexity of any convex hull algorithm would have two parts, computation part and the output part. An algorithm is said to be **output-sensitive** if its complexity depends on the size of the output.

1.2.1 Convex Hulls in Two and Three Dimensions

For an arbitrary set of n points in two and three dimensions, we can compute its convex hull using the *Graham scan*, *gift-wrapping* method, or *divide-and-conquer* paradigm, which are briefly described below.

Note that the convex hull of an arbitrary set S of points in two dimensions is a convex polygon. We'll describe algorithms that compute the *upper hull* of S, since the convex hull is just the union of the upper and lower hulls. Let v_0 denote the point with minimum x-coordinate; if there are more than one, pick the one with the maximum y-coordinate. Let v_{n-1} be similarly defined except that it denotes the point with the maximum x-coordinate. In two dimensions, the upper hull consists of two vertical lines passing through v_0 and v_{n-1}, respectively and a sequence of edges, known as a *polygonal chain*, $\mathcal{C} = \{\overline{v_{j_{i-1}}, v_{j_i}} \mid i = 1, 2, \ldots, k\}$, where $v_{j_0} = v_0$ and $v_{j_k} = v_{n-1}$, such that the entire set S of points lies on one side of or *below* the lines $\mathcal{L}_i$ containing each edge $\overline{v_{j_{i-1}}, v_{j_i}}$. See Figure 1.1a for an illustration of the upper hull. The lower hull is similarly defined.

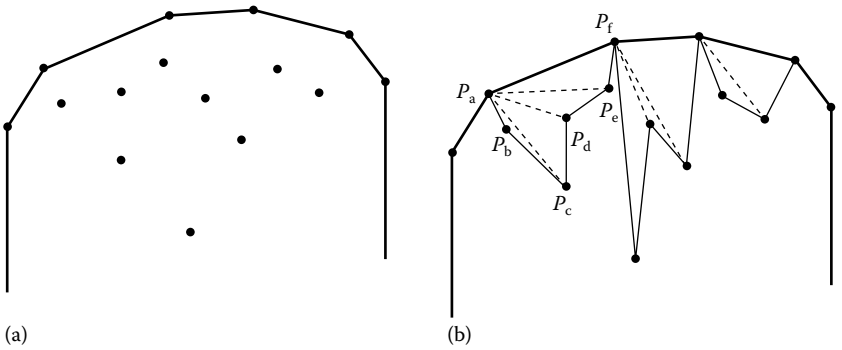

FIGURE 1.1 The upper hull of a set of points (a) and illustration of the Graham scan (b).

The *Graham scan* computes the convex hull by (1) sorting the input set of points in ascending order of their x-coordinates (in case of ties, in ascending order of their y-coordinates), (2) connecting these points into a polygonal chain P stored as a doubly linked list L, and (3) performing a linear scan to compute the upper hull of the polygon [66].

The triple (v_i, v_j, v_k) of points is said to form a *right* turn if and only if the determinant

$$\begin{vmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_k & y_k & 1 \end{vmatrix} < 0,$$

where (x_i, y_i) are the x- and y-coordinates of v_i. If the determinant is positive, then the triple (v_i, v_j, v_k) of points is said to form a *left* turn. The points v_i, v_j, and v_k are collinear if the determinant is zero. This is also known as the **side test**, determining on which side of the line defined by points v_i and v_j the point v_k lies.

It is obvious that when we scan points in L in ascending order of x-coordinate, the middle point of a triple (v_i, v_j, v_k) that does not form a right turn is not on the upper hull and can be deleted. The following is the algorithm.

ALGORITHM GRAHAM_SCAN

Input: A set S of points sorted in lexicographically ascending order of their (x, y)-coordinate values.

Output: A sorted list L of points in ascending x-coordinates.

```
begin
    if (|S| == 2) return (v_0, v_{n-1});
    i = 0; v_{n-1} = next(v_{n-1}); /* set sentinel */
    p_a = v_0; p_b = next(p_a), p_c = next(p_b);
    while (p_b ≠ v_{n-1}) do
        if (p_a, p_b, p_c) forms a right turn
        then begin /* advance */
                p_a = p_b; p_b = p_c;
                p_c = next(p_b);
             end
        else begin /* backtrack */
                delete p_b;
```

```
                    if (p_a ≠ v_0)
                    then p_a = prev(p_a);
                    p_b = next(p_a); p_c = next(p_b);
                end
    p_t = next(v_0);
    L = {v_0, p_t};
    while (p_t ≠ v_{n-1}) do
        begin
            p_u = next(p_t);
            L = L ∪ {p_t, p_u};
            p_t = p_u;
        end;
    return (L);
end.
```

Step (i) being the dominating step, ALGORITHM GRAHAM_SCAN, takes $O(n \log n)$ time. Figure 1.1b shows the initial list L and vertices not on the upper hull are removed from L. For example, p_b is removed since (p_a, p_b, p_c) forms a left turn; p_c is removed since (p_a, p_c, p_d) forms a left turn; p_d, and p_e are removed for the same reason.

One can also use the *gift-wrapping* technique to compute the upper hull. Starting with a vertex that is known to be on the upper hull, say the point $v_0 = v_{i_0}$. We sweep clockwise the half-line emanating from v_0 in the direction of the positive y-axis. The first point v_{i_1} this half-line hits will be the next point on the upper hull. We then march to v_{i_1}, repeat the same process by sweeping clockwise the half-line emanating from v_{i_1} in the direction from v_{i_0} to v_{i_1}, and find the next vertex v_{i_2}. This process terminates when we reach v_{n-1}. This is similar to wrapping an object with a *rope.* Finding the next vertex takes time proportional to the number of points not yet known to be on the upper hull. Thus, the total time spent is $O(n\mathcal{H})$, where $\mathcal{H}$ denotes the number of points on the upper hull. The gift-wrapping algorithm is output-sensitive, and is more efficient than the ALGORITHM GRAHAM_SCAN if the number of points on the upper hull is small, i.e., $O(\log n)$.

One can also compute the upper hull recursively by divide-and-conquer. This method is more amenable to parallelization. The divide-and-conquer paradigm consists of the following steps.

ALGORITHM UPPER_HULL_D&C (*2d-Point S*)

Input: A set S of points.

Output: A sorted list L of points in ascending x-coordinates.

1. If $|S| \leq 3$, compute the upper hull UH(S) explicitly and **return** (UH(S)).
2. Divide S by a vertical line $\mathcal{L}$ into two approximately equal subsets S_l and S_r such that S_l and S_r lie, respectively, to the left and to the right of $\mathcal{L}$.
3. UH(S_l) = Upper_Hull_D&C(S_l).
4. UH(S_r) = Upper_Hull_D&C(S_r).
5. UH(S) = **Merge**($UH(S_l)$, $UH(S_r)$).
6. **return** (UH(S)).

The key step is the **Merge** of two upper hulls, each of which is the solution to a subproblem derived from the recursive step. These two upper hulls are separated by a vertical line $\mathcal{L}$. The **Merge** step basically calls for computation of a common tangent, called *bridge* over line $\mathcal{L}$, of these two upper hulls (Figure 1.2).

The computation of the bridge begins with a segment connecting the rightmost point l of the left upper hull to the leftmost point r of the right upper hull, resulting in a sorted list L. Using the Graham

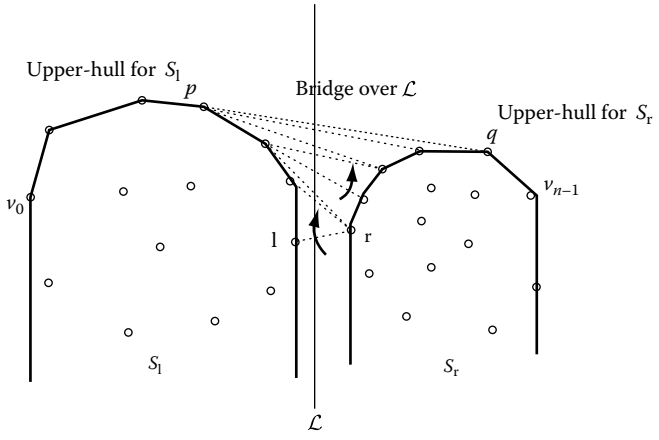

FIGURE 1.2 The bridge $\overline{p,q}$ over the vertical line $\mathcal{L}$.

scan one can obtain in linear time the two endpoints of the bridge, ($\overline{p,q}$ shown in Figure 1.2), such that the entire set of points lies on one side of the line, called *supporting line,* containing the bridge. The running time of the divide-and-conquer algorithm is easily shown to be $O(n \log n)$ since the **Merge** step can be done in $O(n)$ time.

A more sophisticated output-sensitive and *optimal* algorithm which runs in $O(n \log \mathcal{H})$ time has been developed by Kirkpatrick and Seidel [48]. It is based on a variation of the divide-and-conquer paradigm, called **divide-and-marriage-before-conquest** method. It has been shown to be asymptotically optimal; a lower bound proof of $\Omega(n \log \mathcal{H})$ can be found in [48]. The main idea in achieving the optimal result is that of eliminating redundant computations. Observe that in the divide-and-conquer approach after the bridge is obtained, some vertices belonging to the left and right upper hulls that are below the bridge are deleted. Had we known that these vertices are not on the final hull, we could have saved time without computing them. Kirkpatrick and Seidel capitalized on this concept and introduced the marriage-before-conquest principle putting **Merge** step before the two recursive calls.

The divide-and-conquer scheme can be easily generalized to three dimensions. The Merge step in this case calls for computing common supporting faces that wrap two recursively computed convex polyhedra. It is observed by Preparata and Shamos [66] that the common supporting faces are computed from connecting two cyclic sequences of edges, one on each polyhedron (Figure 1.3). See [3] for a characterization of the two cycles of seam edges. The computation of these supporting

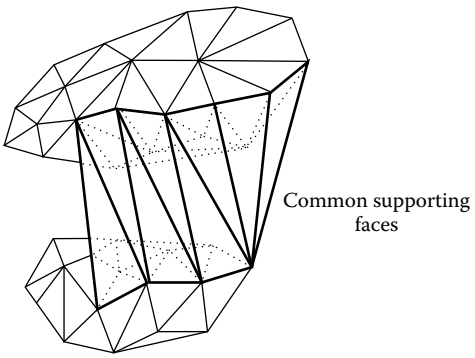

FIGURE 1.3 Common supporting faces of two disjoint convex polyhedra.

faces can be accomplished in linear time, giving rise to an $O(n \log n)$ time algorithm. By applying the marriage-before-conquest principle Edelsbrunner and Shi [28] obtained an $O(n \log^2 \mathcal{H})$ algorithm.

The gift-wrapping approach for computing the convex hull in three dimensions would mimic the process of wrapping a gift with a piece of paper. One starts with a plane supporting S, i.e., a plane determined by three points of S such that the entire set of points lie on one side. In general, the supporting face is a triangle $\Delta(a, b, c)$. Pivoting at an edge, say (a, b) of this triangle, one rotates the plane in space until it hits a third point v, thereby determining another supporting face $\Delta(a, b, v)$. This process repeats until the entire set of points are wrapped by a collection of supporting faces. These supporting faces are called 2-faces, the edges common to two supporting faces, 1-faces, and the vertices (or extreme points) common to 2-faces and 1-faces are called 0-faces. The gift-wrapping method has a running time of $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, 2$.

The following optimal output-sensitive algorithm that runs in $O(n \log \mathcal{H})$ time in two and three dimensions is due to Chan [13]. It is a modification of the *gift-wrapping* method (also known as the Jarvis' March method) and uses a *grouping* technique.

ALGORITHM 2DHULL (S)

1. For $i = 1, 2, \ldots$ do
2. $P \leftarrow$ HULL2D ($S, \mathcal{H}_0, \mathcal{H}_0$), where $\mathcal{H}_0 = \min\{2^{2^i}, n\}$
3. If $P \neq nil$ then return P.

FUNCTION HULL2D ($S, m, \mathcal{H}_0$)

1. Partition S into subsets $S_1, S_2, \ldots, S_{\lceil \frac{n}{m} \rceil}$, each of size at most m
2. For $i = 1, 2, \ldots, \lceil \frac{n}{m} \rceil$ do
3. Compute $CH(S_i)$ and preprocess it in a suitable data structure
4. $p_0 \leftarrow (0, -\infty)$, $p_1 \leftarrow$ the rightmost point of S
5. For $j = 1, 2, \ldots, \mathcal{H}_0$ do
6. For $i = 1, 2, \ldots, \lceil \frac{n}{m} \rceil$ do
7. Compute a point $q_i \in S_i$ that maximizes $\angle p_{j-1} p_j q_i$
8. $p_{j+1} \leftarrow$ a point q from $\{q_1, \ldots, q_{\lceil \frac{n}{m} \rceil}\}$ maximizing $\angle p_{j-1} p_j q$
9. If $p_{j+1} = p_1$ then return list $(p_1, \ldots, p_j)$
10. return *nil*

Let us analyze the complexity of the algorithm. In Step 2, we use an $O(m \log m)$ time algorithm for computing the convex hull for each subset of m points, e.g., Graham's scan for S in two dimensions, and Preparata–Hong algorithm for S in three dimensions. Thus, it takes $O((\frac{n}{m}) m \log m) = O(n \log m)$ time. In Step 5 we build a suitable data structure that supports the computation of the supporting vertex or supporting face in logarithmic time. In two dimensions we can use an array that stores the vertices on the convex hull in say, clockwise order. In three dimensions we use Dobkin–Kirkpatrick hierarchical representation of the faces of the convex hull [24]. Thus, Step 5 takes $\mathcal{H}_0(\frac{n}{m})O(\log m)$ time. Setting $m = \mathcal{H}_0$ gives an $O(n \log \mathcal{H}_0)$ time. Note that setting $m = 1$ we have the Jarvis' March, and setting $m = n$ the two-dimensional (2D) convex hull algorithm degenerates to the Graham's scan. Since we do not know $\mathcal{H}$ in advance, we use in Step 2 of ALGORITHM 2DHULL(S) a sequence $\mathcal{H}_i = 2^{2^i}$ such that $\mathcal{H}_1 + \cdots + \mathcal{H}_{k-1} < \mathcal{H} \leq \mathcal{H}_1 + \cdots + \mathcal{H}_k$ to guess it. The total running time is

$$O\left(\sum_{i=1}^{k} n \log \mathcal{H}_i\right) = O\left(\sum_{i=1}^{\lceil \log \log \mathcal{H} \rceil} n 2^i\right) = O\left(n \log \mathcal{H}\right)$$

1.2.2 Convex Hulls in k Dimensions, $k > 3$

For convex hulls of higher dimensions, Chazelle [16] showed that the convex hull can be computed in time $O(n \log n + n^{\lfloor k/2 \rfloor})$, which is optimal in all dimensions $k \geq 2$ in the worst case. But this result is insensitive to the output size. The gift-wrapping approach generalizes to higher dimensions and yields an output-sensitive solution with running time $O(n\mathcal{H})$, where $\mathcal{H}$ is the total number of i-faces, $i = 0, 1, \ldots, k-1$ and $\mathcal{H} = O(n^{\lfloor k/2 \rfloor})$ [27]. One can also use *beneath–beyond* method [66] of adding points one at a time in ascending order along one of the coordinate axis.* We compute the convex hull $\mathrm{CH}(S_{i-1})$ for points $S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$. For each added point p_i we update $\mathrm{CH}(S_{i-1})$ to get $\mathrm{CH}(S_i)$ for $i = 2, 3, \ldots, n$ by deleting those t-faces, $t = 0, 1, \ldots, k-1$, that are internal to $\mathrm{CH}(S_{i-1} \cup \{p_i\})$. It has been shown by Seidel [27] that $O(n^2 + \mathcal{H} \log h)$ time is sufficient, where h is the number of extreme points. Later Chan [13] obtained an algorithm based on gift-wrapping method using the data structures for ray-shooting queries in polytopes developed by Agarwal and Matoušek [1] and refined by Matoušek and Schwarzkopf [58], that runs in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time. Note that the algorithm is optimal when $k = 2, 3$. In particular, it is optimal when $\mathcal{H} = O(n^{1/(\lfloor k/2 \rfloor)} / \log^{\delta} n)$ for a sufficiently large δ.

We conclude this section with the following theorem [13].

THEOREM 1.1 *The convex hull of a set S of n points in $\Re^k$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$ or $k = 3$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor k/2 \rfloor + 1)} \log^{O(1)} n)$ time for $k > 3$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k-1$.*

1.2.3 Convex Layers of a Planar Set

The convex layers $\mathcal{C}(S)$ of a set S of n points in the Euclidean plane is obtained by a process, known as *onion peeling,* i.e., compute the convex hull of S and remove its vertices from S, until S becomes empty. Figure 1.4 shows the convex layer of a point set. This onion peeling process of a point set is central in the study of robust estimators in statistics, in which the outliers, points lying on the outermost convex layers, should be removed. In this section we describe an efficient algorithm due to Chazelle [14] that runs in optimal $O(n \log n)$ time.

As described in Section 1.2.1, each convex layer of $\mathcal{C}(S)$ can be decomposed into two convex polygonal chains, called upper and lower hulls (Figure 1.5).

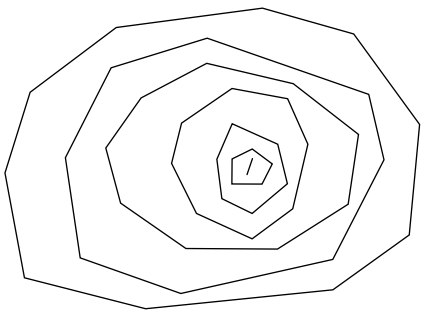

FIGURE 1.4 Convex layers of a point set.

* If the points of S are not given *a priori,* the algorithm can be made **on-line** by adding an extra step of checking if the newly added point is internal or external to the current convex hull. If it is internal, just discard it.

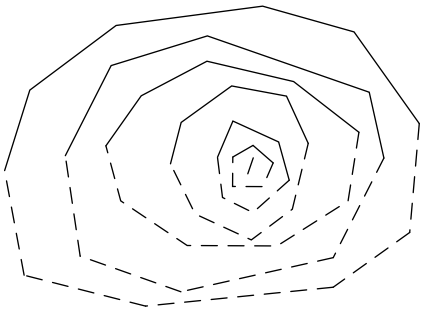

FIGURE 1.5 Decomposition of each convex layer into upper and lower hulls.

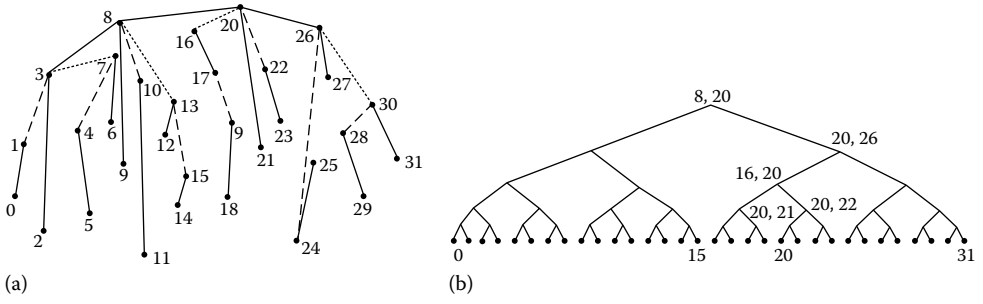

FIGURE 1.6 The hull graph of upper hull (a) and a complete binary tree representation (b).

Let l and r denote the points with the minimum and maximum x-coordinate, respectively, in a convex layer. The upper (respectively, lower) hull of this layer runs clockwise (respectively, counterclockwise) from l to r. The upper and lower hulls are the same if the convex layer has one or two points. Assume that the set S of points $p_0, p_1, \ldots, p_{n-1}$ are ordered in nondescending order of their x-coordinates. We shall concentrate on the computation of upper hulls of $\mathcal{C}(S)$; the other case is symmetric. Consider the complete binary tree $\mathcal{T}(S)$ with leaves $p_0, p_1, \ldots, p_{n-1}$ from left to right. Let $S(v)$ denote the set of points stored at the leaves of the subtree rooted at node v of $\mathcal{T}$ and let $U(v)$ denote its upper hull of the convex hull of $S(v)$. Thus, $U(\rho)$, where ρ denotes the root of $\mathcal{T}$, is the upper hull of the convex hull of S in the outermost layer. The union of all the upper hulls $U(v)$ for all nodes v is a tree, called *hull graph* [14]. (A similar graph is also computed for the lower hull of the convex hull.) To minimize the amount of space, at each internal node v we store the bridge (common tangent) connecting a point in $U(v_l)$ and a point in $U(v_r)$, where v_l and v_r are the left and right children of node v, respectively. Figure 1.6a and b illustrates the binary tree $\mathcal{T}$ and the corresponding hull graph, respectively.

Computation of the hull graph proceeds from bottom up. Computing the bridge at each node takes time linear in the number of vertices on the respective upper hulls in the left and right subtrees. Thus, the total time needed to compute the hull graph is $O(n \log n)$. The bridges computed at each node v which are incident upon a vertex p_k are naturally separated into two subsets divided by the vertical line $\mathcal{L}(p_k)$ passing through p_k. Those on the left are arranged in a list $L(p_k)$ in counterclockwise order from the positive y direction of $\mathcal{L}(p_k)$, and those on the right are arranged in a list $R(p_k)$ in clockwise order. This adjacency list at each vertex in the hull graph can be maintained fairly easily. Suppose the bridge at node v connects vertex p_j in the left subtree and vertex p_k in the right subtree. The edge $\overline{p_j, p_k}$ will be inserted at the *first* position in the current lists $R(p_j)$ and $L(p_k)$. That is, edge

$\overline{p_j, p_k}$ is the *top* edge in both lists $R(p_j)$ and $L(p_k)$. It is easy to retrieve the vertices on the upper hull of the outermost layer from the hull graph beginning at the root node of $\mathcal{T}$.

To compute the upper hull of the next convex layer, one needs to remove those vertices on the first layer (including those vertices in the lower hull). Thus, update of the hull graph includes deletion of vertices on both upper hull and lower hull. Deletions of vertices on the upper hull can be performed in an arbitrary order. But if deletions of vertices on the lower hull from the hull graph are done in say clockwise order, then the update of the adjacency list of each vertex p_k can be made easy, e.g., $R(p_k) = \emptyset$. The deletion of a vertex p_k on the upper hull entails removal of edges incident on p_k in the hull graph. Let $v_1, v_2, \ldots, v_l$ be the list of internal nodes on the leaf-to-root path from p_k. The edges in $L(p_k)$ and $R(p_k)$ are deleted from bottom up in $O(1)$ time each, i.e., the top edge in each list gets deleted last. Figure 1.6b shows the leaf-to-root path when vertex p_{20} is deleted. Figure 1.7a–f shows the updates of bridges when p_{20} is deleted and Figure 1.7g is the final upper hull after the update is finished. It can be shown that the overall time for deletions can be done in $O(n \log n)$ time [14].

THEOREM 1.2 *The convex layers of a set of n points in the plane can be computed in $O(n \log n)$ time.*

Nielsen [64] considered the problem of computing the first k layers of convex hull of a planar point set S that arises in statistics and pattern recognition, and gave an $O(n \log \mathcal{H}_k)$ time algorithm, where $\mathcal{H}_k$ denotes the number of points on the first k layers of convex hull of S, using the grouping scheme of Chan [13].

1.2.4 Applications of Convex Hulls

Convex hulls have applications in clustering, linear regression, and Voronoi diagrams (see Chapter 2). The following problems have solutions derived from the convex hull.

Problem C1 (Set Diameter) Given a set S of n points, find the two points that are the farthest apart, i.e., find $p_i, p_j \in S$ such that $d(p_i, p_j) = \max\{d(p_k, p_l)\}\ \forall p_k, p_l \in S$, where $d(p, q)$ denotes the Euclidean distance between p and q.

In two dimensions $O(n \log n)$ time is both sufficient and necessary in the worst case [66]. It is easy to see that the farthest pair must be extreme points of the convex hull of S. Once the convex hull is computed, the farthest pair in two dimensions can be found in linear time by observing that it admits a pair of parallel supporting lines. Various attempts, including geometric sampling and parametric search method, have been made to solve this problem in three dimensions. See e.g., [59].

Clarkson and Shor [20] gave a randomized algorithm with an optimal expected $O(n \log n)$ time. Later Ramos [67] gave an optimal deterministic algorithm, based on a simplification of the randomization scheme of Clarkson and Shor [20], and derandomization making use of the efficient construction of ϵ-nets by Matoušek [57].

Problem C2 (Smallest enclosing rectangle) Given a set S of n points, find the smallest rectangle that encloses the set.

Problem C3 (Regression line) Given a set S of n points, find a line such that the maximum distance from S to the line is minimized.

These two problems can be solved in optimal time $O(n \log n)$ using the convex hull of S [54] in two dimensions. In k dimensions Houle et al. [43] gave an $O(n^{\lfloor k/2+1 \rfloor})$ time and $O(n^{\lfloor (k+1)/2 \rfloor})$ space algorithm. The time complexity is essentially that of computing the convex hull of the point set.

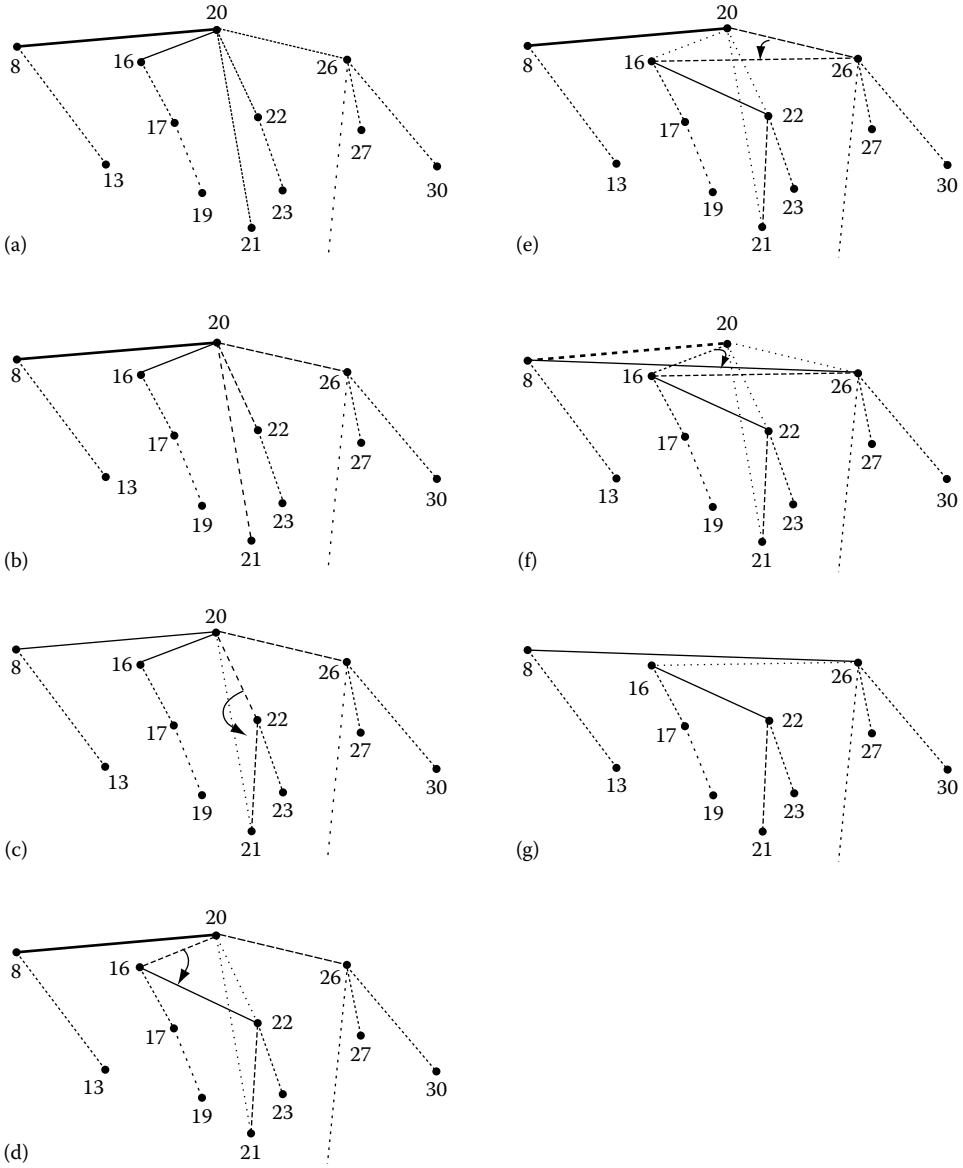

FIGURE 1.7 Update of hull graph.

1.3 Maxima Finding

In this section we discuss a problem concerned with the *extremes* of a point set which is somewhat related to that of convex hull problems. Consider a set S of n points in $\Re^k$ in the Cartesian coordinate system. Let $(x_1(p), x_2(p), \ldots, x_k(p))$ denote the coordinates of point $p \in \Re^k$. Point p is said to *dominate* point q, denoted $p \succeq q$, (or q is dominated by p, denoted $q \preceq p$) if $x_i(p) \geq x_i(q)$ for all $1 \leq i \leq k$. A point p is said to be *maximal* (or a *maximum*) in S if no point in S dominates p. The maxima-finding problem is that of finding the set $\mathcal{M}(S)$ of maximal elements for a set S of points in $\Re^k$.

1.3.1 Maxima in Two and Three Dimensions

In two dimensions the problem can be done fairly easily by a plane-sweep technique. (For a more detailed description of plane-sweep technique, see, e.g., [50] or Section 1.5.1) Assume that the set S of points $p_1, p_2, \ldots, p_n$ are ordered in nondescending order of their x-coordinates, i.e., $x(p_1) \leq x(p_2) \leq \cdots \leq x(p_n)$.

We shall scan the points from right to left. The point p_n is necessarily a maximal element. As we scan the points, we maintain the maximum y-coordinate among those that have been scanned so far. Initially, $\max_y = y(p_n)$. The next point p_i is a maximal element if and only if $y(p_i) > \max_y$. If $y(p_i) > \max_y$, then $p_i \in \mathcal{M}(S)$, and $\max_y$ is set to $y(p_i)$, and we continue. Otherwise $p_i \preceq p_j$ for some $j > i$. Thus, after the initial sorting, the set of maxima can be computed in linear time. Note that the set of maximal elements satisfies the property that their x- and y-coordinates are totally ordered: If they are ordered in strictly ascending x-coordinate, their y-coordinates are ordered in strictly descending order.

In three dimensions we can use the same strategy. We will scan the set in descending order of the x-coordinate by a plane $\mathcal{P}$ orthogonal to the x-axis. Point p_n as before is a maximal element. Suppose we have computed $\mathcal{M}(S_{i+1})$, where $S_{i+1} = \{p_{i+1}, \ldots, p_n\}$, and we are scanning point p_i. Consider the orthogonal projection S_{i+1}^x of the points in S_{i+1} to $\mathcal{P}$ with $x = x(p_i)$. We now have an instance of an *on-line* 2D maximal problem, i.e., for point p_i, if $p_i^x \preceq p_j^x$ for some $p_j^x \in S_{i+1}^x$, then it is not a maximal element, otherwise it is (p_i^x denotes the projection of p_i onto $\mathcal{P}$). If we maintain the points in $\mathcal{M}(S_{i+1}^x)$ as a **height-balanced binary search tree** in either y- or z-coordinate, then testing whether p_i is maximal or not can be done in logarithmic time. If it is dominated by some point in $\mathcal{M}(S_{i+1}^x)$, then it is ignored. Otherwise, it is in $\mathcal{M}(S_{i+1}^x)$ (and also in $\mathcal{M}(S_{i+1})$); $\mathcal{M}(S_{i+1}^x)$ will then be updated to be $\mathcal{M}(S_i^x)$ accordingly. The update may involve deleting points in $\mathcal{M}(S_{i+1}^x)$ that are no longer maximal because they are dominated by p_i^x. Figure 1.8 shows the effect of adding a maximal element p_i^x to the set $\mathcal{M}(S_{i+1}^x)$ of maximal elements. Points in the shaded area will be deleted. Thus, after the initial sorting, the set of maxima in three dimensions can be computed in $O(n \log \mathcal{H})$ time, as the *on-line* 2D maximal problem takes $O(\log \mathcal{H})$ time to maintain $\mathcal{M}(S_i^x)$ for each point p_i, where $\mathcal{H}$ denotes the size of $\mathcal{M}(S)$.

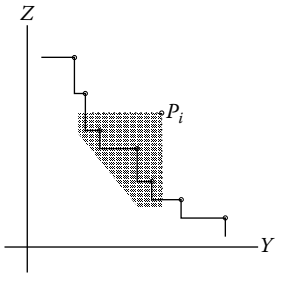

FIGURE 1.8 Update of maximal elements.

Since the total number of points deleted is at most n, we conclude the following.

LEMMA 1.1 Given a set of n points in two and three dimensions, the set of maxima can be computed in $O(n \log n)$ time.

For two and three dimensions one can solve the problem in optimal time $O(n \log \mathcal{H})$, where $\mathcal{H}$ denotes the size of $\mathcal{M}(S)$. The key observation is that we need not *sort* S in its entirety. For instance, in two dimensions one can solve the problem by divide-and-marriage-before-conquest paradigm. We first use a linear time median finding algorithm to divide the set into two halves L and R with points in R having larger x-coordinate values than those of points in L. We then recursively compute $\mathcal{M}(R)$. Before we recursively compute $\mathcal{M}(L)$ we note that points in L that are dominated by points in $\mathcal{M}(R)$ can be eliminated from consideration. We trim L before we invoke the algorithm recursively. That is, we compute $\mathcal{M}(L')$ recursively, where $L' \subseteq L$ consists of points $q \not\preceq p$ for all $p \in \mathcal{M}(R)$. A careful analysis of the running time shows that the complexity of this algorithm is $O(n \log \mathcal{H})$. For three dimensions we note that other than the initial sorting step, the subsequent plane-sweep step takes $O(n \log \mathcal{H})$ time. It turns out that one can replace the full-fledged $O(n \log n)$ sorting step with a so-called *lazy sorting* of S using a technique similar to those described in Section 1.2.1 to derive an output-sensitive algorithm.

THEOREM 1.3 *Given a set S of n points in two and three dimensions, the set $\mathcal{M}(S)$ of maxima can be computed in $O(n \log \mathcal{H})$ time, where $\mathcal{H}$ is the size of $\mathcal{M}(S)$.*

1.3.2 Maxima in Higher Dimensions

The set of maximal elements in $\Re^k, k \geq 4$, can be solved by a generalization of plane-sweep method to higher dimensions. We just need to maintain a data structure for $\mathcal{M}(S_{i+1})$, where $S_{i+1} = \{p_{i+1}, \ldots, p_n\}$, and test for each point p_i if it is a maximal element in S_{i+1}, reducing the problem to one dimension lower, assuming that the points in S are sorted and scanned in descending lexicographical order. Thus, in a straightforward manner we can compute $\mathcal{M}(S)$ in $O(n^{k-2} \log n)$ time. However, we shall show below that one can compute the set of maxima in $O(n \log^{k-2} n)$ time, for $k > 3$ by, divide-and-conquer. Gabow et al. [36] gave an algorithm which improved the time by a $O\left(\frac{\log n}{\log \log n}\right)$ factor to $O(n \log \log n \log^{k-3} n)$.

Let us first consider a *bichromatic maxima-finding* problem. Consider a set of n *red* and a set of m *blue* points, denoted R and B, respectively. The bichromatic maxima-finding problem is to find a subset of points in R that are not dominated by any points in B and vice versa. That is, find $\mathcal{M}(R, B) = \{r | r \not\preceq b, b \in B\}$ and $\mathcal{M}(B, R) = \{b | b \not\preceq r, r \in R\}$.

In three dimensions, this problem can be solved by plane-sweep method in a manner similar to the maxima-finding problem as follows. As before, the sets R and B are sorted in nondescending order of x-coordinates and we maintain two subsets of points $\mathcal{M}\left(R^x_{i+1}\right)$ and $\mathcal{M}\left(B^x_{j+1}\right)$, which are the maxima of the projections of R_{i+1} and B_{j+1} onto the yz-plane for $R_{i+1} = \{r_{i+1}, \ldots, r_n\} \subseteq R$ and $B_{j+1} = \{b_{j+1}, \ldots, b_m\} \subseteq B$, respectively. When the next point $r_i \in R$ is scanned, we test if r^x_i is dominated by any points in $\mathcal{M}\left(B^x_{j+1}\right)$. The point $r_i \in \mathcal{M}(R, B)$, if r^x_i is not dominated by any points in $\mathcal{M}\left(B^x_{j+1}\right)$. We then update the set of maxima for $R^x_i = R^x_{i+1} \cup \{r^x_i\}$. That is, if $r^x_i \preceq q$ for $q \in \mathcal{M}\left(R^x_{i+1}\right)$, then $\mathcal{M}\left(R^x_i\right) = \mathcal{M}\left(R^x_{i+1}\right)$. Otherwise, the subset of $\mathcal{M}\left(R^x_{i+1}\right)$ dominated by r^x_i is removed, and r^x_i is included in $\mathcal{M}\left(R^x_i\right)$. If the next point scanned is $b_j \in B$, we perform similar operations. Thus, for each point scanned we spend $O(\log n + \log m)$ time.

LEMMA 1.2 The bichromatic maxima-finding problem for a set of n red and m blue points in three dimensions can be solved in $O(N \log N)$ time, where $N = m + n$.

Using Lemma 1.2 as basis, one can solve the bichromatic maxima-finding problem in $\Re^k$ in $O(N \log^{k-2} N)$ time for $k \geq 3$ using multidimensional divide-and-conquer.

LEMMA 1.3 The bichromatic maxima-finding problem for a set of n red and m blue points in $\Re^k$ can be solved in $O(N \log^{k-2} N)$ time, where $N = m + n$, and $k \geq 3$.

Let us now turn to the maxima-finding problem in $\Re^k$. We shall use an ordinary divide-and-conquer method to solve the maxima-finding problem. Assume that the points in $S \subseteq \Re^k$ have been sorted in all dimensions. Let L_x denote the median of all the x-coordinate values. We first divide S into two subsets S_1 and S_2, each of size approximately $|S|/2$ such that the points in S_1 have x-coordinates larger than L_x and those of points in S_2 are less than L_x. We then recursively compute $\mathcal{M}(S_1)$ and $\mathcal{M}(S_2)$. It is clear that $\mathcal{M}(S_1) \subseteq \mathcal{M}(S)$. However, some points in $\mathcal{M}(S_2)$ may be dominated by points in $\mathcal{M}(S_1)$, and hence, are not in $\mathcal{M}(S)$. We then project points in S onto the hyperplane $\mathcal{P}$: $x = L_x$. The problem now reduces to the bichromatic maxima-finding problem in $\Re^{k-1}$, i.e., finding among $\mathcal{M}(S_2)$ those that are maxima with respect to $\mathcal{M}(S_1)$. By Lemma 1.3 we

know that this bichromatic maxima-finding problem can be solved in $O(n \log^{k-3} n)$ time. Since the merge step takes $O(n \log^{k-3} n)$ time, we conclude the following.

THEOREM 1.4 *The maxima-finding problem for a set of n points in $\Re^k$ can be solved in $O(n \log^{k-2} n)$ time, for $k \geq 3$.*

We note here also that if we apply the *trimming* operation of S_2 with $\mathcal{M}(S_1)$, i.e., removing points in S_2 that are dominated by points in $\mathcal{M}(S_1)$, before recursion, one can compute $\mathcal{M}(S)$ more efficiently as stated in the following theorem.

THEOREM 1.5 *The maxima-finding problem for a set S of n points in $\Re^k$, $k \geq 4$, can be solved in $O(n \log^{k-2} \mathcal{H})$ time, where $\mathcal{H}$ is the number of maxima in S.*

1.3.3 Maximal Layers of a Planar Set

The maximal layers of a set of points in the plane can be obtained by a process similar to that of convex layers discussed in Section 1.2.3. A brute-force method would yield an $O(\delta \cdot n \log \mathcal{H})$ time, where δ is the number of layers and $\mathcal{H}$ is the maximum number of maximal elements in any layer. In this section we shall present an algorithm due to Atallah and Kosaraju [7] for computing not only the maximal layers, but also some other functions associated with dominance relation.

Consider a set S of n points. As in the previous section, let $\mathcal{D}_S(p)$ denote the set of points in S dominated by p, i.e., $\mathcal{D}_S(p) = \{q \in S | q \preceq p\}$. Since p is always dominated by itself, we shall assume $\mathcal{D}_S(p)$ does not include p, when $p \in S$. The first subproblem we consider is the *maxdominance problem,* which is defined as follows: for each $p \in S$, find $\mathcal{M}(\mathcal{D}_S(p))$. That is, for each $p \in S$ we are interested in computing the set of maximal elements among those points that are dominated by p. Another related problem is to compute the labels of each point p from the labels of those points in $\mathcal{M}(\mathcal{D}_S(p))$. More specifically, suppose each point is associated with a weight $w(p)$. The label $l_S(p)$ is defined to be $w(p)$ if $\mathcal{D}_S(p) = \emptyset$ and is $w(p) + \max\{l_S(q), q \in \mathcal{M}(\mathcal{D}_S(p))\}$. The max function can be replaced with min or any other associative functional operation. In other words, $l_S(p)$ is equal to the maximum among the labels of all the points dominated by p. Suppose we let $w(p) = 1$ for all $p \in S$. Then those points with labels equal to 1 are points that do not dominate any points. These points can be thought of as *minimal* points in S. That a point p_i has label λ implies there exists a sequence of λ points $p_{j_1}, p_{j_2}, \ldots, p_{j_\lambda} = p_i$, such that $p_{j_1} \preceq p_{j_2} \preceq \cdots \preceq p_{j_\lambda} = p_i$. In general, points with label λ are on the λ^{th} minimal layer and the maximum label gives the number of minimal layers. If we modify the definition of domination to be p dominates q if and only if $x(p) \leq x(q)$ and $y(p) \leq y(q)$, then the minimal layers obtained using the method to be described below correspond to the maximal layers.

Let us now discuss the labeling problem defined earlier. We recall a few terms as used in [7].*

Let L and R denote two subsets of points of S separated by a vertical line, such that $x(l) \leq x(r)$ for all $l \in L$ and $r \in R$. $\text{leader}_R(p), p \in R$ is the point $\mathcal{H}_p$ in $\mathcal{D}_R(p)$ with the largest y-coordinate. $\text{Strip}_L(p, R), p \in R$ is the subset of points of $\mathcal{D}_L(p)$ dominated by p but with y-coordinates greater than $\text{leader}_R(p)$, i.e., $\text{Strip}_L(p, R) = \{q \in \mathcal{D}_L(p) | y(q) > y(\mathcal{H}_p)\}$ for $p \in R$. $\text{Left}_L(p, R), p \in R$, is defined to be the largest $l_S(q)$ over all $q \in \text{Strip}_L(p, R)$ if $\text{Strip}_L(p, R)$ is nonempty, and $-\infty$ otherwise.

Observe that for each $p \in R$ $\mathcal{M}(\mathcal{D}_S(p))$ is the concatenation of $\mathcal{M}(\mathcal{D}_R(p))$ and $\text{Strip}_L(p, R)$. Assume that the points in $S = \{p_1, p_2, \ldots, p_n\}$ have been sorted as $x(p_1) < x(p_2) < \cdots < x(p_n)$. We shall present a divide-and-conquer algorithm that can be called with $R = S$ and $\text{Left}_\emptyset(p, S) = -\infty$ for all $p \in S$ to compute $l_S(p)$ for all $p \in S$. The correctness of the algorithm hinges on the following lemma.

* Some of the notations are slightly modified. In [7] min is used in the *label* function, instead of max. See [7] for details.

LEMMA 1.4 For any point $p \in R$, if $\mathcal{D}_S(p) \neq \emptyset$, then $l_S(p) = w(p) + \max\{Left_L(p, R), \max\{l_S(q), q \in \mathcal{M}(\mathcal{D}_R(p))\}\}$.

ALGORITHM MAXDOM_LABEL(R)

Input: A consecutive sequence of m points of S, i.e., $R = \{p_r, p_{r+1}, \ldots, p_{r+m-1}\}$ and for each $p \in R$, $\text{Left}_L(p, R)$, where $L = \{p_1, p_2, \ldots, p_{r-1}\}$. Assume a list Q_R of points of R sorted by increasing y-coordinate.

Output: The labels $l_S(q), q \in R$.

1. If $m = 1$ then we set $l_S(p_r)$ to $w(p_r) + \text{Left}_L(p_r, R)$, if $\text{Left}_L(p_r, R) \neq -\infty$ and to $w(p_r)$ if $\text{Left}_L(p_r, R) = -\infty$, and **return**.
2. Partition R by a vertical line $\mathcal{V}$ into subsets R_1 and R_2 such that $|R_1| = |R_2| = m/2$ and R_1 is to the left of R_2. Extract from Q_R the lists Q_{R_1} and Q_{R_2}.
3. Call MAXDOM_LABEL(R_1). Since $\text{Left}_L(p, R_1)$ equals $\text{Left}_L(p, R)$, this call will return the labels for all $q \in R_1$ which are the final labels for $q \in R$.
4. Compute $\text{Left}_{R_1}(p, R_2)$.
5. Compute $\text{Left}_{L \cup R_1}(p, R_2)$, given $\text{Left}_{R_1}(p, R_2)$ and $\text{Left}_L(p, R)$. That is, for each $p \in R_2$, set $\text{Left}_{L \cup R_1}(p, R_2)$ to be $\max\{\text{Left}_{R_1}(p, R_2), \text{Left}_L(p, R)\}$.
6. Call MAXDOM_LABEL(R_2). This will return the labels for all $q \in R_2$ which are the final labels for $q \in R$.

All steps other than Step 4 are self-explanatory. Steps 4 and 5 are needed in order to set up the correct invariant condition for the second recursive call. The computation of $\text{Left}_{R_1}(p, R_2)$ and its complexity is the key to the correctness and time complexity of the algorithm MAXDOM_LABEL(R). We briefly discuss this problem and show that this step can be done in $O(m)$ time. Since all other steps take linear time, the overall time complexity is $O(m \log m)$.

Consider in general two subsets L and R of points separated by a vertical line $\mathcal{V}$, with L lying to the left of R and points in $L \cup R$ are sorted in ascending y-coordinate (Figure 1.9). Suppose we have computed the labels $l_L(p), p \in L$. We compute $\text{Left}_L(p, R)$ by using a plane-sweep technique scanning

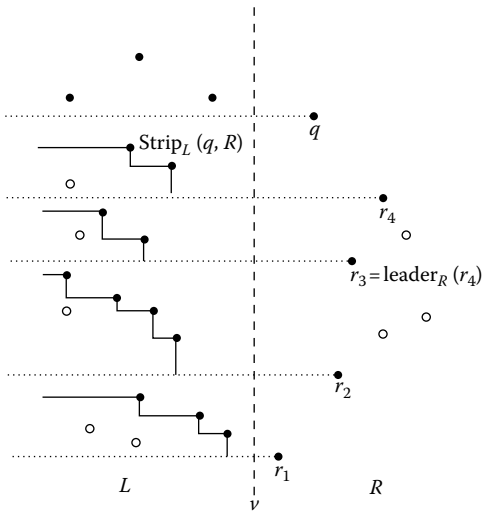

FIGURE 1.9 Computation of $\text{Left}_L(p, R)$.

points in $L \cup R$ in ascending y-coordinate. We will maintain for each point $r \in R$ $\text{Strip}_L(r, R)$ along with the highest and rightmost points in the subset, denoted $\text{1st}_L(r, R)$ and $\text{last}_L(r, R)$, respectively, and $\text{leader}_R(r)$. For each point $p \in L \cap \text{Strip}_L(r, R)$ for some $r \in R$ we maintain a label $\text{max\_}l(p)$, which is equal to $\max\{l_L(q) | q \in \text{Strip}_L(r, R) \text{ and } y(q) < y(p)\}$.

A stack ST_R will be used to store $\text{leader}_R(r_i)$ of $r_i \in R$ such that any element r_t in ST_R is $\text{leader}_R(r_{t+1})$ for point r_{t+1} above r_t, and the top element r_i of ST_R is the last scanned point in R. For instance in Figure 1.9 ST_R contains r_4, r_3, r_2, and r_1 when r_4 is scanned. Another stack ST_L is used to store $\text{Strip}_L(r, R)$ for a yet-to-be-scanned point $r \in R$. (The staircase above r_4 in Figure 1.9 is stored in ST_L. The solid staircases indicate $\text{Strip}_L(r_i, R)$ for r_i, $i = 2, 3, 4$.)

Let the next point scanned be denoted q. If $q \in L$, we pop off the stack ST_L all points that are dominated by q until q'. And we compute $\text{max\_}l(q)$ to be the larger of $l_L(q)$ and $\text{max\_}l(q')$. We then push q onto ST_L. That is, we update ST_L to make sure that all the points in ST_L are maximal.

Suppose $q \in R$. Then $\text{Strip}_L(q, R)$ is initialized to be the entire contents of ST_L and let $\text{1st}_L(q, R)$ be the top element of ST_L and $\text{last}_L(q, R)$ be the bottom element of ST_L.

If the top element of ST_R is equal to $\text{leader}_R(q)$, we set $\text{Left}_L(q, R)$ to $\text{max\_}l(q')$, where q' is $\text{1st}_L(q, R)$, initialize ST_L to be empty, and continue to scan the next point. Otherwise we need to pop off the stack ST_R all points that are *not* dominated by q, until q', which is $\text{leader}_R(q)$. As shown in Figure 1.9, $r_i, i = 4, 3, 2$ will be popped off ST_R when q is scanned. As point r_i is popped off ST_R, $\text{Strip}_L(r_i, R)$ is concatenated with $\text{Strip}_L(q, R)$ to maintain its maximality. That is, the points in $\text{Strip}_L(r_i, R)$ are scanned from $\text{1st}_L(r_i, R)$ to $\text{last}_L(r_i, R)$ until a point, if any, α_i is encountered such that $x(\alpha_i) > x(\text{last}_L(q, R))$. $\text{max\_}l(q')$, $q' = 1st_L(q, R)$, is set to be the larger of $\text{max\_}l(q')$ and $\text{max\_}l(\alpha)$, and $\text{last}_L(q, R)$ is temporarily set to be $\text{last}_L(r_i, R)$. If no such α_i exists, then the entire $\text{Strip}_L(r_i, R)$ is ignored. This process repeats until $\text{leader}_R(q)$ of q is on top of ST_R. At that point, we would have computed $\text{Strip}_L(q, R)$ and $\text{Left}_L(q, R)$ is $\text{max\_}l(q')$, where $q' = \text{1st}_L(q, R)$. We initialize ST_L to be empty and continue to scan the next point.

It has been shown in [7] that this scanning operation takes linear time (with path compression), so the overall algorithm takes $O(m \log m)$ time.

THEOREM 1.6 *Given a set S of n points with weights $w(p_i), p_i \in S, i = 1, 2, \ldots, n$, Algorithm MaxDom_Label(S) returns $l_S(p)$ for each point $p \in S$ in $O(n \log n)$ time.*

Now let us briefly describe the algorithm for the *maxdominance problem*. That is to find for each $p \in S$, $\mathcal{M}(\mathcal{D}_S(p))$.

Algorithm MaxDom_List(S)

Input: A sorted sequence of n points of S, i.e., $S = \{p_1, p_2, \ldots, p_n\}$, where $x(p_1) < x(p_2) < \cdots < x(p_n)$.

Output: $\mathcal{M}(\mathcal{D}_S(p))$ for each $p \in S$ and the list Q_S containing the points of S in ascending y-coordinates.

1. If $n = 1$ then we set $\mathcal{M}(\mathcal{D}_S(p_1)) = \emptyset$ and **return**.
2. Call Algorithm MaxDom_List(L), where $L = \{p_1, p_2, \ldots, p_{n/2}\}$. This call returns $\mathcal{M}(\mathcal{D}_S(p))$ for each $p \in L$ and the list Q_L.
3. Call Algorithm MaxDom_List(R), where $R = \{p_{n/2+1}, \ldots, p_n\}$. This call returns $\mathcal{M}(\mathcal{D}_R(p))$ for each $p \in R$ and the list Q_R.
4. Compute for each $r \in R$ $\text{Strip}_L(r, R)$ using the algorithm described in Step 4 of Algorithm MaxDom_Label(R).
5. For every $r \in R$ compute $\mathcal{M}(\mathcal{D}_S(p))$ by concatenating $\text{Strip}_L(r, R)$ and $\mathcal{M}(\mathcal{D}_R(p))$.
6. Merge Q_L and Q_R into Q_S and return.

Since Steps 4, 5, and 6, excluding the output time, can be done in linear time, we have the following.

THEOREM 1.7 *The maxdominance problem of a set S of n points in the plane can be solved in* $O(n \log n + \mathcal{F})$ *time, where* $\mathcal{F} = \sum_{p \in S} |\mathcal{M}(\mathcal{D}_S(p))|$.

Note that the problem of computing the layers of maxima in three dimensions can be solved in $O(n \log n)$ time [12].

1.4 Row Maxima Searching in Monotone Matrices

The *row maxima-searching problem* in a matrix is that given an $n \times m$ matrix M of real entries, find the leftmost maximum entry in each row.

A matrix is said to be *monotone*, if $i_1 > i_2$ implies that $j(i_1) \geq j(i_2)$, where $j(i)$ is the index of the leftmost column containing the maximum in row i. It is totally monotone if all of its submatrices are monotone.

In fact if every 2×2 submatrix $M[i, j; k, l]$ with $i < j$ and $k < l$ is monotone, then the matrix is totally monotone. Or equivalently if $M(i, k) < M(i, l)$ implies $M(j, k) < M(j, l)$ for any $i < j$ and $k < l$, then M is totally monotone.

The algorithm for solving the row maxima-searching problem is due to Aggarwal et al. [2], and is commonly referred to as the SMAWK algorithm. Specifically the following results were obtained: $O(m \log n)$ time for an $n \times m$ monotone matrix, and $\theta(m)$ time, $m \geq n$, and $\theta(m(1 + \log(n/m)))$ time, $m < n$, if the matrix is totally monotone.

We use as an example the distance matrix between pairs of vertices of a convex n-gon P, represented as a sequence of vertices $p_1, p_2, \ldots, p_n$ in counterclockwise order. For an integer j, let $*j$ denote $((j-1) \bmod n) + 1$. Let M be an $n \times (2n-1)$ matrix defined as follows. If $i < j \leq i + n - 1$ then $M[i, j] = d(p_i, p_{*j})$, where $d(p_i, p_j)$ denotes the Euclidean distance between two vertices p_i and p_j. If $j \leq i$ then $M[i, j] = j - i$, and if $j \geq i + n$ then $M[i, j] = -1$. The problem of computing for each vertex its farthest neighbor is now the same as the row maxima-searching problem.

Consider submatrix $M[i, j; k, l]$, with $i < j$ and $k < l$, that has only positive entries, i.e., $i < j < k < l < i + n$. In this case vertices p_i, p_j, p_{*k}, and p_{*l} are in counterclockwise order around the polygon. From the triangle inequality we have $d(p_i, p_{*k}) + d(p_j, p_{*l}) \geq d(p_i, p_{*l}) + d(p_j, p_{*k})$. Thus, $M[i, j; k, l]$ is monotone. The nonpositive entries ensure that all other 2×2 submatrices are monotone. We will show below that the all farthest neighbor problem for each vertex of a convex n-gon can be solved in $O(n)$ time.

A straightforward divide-and-conquer algorithm for the row maxima-searching problem in monotone matrices is as follows.

ALGORITHM MAXIMUM_D&C

1. Find the maximum entry $j = j(i)$, in the ith row, where $i = \lceil \frac{n}{2} \rceil$.
2. Recursively solve the row maxima-searching problem for the submatrices $M[1, \ldots, \mathrm{i} - 1; 1, \ldots, j]$ when $i, j > 1$ and $M[i + 1, \ldots, n; j, \ldots, m]$ when $i < n$ and $j < m$.

The time complexity required by the algorithm is given by the recurrence

$$f(n, m) \leq m + \max_{1 \leq j \leq m} (f(\lceil n/2 \rceil - 1, j) + f(\lfloor n/2 \rfloor, m - j + 1))$$

with $f(0, m) = f(n, 1) =$ constant. We have $f(n, m) = O(m \log n)$.

Now let us consider the case when the matrix is totally monotone. We distinguish two cases: (a) $m \geq n$ and (b) $m < n$.

Case (a): Wide matrix $m \geq n$.
An entry $M[i, j]$ is bad if $j \neq j(i)$, i.e., column j is not a solution to row i. Column j, $M[*, j]$ is bad if all $M[i, j], 1 \leq i \leq n$ are bad.

LEMMA 1.5 For $j_1 < j_2$ if $M[r, j_1] \geq M[r, j_2]$, then $M[i, j_2], 1 \leq i \leq r$, are bad; otherwise $M[i, j_1], r \leq i \leq n$, are bad.

Consider an $n \times n$ matrix C, the *index* of C is defined to be the largest k such that $C[i, j], 1 \leq i < j$, $1 \leq j \leq k$ are bad.

The following algorithm REDUCE reduces in $O(m)$ time a totally monotone $m \times n$ matrix M to an $n \times n$ matrix C, a submatrix of M, such that for $1 \leq i \leq n$ it contains column $M^{j(i)}$. That is, bad columns of M (which are known not to contain solutions) are eliminated.

ALGORITHM REDUCE(M)

1. $C \leftarrow M; k \leftarrow 1;$
2. **while** C has more than n columns **do**

   case $C(k, k) \geq C(k, k+1)$ and $k < n$: $k \leftarrow k + 1;$
   $C(k, k) \geq C(k, k+1)$ and $k = n$: Delete column C^{k+1};
   $C(k, k) < C(k, k+1)$: Delete column C^k; **if** $k > 1$ **then** $k \leftarrow k - 1$
   end case

3. **return**(C)

The following algorithm solves the maxima-searching problem in an $n \times m$ totally monotone matrix, where $m \geq n$.

ALGORITHM MAX_COMPUTE(M)

1. $B \leftarrow$ REDUCE(M);
2. **if** $n = 1$ **then** output the maximum and **return**;
3. $C \leftarrow B[2, 4, \ldots, 2\lfloor n/2 \rfloor; 1, 2, \ldots, n]$;
4. Call MAX_COMPUTE(C);
5. From the known positions of the maxima in the even rows of B, find the maxima in the odd rows.

The time complexity of this algorithm is determined by the following recurrence:

$$f(n, m) \leq c_1 n + c_2 m + f(n/2, n)$$

with $f(0, m) = f(n, 1) =$ constant. We therefore have $f(n, m) = O(m)$.

Case (b): Narrow matrix $m < n$.
In this case we decompose the problem into m subproblems each of size $\lfloor n/m \rfloor \times m$ as follows. Let $r_i = \lfloor in/m \rfloor$, for $0 \leq i \leq m$. Apply MAX_COMPUTE to the $m \times m$ submatrix $M[r_1, r_2, \ldots, r_m; 1, 2, \ldots, m]$ to get $c_1, c_2, \ldots, c_m$, where $c_i = j(r_i)$. This takes $O(m)$ time. Let $c_0 = 1$. Consider submatrices $B_i = M[r_{i-1}+1, r_{i-1}+2, \ldots, r_i - 1; c_{i-1}, c_{i-1}+1, \ldots, c_i]$ for $1 \leq i \leq m$ and $r_{i-1} \leq r_i - 2$. Applying the straightforward divide-and-conquer algorithm to the submatrices, B_i, we obtain the column positions of the maxima for all

remaining rows. Since each submatrix has at most $\lfloor n/m \rfloor$ rows, the time for finding the maxima is at most $c(p_i - p_{i-1} + 1)\log(n/m)$ for some constant c. Summing over all $1 \leq i \leq m$ we get the total time, which is $O(m(1 + \log(n/m)))$. The bound can be shown to be tight [2].

The applications of the matrix-searching algorithm include the problems of finding all farthest neighbors for all vertices of a convex n-gon ($O(n)$ time), and finding the extremal (maximum perimeter or area) polygons (inscribed k-gons) of a convex n-gon ($O(kn + n\log n)$). If one adopts the algorithm by Hershberger and Suri [42] the above problems can be solved in $O(n)$ time. It is also used in solving the Largest Empty Rectangle Problem discussed in Section 2.3.6 of this book.

1.5 Decomposition

Polygon decomposition arises in pattern recognition [77] in which recognition of a shape is facilitated by first decomposing it into simpler components, called primitives, and comparing them to templates previously stored in a library via some similarity measure. This class of decomposition is called *component-directed decomposition*. The primitives are often convex.

1.5.1 Trapezoidalization

We will consider first *trapezoidalization* of a polygon P with n vertices, i.e., decomposition of the interior of a polygon into a collection of *trapezoids* with two horizontal sides, one of which may degenerate into a point, reducing a trapezoid to a triangle. Without loss of generality let us assume that no edge of P is horizontal. For each vertex v let us consider the horizontal line passing through v, denoted $\mathcal{H}_v$. The vertices of P are classified into three types. A vertex v is *regular* if the other two vertices adjacent to v lie on different sides of $\mathcal{H}_v$. A vertex v is a V-*cusp* if the two vertices adjacent to v are above $\mathcal{H}_v$, and is a Λ-*cusp* if the two vertices adjacent to v are below $\mathcal{H}_v$. In general the intersection of $\mathcal{H}_v$ and the interior of P consists of a number of horizontal segments, one of which contains v. Let this segment be denoted $\overline{v_\ell, v_r}$, where v_ℓ and v_r are called the left and right projections of v on the boundary of P, denoted ∂P, respectively. If v is regular, either $\overline{v, v_\ell}$ or $\overline{v, v_r}$ lies totally in the interior of P. If v is a V-cusp or Λ-cusp, then $\overline{v_\ell, v_r}$ either lies totally in the interior of P or degenerates to v itself.

Consider only the segments $\overline{v_\ell, v_r}$ that are *nondegenerate*. These segments collectively partition the interior of P into a collection of trapezoids, each of which contains no vertex of P in its interior (Figure 1.10a).

The trapezoidalization can be generalized to a **planar straight-line graph** $G(V, E)$, where the entire plane is decomposed into trapezoids, some of which are unbounded. This trapezoidalization is sometimes referred to as horizontal **visibility map** of the edges, as the horizontal segments connect two edges of G that are *visible* (horizontally) (Figure 1.10b). The trapezoidalization of a planar straight-line graph $G(V, E)$ can be computed by plane-sweep technique in $O(n \log n)$ time, where $n = |V|$ [66], while the trapezoidalization of a simple polygon can be found in linear time [15].

The plane-sweep algorithm works as follows. The vertices of the graph $G(V, E)$ are sorted in descending y-coordinates. We will sweep the plane by a horizontal sweep-line from top down. Associated with this approach there are two basic data structures containing all relevant information that should be maintained.

1. Sweep-line status, which records the information of the geometric structure that is being swept. In this example the *sweep-line status* keeps track of the set of edges intersecting the current sweep-line.

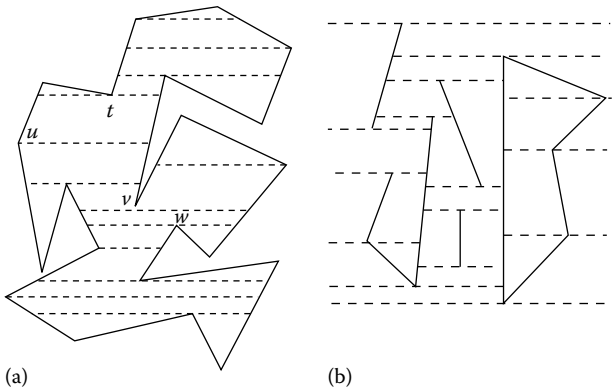

FIGURE 1.10 Trapezoidalization of a polygon (a) and horizontal visibility map of a planar straight line graph (b).

2. *Event schedule,* which defines a sequence of *event points* that the sweep-line status will change. In this example, the sweep-line status will change only at the vertices.

The event schedule is normally represented by a data structure, called *priority queue.* The content of the queue may not be available entirely at the start of the plane-sweep process. Instead, the list of events may change dynamically. In this case, the events are static; they are the y-coordinates of the vertices. The sweep-line status is represented by a suitable data structure that supports insertions, deletions, and computation of the left and right projections, $v_{\rm l}$ and $v_{\rm r}$, of each vertex v. In this example a *red-black tree* or any *height-balanced binary search tree* is sufficient for storing the edges that intersect the sweep-line according to the x-coordinates of the intersections. Suppose at event point v_{i-1} we maintain a list of edges intersecting the sweep-line from left to right. Analogous to the trapezoidalization of a polygon, we say that a vertex v is regular if there are edges incident on v that lie on different sides of $\mathcal{H}_v$; a vertex v is a V-cusp if all the vertices adjacent to v are above $\mathcal{H}_v$; v is a Λ-cusp if all the vertices adjacent to v are below $\mathcal{H}_v$. For each event point v_i we do the following.

1. v_i is regular. Let the leftmost and rightmost edges that are incident on v_i and above $\mathcal{H}_{v_i}$ are $E_\ell(v_i)$ and $E_{\rm r}(v_i)$, respectively. The left projection $v_{i\ell}$ of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the left of $E_\ell(v_i)$ in the sweep-line status. Similarly the right projection $v_{i\rm r}$ of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the right of $E_{\rm r}(v_i)$ in the sweep-line status. All the edges between $E_\ell(v_i)$ and $E_{\rm r}(v_i)$ in the sweep-line status are replaced in an order-preserving manner by the edges incident on v_i that are below $\mathcal{H}_{v_i}$.
2. v_i is a V-cusp. The left and right projections of v_i are computed in the same manner as in Step 1 above. All the edges incident on v_i are then deleted from the sweep-line status.
3. v_i is a Λ-cusp. We use binary search in the sweep-line status to look for the two adjacent edges $E_\ell(v_i)$ and $E_{\rm r}(v_i)$ such that v_i lies in between. The left projection $v_{i\ell}$ of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_\ell(v_i)$ and the right projection $v_{i\rm r}$ of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_{\rm r}(v_i)$. All the edges incident on v_i are then inserted in an order-preserving manner between $E_\ell(v_i)$ and $E_{\rm r}(v_i)$ in the sweep-line status.

Figure 1.11 illustrates these three cases. Since the update of the sweep-line status for each event point takes $O(\log n)$ time, the total amount of time needed is $O(n \log n)$.

THEOREM 1.8 *Given a planar straight-line graph $G(V, E)$, the horizontal visibility map of G can be computed in $O(n \log n)$ time, where $n = |V|$. However, if G is a simple polygon then the horizontal visibility map can be computed in linear time.*

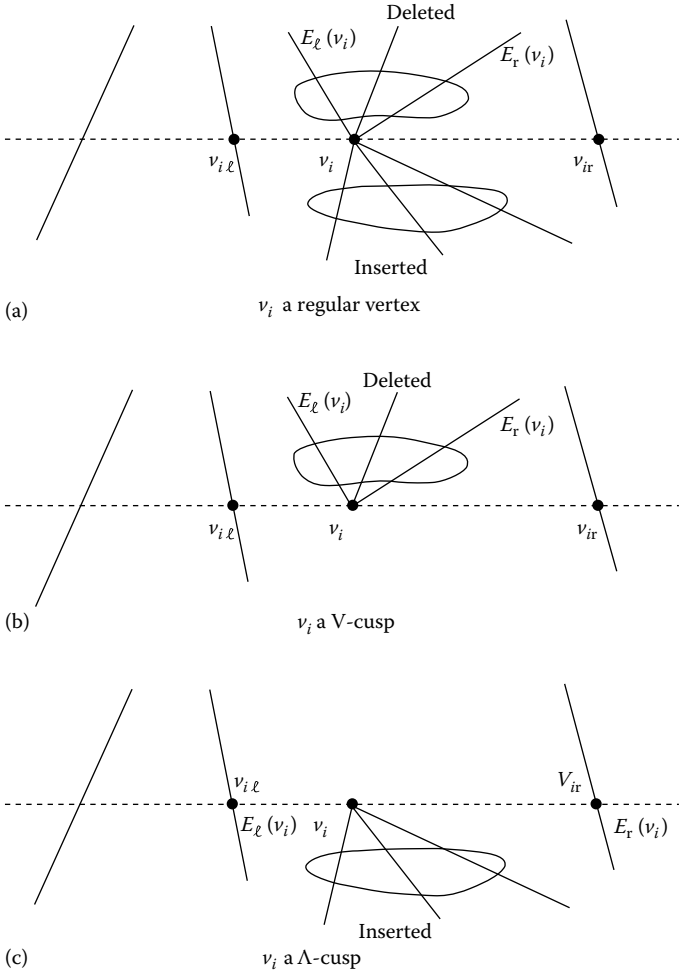

FIGURE 1.11 Updates of sweep-line status. (a) v_i is regular (b) v_i is a V-cusp and (c) v_i is a Λ-cusp.

1.5.2 Triangulation

In this section we consider triangulating a planar straight-line graph by introducing noncrossing edges so that each face in the final graph is a triangle and the outermost boundary of the graph forms a convex polygon. Triangulation of a set of (discrete) points in the plane is a special case. This is a fundamental problem that arises in computer graphics, geographical information systems, and finite element methods. Let us start with the simplest case.

1.5.2.1 Polygon Triangulation

Consider a simple polygon P with n vertices. It is obvious that to triangulate the interior of P (into $n-2$ triangles) one needs to introduce at most $n-3$ diagonals. A pioneering work is due to Garey et al. [37] who gave an $O(n \log n)$ algorithm and a linear algorithm if the polygon is *monotone*. A polygon is monotone if there exists a straight line $\mathcal{L}$ such that the intersection of ∂P and any line orthogonal to $\mathcal{L}$ consists of no more than two points. The shaded area in Figure 1.12 is a monotone polygon.

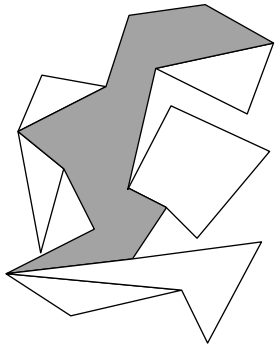

FIGURE 1.12 Decomposition of a simple polygon into monotone subpolygons.

The $O(n \log n)$ time algorithm can be illustrated by the following two-step procedure.

1. Decompose P into a collection of monotone subpolygons with respect to the y-axis in time $O(n \log n)$.
2. Triangulate each monotone subpolygons in linear time.

To find a decomposition of P into a collection of monotone polygons we first obtain the horizontal visibility map described in Section 1.5.1. In particular we obtain for each cusp v the left and right projections and the associated trapezoid below $\mathcal{H}_v$ if v is a V-cusp, and above $\mathcal{H}_v$ if v is a Λ-cusp. (Recall that $\mathcal{H}_v$ is the horizontal line passing through v.) For each V-cusp v we introduce an edge $\overline{v, w}$ where w is the vertex through which the other base of the trapezoid below passes. $\overline{t, u}$ and $\overline{v, w}$ in Figure 1.10a illustrate these two possibilities, respectively. For each Λ-cusp we do the same thing. In this manner we convert each vertex into a regular vertex, except the cusps v for which $\overline{v_l, v_r}$ lies totally outside of P, where v_l and v_r are the left and right projections of v in the horizontal visibility map. This process is called *regularization* [66]. Figure 1.12 shows a decomposition of the simple polygon in Figure 1.10a into a collection of monotone polygons.

We now describe an algorithm that triangulates a monotone polygon P in linear time. Assume that the monotone polygon has v_0 as the topmost vertex and v_{n-1} as the lowest vertex. We have two polygonal chains from v_0 to v_{n-1}, denoted $\mathcal{L}$ and $\mathcal{R}$, that define the left and right boundary of P, respectively. Note that vertices on these two polygonal chains are already sorted in descending order of their y-coordinates. The algorithm is based on a *greedy* method, i.e., whenever a triangle can be formed by connecting vertices either on the same chain or on opposite chains, we do so immediately. We shall examine the vertices in order, and maintain a polygonal chain $\mathcal{C}$ consisting of vertices whose internal angles are greater than π. Initially $\mathcal{C}$ consists of two vertices v_0 and v_1 that define an edge $\overline{v_0, v_1}$. Suppose $\mathcal{C}$ consists of vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_k}$, $k \geq 1$. We distinguish two cases for each vertex v_ℓ examined, $l < n - 1$. Without loss of generality we assume $\mathcal{C}$ is a left chain, i.e., $v_{i_k} \in \mathcal{L}$. The other case is treated symmetrically.

1. $v_\ell \in \mathcal{L}$. Let v_{i_j} be the last vertex on $\mathcal{C}$ that is visible from v_ℓ. That is, the internal angle $\cap(v_{i_j}, v_{i_{j'}}, v_\ell)$, where $j < j' \leq k$, is less than π, and either $v_{i_j} = v_{i_0}$ or the internal angle $\cap(v_{i_{j-1}}, v_{i_j}, v_\ell)$ is greater than π. Add diagonals $\overline{v_\ell, v_{i_{j'}}}$, for $j \leq j' < k$. Update $\mathcal{C}$ to be composed of vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_j}, v_\ell$.
2. $v_\ell \in \mathcal{R}$. In this case we add diagonals $\overline{v_\ell, v_{i_{j'}}}$, for $0 \leq j' \leq k$. $\mathcal{C}$ is updated to be composed of v_{i_k} and v_ℓ and it becomes a right chain.

Figure 1.13a and b illustrates these two cases, respectively, in which the shaded portion has been triangulated.

Fournier and Montuno [34] and independently Chazelle and Incerpi [17] showed that triangulation of a polygon is linear-time equivalent to computing the horizontal visibility map. Based on this result Tarjan and Van Wyk [76] first devised an $O(n \log \log n)$ time algorithm that computes the horizontal visibility map and hence, an $O(n \log \log n)$ time algorithm for triangulating a simple polygon. A breakthrough result of Chazelle [15] finally settled the longstanding open problem, i.e., triangulating a simple polygon in linear time. But the method is quite involved. As a result of this linear triangulation algorithm, a number of problems can be solved asymptotically in linear time. Note that if the polygons have holes, the problem of triangulating the interior is shown to require $\Omega(n \log n)$ time [5].

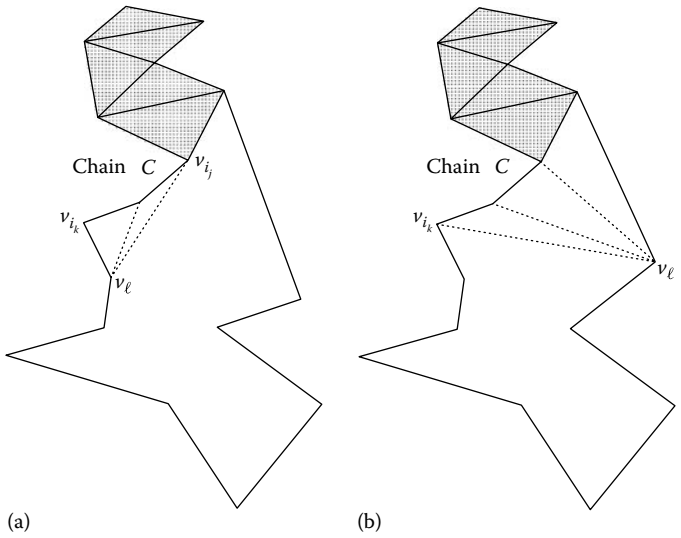

FIGURE 1.13 Triangulation of a monotone polygon. (a) v_ℓ on the left chain and (b) v_ℓ on the right chain.

1.5.2.2 Planar Straight-Line Graph Triangulation

The triangulation is also known as the *constrained triangulation.* This problem includes the triangulation of a set of points. A triangulation of a given planar straight-line graph $G(V, E)$ with $n = |V|$ vertices is a planar graph $\mathcal{G}(V, \mathcal{E})$ such that $E \subseteq \mathcal{E}$ and each face is a triangle, except the exterior one, which is unbounded. A constrained triangulation $\mathcal{G}(V, \mathcal{E})$ of a $G(V, E)$ can be obtained as follows.

1. Compute the convex hull of the set of vertices, ignoring all the edges. Those edges that belong to the convex hull are necessarily in the constrained triangulation. They define the boundary of the exterior face.
2. Compute the horizontalvisibility map for the graph, $G' = G \cup \mathrm{CH}(V)$, where $\mathrm{CH}(V)$ denotes the convex hull of V, i.e., for each vertex, its left and right projections are calculated, and a collection of trapezoids are obtained.
3. Apply the *regularization* process by introducing edges to vertices in the graph G' that are not regular. An isolated vertex requires two edges, one from above and one from below. Regularization will yield a collection of *monotone* polygons that comprise collectively the interior of $\mathrm{CH}(V)$.
4. Triangulate each monotone subpolygon.

It is easily seen that the algorithm runs in time $O(n \log n)$, which is asymptotically optimal. (This is because the problem of sorting is linearly reducible to the problem of constrained triangulation.)

1.5.2.3 Delaunay and Other Special Triangulations

Sometimes we want to look for quality triangulation, instead of just an arbitrary one. For instance, triangles with large or small angles is not desirable. The Delaunay triangulation of a set of points in the plane is a triangulation that satisfies the *empty circumcircle property,* i.e., the circumcircle of each triangle does not contain any other points in its interior. It is well-known that the Delaunay triangulation of points in general position is unique and it will maximize the minimum angle. In fact, the **characteristic angle vector** of the Delaunay triangulation of a set of points is *lexicographically maximum* [49]. The notion of Delaunay triangulation of a set of points can be generalized to a

planar straight-line graph $G(V, E)$. That is, we would like to have G as a subgraph of a triangulation $\mathcal{G}(V, \mathcal{E}'), E \subseteq \mathcal{E}'$, such that each triangle satisfies the empty circumcircle property: no vertex *visible* from the vertices of triangle is contained in the interior of the circle. This *generalized* Delaunay triangulation is thus a constrained triangulation that maximizes the minimum angle. The generalized Delaunay triangulation was first introduced by the author and an $O(n^2)$ (respectively, $O(n \log n)$) algorithm for constructing the generalized triangulation of a planar graph (respectively a simple polygon) with n vertices was given in [52]. As the generalized Delaunay triangulation (also known as *constrained* Delaunay triangulation) is of fundamental importance, we describe in Section 1.5.2.4 an optimal algorithm due to Chew [19] that computes the constrained Delaunay triangulation for a planar straight-line graph $G(V, E)$ with n vertices in $O(n \log n)$ time. Triangulations that minimize the maximum angle or maximum edge length [29] were also studied. But if the constraints are on the measure of the triangles, for instance, each triangle in the triangulation must be nonobtuse, then **Steiner points** must be introduced. See Bern and Eppstein (in [26, pp. 23–90]) for a survey of triangulations satisfying different criteria and discussions of triangulations in two and three dimensions. Bern and Eppstein gave an $O(n \log n + \mathcal{F})$ algorithm for constructing a nonobtuse triangulation of polygons using $\mathcal{F}$ triangles. Bern et al. [11] showed that $\mathcal{F}$ is $O(n)$ and gave an $O(n \log n)$ time algorithm for simple polygons without holes, and an $O(n \log^2 n)$ time algorithm for polygons with holes. For more results about acute triangulations of polygons and special surfaces see [56,83,84] and the references therein. The problem of finding a triangulation of a set of points in the plane whose total edge length is minimized, known as the *minimum weight triangulation*, is listed as an *open* problem (called *minimum length triangulation*) in Johnson's NP-complete column [45]. On the assumption that this problem is NP-hard, many researchers have obtained polynomial-time approximation algorithms for it. See Bern and Eppstein [10] for a survey of approximation algorithms. Only recently this problem was settled in the affirmative by Mulzer and Rote [63].

The problem of triangulating a set P of points in $\Re^k$, $k \geq 3$, is less studied. In this case the convex hull of P is to be partitioned into $\mathcal{F}$ nonoverlapping simplices, the vertices of which are points in P. A simplex in k-dimensions consists of exactly $k + 1$ points, all of which are extreme points. In $\Re^3$ $O(n \log n + \mathcal{F})$ time suffices, where $\mathcal{F}$ is linear if no three points are collinear, and $O(n^2)$ otherwise. Recently Saraf [71] gave a simpler proof that acute triangulations for general polyhedral surfaces exist and also showed that it is possible to obtain a nonobtuse triangulation of a general polyhedral surface. See [26] for more references on 3D triangulations and Delaunay triangulations in higher dimensions.

1.5.2.4 Constrained Delaunay Triangulation

Consider a **planar straight-line graph** $G(V, E)$, where V is a set of points in the plane, and edges in E are nonintersecting except possibly at the endpoints. Let $n = |V|$. Without loss of generality we assume that the edges on the convex hull $\mathrm{CH}(V)$ are all in E. These edges, if not present, can be computed in $O(n \log n)$ time (cf. Section 1.2.1).

In the constrained Delaunay triangulation $G_{DT}(V, \mathcal{E})$ the edges in $\mathcal{E} \setminus E$ are called *Delaunay* edges. It can be shown that two points $p, q \in V$ define a Delaunay edge if there exists a circle $\mathcal{K}$ passing through p and q which does not contain in its interior any other point visible from p and from q.

Let us assume without loss of generality that the points are in general position that no two have the same x-coordinate, and no four are cocircular. Let the points in V be sorted by ascending order of x-coordinate so that $x(p_i) < x(p_j)$ for $i < j$. Let us associate this set V (and graph $G(V, E)$) with it a bounding rectangle R_V with diagonal points $U(u.x, u.y), L(l.x, l.y)$, where $u.x = x(p_n), u.y = \max y(p_i), l.x = x(p_1), l.y = \min y(p_i)$. That is, L is at the lower left corner with x- and y-coordinates equal to the minimum of the x- and y-coordinates of all the points in V, and U is at the upper right corner. Given an edge $\overline{p_i, p_j}, i < j$, its x-interval is the interval $(x(p_i), x(p_j))$. The x-interval of the

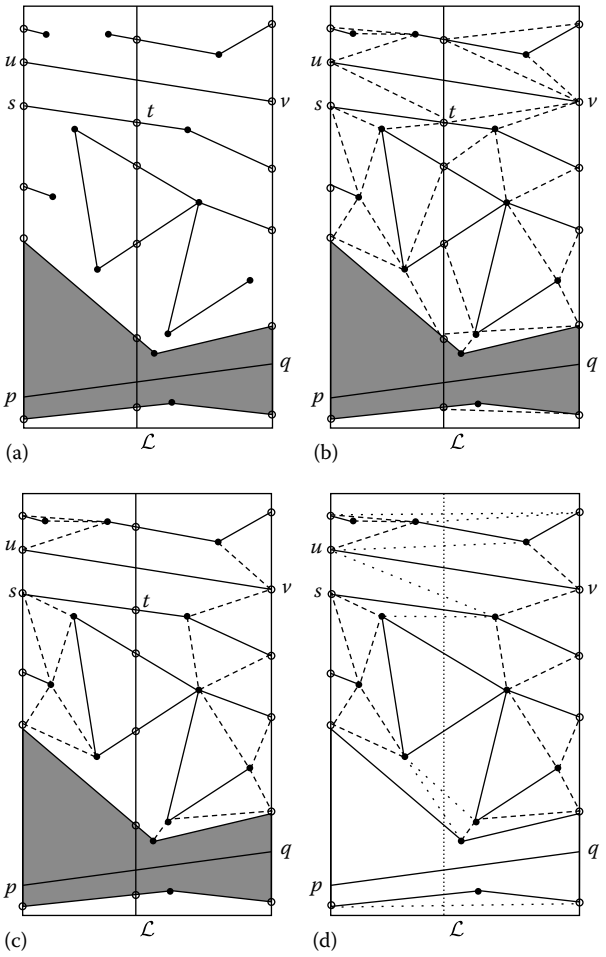

FIGURE 1.14 Computation of constrained Delaunay triangulation for subgraphs in adjacent bounding rectangles.

bounding rectangle R_V, denoted by $\mathcal{X}_V$, is the interval $(x(p_1), x(p_n))$. The set V will be recursively divided by vertical lines $\mathcal{L}$'s and so will be the bounding rectangles. We first divide V into two halves V_l and V_r by a line $\mathcal{L}(X = m)$, where $m = \frac{1}{2}(x(p_{\lfloor n/2 \rfloor}) + x(p_{\lfloor n/2 \rfloor + 1}))$. The edges in E that lie totally to the left and to the right of $\mathcal{L}$ are assigned respectively to the left graph $G_\ell(V_\ell, E_\ell)$ and to the right graph $G_\mathrm{r}(V_\mathrm{r}, E_\mathrm{r})$, which are associated respectively with bounding rectangle R_{V_ℓ} and R_{V_r} whose x-intervals are $\mathcal{X}_{V_\ell} = (x(p_1), m)$ and $\mathcal{X}_{V_\mathrm{r}} = (m, x(p_n))$, respectively. The edges $\overline{p, q} \in E$ that are intersected by the dividing line and that do not *span** the associated x-interval $\mathcal{X}_V$ will each get cut into *two edges* and a *pseudo* point $\epsilon(p, q)$ on the edge is introduced. Two edges, called *half-edges*, $\overline{p, \epsilon(p, q)} \in E_\ell$ and $\overline{\epsilon(p, q), q} \in E_\mathrm{r}$ are created. Figure 1.14a shows the creation of half-edges with pseudo points shown in hollow circles. Note that the edge $\overline{p, q}$ in the shaded area is not considered "present" in the associated bounding rectangle, $\overline{u, v}$ spans the x-interval for which no pseudo point is created, and $\overline{s, t}$ is a half-edge that spans the x-interval of the bounding rectangle to the left of the dividing line ($\mathcal{L}$).

* An edge $\overline{p, q}, x(p) < x(q)$ is said to span an x-interval (a, b), if $x(p) < a$ and $x(q) > b$.

It can be shown that for each edge $\overline{p,q} \in E$, the number of half-edges so created is at most $O(\log n)$. Within each bounding rectangle the edges that span its x-interval will divide the bounding rectangle into various parts. The constrained Delaunay triangulation gets computed for each part recursively. At the bottom of recursion each bounding rectangle contains at most three vertices of V, the edges incident on them, plus a number of half-edges spanning the x-interval including the pseudo endpoints of half-edges. Figure 1.14b illustrates an example of the constrained Delaunay triangulation at some intermediate step. No edges shall intersect one another, except at the endpoints.

As is usually the case for divide-and-conquer paradigm, the Merge step is the key to the method. We describe below the Merge step that combines constrained Delaunay triangulations in adjacent bounding rectangles that share a common dividing vertical edge $\mathcal{L}$.

1. Eliminate the pseudo points along the boundary edge $\mathcal{L}$, including the Delaunay edges incident on those pseudo points. This results in a partial constrained Delaunay triangulation within the union of these two bounding rectangles. Figure 1.14c illustrates the partial constrained Delaunay triangulation as a result of the removal of the Delaunay edges incident on the pseudo points on the border of the constrained Delaunay triangulation shown in Figure 1.14b.
2. Compute new Delaunay edges that cross $\mathcal{L}$ as follows. Let A and B be the two endpoints of an edge that crosses $\mathcal{L}$ with A on the left and B on the right of $\mathcal{L}$, and $\overline{A,B}$ is known to be part of the desired constrained Delaunay triangulation. That is, either $\overline{A,B}$ is an edge of E or a Delaunay edge just created. Either A or B may be a pseudo point. Let $\overline{A,C}$ be the first edge counterclockwise from edge $\overline{A,B}$. To decide if $\overline{A,C}$ remains to be a Delaunay edge in the desired constrained Delaunay triangulation, we consider the next edge $\overline{A,C_1}$, if it exists counterclockwise from $\overline{A,C}$. If $\overline{A,C_1}$ does not exist, or $\overline{A,C}$ is in E, $\overline{A,C}$ will remain. Otherwise we test if the circumcircle $\mathcal{K}(A,C,C_1)$ contains B in its interior. If so, $\overline{A,C}$ is eliminated, and the test continues on $\overline{A,C_1}$. Otherwise, $\overline{A,C}$ stays. We do the same thing to test edges incident on B, except that we consider edges incident on B in clockwise direction from $\overline{B,A}$. Assume now we have determined that both $\overline{A,C}$ and $\overline{B,D}$ remain. The next thing to do is to decide which of edge $\overline{B,C}$ and $\overline{A,D}$ should belong to the desired constrained Delaunay triangulation. We apply the circle test: test if circle $\mathcal{K}(A,B,C)$ contains D in the interior. If not, $\overline{B,C}$ is the desired Delaunay edge. Otherwise $\overline{A,D}$ is. We then repeat this step.

Step 2 of the merge step is similar to the method of constructing unconstrained Delaunay triangulation given in [52] and can be accomplished in linear time in the number of edges in the combined bounding rectangle. The dotted lines in Figure 1.14d are the Delaunay edges introduced in Step 2. We therefore conclude with the following theorem.

THEOREM 1.9 *Given a planar straight-line graph $G(V,E)$ with n vertices, the constrained Delaunay triangulation of G can be computed in $O(n \log n)$ time, which is asymptotically optimal.*

An implementation of this algorithm can be found in http://www.opencps.org/Members/ta/Constrained Delaunay Triangulation/cpstype_view

A plane-sweep method, for constructing the constrained Delaunay triangulation was also presented by De Floriani and Puppo [23] and by Domiter and Žalik [25]. The constrained Delaunay triangulation has been used as a basis for the so-called *Delaunay refinement* [31,40,61,68] for generating triangular meshes satisfying various *angle conditions* suitable for use in interpolation and the finite element method. Shewchuk [75] gave a good account of this subject and has also developed a tool, called Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator [74]. See http://www.cs.cmu.edu/~quake/triangle.html for more details.

1.5.3 Other Decompositions

Partitioning a simple polygon into shapes such as convex polygons, **star-shaped polygons**, spiral polygons, etc., has also been investigated. After a polygon has been triangulated one can partition the polygon into star-shaped polygons in linear time. This algorithm provided a very simple proof of the traditional art gallery problem originally posed by Klee, i.e., $\lfloor n/3 \rfloor$ vertex guards are always sufficient to see the entire region of a simple polygon with n vertices. But if a partition of a simple polygon into a minimum number of star-shaped polygons is desired, Keil [47] gives an $O(n^5N^2 \log n)$ time, where N denotes the number of reflex vertices. However, the problem of decomposing a simple polygon into a minimum number of star-shaped parts that may overlap is shown to be **NP-hard** [53,65]. This problem sometimes is referred to as the *covering* problem, in contrast to the *partitioning* problem, in which the components are not allowed to overlap. The problem of partitioning a polygon into a minimum number of convex parts can be solved in $O(N^2 n \log n)$ time [47]. It is interesting to note that it may not be possible to partition a simple polygon into convex quadrilaterals, but it is always possible for rectilinear polygons. The problem of determining if a convex quadrilateralization of a polygonal region (with holes) exists is NP-complete. It is interesting to note that $\lfloor n/4 \rfloor$ vertex guards are always sufficient for the art gallery problem in a rectilinear polygon. An $O(n \log n)$ algorithm for computing a convex quadrilateralization or positioning at most $\lfloor n/4 \rfloor$ guards is known (see [65] for more information). Modifying the proof of Lee and Lin [53], Schuchardt and Hecker [72] showed that the minimum covering problem by star-shaped polygons for rectilinear polygons is also NP-hard. Most recently Katz and Roisman [46] proved that even guarding the vertices of a rectilinear polygon with minimum number of guards is NP-hard. However, the minimum covering problem by **r-star-shaped polygons** or by **s-star-shaped polygons** for rectilinear polygons can be solved in polynomial time [62,80]. When the rectilinear polygon to be covered is monotone or *uni-s-star-shaped*,* then the minimum covering by r-star-shaped polygons can be found in linear time [38,55].

For variations and results of art gallery problems the reader is referred to [65,73,78]. Polynomial time algorithms for computing the minimum partition of a simple polygon into simpler parts while allowing Steiner points can be found in [5].

The minimum partition problem for simple polygons becomes NP-hard when the polygons are allowed to have *holes* [47]. Asano et al. [4] showed that the problem of partitioning a simple polygon with h holes into a minimum number of trapezoids with two horizontal sides can be solved in $O(n^{h+2})$ time, and that the problem is NP-complete if h is part of the input. An $O(n \log n)$ time 3-approximation algorithm was presented.

The problem of partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points) arises in VLSI artwork data. Imai and Asano [44] gave an $O(n^{3/2} \log n)$ time and $O(n \log n)$ space algorithm for partitioning a rectilinear polygon with holes into a minimum number of rectangles (allowing Steiner points). The problem of covering a rectilinear polygon (without holes) with a minimum number of rectangles, however, is NP-hard [21,41].

Given a polyhedron with n vertices and r notches (features causing nonconvexity), $\Omega(r^2)$ convex components are required for a complete convex decomposition in the worst case. Chazelle and Palios [18] gave an $O((n + r^2) \log r)$ time $O(n + r^2)$ space algorithm for this problem. Bajaj and Dey addressed a more general problem where the polyhedron may have holes and internal voids [9]. The problem of minimum partition into convex parts and the problem of determining if a nonconvex polyhedron can be partitioned into tetrahedra without introducing Steiner points are NP-hard [69].

---

* Recall that a rectilinear polygon P is s-star-shaped, if there exists a point p in P such that all the points of P are *s-visible* from p. When the staircase paths connecting p and all other *s-visible* points q in P are of the same (uniform) orientation, then the rectilinear polygon is *uni-s-star-shaped*.

1.6 Research Issues and Summary

We have covered in this chapter a number of topics in computational geometry, including convex hulls, maximal-finding problems, decomposition, and maxima searching in monotone matrices. Results, some of which are classical, and some which represent the state of the art of this field, were presented. More topics will be covered in Chapter 2.

In Section 1.2.3 an optimal algorithm for computing the layers of planar convex hulls is presented. It shows an interesting fact: within the same time as computing the convex hull (the outermost layer) of a point set in two dimensions, one can compute *all* layers of convex hull. Whether or not one can do the same for higher dimensions $\Re^k, k > 2$, remains to be seen. Although the triangulation problem of a simple polygon has been solved by Chazelle [15], the algorithm is far from being practical. As this problem is at the heart of this field, a simpler and more practical algorithm is of great interest. Recall that the following two minimum covering problems have been shown to be NP-hard: the minimum covering of rectilinear polygons by *rectangles* [21,41], and the minimum covering of rectilinear polygons by star-shaped polygons [46,72]. However, the minimum covering of rectilinear polygons by r-star-shaped polygons or by s-star-shaped polygons, as described above, can be found in polynomial time. There seems to be quite a complexity gap. Note that the components in the decomposition in these two cases, rectangles versus r-star-shaped polygons and star-shape polygons versus s-star-shaped polygons, are related in the sense that the former one is a special case of the latter. That is, a rectangle is an r-star-shaped polygon and a star-shaped primitive of a rectilinear polygon is s-star-shaped. But the converse is not true. When the primitives used in the covering problem are more restricted, they seem to make the minimum covering problem harder computationally. But when and if the fact that the minimum covering of a rectilinear polygon by s-star-shaped polygons solvable in polynomial time can be used to approximate the minimum covering by primitives of more restricted types is not clear. The following results, just to give some examples, were known. Eidenbenz et al. [30] presented inapproximability results of some art gallery problems and Fragoudakis et al. [35] showed certain maximization problems of guarded boundary of an art gallery to be APX-complete. There is a good wealth of problems related to art gallery or polygon decomposition that are worth further investigation.

1.7 Further Information

For some problems we present efficient algorithms in pseudocode and for others that are of more theoretical interest we only give a sketch of the algorithms and refer the reader to the original articles. The text book by de Berg et al. [22] contains a very nice treatment of this topic. The reader who is interested in *parallel* computational geometry is referred to [6]. For current research results, the reader may consult the *Proceedings of the Annual ACM Symposium on Computational Geometry*, *Proceedings of the Annual Canadian Conference on Computational Geometry, and the following three journals, Discrete & Computational Geometry, International Journal of Computational Geometry & Applications,* and *Computational Geometry: Theory and Applications.* More references can be found in [39,50,70,81]. The ftp site <ftp://ftp.cs.usask.ca/pub/geometry/geombib.tar.gz> contains close to 14,000 entries of bibliography in this field.

David Avis announced a convex hull/vertex enumeration code, *lrs,* based on reverse search and made it available. It finds all vertices and rays of a polyhedron in $\Re^k$ for any k, defined by a system of inequalities, and finds a system of inequalities describing the convex hull of a set of vertices and rays. More details can be found at this site http://cgm.cs.mcgill.ca/~avis/C/lrs.html. See Avis et al. [8] for more information about other convex hull codes. Those who are interested in the implementations or would like to have more information about other software available can consult http://www.geom.umn.edu/software/cglist/.

The following WWW page on *Geometry in Action* maintained by David Eppstein at http://www.ics.uci.edu/∼eppstein/geom.html and the Computational Geometry Pages by J. Erickson at http://compgeom.cs. uiuc.edu/∼jeffe/compgeom/ give a comprehensive description of research activities of computational geometry.

Defining Terms

Asymptotic time or space complexity: Asymptotic behavior of the time (or space) complexity of an algorithm when the size of the problem approaches infinity. This is usually denoted in big-Oh notation of a function of input size. A time or space complexity $T(n)$ is $O(f(n))$ means that there exists a constant $c > 0$ such that $T(n) \leq c \cdot f(n)$ for sufficiently large n, i.e., $n > n_0$, for some n_0.

CAD/CAM: Computer-aided design and computer-aided manufacturing, a discipline that concerns itself with the design and manufacturing of products aided by a computer.

Characteristic angle vector: A vector of minimum angles of each triangle in a triangulation arranged in nondescending order. For a given point set the number of triangles is the same for all triangulations, and therefore each of these triangulation has a characteristic angle vector.

Divide-and-marriage-before-conquest: A problem-solving paradigm derived from divide-and-conquer. A term coined by the Kirkpatrick and Seidel [48], authors of this method. After the divide step in a divide-and-conquer paradigm, instead of conquering the subproblems by recursively solving them, a merge operation is performed first on the subproblems. This method is proven more effective than conventional divide-and-conquer for some applications.

Extreme point: A point in S is an extreme point if it cannot be expressed as a convex combination of other points in S. A convex combination of points $p_1, p_2, \ldots, p_n$ is $\Sigma_{i=1}^{n} \alpha_i p_i$, where $\alpha_i, \forall i$ is nonnegative, and $\Sigma_{i=1}^{n} \alpha_i = 1$.

Geometric duality: A transform between a point and a hyperplane in $\Re^k$ that preserves incidence and order relation. For a point $p = (\mu_1, \mu_2, \ldots, \mu_k)$, its dual $\mathcal{D}(p)$ is a hyperplane denoted by $x_k = \Sigma_{j=1}^{k-1} \mu_j x_j - \mu_k$; for a hyperplane $\mathcal{H} : x_k = \Sigma_{j=1}^{k-1} \mu_j x_j + \mu_k$, its dual $\mathcal{D}(\mathcal{H})$ is a point denoted by $(\mu_1, \mu_2, \ldots, -\mu_k)$. See [22,27] for more information.

Height-balanced binary search tree: A data structure used to support membership, insert/delete operations each in time logarithmic in the size of the tree. A typical example is the *AVL* tree or *red-black* tree.

NP-hard problem: A complexity class of problems that are intrinsically *harder* than those that can be solved by a Turing machine in nondeterministic polynomial time. When a decision version of a combinatorial optimization problem is proven to belong to the class of NP-complete problems, which includes well-known problems such as satisfiability, traveling salesman problem, etc., an optimization version is NP-hard. For example, to decide if there exist k star-shaped polygons whose union is equal to a given simple polygon, for some parameter k, is NP-complete. The optimization version, i.e., finding a minimum number of star-shaped polygons whose union is equal to a given simple polygon, is NP-hard.

On-line algorithm: An algorithm is said to be online if the input to the algorithm is given one at a time. This is in contrast to the off-line case where the input is known in advance. The algorithm that works online is similar to the off-line algorithms that work incrementally, i.e., it computes a partial solution by considering input data one at a time.

Planar straight-line graph: A graph that can be embedded in the plane without crossings in which every edge in the graph is a straight line segment. It is sometimes referred to as planar subdivision or map.

Star-shaped polygon: A polygon P in which there exists an interior point p such that all the points of P are visible from p. That is, for any point q on the boundary of P, the intersection of the line segment $\overline{p,q}$ with the boundary of P is the point q itself.

r-Star-shaped polygon: A (rectilinear) polygon P in which there exists an interior point p such that all the points of P are r-visible from p. Two points in P are said to be r-visible if there exists a rectangle in P that totally contains these two points.

s-Star-shaped polygon: A (rectilinear) polygon P in which there exists an interior point p such that all the points of P are s-visible from p. Two points in P are said to be s-visible if there exists a staircase path in P that connects these two points. Recall that a staircase path is a rectilinear path that is monotone in both x- and y-directions, i.e., its intersection with every horizontal or vertical line is either empty, a point, or a line segment.

Steiner point: A point that is not part of the input set. It is derived from the notion of Steiner tree. Consider a set of three points determining a triangle $\Delta(a, b, c)$ all of whose angles are smaller than 120°, in the Euclidean plane. Finding a shortest tree interconnecting these three points is known to require a fourth point s in the interior such that each side of $\Delta(a, b, c)$ subtends the angle at s equal to 120°. The optimal tree is called the Steiner tree of the three points, and the fourth point is called the Steiner point.

Visibility map: A planar subdivision that encodes the visibility information. Two points p and q are visible if the straight line segment $\overline{p,q}$ does not intersect any other object. A horizontal (or vertical) visibility map of a planar straight-line graph is a partition of the plane into regions by drawing a horizontal (or vertical) straight line through each vertex p until it intersects an edge e of the graph or extends to infinity. The edge e is said to be horizontally (or vertically) visible from p.

References

1. Agarwal, P.K. and Matoušek, J., Ray shooting and parametric search, *SIAM J. Comput.*, 22, 794–806, 1993.
2. Aggarwal, A., Klawe, M.M., Moran, S., Shor, P., and Wilber, R., Geometric applications of a matrix-searching algorithm, *Algorithmica,* 2(2), 195–208, 1987.
3. Amato, N. and Preparata, F.P., The parallel 3D convex hull problem revisited, *Int. J. Comput. Geom. Appl.,* 2(2), 163–173, June 1992.
4. Asano, Ta., Asano, Te., and Imai, H., Partitioning a polygonal region into trapezoids, *J. Assoc. Comput. Mach.,* 33(2), 290–312, April 1986.
5. Asano, Ta., Asano, Te., and Pinter, R.Y., Polygon triangulation: Efficiency and minimality, *J. Algorithms,* 7, 221–231, 1986.
6. Atallah, M.J., Parallel techniques for computational geometry, *Proc. IEEE,* 80(9), 1435–1448, September 1992.
7. Atallah, M.J. and Kosaraju, S.R., An efficient algorithm for maxdominance with applications, *Algorithmica,* 4, 221–236, 1989.
8. Avis, D., Bremner, D., and Seidel, R., How good are convex hull algorithms, *Comput. Geom. Theory Appl.,* 7(5/6), 265–301, April 1997.
9. Bajaj, C. and Dey, T.K., Convex decomposition of polyhedra and robustness, *SIAM J. Comput.,* 21, 339–364, 1992.
10. Bern, M. and Eppstein, D., Approximation algorithms for geometric problems, in *Approximation Algorithms for NP-Hard Problems,* Hochbaum, D. (Ed.), PWS Publishing, Boston, MA, pp. 296–345, 1996.
11. Bern, M., Mitchell, S., and Ruppert, J., Linear-size nonobtuse triangulation of polygons, *Discrete Comput. Geom.,* 14, 411–428, 1995.

12. Buchsbaum, A.L. and Goodrich, M.T., Three dimensional layers of maxima, *Algorithmica,* 39(4), 275–286, May 2004.
13. Chan, T.M., Output-sensitive results on convex hulls, extreme points, and related problems, *Discrete Comput. Geom.,* 16(4), 369–387, April 1996.
14. Chazelle, B., On the convex layers of a planar set, *IEEE Trans. Inf. Theory,* IT-31, 509–517, 1985.
15. Chazelle, B., Triangulating a simple polygon in linear time, *Discrete Comput. Geom.,* 6, 485–524, 1991.
16. Chazelle, B., An optimal convex hull algorithm for point sets in any fixed dimension, *Discrete Comput. Geom.,* 8, 145–158, 1993.
17. Chazelle, B. and Incerpi, J., Triangulation and shape-complexity, *ACM Trans. Graph.,* 3(2), 135–152, 1984.
18. Chazelle, B. and Palios, L., Triangulating a non-convex polytope, *Discrete Comput. Geom.,* 5, 505–526, 1990.
19. Chew, L.P., Constrained Delaunay triangulations, *Algorithmica,* 4(1), 97–108, 1989.
20. Clarkson, K.L. and Shor, P.W., Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.,* 4, 387–421, 1989.
21. Culberson, J.C. and Reckhow, R.A., Covering polygons is hard, *J. Algorithms,* 17(1), 2–44, July 1994.
22. de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M., *Computational Geometry: Algorithms and Applications,* Springer-Verlag, Berlin, Germany, 386 pp., 2008.
23. De Floriani, L. and Puppo, E., An on-line algorithm for constrained Delaunay triangulation, *CVGIP Graph. Models Image Process.,* 54(4), 290–300, July 1992.
24. Dobkin, D.P. and Kirkpatrick, D.G., Fast detection of polyhedral intersection, *Theore. Comput. Sci.,* 27, 241–253, 1983.
25. Domiter, V. and Žalik, B., Sweep-line algorithm for constrained Delaunay triangulation, *Int. J. Geogr. Inf. Sci.,* 22(4), 449–462, 2008.
26. Du, D.Z. and Hwang, F.K. (Eds.), *Computing in Euclidean Geometry,* World Scientific, Singapore, 1992.
27. Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, Berlin, Germany, 1987.
28. Edelsbrunner, H. and Shi, W., An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem, *SIAM J. Comput.,* 20(2), 259–269, April 1991.
29. Edelsbrunner, H. and Tan, T.S., A quadratic time algorithm for the minmax length triangulation, *SIAM J. Comput.,* 22, 527–551, 1993.
30. Eidenbenz, S., Stamm, C., and Widmayer, P., Inapproximability results for guarding polygons and terrains, *Algorithmica,* 31(1), 79–113, 2001.
31. Erten, H. and Üngör, A., Triangulations with locally optimal Steiner points, *Proceedings of the 5th Eurographics Symposium on Geometry Processing,* Barcelona, Spain, pp. 143–152, 2007.
32. Fabri, A., Giezeman, G., Kettner, L., Schirra, S., and Schönherr, S., On the design of CGAL a computational geometry algorithms library, *Softw. Prac. Exp.,* 30(11), 1167–1202, Aug. 2000. http://www.cgal.org/
33. Fortune, S., Progress in computational geometry, in *Directions in Computational Geometry,* Martin, R. (Ed.), Information Geometers, Winchester, UK, pp. 81–128, 1993.
34. Fournier, A. and Montuno, D.Y., Triangulating simple polygons and equivalent problems, *ACM Trans. Graph.,* 3(2), 153–174, 1984.
35. Fragoudakis, C., Markou, E., and Zachos, S., Maximizing the guarded boundary of an art gallery is APX-complete, *Comput. Geom. Theory Appl.*, 38(3), 170–180, October 2007.
36. Gabow, H.N., Bentley, J.L., and Tarjan, R.E., Scaling and related techniques for geometry problems, *Proceedings of the 16th ACM Symposium on Theory of Computing,* New York, pp. 135–143, 1984.
37. Garey, M.R., Johnson, D.S., Preparata, F.P., and Tarjan, R.E., Triangulating a simple polygon, *Inf. Proc. Lett.,* 7, 175–179, 1978.

38. Gewali, L., Keil, M., and Ntafos, S., On covering orthogonal polygons with star-shaped polygons, *Inf. Sci.,* 65(1–2), 45–63, November 1992.
39. Goodman, J.E. and O'Rourke, J. (Eds)., *The Handbook of Discrete and Computational Geometry,* CRC Press LLC, Boca Raton, FL, 1997.
40. Har-Peled, S. and Üngör, A., A time-optimal Delaunay refinement algorithm in two dimensions, *Proceedings of the 21st ACM Symposium on Computational Geometry,* ACM, New York, 228–236, June 2005.
41. Heinrich-Litan, L. and Lübbecke, M.E., Rectangle covers revisited computationally, *J. Exp. Algorithmics,* 11, 1–21, 2006.
42. Hershberger, J. and Suri, S., Matrix searching with the shortest path metric, *SIAM J. Comput.,* 26(6), 1612–1634, December 1997.
43. Houle, M.E., Imai, H., Imai, K., Robert, J.-M., and Yamamoto, P., Orthogonal weighted linear L_1 and L_∞ approximation and applications, *Discrete Appl. Math.,* 43, 217–232, 1993.
44. Imai, H. and Asano, Ta., Efficient algorithms for geometric graph search problems, *SIAM J. Comput.,* 15(2), 478–494, May 1986.
45. Johnson, D.S., The NP-completeness column, *ACM Trans. Algorithms,* 1(1), 160–176, July 2005.
46. Katz, M. and Roisman, G.S., On guarding the vertices of rectilinear domains, *Comput. Geom. Theory Appl.,* 39(3), 219–228, April 2008.
47. Keil, J.M., Decomposing a polygon into simpler components, *SIAM J. Comput.,* 14(4), 799–817, 1985.
48. Kirkpatrick, D.G. and Seidel, R., The ultimate planar convex Hull algorithm? *SIAM J. Comput.,* 15(1), 287–299, February 1986.
49. Lee, D.T., Proximity and reachability in the plane, PhD thesis, Tech. Rep. R-831, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1978.
50. Lee, D.T., Computational geometry, in *Computer Science and Engineering Handbook,* Tucker, A. (Ed)., CRC Press, Boca Raton, FL, pp. 111–140, 1996.
51. Lee, D.T., Lee, G.C., and Huang, Y.W., Knowledge management for computational problem solving, *J. Universal Comput. Sci.,* 9(6), 563–570, 2003.
52. Lee, D.T. and Lin, A.K., Generalized Delaunay triangulation for planar graphs, *Discrete Comput. Geom.,* 1, 201–217, 1986.
53. Lee, D.T. and Lin, A.K., Computational complexity of art gallery problems, *IEEE Trans. Inf. Theory,* IT-32, 276–282, 1986.
54. Lee, D.T. and Wu, Y.F., Geometric complexity of some location problems, *Algorithmica,* 1(1), 193–211, 1986.
55. Lingas, A., Wasylewicz, A., and Zylinski, P., Note on covering monotone orthogonal polygons with star-shaped polygons, *Inf. Proc. Lett.,* 104(6), 220–227, December 2007.
56. Maehara, H., Acute triangulations of polygons, *Eur. J. Combin.,* 23, 45–55, 2002.
57. Matoušek, J., Efficient partition trees, *Discrete Comput. Geom.,* 8(1), 315–334, 1992.
58. Matoušek, J. and Schwarzkopf, O., On ray shooting on convex polytopes, *Discrete Comput. Geom.,* 10, 215–232, 1993.
59. Matoušek, J. and Schwarzkopf, O., A deterministic algorithm for the three-dimensional diameter problem, *Comput. Geom. Theory Appl.,* 6(4), 253–262, July 1996.
60. Mehlhorn, K. and Näher, S., *LEDA, A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, Cambridge, U.K., 1999.
61. Miller, G.L., A time efficient Delaunay refinement algorithm, *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, 400–409, 2004.
62. Motwani, R., Raghunathan, A., and Saran, H., Covering orthogonal polygons with star polygons: The perfect graph approach, *J. Comput. Syst. Sci.*, 40(1), 19–48, 1990.
63. Mulzer, W. and Rote, G., Minimum weight triangulation is NP-Hard, *Proceedings of the 22nd ACM Symposium on Computational Geometry,* Sedona, AZ, 1–10, June 2006. Revised version to appear in *J. Assoc. Comput. Mach.,* 55(2), May 2008.

64. Nielsen, F., Output-sensitive peeling of convex and maximal layers, *Inf. Proc. Lett.,* 59(5), 255–259, September 1996.
65. O'Rourke, J., *Art Gallery Theorems and Algorithms,* Oxford University Press, New York, 1987.
66. Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction,* Springer-Verlag, New York, 1988.
67. Ramos, E.A., An optimal deterministic algorithm for computing the diameter of a three-dimensional point set, *Discrete Comput. Geom.*, 26(2), 233–244, 2001.
68. Ruppert, J., A Delaunay refinement algorithm for quality 2-dimensional mesh generation, *J. Algorithms,* 18(3), 548–585, May 1995.
69. Ruppert, J. and Seidel, R., On the difficulty of triangulating three-dimensional non-convex polyhedra, *Discrete Comput. Geom.,* 7, 227–253, 1992.
70. Sack, J. and Urrutia, J. (Eds.), *Handbook of Computational Geometry,* Elsevier Science Publishers, B.V. North-Holland, Amsterdam, the Netherlands, 2000.
71. Saraf, S., Acute and nonobtuse triangulations of polyhedral surfaces, *Eur. J. Combin.,* on line September 2008, doi:10.1016/j.ejc.2008.08.004.
72. Schuchardt, D. and Hecker, H.D., Two NP-hard art-gallery problems for ortho-polygons, *Math. Log. Q.,* 41, 261–267, 1995.
73. Shermer, T.C., Recent results in art galleries, *Proc. IEEE,* 80(9), 1384–1399, September 1992.
74. Shewchuk, J.R., Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator, in *Applied Computational Geometry: Towards Geometric Engineering,* Lin, M.C. and Manocha, D. (Eds.), Springer-Verlag, Berlin, Germany, pp. 203–222, May 1996.
75. Shewchuk, J.R., Delaunay refinement algorithms for triangular mesh generation, *Comput. Geom. Theory Appl.,* 22(1–3), 21–74, May 2002.
76. Tarjan, R.E. and Van Wyk, C.J., An $O(n \log \log n)$-time algorithm for triangulating a simple polygon, *SIAM J. Comput.,* 17(1), 143–178, February 1988. Erratum: 17(5), 1061, 1988.
77. Toussaint, G.T., New results in computational geometry relevant to pattern recognition in practice, in *Pattern Recognition in Practice II,* Gelsema, E.S. and Kanal, L.N. (Eds.), North-Holland, Amsterdam, the Netherlands, pp. 135–146, 1986.
78. Urrutia, J., Art gallery and illumination problems, in *Handbook on Computational Geometry,* Sack, J.R. and Urrutia, J. (Eds.), Elsevier Science Publishers, Amsterdam, the Netherlands, pp. 973–1026, 2000.
79. Wei, J.D., Tsai, M.H., Lee, G.C., Huang, J.H., and Lee, D.T., GeoBuilder: A geometric algorithm visualization and debugging system for 2D and 3D geometric computing, *IEEE Trans. Vis. Comput. Graph.,* 15(2), 234–248, March 2009.
80. Worman, C. and Keil, J.M., Polygon decomposition and the orthogonal art gallery problem, *Int. J. Comput. Geom. Appl.,* 17(2), 105–138, April 2007.
81. Yao, F.F., Computational geometry, in *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, van Leeuwen, J. (Ed.), MIT Press, Cambridge, MA, pp. 343–389, 1994.
82. Yap, C., Towards exact geometric computation, *Comput. Geom. Theory Appl.,* 7(3), 3–23, February 1997.
83. Yuan, L., Acute triangulations of polygons, *Discrete Comput. Geom.,* 34, 697–706, 2005.
84. Yuan, L. and Zamfirescu, T., Acute triangulations of flat möbius strips, *Discrete Comput. Geom.,* 37, 671–676, 2007.

2

Computational Geometry II

D.T. Lee
Academia Sinica

2.1 Introduction

This chapter is a follow-up of Chapter 1, which deals with geometric problems and their efficient solutions. The classes of problems that we address in this chapter include proximity, optimization, intersection, searching, point location, and some discussions of geometric software that has been developed.

2.2 Proximity

Geometric problems abound pertaining to the questions of how close two geometric entities are among a collection of objects or how similar two geometric patterns match each other. For example, in pattern classification and clustering, features that are similar according to some metric are to be clustered in a group. The two aircrafts that are the closest at any time instant in the air space will have the largest likelihood of collision with each other. In some cases one may be interested in how far

apart or how dissimilar the objects are. Some of these proximity-related problems will be addressed in this section.

2.2.1 Closest Pair

Consider a set S of n points in $\Re^k$. The closest pair problem is to find in S a pair of points whose distance is the minimum, i.e., find p_i and p_j, such that $d(p_i, p_j) = \min_{k \neq l}\{d(p_k, p_l)$, for all points $p_k, p_l \in S\}$, where $d(a, b)$ denotes the Euclidean distance between a and b. (The result below holds for any distance metric in Minkowski's norm.) Enumerating all pairs of distances to find the pair with the minimum distance would take $O(k \cdot n^2)$ time. As is well-known, in one dimension one can solve the problem much more efficiently: Since the closest pair of points must occur consecutively on the real line, one can sort these points and then scan them in order to solve the closest pair problem in $O(n \log n)$ time. The time complexity turns out to be the best possible, since the problem has a lower bound of $\Omega(n \log n)$, following from a linear time transformation from the *element uniqueness problem* [90].

But unfortunately there is no total ordering for points in $\Re^k$ for $k \geq 2$, and thus, sorting is not applicable. We will show that by using divide-and-conquer approach, one can solve this problem in $O(n \log n)$ optimal time. Let us consider the case when $k = 2$. In the following, we only compute the minimum distance between the closest pair; the actual identity of the closest pair that realizes the minimum distance can be found easily by some straightforward bookkeeping operations. Consider a vertical separating line $\mathcal{V}$ that divides S into S_1 and S_2 such that $|S_1| = |S_2| = n/2$. Let δ_i denote the minimum distance defined by the closest pair of points in $S_i, i = 1, 2$. Observe that the minimum distance defined by the closest pair of points in S is either δ_1, δ_2, or $d(p, q)$ for some $p \in S_1$ and $q \in S_2$. In the former case, we are done. In the latter, points p and q must lie in the vertical strip of width $\delta = \min\{\delta_1, \delta_2\}$ on each side of the separating line $\mathcal{V}$ (Figure 2.1). The problem now reduces to that of finding the closest pair between points in S_1 and S_2 that lie inside the strip $\mathcal{L}$ of width 2δ. This subset of points $\mathcal{L}$ possesses a special property, known as *sparsity*, i.e., for each square box* of length 2δ the number of points in $\mathcal{L}$ is bounded by a constant $c = 4 \cdot 3^{k-1}$, since in each set S_i, there exists no point that lies in the interior of the δ-ball centered at each point in $S_i, i = 1, 2$ [90] (Figure 2.2). It is this sparsity property that enables us to solve the *bichromatic closest pair problem* in $O(n)$ time.

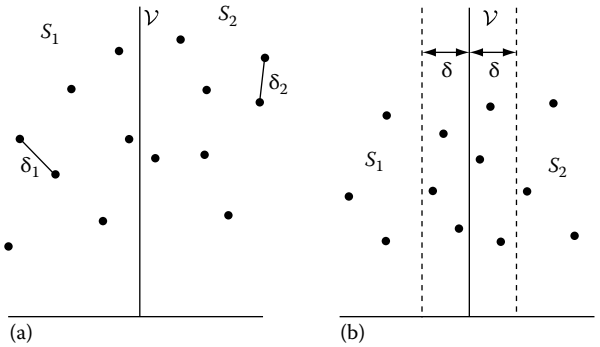

FIGURE 2.1 Divide-and-conquer scheme for closest pair problem. Solutions to subproblems S_1 and S_2 (a) and candidates must lie in the vertical strip of width δ on each side of $\mathcal{V}$ (b).

* A box is a hypercube in higher dimensions.

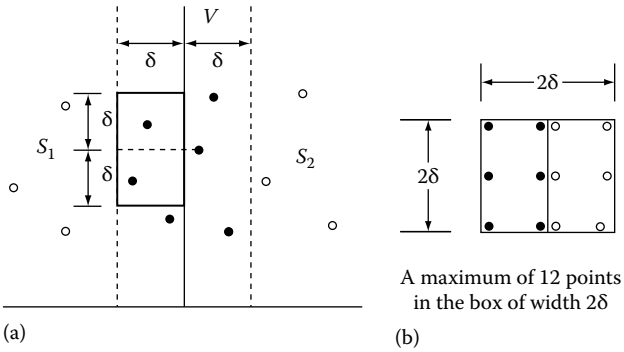

FIGURE 2.2 The box of width 2δ dissected by the separating line has at most 12 points; each point in S_2 needs to examine at most 6 points in S_1 to find its closest neighbor (a) and the box of width 2δ dissected by the separating line has at most 12 points (b).

The *bichromatic closest pair problem* is defined as follows. Given two sets of red and blue points, denoted R and B, find the closest pair $r \in R$ and $b \in B$, such that $d(r, b)$ is minimum among all possible distances $d(u, v), u \in R, v \in B$. Let $\overline{S}_i \subseteq S_i$ denote the set of points that lie in the vertical strip. In two dimensions, the sparsity property ensures that for each point $p \in \overline{S}_1$ the number of candidate points $q \in \overline{S}_2$ for the closest pair is at most six (Figure 2.2). We, therefore, can scan these points $\overline{S}_1 \cup \overline{S}_2$ in order along the separating line $\mathcal{V}$ and compute the distance between each point in $\overline{S}_1$ (respectively, $\overline{S}_2$) scanned and its six candidate points in $\overline{S}_2$ (respectively, $\overline{S}_1$). The pair that gives the minimum distance δ_3 is the bichromatic closest pair. The minimum distance of all pairs of points in S is then equal to $\delta_S = \min\{\delta_1, \delta_2, \delta_3\}$.

Since the merge step takes linear time, the entire algorithm takes $O(n \log n)$ time. This idea generalizes to higher dimensions, except that to ensure the sparsity property of the set $\mathcal{L}$, the separating hyperplane should be appropriately chosen so as to obtain an $O(n \log n)$-time algorithm [90], which is asymptotically optimal.

We note that the bichromatic closest pair problem is in general more difficult than the closest pair problem. Edelsbrunner and Sharir [46] showed that in three dimensions the number of possible closest pairs is $O((|R| \cdot |B|)^{2/3} + |R| + |B|)$. Agarwal et al. [3] gave a randomized algorithm with an expected running time of $O((mn \log m \log n)^{2/3} + m \log^2 n + n \log^2 m)$ in $\Re^3$ and $O((mn)^{1-1/(\lceil k/2 \rceil + 1)+\epsilon} + m \log n + n \log m)$ in $\Re^k$, $k \geq 4$, where $m = |R|$ and $n = |B|$. Only when the two sets possess the sparsity property defined above can the problem be solved in $O(n \log n)$ time, where $n = |R| + |B|$. A more general problem, known as the fixed radius all nearest neighbor problem in a sparse set [90], i.e., given a set M of points in $\Re^k$ that satisfies the sparsity condition, finds that all pairs of points whose distance is less than a given parameter δ can be solved in $O(|M| \log |M|)$ time [90].

The closest pair of vertices u and v of a simple polygon P such that $\overline{u, v}$ lies totally within P can be found in linear time [55]; $\overline{u, v}$ is also known as a diagonal of P.

2.2.2 Voronoi Diagrams

The Voronoi diagram $\mathcal{V}(S)$ of a set S of points, called *sites*, $S = \{p_1, p_2, \ldots, p_n\}$ in $\Re^k$ is a partition of $\Re^k$ into Voronoi cells $V(p_i)$, $i = 1, 2, \ldots, n$, such that each cell contains points that are closer to site p_i than to any other site $p_j, j \neq i$, i.e.,

$$V(p_i) = \left\{ x \in R^k \,\middle|\, d(x, p_i) \leq d(x, p_j) \, \forall p_j \in R^k, j \neq i \right\}.$$

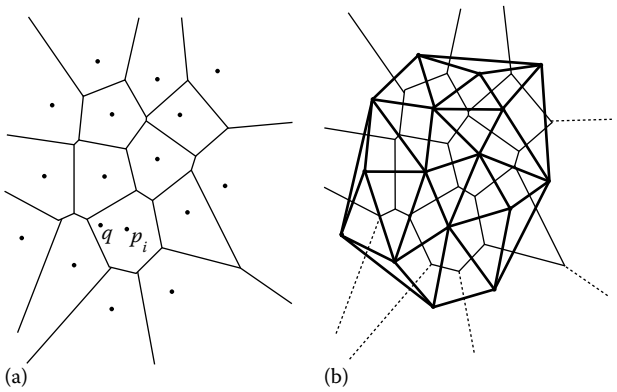

FIGURE 2.3 The Voronoi diagram of a set of 16 points in the plane (a) and its dual graph a Delaunay triangulation (b).

In two dimensions, $\mathcal{V}(S)$ is a planar graph and is of size linear in $|S|$. In dimensions $k \geq 2$, the total number of d-faces of dimensions $d = 0, 1, \ldots, k-1$, in $\mathcal{V}(S)$ is $O(n^{\lceil k/2 \rceil})$.

Figure 2.3a shows the Voronoi diagram of 16 point sites in two dimensions. Figure 2.3b shows the straight-line dual graph of the Voronoi diagram, which is called the Delaunay triangulation (cf. Section 1.5.2). In this triangulation, the vertices are the sites, and the two vertices are connected by an edge, if their Voronoi cells are adjacent.

2.2.2.1 Construction of Voronoi Diagrams in Two Dimensions

The Voronoi diagram possesses many proximity properties. For instance, for each site p_i, the closest site must be among those whose Voronoi cells are adjacent to $V(p_i)$. Thus, the closest pair problem for S in $\Re^2$ can be solved in linear time after the Voronoi diagram has been computed. Since this pair of points must be adjacent in the Delaunay triangulation, all one has to do is to examine all the adjacent pairs of points and report the pair with the smallest distance. A divide-and-conquer algorithm to compute the Voronoi diagram of a set of points in the L_p-metric for all $1 \leq p \leq \infty$ is known [62]. There is a rich body of literature concerning the Voronoi diagram. The interested reader is referred to the surveys [41,93].

We give below a brief description of a plane-sweep algorithm, known as the *wavefront approach*, due to Dehne and Klein [38]. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of point sites in $\Re^2$ sorted in ascending x-coordinate value, i.e., $x(p_1) < x(p_2) < \cdots < x(p_n)$. Consider that we sweep a vertical line $\mathcal{L}$ from left to right and as we sweep $\mathcal{L}$, we compute the Voronoi diagram $\mathcal{V}(S_t)$, where

$$S_t = \left\{ p_i \in S \,\middle|\, x\left(p_i\right) < t \right\} \cup \{\mathcal{L}_t\}.$$

Here $\mathcal{L}_t$ denotes the vertical line whose x-coordinate equals t. As is well-known, $\mathcal{V}(S_t)$ will contain not only straight-line segments, which are portions of perpendicular bisectors of two-point sites, but also parabolic curve segments, which are portions of *bisectors* of one-point site and $\mathcal{L}_t$. The wavefront W_t, consisting of a sequence of parabolae, called *waves*, is the boundary of the Voronoi cell $V(\mathcal{L}_t)$ with respect to S_t. Figure 2.4a and b illustrate two instances, $\mathcal{V}(S_t)$ and $\mathcal{V}(S_{t'})$. Those Voronoi cells that do not contribute to the wavefront are final, whereas those that do will change as $\mathcal{L}$ moves to the right. There are two possible events at which the wavefront needs an update. One, called site event, is when a site is hit by $\mathcal{L}$ and a new wave appears. The other, called spike event, is when an old wave disappears. Let p_i and p_j be two sites such that the associated waves are adjacent in W_t. The bisector of p_i and p_j defines an edge of $\mathcal{V}(S_t)$ to the left of W_t. Its extension into the cell $V(\mathcal{L}_t)$ is called a spike. The spikes can be viewed as tracks along which two neighboring waves travel.

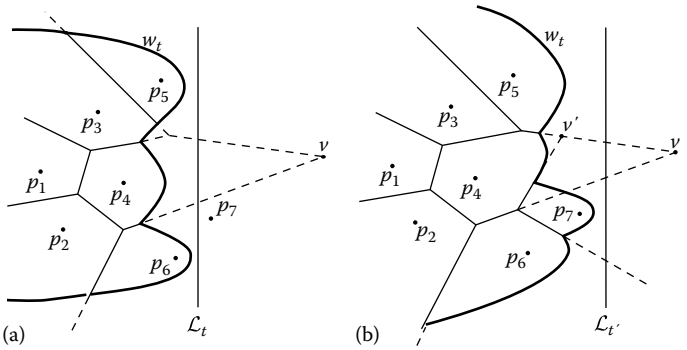

FIGURE 2.4 The Voronoi diagrams of (a) $\mathcal{V}(S_t)$ and (b) $\mathcal{V}(S_{t'})$.

A wave disappears from W_t, once it has reached the point where its two neighboring spikes intersect. In Figure 2.4a dashed lines are spikes and v is a potential spike event point. Without p_7 the wave of p_3 would disappear first and then the wave of p_4. After p_7, a site event, has occurred, a new point v' will be created and it defines an earlier spike event than v. v' will be a spike event point at which the wave of p_4 disappears and waves of p_5 and p_7 become adjacent. Note that the spike event corresponding to v' does not occur at $\mathcal{L}_t$, when $t = x(v')$. Instead, it occurs at $\mathcal{L}_t$, when $t = x(v') + d(v', p_4)$. If there is no site event between $\mathcal{L}_{x(p_7)}$ and $\mathcal{L}_t$, then the wave of p_4 will disappear. It is not difficult to see that after all site events and spike events have been processed at time τ, $\mathcal{V}(S)$ is identical to $\mathcal{V}(S_\tau)$ with the wavefront removed.

Since the waves in W_t can be stored in a *height-balanced binary search tree* and the site events and spike events can be maintained as a *priority queue*, the overall time and space needed are $O(n \log n)$ and $O(n)$, respectively.

Although $\Omega(n \log n)$ is the lower bound for computing the Voronoi diagram for an arbitrary set of n sites, this lower bound does not apply to special cases, e.g., when the sites are on the vertices of a convex polygon. In fact, the Voronoi diagram of a convex polygon can be computed in linear time [5]. This demonstrates further that additional properties of the input can sometimes help reduce the complexity of the problem.

2.2.2.2 Construction of Voronoi Diagrams in Higher Dimensions

The Voronoi diagrams in $\Re^k$ are related to the convex hulls $\Re^{k+1}$ via a **geometric duality** transformation. Consider a set S of n sites in $\Re^k$, which is the hyperplane $\mathcal{H}^0$ in $\Re^{k+1}$ such that $x_{k+1} = 0$, and a paraboloid $\mathcal{P}$ in $\Re^{k+1}$ represented as $x_{k+1} = x_1^2 + x_2^2 + \cdots + x_k^2$. Each site $p_i = (\mu_1, \mu_2, \ldots, \mu_k)$ is transformed into a hyperplane $\mathcal{H}(p_i)$ in $\Re^{k+1}$ denoted as $x_{k+1} = 2\Sigma_{j=1}^k \mu_j x_j - (\Sigma_{j=1}^k \mu_j^2)$. That is, $\mathcal{H}(p_i)$ is tangent to the paraboloid $\mathcal{P}$ at point $\mathcal{P}(p_i) = (\mu_1, \mu_2, \ldots, \mu_k, \mu_1^2 + \mu_2^2 + \cdots + \mu_k^2)$, which is just the vertical projection of site p_i onto the paraboloid $\mathcal{P}$. See Figure 2.5 for an illustration of the transformation in one dimension. The half-space defined by $\mathcal{H}(p_i)$ and containing the paraboloid $\mathcal{P}$ is denoted as $\mathcal{H}^+(p_i)$. The intersection of all half-spaces $\bigcap_{i=1}^n \mathcal{H}^+(p_i)$ is a convex body and the boundary of the convex body is denoted by $\mathrm{CH}(\mathcal{H}(S))$. Any point $q \in \Re^k$ lies in the Voronoi cell $V(p_i)$, if the vertical projection of q onto $\mathrm{CH}(\mathcal{H}(S))$ is contained in $\mathcal{H}(p_i)$. The distance between point q and its closest site p_i can be shown to be equal to the square root of the vertical distance between its vertical projection $\mathcal{P}(q)$ on the paraboloid $\mathcal{P}$ and

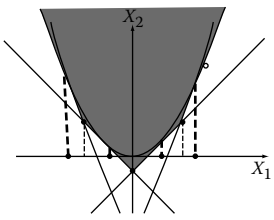

FIGURE 2.5 The paraboloid transformation of a site in one dimension to a line tangent to a parabola.

on CH($\mathcal{H}(S)$). Moreover every κ-face of CH($\mathcal{H}(S)$) has a vertical projection on the hyperplane $\mathcal{H}^0$ equal to the κ-face of the Voronoi diagram of S in $\mathcal{H}^0$.

We thus obtain the result which follows from the theorem for the convex hull in Chapter 1.

THEOREM 2.1 *The Voronoi diagram of a set S of n points in $\Re^k$, $k \geq 2$ can be computed in $O(n \log \mathcal{H})$ time for $k = 2$, and in $O(n \log \mathcal{H} + (n\mathcal{H})^{1-1/(\lfloor (k+1)/2 \rfloor + 1)} \log^{O(1)} n)$ time for $k > 2$, where $\mathcal{H}$ is the number of i-faces, $i = 0, 1, \ldots, k$.*

It has been shown that the Voronoi diagram in $\Re^k$, for $k = 3, 4$, can be computed in $O((n + \mathcal{H}) \log^{k-1} \mathcal{H})$ time [12].

2.2.2.3 Farthest Neighbor Voronoi Diagram

The Voronoi diagram defined in Section 2.2.2 is also known as the *nearest neighbor Voronoi diagram*. The nearest neighbor Voronoi diagram partitions the space into cells such that each site has its own cell, which contains all the points that are closer to this site than to any other site. A variation of this partitioning concept is a partition of the space into cells, each of which is associated with a site, and contains all the points that are farther from the site than from any other site. This diagram is called the *farthest neighbor Voronoi diagram*. Unlike the nearest neighbor Voronoi diagram, the farthest neighbor Voronoi diagram only has a subset of sites which have a Voronoi cell associated with them. Those sites that have a nonempty Voronoi cell are those that lie on the convex hull of S. A similar partitioning of the space is known as the *order κ-nearest neighbor Voronoi diagram*, in which each Voronoi cell is associated with a subset of κ sites in S for some fixed integer κ such that these κ sites are the closest among all other sites. For $\kappa = 1$, we have the nearest neighbor Voronoi diagram, and for $\kappa = n - 1$, we have the farthest neighbor Voronoi diagram. The construction of the order κ-nearest neighbor Voronoi diagram in the plane can be found in, e.g., [90]. *The order κ Voronoi diagrams* in $\Re^k$ are related to the levels of hyperplane arrangements in $\Re^{k+1}$ using the paraboloid transformation discussed in Section 2.2.2.2. See, e.g., [2] for details. Below is a discussion of the farthest neighbor Voronoi diagram in two dimensions.

Given a set S of sites $s_1, s_2, \ldots, s_n$, the f-neighbor Voronoi cell of site s_i is the locus of points that are farther from s_i than from any other site s_j, $i \neq j$, i.e.,

$$f\_V(s_i) = \left\{ p \in \Re^2 \middle| d(p, s_i) \geq d(p, s_j), s_i \neq s_j \right\}.$$

The union of these f-neighbor Voronoi cells is called the farthest neighbor Voronoi diagram of S. Figure 2.6 shows the farthest neighbor Voronoi diagram for a set of 16 sites. Note that only sites that are on the convex hull CH(S) will have a nonempty f-neighbor Voronoi cell [90] and that all the f-neighbor Voronoi cells are unbounded.

Since the farthest neighbor Voronoi diagram in the plane is related to the convex hull of the set of sites, one can use the divide-and-marriage-before-conquest paradigm to compute the farthest neighbor Voronoi diagram of S in two dimensions in time $O(n \log \mathcal{H})$, where $\mathcal{H}$ is the number of sites on the convex hull. Once the convex hull is available, the linear time algorithm [5] for computing the Voronoi diagram for a convex polygon can be applied.

2.2.2.4 Weighted Voronoi Diagrams

When the sites have weights such that the distance from a point to the sites is weighted, the structure of the Voronoi diagram can be drastically different than the unweighted case. We consider a few examples.

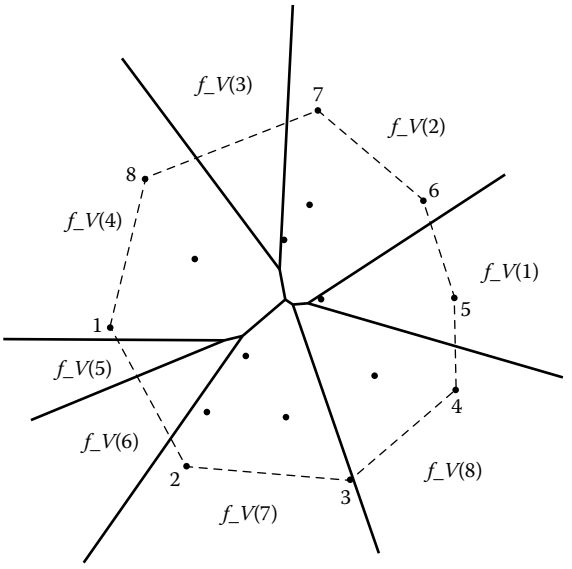

FIGURE 2.6 The farthest neighbor Voronoi diagram of a set of 16 sites in the plane.

Example 2.1: Power Diagrams

Suppose each site s in $\Re^k$ is associated with a nonnegative weight, w_s. For an arbitrary point p in $\Re^k$ the weighted distance from p to s is defined as

$$\delta(s,p) = d(s,p)^2 - w_s^2.$$

If w_s is positive, and if $d(s,p) \geq w_s$, then $\sqrt{\delta(s,p)}$ is the length of the tangent of p to the ball, $b(s)$, of radius w_s and centered at s. $\delta(s,p)$ is also called the *power of p* with respect to the ball $b(s)$. The locus of points p equidistant from two sites $s \neq t$ of equal weight will be a hyperplane called the *chordale* of s and t (see Figure 2.7). Point q is equidistant to sites a and b, and the distance is the length of the tangent line $\overline{q,c} = \overline{q,d}$.

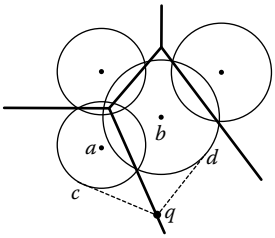

FIGURE 2.7 The power diagram in two dimensions. $\delta(q,a) = \delta(q,b) =$ length of $\overline{q,c}$.

The power diagram in two dimensions can be used to compute the contour of the union of n disks, and the connected components of n disks in $O(n \log n)$ time, and in higher dimensions, it can be used to compute the union or intersection of n axis-parallel cones in $\Re^k$ with apexes in a common hyperplane in time $O(\mathrm{CH}_{k+1}(n))$, the multiplicative-weighted nearest neighbor Voronoi diagram (defined below) for n points in $\Re^k$ in time $O(\mathrm{CH}_{k+2}(n))$, and the Voronoi diagrams for n spheres in $\Re^k$ in time $O(\mathrm{CH}_{k+2}(n))$, where $\mathrm{CH}_\ell(n)$ denotes the time for constructing the convex hull of n points in $\Re^\ell$ [93]. For the best time bound for $\mathrm{CH}_\ell(n)$, See Section 1.1. For more results on the union of spheres and the volumes see [45].

Example 2.2: Multiplicative-Weighted Voronoi Diagrams

Suppose each site $s \in \Re^k$ is associated with a positive weight w_s. The distance from a point $p \in \Re^k$ to s is defined as

$$\delta_{\mathrm{multi}-w}(s,p) = d(p,s)/w_s.$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a disk, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s,t}$, if $w_s = w_t$. Each cell associated with a site s consists

of all points closer to s than to any other site and may be disconnected. In the worst case, the multiplicative-weighted nearest neighbor Voronoi diagram of a set S of n points in two dimensions can have $O(n^2)$ regions and can be computed in $O(n^2)$ time. But in one dimension, the diagram can be computed optimally in $O(n \log n)$ time. On the other hand, the multiplicative-weighted farthest neighbor Voronoi diagram has a very different characteristic. Each Voronoi cell associated with a site remains connected, and the size of the diagram is still linear in the number of sites. An $O(n \log^2 n)$-time algorithm for constructing such a diagram is given in [66]. See [80] for more applications of the diagram.

Example 2.3: Additive-Weighted Voronoi Diagrams

Suppose each site $s \in \Re^k$ is associated with a positive weight w_s. The distance of a point $p \in \Re^k$ to a site s is defined as

$$\delta_{\text{add}-w}(s,p) = d(p,s) - w_s.$$

In two dimensions, the locus of points equidistant to two sites $s \neq t$ is a branch of a hyperbola, if $w_s \neq w_t$, and a perpendicular bisector of line segment $\overline{s,t}$, if $w_s = w_t$. The Voronoi diagram has properties similar to the ordinary unweighted diagram. For example, each cell is still connected and the size of the diagram is linear. If the weights are positive, the diagram is the same as the Voronoi diagram of a set of spheres centered at site s and of radius w_s, and in two dimensions, this diagram for n disks can be computed in $O(n \log n)$ time [15,81], and in $k \geq 3$, one can use the notion of power diagram (cf. Example 2.1) to compute the diagram [93].

2.2.2.5 Generalizations of Voronoi Diagrams

We consider two variations of Voronoi diagrams that are of interest and have applications.

Example 2.4: Geodesic Voronoi Diagrams

The nearest neighbor geodesic Voronoi diagram is a Voronoi diagram of sites in the presence of obstacles. The distance from point p to a site s, called the *geodesic distance* between p and s, is the length of the shortest path from p to s avoiding all the obstacles (cf. Section 2.6.1). The locus of points equidistant to two sites s and t is in general a collection of hyperbolic segments. The cell associated with a site is the locus of points whose geodesic distance to the site is shorter than to any other site [86]. The farthest neighbor geodesic Voronoi diagram can be similarly defined. Efficient algorithms for computing either kind of geodesic Voronoi diagram for k point sites in an n-sided simple polygon in $O((n+k)\log(n+k))$ time can be found in [86]. Figure 2.8 illustrates the geodesic Voronoi diagram of a set of point sites within a simple polygon; the whole shaded region is $V(s_i)$.

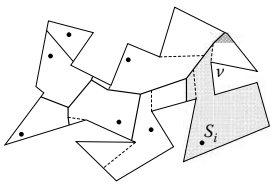

FIGURE 2.8 The geodesic Voronoi diagram within a simple polygon.

Example 2.5: Skew Voronoi Diagrams

A *directional distance* function between two points in the plane is introduced in Aichholzer et al. [8] that models a more realistic distance measure. The distance, called *skew distance*, from point p to point q is defined as

$$\tilde{d}(p,q) = d(p,q) + k \cdot d_y(p,q),$$

where $d_y(p,q) = y(q) - y(p)$, and $k \geq 0$ is a parameter. This distance function is *asymmetric* and satisfies $\tilde{d}(p,q) + \tilde{d}(q,p) = 2d(p,q)$, and the triangle inequality. Imagine we have a tilted plane $\boldsymbol{\mathcal{T}}$

obtained by rotating the *xy*-plane by an angle α about the *x*-axis. The height (*z*-coordinate) $h(p)$ of a point p on $\mathcal{T}$ is related to its *y*-coordinate by $h(p) = y(p) \cdot \sin \alpha$.

The distance function defined above reflects the cost that is proportional to the difference of their heights; the distance is *smaller* going *downhill* than going *uphill*. That is, the distance from p to q defined as $\tilde{d}(p,q) = d(p,q) + \kappa \cdot (h(q) - h(p))$ for $\kappa > 0$ serves this purpose; $\tilde{d}(p,q)$ is less than $\tilde{d}(q,p)$, if $h(q)$ is smaller than $h(p)$.

Because the distance is directional, one can define two kinds of Voronoi diagrams defined by the set of sites. A skew Voronoi cell from a site p, $\mathcal{V}_{\text{from}}(p)$, is defined as the set of points that are closest to p than to any other site. That is,

$$\mathcal{V}_{\text{from}}(p) = \left\{x \,\middle|\, \tilde{d}(p,x) \leq \tilde{d}(q,x)\right\}$$

for all $q \neq p$. Similarly one can define a skew Voronoi cell to a site p as follows:

$$\mathcal{V}_{\text{to}}(p) = \left\{x \,\middle|\, \tilde{d}(x,p) \leq \tilde{d}(x,q)\right\}$$

for all $q \neq p$.

The collection of these Voronoi cells for all sites is called the skew (or directional) Voronoi diagram.

For each site p, we define an r-disk centered at p, denoted $\text{from}_r(p)$ to be the set of points to which the skew distance from p is r. That is, $\text{from}_r(p) = \{x | \tilde{d}(p,x) = r\}$. Symmetrically, we can also define an r-disk centered at p, denoted $\text{to}_r(p)$ to be the set of points from which the skew distance to p is r. That is, $\text{to}_r(p) = \{x | \tilde{d}(x,p) = r\}$. The subscript r is omitted, when $r = 1$. It can be shown that $\text{to}_r(p)$ is just a mirror reflection of $\text{from}_r(p)$ about the horizontal line passing through p. We shall consider only the skew Voronoi diagram which is the collection of the cells $\mathcal{V}_{\text{from}}(p)$ for all $p \in S$.

LEMMA 2.1 For $k > 0$, the unit disk *from*(p) is a conic with focus p, directrix the horizontal line at y-distance $1/k$ above p, and eccentricity k. Thus, *from*(p) is an ellipse for $k < 1$, a parabola for $k = 1$, and a hyperbola for $k > 1$. For $k = 0$, *from*(p) is a disk with center p (which can be regarded as an ellipse of eccentricity zero).

Note that when k equals 0, the skew Voronoi diagram reduces to the ordinary nearest neighbor Voronoi diagram. When $k < 1$, it leads to known structures: By Lemma 2.1, the skew distance $\tilde{d}$ is a *convex distance function* and the Voronoi diagrams for convex distance functions are well studied (see, e.g., [93]). They consist of $O(n)$ edges and vertices, and can be constructed in time $O(n \log n)$ by divide-and-conquer.

When $k \geq 1$, since the unit disks are no longer bounded, the skew Voronoi diagrams have different behavior from the ordinary ones. As it turns out, some of the sites do not have nonempty skew Voronoi cells in this case. In this regard, it looks like ordinary farthest neighbor Voronoi diagram discussed earlier.

Let $L_0(p,k)$ denote the locus of points x such that $\tilde{d}(p,x) = 0$. It can be shown that for $k = 1$, $L_0(p,k)$ is a vertical line emanating downwards from p; and for $k > 1$, it consists of two rays, emanating from, and extending below, p, with slopes $1/(\sqrt{k^2-1})$ and $-1/(\sqrt{k^2-1})$, respectively. Let $N(p,k)$ denote the area below $L_0(p,k)$ (for $k > 1$). Let the *0-envelope*, $E_0(S)$, be the upper boundary of the union of all $N(p,k)$ for $p \in S$. $E_0(S)$ is the upper envelope of the graphs of all $L_0(p,k)$, when being seen as functions of the x-coordinate. For each point u lying above $E_0(S)$, we have $\tilde{d}(p,u) > 0$ for all $p \in S$, and for each point v lying below $E_0(S)$, there is at least one $p \in S$ with $\tilde{d}(p,v) < 0$. See Figure 2.9 for an example of a 0-envelope (shown as the dashed polygonal line) and the corresponding skew Voronoi diagram. Note that the skew Voronoi cells associated with sites q and t are empty. The following results are obtained [8].

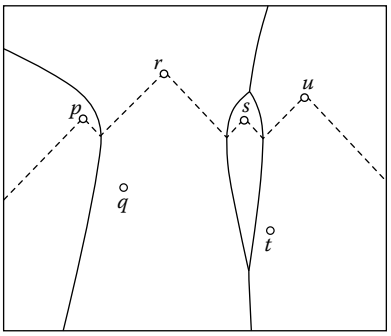

FIGURE 2.9 The 0-envelope and the skew Voronoi diagram when $k = 1.5$.

LEMMA 2.2 For $k > 1$, the 0-envelope $E_0(S)$ of a set S of n sites can be computed in $O(n \log \mathcal{H})$ time and $O(n)$ space, where $\mathcal{H}$ is the number of edges of $E_0(S)$.

LEMMA 2.3 Let $p \in S$ and $k > 1$, then $\mathcal{V}_{\text{from}}(p) \neq \emptyset$, if and only if $p \in E_0(S)$. $\mathcal{V}_{\text{from}}(p)$ is unbounded if and only if p lies on the upper hull of $E_0(S)$. For $k = 1$, $\mathcal{V}_{\text{from}}(p)$ is unbounded for all p.

THEOREM 2.2 *For any $k \geq 0$, the skew Voronoi diagram for n sites can be computed in $O(n \log \mathcal{H})$ time and $O(n)$ space, where $\mathcal{H}$ is the number of nonempty skew Voronoi cells in the resulting Voronoi diagram.*

The sites mentioned so far are point sites. They can be of different shapes. For instance, they can be line segments or polygonal objects. The Voronoi diagram for the edges of a simple polygon P that divides the interior of P into Voronoi cells is also known as the *medial axis* or *skeleton* of P [90]. The distance function used can also be the convex distance function or other norms.

2.3 Optimization

The geometric optimization problems arise in operations research, Very Large-Scale Integrated Circuit (VLSI) layout, and other engineering disciplines. We give a brief description of a few problems in this category that have been studied in the past.

2.3.1 Minimum Cost Spanning Tree

The minimum (cost) spanning tree (MST) of an undirected, weighted graph $G(V, E)$, in which each edge has a nonnegative weight, is a well-studied problem in graph theory and can be solved in $O(|E| \log |V|)$ time [90]. When cast in the Euclidean or other L_p-metric plane in which the input consists of a set S of n points, the complexity of this problem becomes different. Instead of constructing a complete graph with edge weight being the distance between its two endpoints, from which to extract an MST, a sparse graph, known as the *Delaunay triangulation* of the point set, is computed. The Delaunay triangulation of S, which is a planar graph, is the straight-line dual of the Voronoi diagram of S. That is, two points are connected by an edge, if and only if the Voronoi cells of these two sites share an edge. (cf. Section 1.5.2.3). It can be shown that the MST of S is a subgraph of the Delaunay triangulation. Since the MST of a planar graph can be found in linear time [90], the problem can be solved in $O(n \log n)$ time. In fact, this is asymptotically optimal, as the closest pair

of the set of points must define an edge in the MST, and the closest pair problem is known to have an $\Omega(n \log n)$ lower bound, as mentioned in Section 2.2.1.

This problem in dimensions three or higher can be solved in subquadratic time. Agarwal et al. [3] showed that the Euclidean MST problem for a set of N points in $\Re^k$ can be solved in time $O(\mathcal{T}_k(N, N) \log^k N)$, where $\mathcal{T}_k(m, n)$ denotes the time required to compute a bichromatic closest pair among m red and n blue points in $\Re^k$. If $\mathcal{T}_k(N, N) = \Omega(N^{1+\epsilon})$, for some fixed $\epsilon > 0$, then the running time improves to be $O(\mathcal{T}_k(N, N))$. They also gave a randomized algorithm with an expected time of $O((N \log N)^{4/3})$ in three dimensions and of $O(N^{2(1-1/(\lceil k/2 \rceil + 1))+\epsilon})$ for any positive ϵ in $k \geq 4$ dimensions [3]. Interestingly enough, if we want to find an MST that spans at least k nodes in a planar graph (or in the Euclidean plane), for some parameter $k \leq n$, then the problem, called k-MST problem, is NP-hard [92]. Approximation algorithms for the k-MST problem can be found in [18,75].

2.3.2 Steiner Minimum Tree

The *Steiner minimum tree* (SMT) of a set of vertices $S \subseteq V$ in an undirected weighted graph $G(V, E)$ is a spanning tree of $S \cup Q$ for some $Q \subseteq V$ such that the total weight of the spanning tree is minimum. This problem differs from MST in that we need to identify a set $Q \subseteq V$ of *Steiner vertices* so that the total cost of the spanning tree is minimized. Of course, if $S = V$, SMT is the same as MST. It is the identification of the Steiner vertices that makes this problem intractable. In the plane, we are given a set S of points and are to find the shortest tree interconnecting points in S, while additional Steiner points are allowed. Both Euclidean and rectilinear (L_1-metric) SMT problems are known to be NP-hard. In the geometric setting, the rectilinear SMT problem arises mostly in VLSI net routing, in which a number of terminals need to be interconnected using horizontal and vertical wire segments using the shortest wire length. As this problem is intractable, heuristics are proposed. For more information, the reader is referred to a special issue of *Algorithmica* on Steiner trees, edited by Hwang [57]. Most heuristics for the L_1 SMT problem are based on a classical theorem, known as the *Hanan grid theorem*, which states that the Steiner points of an SMT must be at the grid defined by drawing horizontal and vertical lines through each of the given points. However, when the number of orientations permitted for routing is greater than 2, the Hanan grid theorem no longer holds true. Lee and Shen [65] established a *multi-level grid* theorem, which states that the Steiner points of an SMT for n points must be at the grid defined by drawing λ lines in the feasible orientation recursively for up to $n - 2$ levels, where λ denotes the number of orientations of the wires allowed in routing. That is, the given points are assumed to be at the *0th level*. At each level, λ lines in the feasible orientations are drawn through each new grid point created at the previous level. In this λ-geometry plane, feasible orientations are assumed to make an angle $i\pi/\lambda$ with the positive x-axis. For the rectilinear case, $\lambda = 2$, Figure 2.10 shows that Hanan grid is insufficient for determining a Steiner SMT for $\lambda = 3$. Steiner point s_3 does not lie on the Hanan grid.

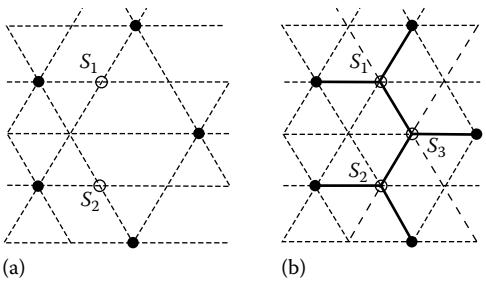

FIGURE 2.10 Hanan grid theorem fails for $\lambda = 3$. Steiner point s_3 does not lie on the Hanan grid (a), but they line on a second-level grid (b).

DEFINITION 2.1 The performance ratio of any approximation $\mathcal{A}$ in metric space $\mathcal{M}$ is defined as

$$\rho_{\mathcal{M}}(\mathcal{A}) = \inf_{P \in \mathcal{M}} \frac{L_s(P)}{L_{\mathcal{A}}(P)}$$

where $L_s(P)$ and $L_{\mathcal{A}}(P)$ denote, respectively, the lengths of a Steiner minimum tree and of the approximation $\mathcal{A}$ on P in space $\mathcal{M}$. When the MST is the approximation, the performance ratio is known as the Steiner ratio, denoted simply as ρ.

It is well-known that the Steiner ratios for the Euclidean and rectilinear SMTs are $\frac{\sqrt{3}}{2}$ and $\frac{2}{3}$, respectively [57]. The λ-*Steiner ratio** for the λ-geometry SMTs is no less than $\frac{\sqrt{3}\cos(\pi/2\lambda)}{2}$. The following interesting result regarding Steiner ratio is reported in [65], which shows that the Steiner ratio is not an increasing function from $\frac{2}{3}$ to $\frac{\sqrt{3}}{2}$, as λ varies from 2 to ∞.

THEOREM 2.3 *The λ-Steiner ratio is $\frac{\sqrt{3}}{2}$, when λ is a multiple of 6, and $\frac{\sqrt{3}\cos(\pi/2\lambda)}{2}$ when λ is a multiple of 3 but not a multiple of 6.*

2.3.3 Minimum Diameter Spanning Tree

The *minimum diameter spanning tree* (MDST) of an undirected weighted graph $G(V, E)$ is a spanning tree such that its **diameter**, i.e., total weight of the longest path in the tree, is minimum. This arises in applications to communication network where a tree is sought such that the maximum delay, instead of the total cost, is to be minimized. Using a graph-theoretic approach one can solve this problem in $O(|E||V| \log |V|)$ time. However, by the triangle inequality one can show that there exists an MDST such that the longest path in the tree consists of no more than *three segments* [56]. Based on this an $O(n^3)$-time algorithm was obtained.

THEOREM 2.4 *Given a set S of n points, the minimum diameter spanning tree for S can be found in $\theta(n^3)$ time and $O(n)$ space.*

We remark that the problem of finding a spanning tree whose total cost and the diameter are both bounded is NP-complete [56]. In [92], the problem of finding a *minimum diameter cost spanning* tree is studied. In this problem for each pair of vertices v_i and v_j there is a weighting function $w_{i,j}$ and the diameter *cost* of a spanning tree is defined to be the maximum over $w_{i,j} * d_{i,j}$, where $d_{i,j}$ denotes the distance between vertices v_i and v_j. To find a spanning tree with minimum diameter cost as defined above is shown to be NP-hard [92].

Another similar problem that arises in VLSI clock tree routing is to find a tree from a source to multiple sinks such that every source-to-sink path is the shortest rectilinear path and the total wire length is to be minimized. This problem, also known as *rectilinear Steiner arborescence problem* (see [57]), has been shown to be NP-complete [98]. A polynomial time approximation scheme of approximation ratio $(1 + 1/\epsilon)$ in time $O(n^{O(\epsilon)} \log n)$ was given by Lu and Ruan [70]. Later a simple 2-approximation algorithm in time $O(n \log n)$ was provided by Ranmath [91]. The problem of finding a minimum spanning tree such that the longest source-to-sink path is bounded by a given parameter is shown also to be NP-complete [96].

* The λ-Steiner ratio is defined as the greatest lower bound of the length of SMT over the length of MST in the λ-geometry plane.

2.3.4 Minimum Enclosing Circle

Given a set S of points the problem is to find the smallest disk enclosing the set. This problem is also known as the (unweighted) one-center problem. That is, find a center such that the maximum distance from the center to the points in S is minimized. More formally, we need to find the center $c \in \Re^2$ such that $\max_{p_j \in S} d(c, p_j)$ is minimized. The weighted one-center problem, in which the distance function $d(c, p_j)$ is multiplied by the weight w_j, is a well-known *min–max problem*, also referred to as the *emergency center problem* in operations research. In two dimensions, the one-center problem can be solved in $O(n)$ time. The minimum enclosing ball problem in higher dimensions is also solved by using linear programming technique [103]. The general p-center problem, i.e., finding p circles whose union contains S such that the maximum radius is minimized, is known to be NP-hard. For a special case when $p = 2$, Eppstein [48] gave an $O(n \log^2 n)$ randomized algorithm based on parametric search technique, and Chan [22] gave a deterministic algorithm with a slightly worse running time. For the problem of finding a minimum enclosing ellipsoid for a point set in $\Re^k$ and other types of geometric location problem see, e.g., [47,103].

2.3.5 Largest Empty Circle

This problem, in contrast to the minimum enclosing circle problem, is to find a circle centered in the interior of the convex hull of the set S of points that does not contain any given point and the radius of the circle is to be maximized. This is mathematically formalized as a *max–min problem*, the minimum distance from the center to the set is maximized. The weighted version is also known as the *obnoxious center problem* in facility location. For the unweighted version, the center must be either at a vertex of the Voronoi diagram for S in the convex hull or at the intersection of a Voronoi edge and the boundary of the convex hull. $O(n \log n)$ time is sufficient for this problem. Following the same strategy one can solve the largest empty square problem for S in $O(n \log n)$ time as well, using the Voronoi diagram in the L_∞-metric [62]. The time complexity of the algorithm is asymptotically optimal, as the maximum gap problem, i.e., finding the maximum gap between two consecutive numbers on the real line, which requires $\Omega(n \log n)$ time, is reducible to this problem [90]. In contrast to the minimum enclosing ellipsoid problem is the largest empty ellipsoid problem, which has also been studied [43].

2.3.6 Largest Empty Rectangle

In Section 1.2.4, we mentioned the smallest enclosing rectangle problem. Here, we look at the problem of finding the largest rectangle that is empty. Mukhopadhyay and Rao [78] gave an $O(n^3)$-time and $O(n^2)$-space algorithm for finding the largest empty arbitrarily oriented rectangle of a set of n points. A special case of this problem is to find the largest area *restricted* rectangle with sides parallel to those of the original rectangle containing a given set S of n points, whose interior contains no points from S. The problem arises in document analysis of printed-page layout in which white space in the black-and-white image of the form of a maximal empty rectangle is to be recognized. A related problem, called the *largest empty corner rectangle problem*, is that given two subsets S_l and S_r of S separated by a vertical line, find the largest rectangle containing no other points in S such that the lower left corner and the upper right corner of the rectangle are in S_l and S_r, respectively. This problem can be solved in $O(n \log n)$ time, where $n = |S|$, using fast matrix searching technique (cf. Section 1.4). With this as a subroutine, one can solve the largest empty restricted rectangle problem in $O(n \log^2 n)$ time. When the points define a rectilinear polygon that is orthogonally convex,the largest empty restricted rectangle that can fit inside the polygon can be found in $O(n\alpha(n))$ time, where $\alpha(n)$ is the slowly growing inverse of Ackermann's function using a result of Klawe and Kleitman [61]. When the polygon P is arbitrary and may contain holes, Daniels et al. [36] gave an

$O(n \log^2 n)$ algorithm, for finding the largest empty restricted rectangle in P. Orlowski [82] gave an $O(n \log n + s)$ algorithm, where s is the number of restricted rectangles that are possible candidates, and showed that s is $O(n^2)$ in the worst case and the expected value is $O(n \log n)$.

2.3.7 Minimum-Width Annulus

Given a set S of n points find an annulus (defined by two concentric circles) whose center lies internal to the convex hull of S such that the width of the annulus is minimized. This problem arises in dimensional tolerancing and metrology which deals with the specification and measurement of error tolerances in geometric shapes. To measure if a manufactured circular part is round, an American National Standards Institute (**ANSI**) standard is to use the width of an annulus covering the set of points obtained from a number of measurements. This is known as the roundness problem [51,99,100]. It can be shown that the center of the annulus can be located at the intersection of the nearest neighbor and the farthest neighbor Voronoi diagrams, as discussed in Section 2.2.2. The center can be computed in $O(n \log n)$ time [51]. If the input is defined by a simple polygon P with n vertices, then the problem is to find a minimum-width annulus that contains the boundary of P. The center of the smallest annulus can be located at the medial axis of P [100]. In particular, the problem can be solved in $O(n \log n + k)$, where k denotes the number of intersection points of the medial axis of the simple polygon and the farthest neighbor Voronoi diagram of the vertices of P. In [100], k is shown to be $\theta(n^2)$. However, if the polygon is convex, one can solve this problem in linear time [100]. Note that the minimum-width annulus problem is equivalent to the *best circle approximation problem*, in which a circle approximating a given shape (or a set of points) is sought such that the error is minimized. The error of the approximating circle is defined to be the maximum over all distances between points in the set and the approximating circle. To be more precise, the error is equal to one half of the width of the smallest annulus. See Figure 2.11.

If the center of the smallest annulus of a point set can be arbitrarily placed, the center may lie at infinity and the annulus degenerates to a pair of parallel lines enclosing the set of points. When the center is to be located at infinity, the problem becomes the well-known *minimum-width problem*, which is to find a pair of parallel lines enclosing the set such that the distance between them is minimized. The width of a set of n points can be computed in $O(n \log n)$ time, which is optimal [67]. In three dimensions the width of a set is also used as a measure for flatness of a plate, a flatness problem in computational metrology. Chazelle et al. [28] gave an $O(n^{8/5+\epsilon})$-time algorithm for this problem, improving over a previously known algorithm that runs in $O(n^2)$ time.

Shermer and Yap [97] introduced the notion of *relative roundness*, where one wants to minimize the ratio of the annulus width and the radius of the inner circle. An $O(n^2)$ algorithm was presented. Duncan et al. [42] define another notion of roundness, called *referenced roundness*, which becomes equivalent to the flatness problem when the radius of the reference circle is set to infinity. Specifically given a reference radius ρ of an annulus A that contains S, i.e., ρ is the mean of the two concentric

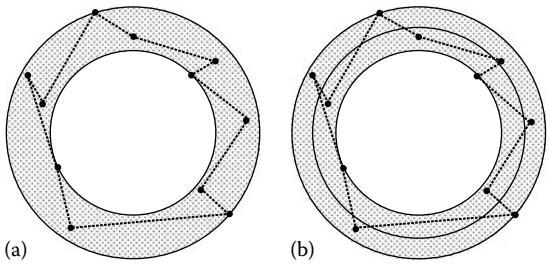

FIGURE 2.11 Minimum-width annulus (a) and the best circle approximation (b).

circles defining the annulus, find an annulus of a minimum width among all annuli with radius ρ containing S, or for a given $\epsilon > 0$, find an annulus containing S whose width is upper bounded by ϵ. They presented an $O(n \log n)$ algorithm for two dimensions and a near quadratic-time algorithm for three dimensions. In contrast to the minimum-width annulus problem is the *largest empty annulus problem*, in which we want to find the largest-width annulus that contains *no* points. The problem is much harder and can be solved in $O(n^3 \log n)$ time [39].

2.4 Geometric Matching

Matching in general graphs is one of the classical subjects in combinatorial optimization and has applications in operations research, pattern recognition, and VLSI design. Only geometric versions of the matching problem are discussed here. For graph-theoretic matching problems, see [83].

Given a weighted undirected complete graph on a set of $2n$ vertices, a *complete matching* is a set of n edges such that each vertex has exactly one edge incident on it. The weight of a matching is the sum of the weights of the edges in the matching. In a metric space, the vertices are points in the plane and the weight of an edge between two points is the distance between them. The *Euclidean minimum weight matching problem* is that given $2n$ points, find n matching pairs of points (p_i, q_i) such that $\Sigma d(p_i, q_i)$ is minimized.

It was not known if geometric properties can be exploited to obtain an algorithm that is faster than the $\theta(n^3)$ algorithm for general graphs (see [83]). Vaidya [101] settled this question in the affirmative. His algorithm is based on a well-studied primal-dual algorithm for weighted matching. Making use of additive-weighted Voronoi diagram discussed in Section 2.2.2.4 and the range search tree structure (see Section 2.7.1), Vaidya solved the problem in $O(n^{2.5} \log^4 n)$ time. This algorithm also generalizes to $\Re^k$ but the complexity is increased by a $\log^k n$ factor.

The *bipartite minimum weight matching* problem is defined similarly, except that we are given a set of red points $R = \{r_1, r_2, \ldots, r_n\}$ and a set of blue points $B = \{b_1, b_2, \ldots, b_n\}$ in the plane, and look for n matching pairs of points $(r, b) \in R \times B$ with minimum cost. In [101] Vaidya gave an $O(n^{2.5} \log n)$-time algorithm for Euclidean metric and an $O(n^2 \log^3 n)$ algorithm for L_1-metric. Approximation algorithms for this problem can be found in [7] and in [58].

If these $2n$ points are given as vertices of a polygon, the problems of minimum weight matching and bipartite matching can be solved in $O(n \log n)$ time if the polygon is convex and in $O(n \log^2 n)$ time if the polygon is simple. In this case, the weight of each matching pair of vertices is defined to be the geodesic distance between them [71]. However, if a *maximum weight matching* is sought, a $\log n$ factor can be shaved off [71].

Because of the triangle inequality, one can easily show that in a minimum weight matching, the line segments defined by the matched pairs of points cannot intersect one another. Generalizing this *nonintersecting property* the following *geodesic minimum matching problem* in the presence of obstacles can be formulated. Given $2m$ points and polygonal obstacles in the plane, find a matching of these $2m$ points such that the sum of the geodesic distances between the matched pairs is minimized. These m paths must not cross each other (they may have portions of the paths overlapping each other). There is no efficient algorithm known to date, except for the obvious method of reducing it to a minimum matching of a complete graph, in which the weight of an edge connecting any two points is the geodesic distance between them. Note that finding a geodesic matching without optimization is trivial, since these m noncrossing paths can always be found. This geodesic minimum matching problem in the general polygonal domain seems nontrivial. The *noncrossing* constraint and the optimization objective function (minimizing total weight) makes the problem hard.

When the matching of these $2m$ points is given *a priori*, finding m noncrossing paths minimizing the total weight seems very difficult. This resembles global routing problem in VLSI for which m two-terminal nets are given, and a routing is sought that optimizes a certain objective function,

including total wire length, subject to some capacity constraints. The noncrossing requirement is needed when single-layer routing or planar routing model is used. Global routing problems in general are NP-hard. Since the paths defined by matching pairs in an optimal routing cannot cross each other, paths obtained by earlier matched pairs become *obstacles* for subsequently matched pairs. Thus, the *sequence* in which the pairs of points are matched is very crucial. In fact, the path defined by a matched pair of points need not be the shortest. Thus, to route the matched pairs in a greedy manner sequentially does not give an optimal routing. Consider the configuration shown in Figure 2.12 in which $R = X, Y, Z$, $B = x, y, z$, and points (X, x), (Y, y), and (Z, z) are to be matched. Note that in this optimal routing, none of the matched pairs is realized by the shortest path, i.e., a straight line. This problem is referred to as the *shortest k-pair noncrossing path problem*. However, if the m matching pairs of points are on the boundary of a simple polygon, and the path must be confined to the interior of the polygon, Papadopoulou [84] gave an $O(n+m)$ algorithm for finding an optimal set of m noncrossing paths, if a solution exists, where n is the number of vertices of the polygon.

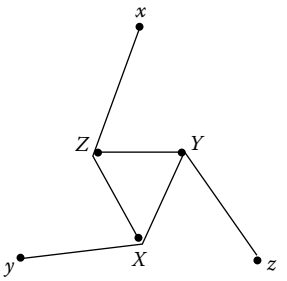

FIGURE 2.12 An instance of three noncrossing pair matching problem.

Atallah and Chen [14] considered the following bipartite matching problem: Given n red and n blue disjoint isothetic rectangles in the plane, find a matching of these n red–blue pairs of rectangles such that the rectilinear paths connecting the matched pairs are noncrossing and monotone. Surprisingly enough, they showed that such a matching satisfying the constraints always exists and gave an asymptotically optimal $O(n \log n)$ algorithm for finding such a matching.

To conclude this section we remark that the min–max versions of the general matching or bipartite matching problems are open. In the red-blue matching, if one of the sets is allowed to translate, rotate, or scale, we have a different matching problem. In this setting, we often look for the *best match* according to min–max criterion, i.e., the maximum error in the matching is to be minimized. A dual problem can also be defined, i.e., given a maximum error bound, determine if a matching exists, and if so, what kind of *motions* are needed. For more information about various types of matching, see a survey by Alt and Guibas [9].

2.5 Planar Point Location

Planar point location is a fundamental problem in computational geometry. Given a planar subdivision, and a query point, we want to find the region that contains the query point. Figure 2.13 shows an example of a planar subdivision. This problem arises in geographic information systems, in which one often is interested in locating, for example, a certain facility in a map. Consider the skew Voronoi diagram, discussed earlier in Section 2.2.2.5, for a set S of emergency dispatchers. Suppose an emergency situation arises at a location q and that the nearest dispatcher p is to be called so that the distance $\tilde{d}(p, q)$ is the smallest among all distances $\tilde{d}(r, q)$, for $r \in S$. This is equivalent to locating q in the Voronoi cell $\mathcal{V}_{\text{from}}(p)$ of the skew Voronoi diagram that contains q. In situations like this, it is vital that the nearest dispatcher be located quickly. We therefore address the point location problem under the assumption that the underlying planar map is fixed and the main objective is to have a fast response time to each query. Toward this end we preprocess the planar map into a suitable structure so that it would facilitate the point location task.

An earlier preprocessing scheme is based on the *slab method* [90], in which parallel lines are drawn through each vertex, thus, partitioning the plane into parallel slabs. Each parallel slab is further divided into subregions by the edges of the subdivision that can be linearly ordered. Any given query point q can thus be located by two binary searches; one to locate among the $n+1$ horizontal slabs the

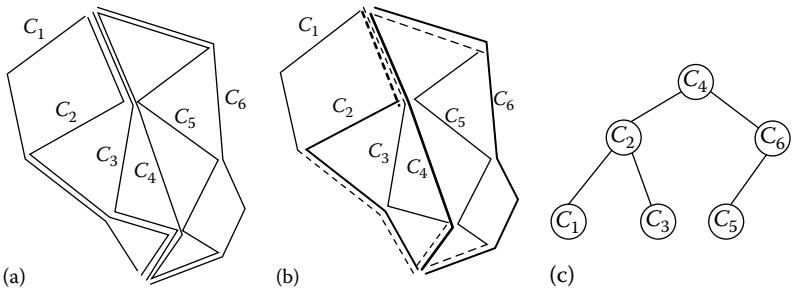

FIGURE 2.13 Chain decomposition method for a planar subdivision.

slab containing q, and followed by another to locate the region defined by a pair of consecutive edges which are ordered from left to right. We use a three tuple, $(P(n), S(n), Q(n))$= (preprocessing time, space requirement, query time) to denote the performance of the search strategy. The slab method gives an $(O(n^2), O(n^2), O(\log n))$ algorithm. Since preprocessing time is only performed once, the time requirement is not as critical as the space requirement, which is permanently engaged. The primary goal of any search strategy is to minimize the query time and the space required. Lee and Preparata [90] first proposed a *chain decomposition* method to decompose a monotone planar subdivision with n points into a collection of $m \leq n$ monotone chains organized in a complete binary tree. Each node in the binary tree is associated with a monotone chain of at most n edges, ordered in y-coordinate. This set of monotone chains forms a totally ordered set partitioning the plane into collections of regions. In particular, between two adjacent chains, there are a number of disjoint regions. The point location process begins with the root node of the complete binary tree. When visiting a node, the query point is compared with the node, hence, the associated chain, to decide on which side of the chain the query point lies. Each chain comparison takes $O(\log n)$ time, and the total number of nodes visited is $O(\log m)$. The search on the binary tree will lead to two adjacent chains, and, hence, identify a region that contains the point. Thus, the query time is $O(\log m \log n) = O(\log^2 n)$. Unlike the slab method in which each edge may be stored as many as $O(n)$ times, resulting in $O(n^2)$ space, it can be shown that with an appropriate chain assignment scheme, each edge in the planar subdivision is stored only once. Thus, the space requirement is $O(n)$. For example, in Figure 2.13, the edges shared by the root chain C_4 and its descendant chains are assigned to the root chain; in general, any edge shared by two nodes on the same root-to-leaf path will be assigned to the node that is an ancestor of the other node. The chain decomposition scheme gives rise to an $(O(n \log n), O(n), O(\log^2 n))$ algorithm. The binary search on the chains is not efficient enough. Recall that after each chain comparison, we will move down the binary search tree to perform the next chain comparison and start over another binary search on the same y-coordinate of the query point to find an edge of the chain, against which a comparison is made to decide if the point lies to the left or right of the chain. A more efficient scheme is to be able to perform a binary search of the y-coordinate at the root node and to spend only $O(1)$ time per node as we go down the chain tree, shaving off an $O(\log n)$ factor from the query time. This scheme is similar to the ones adopted by Chazelle and Guibas [37,90] in fractional cascading search paradigm and by Willard [37] in his range tree-search method. With the linear time algorithm for triangulating a simple polygon (cf. Section 1.5), we conclude with the following optimal search structure for planar point location.

THEOREM 2.5 *Given a planar subdivision of n vertices, one can preprocess the subdivision in linear time and space such that each point location query can be answered in $O(\log n)$ time.*

The point location problem in arrangements of hyperplanes is also of significant interest. See, e.g., [30]. **Dynamic** versions of the point location problem, where the underlying planar subdivision is subject to changes (insertions and deletions of vertices or edges), have also been investigated. See [34] for a survey of dynamic computational geometry.

2.6 Path Planning

This class of problems is mostly cast in the following setting. Given is a set of obstacles O, an object, called *robot*, and an initial and final position, called source and destination, respectively. We wish to find a path for the robot to move from the source to the destination avoiding all the obstacles. This problem arises in several contexts. For instance, in robotics this is referred to as the *piano movers' problem* or *collision avoidance problem*, and in VLSI design this is the routing problem for two-terminal nets. In most applications, we are searching for a collision avoidance path that has the shortest length, where the distance measure is based on the Euclidean or L_1-metric. For more information regarding motion planning see, e.g., [11,94].

2.6.1 Shortest Paths in Two Dimensions

In two dimensions, the Euclidean shortest path problem in which the robot is a point, and the obstacles are simple polygons, is well studied. A most fundamental approach is by using the notion of the *visibility graph*. Since the shortest path must make turns at polygonal vertices, it is sufficient to construct a graph whose vertices include the vertices of the polygonal obstacles, the source and the destination, and whose edges are determined by vertices that are mutually visible, i.e., the segment connecting the two vertices does not intersect the interior of any obstacle. Once the visibility graph is constructed with edge weight equal to the Euclidean distance between the two vertices, one can then apply the Dijkstra's shortest path algorithms [90] to find the shortest path between the source and destination. The Euclidean shortest path between two points is referred to as the *geodesic path* and the distance as the *geodesic distance*. The visibility graph for a set of polygonal obstacles with a total of n vertices can be computed trivially in $O(n^3)$ time. The computation of the visibility graph is the dominating factor for the complexity of any visibility graph-based shortest path algorithm. Research results aiming at more efficient algorithms for computing the visibility graph and the geodesic path in time proportional to the size of the graph have been obtained. For example, in [89] Pocchiola and Vetger gave an optimal output-sensitive algorithm that runs in $O(\mathcal{F} + n \log n)$ time and $O(n)$ space for computing the visibility graph, where $\mathcal{F}$ denotes the number of edges in the graph.

Mitchell [74] used the *continuous Dijkstra* wavefront approach to the problem for general polygonal domain of n obstacle vertices and obtained an $O(n^{3/2+\epsilon})$-time algorithm. He constructed the *shortest path map* that partitions the plane into regions such that all points q that lie in the same region have the same vertex sequence in the shortest path from the given source to q. The shortest path map takes $O(n)$ space and enables us to perform the shortest path queries, i.e., find the shortest path from the given source to any query points, in $O(\log n)$ time. Hershberger and Suri [54] on the other hand used plane subdivision approach and presented an $O(n \log^2 n)$-time and $O(n \log n)$-space algorithm to compute the shortest path map of a given source point. They later improved the time bound to $O(n \log n)$. It remains an open problem if there exists an algorithm for finding the Eucliedan shortest path in a general polygonal domain of h obstacles and n vertices in $O(n + h \log h)$ time and $O(n)$ space. If the source-destination path is confined in a simple polygon with n vertices, the shortest path can be found in $O(n)$ time [37].

In the context of VLSI routing, one is mostly interested in rectilinear paths (L_1-metric) whose edges are either horizontal or vertical. As the paths are restricted to be rectilinear, the shortest path problem can be solved more easily. Lee et al. [68] gave a survey on this topic.

In a two-layer routing model, the number of segments in a rectilinear path reflects the number of *vias*, where the wire segments change layers, which is a factor that governs the fabrication cost. In robotics, a straight-line motion is not as costly as *making turns*. Thus, the number of segments (or turns) has also become an objective function. This motivates the study of the problem of finding a path with the least number of segments, called the *minimum link path problem* [76].

These two cost measures, length and number of links, are in conflict with each other. That is, the shortest path may have far too many links, whereas a minimum link path may be arbitrarily long compared with the shortest path. A path that is optimal in both criteria is called the *smallest path*. In fact it can be easily shown that in a general polygonal domain, the smallest path does not exist. However, the smallest rectilinear path in a simple rectilinear polygon exists, and can be found in linear time. Instead of optimizing both measures *simultaneously* one can either seek a path that optimizes a linear function of both length and the number of links, known as the *combined* metric [104], or optimizes them in a lexicographical order. For example, we optimize the length first, and then the number of links, i.e., among those paths that have the same shortest length, find one whose number of links is the smallest and vice versa. In the rectilinear case see, e.g., [104]. Mitchell [76] gave a comprehensive treatment of the geometric shortest path and optimization problems.

A generalization of the collision avoidance problem is to allow collision with a cost. Suppose each obstacle has a weight which represents the cost if the obstacle is *penetrated*. Lee et al. [68] studied this problem in the rectilinear case. They showed that the shortest rectilinear path between two given points in the presence of weighted rectilinear polygons can be found in $O(n \log^{3/2} n)$ time and space. Chen et al. [31] showed that a data structure can be constructed in $O(n \log^{3/2} n)$ time and $O(n \log n)$ space that enables one to find the shortest path from a given source to any query point in $O(\log n + \mathcal{H})$ time, where $\mathcal{H}$ is the number of links in the path. Another generalization is to include in the set of obstacles some subset $F \subset O$ of obstacles, whose vertices are *forbidden* for the solution path to make turns. Of course, when the weight of obstacles is set to be ∞, or the forbidden set $F = \emptyset$, these generalizations reduce to the ordinary collision avoidance problem.

2.6.2 Shortest Paths in Three Dimensions

The Euclidean shortest path problem between two points in a three-dimensional polyhedral environment turns out to be much harder than its two-dimensional counterpart. Consider a convex polyhedron P with n vertices in three dimensions and two points s and d on the surface of P. The shortest path from s to d on the surface will cross a sequence of edges, denoted by $\xi(s, d)$. $\xi(s, d)$ is called the *shortest path edge sequence* induced by s and d and consists of distinct edges. For given s and d, the shortest path from s to d is not unique. However, $\xi(s, d)$ is unique. If $\xi(s, d)$ is known, the shortest path between s and d can be computed by a planar unfolding procedure so that these faces crossed by the path lie in a common plane and the path becomes a straight-line segment.

The shortest paths on the surface of a convex polyhedron P possess the following topological properties: (1) they do not pass through the vertices of P and do not cross an edge of P more than once, (2) they do not intersect themselves, i.e., they must be simple, and (3) except for the case of the two shortest paths sharing a common subpath, they intersect transversely in at most one point, i.e., they cross each other. If the shortest paths are grouped into equivalent classes according to the sequences of edges that they cross, then the number of such equivalent classes denoted by $|\xi(P)|$ is $\theta(n^4)$, where n is the number of vertices of P. These equivalent classes can be computed in $O(|\xi(P)|n^3 \log n)$ time. Chen and Han [32] gave an $O(n^2)$ algorithm for finding the shortest path between a fixed source s and any destination d, where n is the number of vertices and edges of the polyhedron, which may or may not be convex. If s and d lie on the surface of two different polyhedra, $O(N^{O(k)})$ time suffices for computing the shortest path between them amidst a set of k polyhedra, where N denotes the total number of vertices of these obstacles.

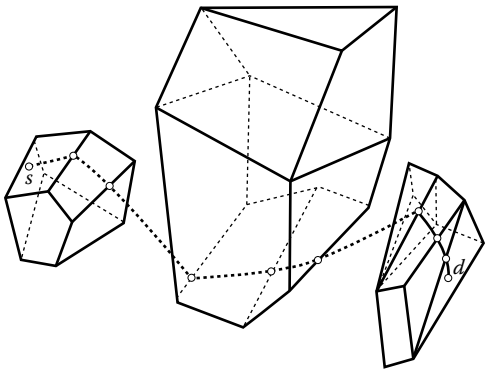

FIGURE 2.14 The possible shortest path edge sequence between points s and d.

The crux of the problem lies in the fact that the number of possible edge sequences may be exponential in the number of obstacles, if s and d lie on the surface of different polyhedra. It was established that the problem of determining the shortest path edge sequence is indeed NP-hard [11]. Figure 2.14 shows an example of the possible shortest path edge sequence induced by s and d in the presence of three convex polyhedra. Approximation algorithms for this problem can be found in, e.g., Choi et al. [35] and a recent article [76] for a survey.

2.7 Searching

This class of problems is cast in the form of query-answering. Given a collection of objects, with preprocessing allowed, one is to find the objects that satisfy the queries. The problem can be **static** or dynamic, depending on the fact that if the database to be searched is allowed to change over the course of query-answering sessions, and it is studied mostly in two modes, *count mode* and *report mode*. In the former case, only the *number* of objects satisfying the query is to be answered, whereas in the latter, the actual identity of the objects is to be reported. In the report mode, the query time of the algorithm consists of two components, the *search time* and the *retrieval time*, and expressed as $Q_{\mathcal{A}}(n) = O(f(n) + \mathcal{F})$, where n denotes the size of the database, $f(n)$ a function of n, and $\mathcal{F}$ the size of output. Sometimes we may need to perform some semigroup operations to those objects that satisfy the query. For instance, we may have weights $w(v)$ assigned to each object v, and we want to compute $\Sigma w(v)$ for all $v \cap q \neq \emptyset$. This is referred to as *semigroup range searching*. The semigroup range searching problem is the most general form: if the semigroup operation is set union, we get report-mode range searching problem and if the semigroup operation is just addition (of uniform weight), we have the count-mode range searching problem. We will not discuss the semigroup range searching here. It is obvious that algorithms that handle the report-mode queries can also handle the count-mode queries ($\mathcal{F}$ is the answer). It seems natural to expect that the algorithms for count-mode queries would be more efficient (in terms of the order of magnitude of the space required and query time), as they need not search for the objects. However, it was argued that in the report-mode range searching, one could take advantage of the fact that since reporting takes time, the more to report, the sloppier the search can be. For example, if we were to know that the ratio $n/\mathcal{F}$ is $O(1)$, we could use a sequential search on a linear list. This notion is known as the *filtering search* [90]. In essence, more objects than necessary are identified by the search mechanism, followed by a filtering process leaving out unwanted objects. As indicated below, the count-mode range searching problem is harder than the report-mode counterpart.

2.7.1 Range Searching

This is a fundamental problem in database applications. We will discuss this problem and the algorithm in the two-dimensional space. The generalization to higher dimensions is straightforward using a known technique, called multidimensional divide-and-conquer [90]. Given is a set of n points in the plane, and the ranges are specified by a product $(l_1, u_1) \times (l_2, u_2)$. We would like to find points $p = (x, y)$ such that $l_1 \leq x \leq u_1$ and $l_2 \leq y \leq u_2$. Intuitively we want to find those points that lie inside a query rectangle specified in the range. This is called *orthogonal range searching*, as opposed to other kinds of range searching problems discussed below, e.g., half-space range searching, and simplex range searching, etc. Unless otherwise specified, a range refers to an orthogonal range. We discuss the static case, as this belongs to the class of **decomposable searching problems**, the **dynamization transformation** techniques can be applied. We note that the range tree structure mentioned below can be made dynamic by using a weight-balanced tree, called $BB(\alpha)$ tree.

For count-mode queries, it can be solved by using the locus method as follows. In two dimensions, we can divide the plane into $O(n^2)$ cells by drawing a horizontal and a vertical line through each point. The answer to the query q, i.e., find the number of points dominated by q (those points whose x- and y-coordinates are both no greater than those of q), can be found by locating the cell containing q. Let it be denoted by $\mathrm{Dom}(q)$. Thus, the answer to the count-mode range queries can be obtained by some simple arithmetic operations of $\mathrm{Dom}(q_i)$ for the four corners, q_1, q_2, q_3, q_4, of the query rectangle. Let q_4 be the northeast corner and q_2 be the southwest corner. The answer will be $\mathrm{Dom}(q_4) - \mathrm{Dom}(q_1) - \mathrm{Dom}(q_3) + \mathrm{Dom}(q_2)$. Thus, in $\Re^k$, we have $Q(k, n) = O(k \log n)$, $S(k, n) = P(k, n) = O(n^k)$. To reduce space requirement at the expense of query time has been a goal of further research on this topic. Bentley introduced a data structure called the *range trees* [90]. Using this structure the following results were obtained: for $k \geq 2$, $Q(k, n) = O(\log^{k-1} n)$, $S(k, n) = P(k, n) = O(n \log^{k-1} n)$. See [1,4] for more recent results.

For report-mode queries, by using filtering search technique the space requirement can be further reduced by a $\log \log n$ factor. If the range satisfies additional conditions, e.g., grounded in one of the coordinates, say $l_1 = 0$, or the aspect ratio of the intervals specifying the range is fixed, less space is needed. For instance, in two dimensions, the space required is linear (a saving of $\log n / \log \log n$ factor) for these two cases. By using the so-called *functional approach* to data structures, Chazelle developed a compression scheme to reduce further the space requirement. Thus, in k-dimensions, $k \geq 2$, for the count-mode range queries, we have $Q(k, n) = O(\log^{k-1} n)$ and $S(k, n) = O(n \log^{k-2} n)$ and for report-mode range queries $Q(k, n) = O(\log^{k-1} n + \mathcal{F})$, and $S(k, n) = O(n \log^{k-2+\epsilon} n)$ for some $0 < \epsilon < 1$ [1,4].

As regards the lower bound of range searching in terms of space–time trade-offs, Chazelle [23] showed that in k-dimensions, if the query time is $O(\log^c n + \mathcal{F})$ for any constant c, the space required is $\Omega(n(\log n / \log \log n)^{k-1})$ for the pointer machine models and the bound is tight for any $c \geq 1$, if $k = 2$, and any $c \geq k - 1 + \epsilon$ (for any $\epsilon > 0$), if $k > 2$. See also [1,4] for more lower-bound results related to orthogonal range searching problems.

2.7.2 Other Range Searching Problems

There are other range searching problems called *simplex range searching problem* and the *half-space range searching problem* that have been well studied. A simplex range in $\Re^k$ is a range whose boundary is specified by $k + 1$ hyperplanes. In two dimensions it is a triangle. For this problem, there is a lower bound on the query time for simplex range queries: let m denote the space required, $Q(k, n) = \Omega((n / \log n) m^{1/k}), k > 2$, and $Q(2, n) = \Omega(n / \sqrt{m})$ [37].

The report-mode half-space range searching problem in the plane can be solved optimally in $Q(n) = O(\log n + \mathcal{F})$ time and $S(n) = O(n)$ space, using geometric duality transform [37]. But this method does not generalize to higher dimensions. In [6], Agarwal and Matoušek obtained

a general result for this problem: for $n \leq m \leq n^{\lfloor k/2 \rfloor}$, with $O(m^{1+\epsilon})$ space and preprocessing, $Q(k, n) = O((n/m^{1/\lfloor k/2 \rfloor}) \log n + \mathcal{F})$. As half-space range searching problem is also decomposable, standard dynamization techniques can be applied.

A general method for simplex range searching is to use the notion of *partition tree*. The search space is partitioned in a hierarchical manner using cutting hyperplanes [25] and a search structure is built in a tree structure. Using a **cutting theorem** of hyperplanes, Matoušek [72] showed that for k-dimensions, there is a linear space search structure for the simplex range searching problem with query time $O(n^{1-1/k})$, which is optimal in two dimensions and within $O(\log n)$ factor of being optimal for $k > 2$. For more detailed information regarding geometric range searching, see [72].

The above discussion is restricted to the case in which the database is a collection of points. One may consider also other kinds of objects, such as line segments, rectangles, triangles, etc., whatever applications may take. The inverse of the orthogonal range searching problem is that of *point enclosure searching problem*. Consider a collection of isothetic rectangles. The point enclosure searching is to find all rectangles that contain the given query point q. We can cast these problems as *intersection searching problem*, i.e., given a set S of objects, and a query object q, find a subset $\mathcal{F}$ of S such that for any $f \in \mathcal{F}$, $f \cap q \neq \emptyset$. We have then the rectangle enclosure searching problem, rectangle containment problem, segment intersection searching problem, etc. Janardan and Lopez [59] generalized the intersection searching in the following manner. The database is a collection of groups of objects, and the problem is to find all the groups of objects intersecting a query object. A group is considered to be intersecting the query object if any object in the group intersects the query object. When each group has only one object, this reduces to the ordinary searching problems.

2.8 Intersection

This class of problems arises in, for example, architectural design, computer graphics, etc. In an architectural design, no two objects can share a common region. In computer graphics, the well-known hidden-line or hidden-surface elimination problems [40] are examples of intersection problems. This class encompasses two types of problems, *intersection detection* and *intersection computation*.

2.8.1 Intersection Detection

The intersection detection problem is of the form: Given a set of objects, do any two intersect? For instance, given n line segments in the plane, are there two that intersect? The intersection detection problem has a lower bound of $\Omega(n \log n)$ [90].

In two dimensions, the problem of detecting if two polygons of r and b vertices intersect was easily solved in $O(n \log n)$ time, where $n = r + b$ using the red–blue segment intersection algorithm [29]. However, this problem can be reduced in linear time to the problem of detecting the self-intersection of a polygonal curve. The latter problem is known as the *simplicity test* and can be solved optimally in linear time by Chazelle's linear time triangulation algorithm (cf. Section 1.5). If the two polygons are convex, then $O(\log n)$ suffices in detecting if they intersect [26]. Note that although detecting if two convex polygons intersect can be done in logarithmic time, detecting if the boundary of the two convex polygons intersect requires $\Omega(n)$ time [26]. Mount [77] investigated the intersection detection of two simple polygons and computed a separator of m links in $O(m \log^2 n)$ time if they do not intersect.

In three dimensions, detecting if two convex polyhedra intersect can be solved in linear time [26] by using a hierarchical representation of the convex polyhedron or by formulating it as a linear programming problem in three variables.

2.8.2 Intersection Reporting/Counting

One of the simplest of such intersecting reporting problems is that of reporting pairwise intersection, e.g., intersecting pairs of line segments in the plane. An earlier result due to Bentley and Ottmann [90] used the plane-sweep technique that takes $O((n + \mathcal{F}) \log n)$ time, where $\mathcal{F}$ is the output size. This is based on the observation that the line segments intersected by a vertical sweep-line can be ordered according to the y-coordinates of their intersection with the sweep-line, and the sweep-line status can be maintained in logarithmic time per event point, which is either an endpoint of a line segment or the intersection of two line segments. It is not difficult to see that the lower bound for this problem is $\Omega(n \log n + \mathcal{F})$; thus, the above algorithm is $O(\log n)$ factor from the optimal. This segment intersection reporting problem has been solved optimally by Chazelle and Edelsbrunner [27], who used several important algorithm design and data structuring techniques, as well as some crucial combinatorial analysis. In contrast to this asymptotically time-optimal deterministic algorithm, a simpler randomized algorithm was obtained [37] for this problem which is both time- and space-optimal. That is, it requires only $O(n)$ space (instead of $O(n + \mathcal{F})$ as reported in [27]). Balaban [16] gave a deterministic algorithm that solves this problem optimally both in time and space.

On a separate front, the problem of finding intersecting pairs of segments from two different sets was considered. This is called *bichromatic line segment intersection problem*.

Chazelle et al. [29] used *hereditary segment trees* structure and fractional cascading and solved both segment intersection reporting and counting problems optimally in $O(n \log n)$ time and $O(n)$ space. (The term $\mathcal{F}$ should be included in case of reporting.) If the two sets of line segments form connected subdivisions, then merging or overlaying these two subdivisions can be computed in $O(n + \mathcal{F})$ [50]. Boissonnat and Snoeyink [20] gave yet another optimal algorithm for the bichromatic line segment intersection problem, taking into account the notion of algebraic degree proposed by Liotta et al. [69].

The *rectangle intersection* reporting *problem* arises in the design of VLSI, in which each rectangle is used to model a certain circuitry component. These rectangles are isothetic, i.e., their sides are all parallel to the coordinate axes. This is a well-studied classical problem, and optimal algorithms ($O(n \log n + \mathcal{F})$ time) have been reported. See [4] for more information. The k-dimensional hyperrectangle intersection reporting (respectively, counting) problem can be solved in $O(n^{k-2} \log n + \mathcal{F})$ time and $O(n)$ space (respectively, in time $O(n^{k-1} \log n)$ and space $O(n^{k-2} \log n)$). Gupta et al. [53] gave an $O(n \log n \log\log n + \mathcal{F} \log\log n)$-time and linear-space algorithm for the *rectangle enclosure reporting problem* that calls for finding all the enclosing pairs of rectangles.

2.8.3 Intersection Computation

Computing the actual intersection is a basic problem, whose efficient solutions often lead to better algorithms for many other problems.

Consider the problem of computing the common intersection of half-planes by divide-and-conquer. Efficient computation of the intersection of the two convex polygons is required during the merge step. The intersection of the two convex polygons can be solved very efficiently by plane-sweep in linear time, taking advantage of the fact that the edges of the input polygons are ordered. Observe that in each vertical strip defined by the two consecutive sweep-lines, we only need to compute the intersection of two trapezoids, one derived from each polygon [90].

The problem of the intersecting two convex polyhedra was first studied by Muller and Preparata [90], who gave an $O(n \log n)$ algorithm by reducing the problem to the problems of intersection detection and convex hull computation. Following this result one can easily derive an $O(n \log^2 n)$ algorithm for computing the common intersection of n half-spaces in three dimensions by divide-and-conquer. However, using geometric duality and the concept of separating plane, Preparata and Muller [90] obtained an $O(n \log n)$ algorithm for computing the common intersection of n half-spaces, which is asymptotically optimal. There appears to be a difference in the approach to

solving the common intersection problem of half-spaces in two and three dimensions. In the latter, we resorted to geometric duality instead of divide-and-conquer. This inconsistency was later resolved. Chazelle [24] combined the hierarchical representation of convex polyhedra, geometric duality, and other ingenious techniques to obtain a linear time algorithm for computing the intersection of two convex polyhedra. From this result, several problems can be solved optimally: (1) the common intersection of half-spaces in three dimensions can now be solved by divide-and-conquer optimally, (2) the merging of two Voronoi diagrams in the plane can be done in linear time by observing the relationship between the Voronoi diagram in two dimensions and the convex hull in three dimensions (cf. Section 1.2), and (3) the medial axis of a simple polygon or the Voronoi diagram of vertices of a convex polygon [5] can be solved in linear time.

2.9 Research Issues and Summary

We have covered in this chapter a number of topics in computational geometry, including proximity, optimization, planar point location, geometric matching, path planning, searching, and intersection. These topics discussed here and in the previous chapter are not meant to be exhaustive. New topics arise as the field continues to flourish.

In Section 2.3, we discussed the problems of the smallest enclosing circle and the largest empty circle. These are the two extremes: either the circle is empty or it contains all the points. The problem of finding the smallest (respectively, largest) circle containing at least (respectively, at most) k points for some integer $0 \leq k \leq n$ is also a problem of interest. Moreover, the shape of the object is not limited to circles. A number of open problems remain. Given two points s and t in a simple polygon, is it NP-complete to decide whether there exists a path with at most k links and of length at most L? What is the complexity of the shortest k-pair noncrossing path problem discussed in Section 2.4? Does there exist an algorithm for finding the Eucliedan shortest path in a general polygonal demain of h obstacles and n vertices in $O(n + h \log h)$ time and $O(n)$ space? How fast can one solve the geodesic minimum matching problem for $2m$ points in the presence of polygonal obstacles? Can one solve the largest empty restricted rectangle problem for a rectilinear polygon in $O(n \log n)$ time? The best known algorithm to date runs in $O(n \log^2 n)$ time [36]. Is $\Omega(n \log n)$ a lower bound of the minimum-width annulus problem? Can the technique used in [51] be applied to the polygonal case to yield an $O(n \log n)$-time algorithm for the minimum-width annulus problem? In [76] Mitchell listed a few open problems worth studying. Although the minimum spanning tree problem for general graph with m edges and n vertices can be solved optimally in $O(\mathcal{T}^*(m, n))$ time, where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons and $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where $\alpha(m, n)$ is the inverse of Ackermann's function [88], an optimal algorithm for the Euclidean minimum spanning tree is still open. The problem of finding a spanning tree of bounded cost and bounded diameter is known to be NP-complete [96], but there is no known approximation algorithm to date.

Researchers in computational geometry have begun to address the issues concerning the actual running times of the algorithms and their robustness when the computations in their implementations are not exact [106]. It is understood that the real-RAM computation model with an implicit infinite-precision arithmetic is unrealistic in practice. In addition to the robustness issue concerning the accuracy of the output of an algorithm, one needs to find a new cost measure to evaluate the efficiency of an algorithm. In the infinite-precision model, the asymptotic time complexity was accepted as an adequate cost measure. However, when the input data have a finite-precision representation and computation time varies with the precision required, an alternative cost measure is warranted. The notion of the **degree** of a geometric algorithm could be an important cost measure for comparing the efficiency of the algorithms when they are actually implemented [69]. For example, Chen et al. [33] showed that the Voronoi diagram of a set of arbitrarily oriented segments can

be constructed by a plane-sweep algorithm with degree 14 for certain regular k-gon metrics. This notion of the degree of robustness could play a similar role as the asymptotic time complexity has in the past for the real-RAM computation model.

On the applied side, there are efforts put into development of geometric software. A project known as the Computational Geometry Algorithms Library (CGAL) project (http://www.cgal.org/) [49] is an ongoing collaborative effort of researchers in Europe to organize a system library containing primitive geometric abstract data types useful for geometric algorithm developers. This is concurrent with the Library of Efficient Data Types and Algorithms (LEDA) project [73] which was initiated at the Max-Planck-Institut für Informatik, Saarbrücken, Germany, and now maintained by Algorithmic Solutions Software GmbH (http://www.algorithmic-solutions.com/leda/about/index.htm.) LEDA is a C++ class library for efficient data types and algorithms, and provides algorithmic in-depth knowledge in geometric computing, combinatorial optimization, etc. In Asia, a web-collaboratory project was initiated at the Institute of Information Science and the Research Center for Information Technology Innovation, Academia Sinica, from which a geometric algorithm visualization and debugging system, GeoBuilder, for 2D and 3D geometric computing has been developed (http://webcollab.iis.sinica.edu.tw/Components/GeoBuilder/). This system facilitates geometric algorithmic researchers in not only testing their ideas and demonstrating their findings, but also teaching algorithm design in the classroom. GeoBuilder is embedded into a knowledge portal [64], called OpenCPS (Open Computational Problem Solving) (http://www.opencps.org/), as a practice platform for a course on geometric computing and algorithm visualization. The GeoBuilder system possesses three important features: First, it is a platform-independent software system based on Java's promise of portability, and can be invoked by Sun's Java Web Start technology in any browser-enabled environment. Second, it has the collaboration capability for multiple users to concurrently develop programs, manipulate geometric objects and control the camera. Finally, its 3D geometric drawing bean provides an optional function that can automatically position the camera to track 3D objects during algorithm visualization [102]. GeoBuilder develops its rich client platform based on Eclipse RCP and has already built in certain functionalities such as remote addition, deletion, and saving of files as well as remote compiling, and execution of LEDA C/C++ programs, etc. based on a multipage editor. Other projects related to the efforts of building geometric software or problem-solving environment, include GeoLab, developed at the Institute of Computing at UNICAMP (State University of Campinas) as a programming environment for implementation, testing, and animation of geometric algorithms (http://www.dcc.unicamp.br/∼rezende/GeoLab.htm), and *XYZ* GeoBench, developed at Zürich, as a programming environment on Macintosh computers for geometric algorithms, providing an interactive user interface similar to a drawing program which can be used to create and manipulate geometric objects such as points, line segments, polygons, etc. (http://www.schorn.ch/geobench/XYZGeoBench.html).

2.10 Further Information

Additional references about various variations of closest pair problems can be found in [17,21,60,95]. For additional results concerning the Voronoi diagrams in higher dimensions and the duality transformation see [15]. For more information about Voronoi diagrams for sites other than points, in various distance functions or norms, see [10,19,81,85,87,93]. A recent textbook by de Berg et al. [37] contains a very nice treatment of computational geometry in general. The book by Narasimhan and Smid [79] covers topics pertaining to geometric spanner networks. More information can be found in [52,63,93,105]. The reader who is interested in parallel computational geometry is referred to [13]. For current research activities and results, the reader may consult the *Proceedings of the Annual ACM Symposium on Computational Geometry, Proceedings of the Annual Canadian Conference*

on Computational Geometry and the following three journals: *Discrete & Computational Geometry, International Journal of Computational Geometry & Applications,* and *Computational Geometry: Theory and Applications.* The following site http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html contains close to 14,000 entries of bibliography in this field.

Those who are interested in the implementations or would like to have more information about available software can consult http://www.cgal.org/.

The following WWW page on *Geometry in Action* maintained by David Eppstein at http://www.ics.uci.edu/~eppstein/geom.html and the Computational Geometry Page by J. Erickson at http://compgeom.cs.uiuc.edu/~jeffe/compgeom give a comprehensive description of research activities of computational geometry.

Defining Terms

ANSI: American National Standards Institute.

Bisector: A bisector of two elements e_i and e_j is defined to be the locus of points equidistant from both e_i and e_j. That is, $\{p|d(p, e_i) = d(p, e_j)\}$. For instance, if e_i and e_j are two points in the Euclidean plane, the bisector of e_i and e_j is the perpendicular bisector to the line segment $\overline{e_i, e_j}$.

Cutting theorem: This theorem [25] states that for any set $\mathcal{H}$ of n hyperplanes in $\Re^k$, and any parameter r, $1 \leq r \leq n$, there always exists a $(1/r)$-cutting of size $O(r^k)$. In two dimensions, a $(1/r)$-cutting of size s is a partition of the plane into s disjoint triangles, some of which are unbounded, such that no triangle in the partition intersects more than n/r lines in $\mathcal{H}$. In $\Re^k$, triangles are replaced by simplices. Such a cutting can be computed in $O(nr^{k-1})$ time.

Decomposable searching problems: A searching problem with query Q is decomposable if there exists an efficiently computable associative, and commutative binary operator @ satisfying the condition: $Q(x, A \cup B) = @(Q(x, A), Q(x, B))$. In other words, one can decompose the searched domain into subsets, find answers to the query from these subsets, and combine these answers to form the solution to the original problem.

Degree of an algorithm or problem: Assume that each input variable is of arithmetic degree 1 and that the arithmetic degree of a polynomial is the common arithmetic degree of its monomials, whose degree is defined to be the sum of the arithmetic degrees of its variables. An algorithm has degree d, if its test computation involves evaluation of multivariate polynomials of arithmetic degree d. A problem has degree d, if any algorithm that solves it has degree at least d [69].

Diameter of a graph: The distance between two vertices u and v in a graph is the sum of weights of the edges of the shortest path between them. (For an unweighted graph, it is the number of edges of the shortest path.) The diameter of a graph is the maximum among all the distances between all possible pairs of vertices.

Dynamic versus static: This refers to cases when the underlying problem domain can be subject to updates, i.e., insertions and deletions. If no updates are permitted, the problem or data structure is said to be static; otherwise, it is said to be dynamic.

Dynamization transformation: A data structuring technique can transform a static data structure into a dynamic one. In so doing, the performance of the dynamic structure will exhibit certain space-time trade-offs. See, e.g., [63,90] for more references.

Geometric duality: A transform between a point and a hyperplane in $\Re^k$, that preserves incidence and order relation. For a point $p = (\mu_1, \mu_2, \ldots, \mu_k)$, its dual $\mathcal{D}(p)$ is a hyperplane denoted by $x_k = \Sigma_{j=1}^{k-1} \mu_j x_j - \mu_k$; for a hyperplane $\mathcal{H} : x_k = \Sigma_{j=1}^{k-1} \mu_j x_j + \mu_k$, its dual $\mathcal{D}(\mathcal{H})$ is a point denoted by $(\mu_1, \mu_2, \ldots, -\mu_k)$. There are other duality transformations. What is described in the text is called the paraboloid transform. See [37,44,90] for more information.

Height-balanced binary search tree: A data structure used to support membership, insert/delete operations each in time logarithmic in the size of the tree. A typical example is the *AVL* tree or red-black tree.

Orthogonally convex rectilinear polygon: A rectilinear polygon P is orthogonally convex if every horizontal or vertical segment connecting two points in P lies totally within P.

Priority queue: A data structure used to support insert and delete operations in time logarithmic in the size of the queue. The elements in the queue are arranged so that the element of the minimum priority is always at one end of the queue, readily available for delete operation. Deletions only take place at that end of the queue. Each delete operation can be done in constant time. However, since restoring the above property after each deletion takes logarithmic time, we often say that each delete operation takes logarithmic time. A heap is a well-known priority queue.

References

1. Agarwal, P.K., Range Searching, in *Handbook of Discrete and Computational Geometry*, Goodman, J.E. and O'Rourke, J. (Eds.), CRC Press LLC, Boca Raton, FL, 2004.
2. Agarwal, P.K., de Berg, M., Matoušek, J., and Schwarzkopf, O., Constructing levels in arrangements and higher order Voronoi diagrams, *SIAM J. Comput.,* 27, 654–667, 1998.
3. Agarwal, P.K., Edelsbrunner, H., Schwarzkopf, O., and Welzl, E., Euclidean minimum spanning trees and bichromatic closest pairs, *Discrete Comput. Geom.,* 6(5), 407–422, 1991.
4. Agarwal, P.K. and Erickson, J., Geometric range searching and its relatives, in *Advances in Discrete and Computational Geometry,* Chazelle, B., Goodman, J.E., and Pollack, R. (Eds.), AMS Press, Providence, RI, pp. 1–56, 1999.
5. Aggarwal, A., Guibas, L.J., Saxe, J., and Shor, P.W., A linear-time algorithm for computing the Voronoi diagram of a convex polygon, *Discrete Comput. Geom.,* 4(6), 591–604, 1989.
6. Agarwal, P.K. and Matoušek, J., Dynamic half-space range reporting and its applications, *Algorithmica,* 13(4) 325–345, Apr. 1995.
7. Agarwal, P. and Varadarajan, K., A near-linear constant-factor approximation for Euclidean bipartite matching? *Proceedings of the 20th Symposium on Computational Geometry,* ACM, New York, pp. 247–252, 2004.
8. Aichholzer, O., Aurenhammer, F., Chen, D.Z., Lee, D.T., Mukhopadhyay, A., and Papadopoulou, E., Skew Voronoi diagrams, *Int. J. Comput. Geom. Appl.*, 9(3), 235–247, June 1999.
9. Alt, H. and Guibas, L.J., Discrete geometric shapes: Matching, interpolation, and approximation, in *Handbook of Computational Geometry*, Sack, J.R. and Urrutia, J. (Eds.), Elsevier Science Publishers, B.V. North-Holland, Amsterdam, the Netherlands, pp. 121–153, 2000.
10. Alt, H. and Schwarzkopf, O., The Voronoi diagram of curved objects, *Discrete Comput. Geom.,* 34(3), 439–453, Sept. 2005.
11. Alt, H. and Yap, C.K., Algorithmic aspect of motion planning: A tutorial, Part 1 & 2, *Algorithms Rev.,* 1, 43–77, 1990.
12. Amato, N.M. and Ramos, E.A., On computing Voronoi diagrams by divide-prune-and-conquer, *Proceedings of the 12th ACM Symposium on Computational Geometry,* ACM, New York, pp. 166–175, 1996.
13. Atallah, M.J., Parallel techniques for computational geometry, *Proc. IEEE,* 80(9), 1435–1448, Sept. 1992.
14. Atallah, M.J. and Chen, D.Z., On connecting red and blue rectilinear polygonal obstacles with nonintersecting monotone rectilinear paths, *Int. J. Comput. Geom. Appl.*, 11(4), 373–400, Aug. 2001.

15. Aurenhammer, F. and Klein, R., Voronoi diagrams, in *Handbook of Computational Geometry,* Sack, J.R. and Urrutia, J. (Eds.), Elsevier Science Publishers, B.V. North-Holland, Amsterdam, the Netherlands, pp. 201–290, 2000.
16. Balaban, I.J., An optimal algorithm for finding segments intersections, *Proceedings of the 11th Symposium on Computational Geometry*, Vancouver, BC, Canada, pp. 211–219, June 1995.
17. Bespamyatnikh, S.N., An optimal algorithm for closest pair maintenance, *Discrete Comput. Geom.,* 19(2), 175–195, 1998.
18. Blum, A., Ravi, R., and Vempala, S., A Constant-factor approximation algorithm for the *k*-MST problem, *J. Comput. Syst. Sci.,* 58(1), 101–108, Feb. 1999.
19. Boissonnat, J.-D., Sharir, M., Tagansky, B., and Yvinec, M., Voronoi diagrams in higher dimensions under certain polyhedra distance functions, *Discrete Comput. Geom.,* 19(4), 473–484, 1998.
20. Boissonnat, J.-D. and Snoeyink, J., Efficient algorithms for line and curve segment intersection using restricted predicates, *Comp. Geom. Theor. Appl.,* 19(1), 35–52, May 2000.
21. Callahan, P. and Kosaraju, S.R., A decomposition of multidimensional point sets with applications to *k*-nearest-neighbors and *n*-body potential fields, *J. Assoc. Comput. Mach.,* 42(1), 67–90, Jan. 1995.
22. Chan, T.M., More planar 2-center algorithms, *Comp. Geom. Theor. Appl.,* 13(3), 189–198, Sept. 1999.
23. Chazelle, B., Lower bounds for orthogonal range searching. I. The reporting case, *J. Assoc. Comput. Mach.,* 37(2), 200–212, Apr. 1990.
24. Chazelle, B., An optimal algorithm for intersecting three-dimensional convex polyhedra, *SIAM J. Comput.,* 21(4), 671–696, 1992.
25. Chazelle, B., Cutting hyperplanes for divide-and-conquer, *Discrete Comput. Geom.,* 9(2), 145–158, 1993.
26. Chazelle, B. and Dobkin, D.P., Intersection of convex objects in two and three dimensions, *J. Assoc. Comput. Mach.,* 34(1), 1–27, 1987.
27. Chazelle, B. and Edelsbrunner, H., An optimal algorithm for intersecting line segments in the plane, *J. Assoc. Comput. Mach.,* 39(1), 1–54, 1992.
28. Chazelle, B., Edelsbrunner, H., Guibas, L.J., and Sharir, M., Diameter, width, closest line pair, and parametric searching, *Discrete Comput. Geom.,* 8, 183–196, 1993.
29. Chazelle, B., Edelsbrunner, H., Guibas, L.J., and Sharir, M., Algorithms for bichromatic line-segment problems and polyhedral terrains, *Algorithmica,* 11(2), 116–132, Feb. 1994.
30. Chazelle, B. and Friedman, J., Point location among hyperplanes and unidirectional ray-shooting, *Comp. Geom. Theor. Appl.,* 4, 53–62, 1994.
31. Chen, D., Klenk, K.S., and Tu, H.-Y.T., Shortest path queries among weighted obstacles in the rectilinear plane, *SIAM J. Comput.,* 29(4), 1223–1246, Feb. 2000.
32. Chen, J. and Han, Y., Shortest paths on a polyhedron, Part I: Computing shortest paths, *Int. J. Comput. Geom. Appl.,* 6(2), 127–144, 1996.
33. Chen, Z. Papadopoulou, E., and Xu J., Robustness of k-gon Voronoi diagram construction, *Inf. Process. Lett.,* 97(4), 138–145, Feb. 2006.
34. Chiang, Y.-J. and Tamassia, R., Dynamic algorithms in computational geometry, *Proc. IEEE,* 80(9), 1412–1434, Sept. 1992.
35. Choi, J., Sellen, J., and Yap, C.K., Approximate Euclidean shortest path in 3-space, *Int. J. Comput. Geom. Appl.,* 7(4), 271–295, 1997.
36. Daniels, K., Milenkovic, V., and Roth, D., Finding the largest area axis-parallel rectangle in a polygon, *Comp. Geom. Theor. Appl.,* 7, 125–148, Jan. 1997.
37. de Berg, M., Cheong, O., van Kreveld, M. and Overmars, M., *Computational Geometry: Algorithms and Applications,* Springer-Verlag, Berlin, Germany, p. 386, 2008.
38. Dehne, F. and Klein, R., The Big Sweep: On the power of the wavefront approach to Voronoi diagrams, *Algorithmica,* 17(1), 19–32, Jan. 1997.

39. Diaz-Banez, J.M., Hurtado, F., Meijer, H., Rappaport, D., and Sellares, J.A., The largest empty annulus problem, *Int. J. Comput. Geom. Appl.*, 13(4), 317–325, Aug. 2003.
40. Dorward, S.E., A survey of object-space hidden surface removal, *Int. J. Comput. Geom. Appl.,* 4(3), 325–362, Sept. 1994.
41. Du, D.Z. and Hwang, F.K., Eds., *Computing in Euclidean Geometry,* World Scientific, Singapore, 1992.
42. Duncan, C.A., Goodrich, M.T., and Ramos, E.A., Efficient approximation and optimization algorithms for computational metrology, *Proceedings of the 8th annual ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, pp. 121–130, 1997.
43. Dwyer, R.A. and Eddy, W.F., Maximal empty ellipsoids, *Int. J. Comput. Geom. Appl.,* 6(2), 169–186, 1996.
44. Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, Berlin, Germany, 1987.
45. Edelsbrunner, H., The union of balls and its dual shape, *Discrete Comput. Geom.,* 13(1), 415–440, 1995.
46. Edelsbrunner, H. and Sharir, M., A hyperplane incidence problem with applications to counting distances, in *Applied Geometry and Discrete Mathematics. The Victor Klee Festschrift,* Gritzmann, P. and Sturmfels, B. (Eds.), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS Press, Providence, RI, pp. 253–263, 1991.
47. Efrat, A. and Sharir, M., A near-linear algorithm for the planar segment center problem, *Discrete Comput. Geom.*, 16(3), 239–257, 1996.
48. Eppstein, D., Fast construction of planar two-centers, *Proceedings of the 8th annual ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, pp. 131–138, 1997.
49. Fabri A., Giezeman G., Kettner L., Schirra, S., and Schonherr S., On the design of CGAL a computational geometry algorithms library, *Software Pract. Ex.,* 30(11), 1167–1202, Aug. 2000. http://www.cgal.org/
50. Finkle, U. and Hinrichs, K., Overlaying simply connected planar subdivision in linear time, *Proceedings of the 11th annual ACM Symposium on Computational Geometry,* ACM, New York, pp. 119–126, 1995.
51. Garcia-Lopez, J., Ramos, P.A., and Snoeyink, J., Fitting a set of points by a circle, *Discrete Comput. Geom.,* 20, 389–402, 1998.
52. Goodman, J.E. and O'Rourke, J., Eds., *The Handbook of Discrete and Computational Geometry,* CRC Press LLC, Boca Raton, FL, 2004.
53. Gupta, P., Janardan, R., Smid, M., and Dasgupta, B., The rectangle enclosure and point-dominance problems revisited, *Int. J. Comput. Geom. Appl.*, 7(5), 437–455, May 1997.
54. Hershberger, J. and Suri, S., Efficient computation of Euclidean shortest paths in the plane, *Proceedings of the 34th Symposium on Foundations of Computer Science*, Palo Alto, CA, 508–517, Nov. 1993.
55. Hershberger, J. and Suri, S., Finding a shortest diagonal of a simple polygon in linear time, *Comput. Geom. Theor. Appl.,* 7(3), 149–160, Feb. 1997.
56. Ho, J.M., Chang, C.H., Lee, D.T., and Wong, C.K., Minimum diameter spanning tree and related problems, *SIAM J. Comput.,* 20(5), 987–997, Oct. 1991.
57. Hwang, F.K., Foreword, *Algorithmica,* 7(2/3), 119–120, 1992.
58. Indyk, P. A near linear time constant factor approximation for Euclidean bichromatic matching (Cost), *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, pp. 39–42, 2007.
59. Janardan, R. and Lopez, M., Generalized intersection searching problems, *Int. J. Comput. Geom. Appl.*, 3(1), 39–69, Mar. 1993.
60. Kapoor, S. and Smid, M., New techniques for exact and approximate dynamic closest-point problems, *SIAM J. Comput.*, 25(4), 775–796, Aug. 1996.

61. Klawe, M.M. and Kleitman, D.J., An almost linear time algorithm for generalized matrix searching, *SIAM J. Discrete Math.,* 3(1), 81–97, Feb. 1990.
62. Lee, D.T., Two dimensional voronoi diagrams in the L_p-metric, *J. Assoc. Comput. Mach.,* 27, 604–618, 1980.
63. Lee, D.T., Computational geometry, in *Computer Science and Engineering Handbook,* Tucker, A. (Ed.), CRC Press, Boca Raton, FL, pp. 111–140, 1997.
64. Lee, D.T., Lee, G.C., and Huang, Y.W., Knowledge management for computational problem solving, *J. Universal Comput. Sci.,* 9(6), 563–570, 2003.
65. Lee, D.T. and Shen, C.F., The Steiner minimal tree problem in the λ-geometry plane, *Proceedings of the 7th International Symposium on Algorithms and Computation,* Asano, T., Igarashi, Y., Nagamochi, H., Miyano, S., and Suri, S. (Eds.), LNCS, Vol. 1173, Springer-Verlag, Berlin, Germany, pp. 247–255, Dec. 1996.
66. Lee, D.T. and Wu, V.B., Multiplicative weighted farthest neighbor Voronoi diagrams in the plane, *Proceedings of International Workshop on Discrete Mathematics and Algorithms,* Hong Kong, pp. 154–168, Dec. 1993.
67. Lee, D.T. and Wu, Y.F., Geometric complexity of some location problems, *Algorithmica,* 1(1), 193–211, 1986.
68. Lee, D.T., Yang, C.D., and Wong, C.K., Rectilinear paths among rectilinear obstacles, *Perspect. Discrete Appl. Math.,* Bogart, K. (Ed.), 70, 185–215, 1996.
69. Liotta, G., Preparata, F.P., and Tamassia, R., Robust proximity queries: An illustration of degree-driven algorithm design, *SIAM J. Comput.,* 28(3), 864–889, Feb. 1998.
70. Lu, B. and Ruan, L., Polynomial time approximation scheme for the rectilinear Steiner arborescence problem, *J. Comb. Optim.,* 4(3), 357–363, 2000.
71. Marcotte, O. and Suri, S., Fast matching algorithms for points on a polygon, *SIAM J. Comput.,* 20(3), 405–422, June 1991.
72. Matoušek, J., Geometric range searching, *ACM Comput. Surv.,* 26(4), 421–461, 1994.
73. Mehlhorn, K. and Näher, S., *LEDA, A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, Cambridge, U.K., 1999.
74. Mitchell, J.S.B., Shortest paths among obstacles in the plane, *Int. J. Comput. Geom. Appl.,* 6(3), 309–332, Sept. 1996.
75. Mitchell, J.S.B., Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and Related Problems, *SIAM J. Comput.,* 28(4), 1298–1309, Aug. 1999.
76. Mitchell, J.S.B., Geometric shortest paths and network optimization, in *Handbook of Computational Geometry,* Sack, J.R. and Urrutia, J. (Eds.), Elsevier Science Publishers, B.V. North-Holland, Amsterdam, the Netherlands, 633–701, 2000.
77. Mount, D.M., Intersection detection and separators for simple polygons, *Proceedings of the 8th Annual ACM Symposium on Computational Geometry,* ACM, New York, pp. 303–311, 1992.
78. Mukhopadhyay, A. and Rao, S.V., Efficient algorithm for computing a largest empty arbitrarily oriented rectangle, *Int. J. Comput. Geom. Appl.,* 13(3), 257–272, June 2003.
79. Narasimhan, G. and Smid, M. *Geometric Spanner Networks*, Cambridge University Press, Cambridge, U.K., 2007.
80. Nielsen, F., Boissonnat, J.-D., and Nock, R., On bregman Voronoi diagrams, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms,* SIAM, Philadelphia, PA, pp. 746–755, 2007.
81. Okabe, A., Boots, B., Sugihara, K., and Chiu, S.N., *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams,* (2nd ed.), John Wiley & Sons, Chichester, U.K., 2000.
82. Orlowski, M., A new algorithm for the largest empty rectangle, *Algorithmica,* 5(1), 65–73, 1990.
83. Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity,* Dover Publications Inc., Mineola, NY, 1998.

84. Papadopoulou, E., *k*-Pairs non-crossing shortest paths in a simple polygon, *Int. J. Comput. Geom. Appl.*, 9(6), 533–552, Dec. 1999.
85. Papadopoulou, E., The hausdorff Voronoi diagram of point clusters in the plane, *Algorithmica,* 40(2), 63–82, July 2004.
86. Papadopoulou, E. and Lee, D.T., A new approach for the geodesic voronoi diagram of points in a simple polygon and other restricted polygonal domains, *Algorithmica,* 20(4), 319–352, Apr. 1998.
87. Papadopoulou, E. and Lee, D.T., The Hausdorff Voronoi diagram of polygonal objects: A divide and conquer approach, *Int. J. Comput. Geom. Appl.*, 14(6), 421–452, Dec. 2004.
88. Pettie, S. and Ramachandran, V., An optimal minimum spanning tree algorithm, *J. Assoc. Comput. Mach.,* 49(1), 16–34, Jan. 2002.
89. Pocchiola, M. and Vegter, G., Topologically sweeping visibility complexes via pseudotriangulations, *Discrete Comput. Geom.,* 16, 419–453, Dec. 1996.
90. Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction,* Springer-Verlag, Berlin, Germany, 1988.
91. Ranmath, S., New Approximations for the rectilinear Steiner arborescence problem, *IEEE Trans. Comput. Aided Design.*, 22(7), 859–869, July 2003.
92. Ravi, R., Sundaram, R., Marathe, M.V., Rosenkrantz, D.J., and Ravi, S.S., Spanning trees short or small, *SIAM J. Discrete Math.,* 9(2), 178–200, May 1996.
93. Sack, J. and Urrutia, J., Eds., *Handbook of Computational Geometry,* Elsevier Science Publishers, B.V. North-Holland, Amsterdam, the Netherlands, 2000.
94. Sharir, M., Algorithmic motion planning, in *Handbook of Discrete and Computational Geometry,* Goodman, J.E. and O'Rourke, J. (Eds.) CRC Press LLC, Boca Raton, FL, 2004.
95. Schwartz, C., Smid, M., and Snoeyink, J., An optimal algorithm for the on-line closest-pair problem, *Algorithmica,* 12(1), 18–29, July 1994.
96. Seo, D.Y., Lee, D.T., and Lin, T.C., Geometric minimum diameter minimum cost spanning tree problem, *Algorithmica,* submitted for publication, 2009.
97. Shermer, T. and Yap, C., Probing near centers and estimating relative roundness, *Proceedings of the ASME Workshop on Tolerancing and Metrology*, University of North Carolina, Charlotte, NC, 1995.
98. Shi, W. and Chen S., The rectilinear Steiner arborescence problem is NP-complete, *SIAM J. Comput.*, 35(3), 729–740, 2005.
99. Smid, M. and Janardan, R., On the width and roundness of a set of points in the plane, *Comput. Geom. Theor. Appl.,* 9(1), 97–108, Feb. 1999.
100. Swanson, K., Lee, D.T., and Wu, V.L., An Optimal algorithm for roundness determination on convex polygons, *Comput. Geom. Theor. Appl.,* 5(4), 225–235, Nov. 1995.
101. Vaidya, P.M., Geometry helps in matching, *SIAM J. Comput.*, 18(6), 1201–1225, Dec. 1989.
102. Wei, J.D., Tsai, M.H., Lee, G.C., Huang, J.H., and Lee, D.T., GeoBuilder: A geometric algorithm visualization and debugging system for 2D and 3D geometric computing, *IEEE Trans. Vis. Comput. Graph*, 15(2), 234–248, Mar. 2009.
103. Welzl, E., Smallest enclosing disks, balls and ellipsoids, in *New Results and New Trends in Computer Science,* Maurer, H.A. (Ed.), LNCS, Vol. 555, Springer-Verlag, Berlin, Germany, pp. 359–370, 1991.
104. Yang, C.D., Lee, D.T., and Wong, C.K., Rectilinear path problems among rectilinear obstacles revisited, *SIAM J. Comput.,* 24(3), 457–472, June 1995.
105. Yao, F.F., Computational geometry, in *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity*, van Leeuwen, J. (Ed.), MIT Press, Cambridge, MA, pp. 343–389, 1994.
106. Yap, C., Exact computational geometry and tolerancing, in *Snapshots of Computational and Discrete Geometry,* Avis, D. and Bose, J. (Eds.), School of Computer Science, McGill University, Montreal, QC, Canada, 1995.