Computational Geometry I

D. T. Lee
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
e-mail: dtlee@ece.nwu.edu

1 Introduction

Computational geometry, since its inception[41] 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, statistics, etc. 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*[25, 49], and the actual running times of their implementions.

In this and the following chapter 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.

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 output.

2.1 Convex Hulls in 2- and 3-Dimensions

For an arbitrary set of n points in 2- and 3-dimensions, we can compute the 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 of points in 2-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 2-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 the lines $\mathcal{L}_i$ containing each edge $\overline{v_{j_{i-1}}, v_{j_i}}$. See Fig. 1(a) for an illustration of the upper hull. The lower hull is similarly defined.

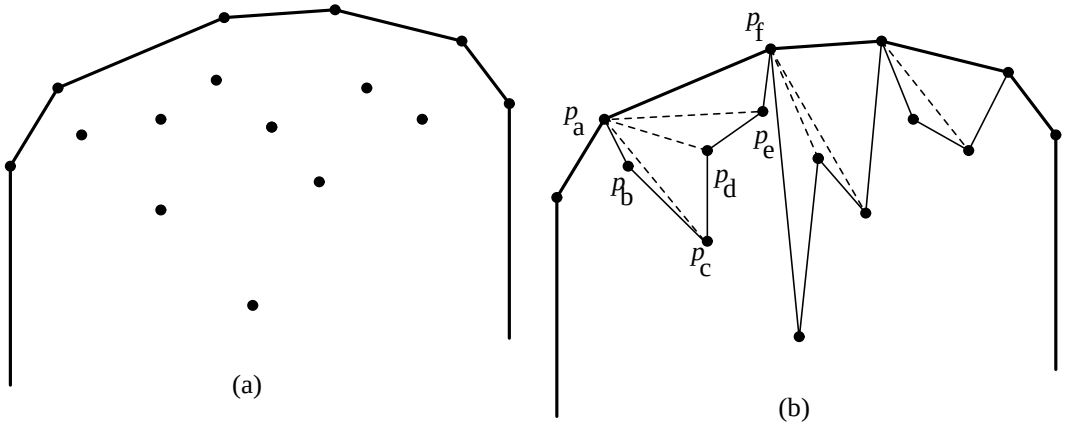

Figure 1: The upper hull of a set of points.

The *Graham Scan* computes the convex hull by (i) sorting the input set of points in ascending order of their x-coordinates; in case of ties, in ascending order of their y-coordinates, (ii) connecting these points into a polygonal chain P stored as a doubly-linked list L, and (iii) performing a linear scan to compute the upper hull of the polygon[41].

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. 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) which 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. Fig. 1(b) 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 (Fig. 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 scan* one can obtain in *linear* time the two endpoints of the bridge, ($\overline{p, q}$ shown in Fig. 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[34]. 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 [34]. 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.

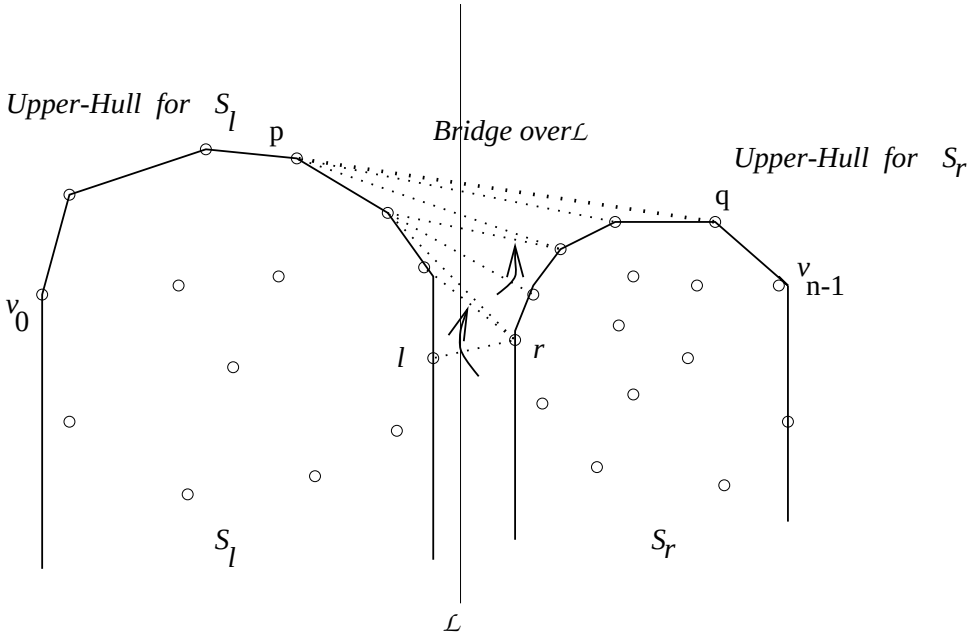

Figure 2: The bridge $\overline{p,q}$ over the vertical line $\mathcal{L}$.

The divide-and-conquer scheme can be easily generalized to 3-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 Hong[41] that the common supporting faces are computed from connecting two *cyclic* sequences of edges, one on each polyhedron (Fig. 3). See [2] for a characterization of the two cycles of *seam* edges. The computation of these supporting 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[23] obtained an $O(n \log^2 \mathcal{H})$ algorithm.

The gift-wrapping approach for computing the convex hull in 3-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 2-dimensions is due to Chan[10]. A similar algorithm for 3-dimensions can be obtained. 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

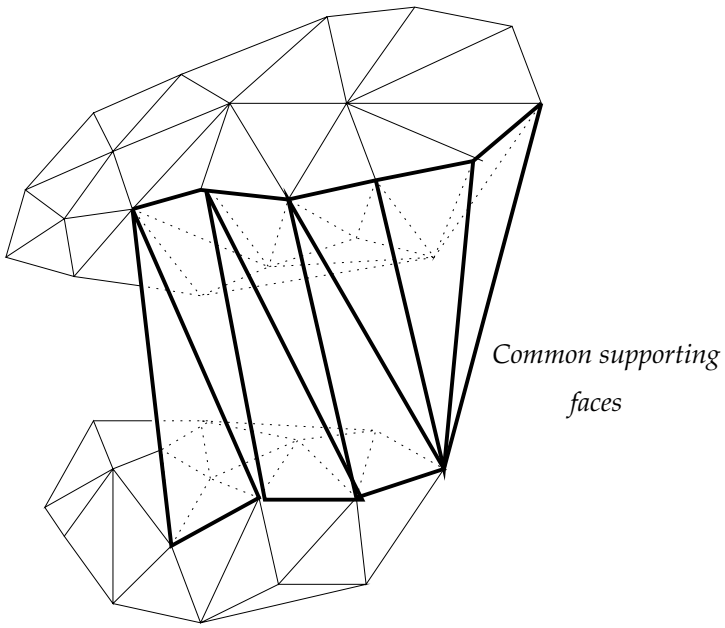

Figure 3: Common supporting faces of two disjoint convex polyhedra.

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 2-dimensions, and Preparata-Hong algorithm for S in 3-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 2-dimensions we can use an array that stores the vertices on the convex hull in say, clockwise, order. In 3-dimensions we use Dobkin-Kirkpatrick hierarchical representation of the faces of the convex hull[20]. 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 2-dimensional 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 $\mathcal{H}_1 + \ldots \mathcal{H}_{k-1} < \mathcal{H} \leq \mathcal{H}_1 + \ldots \mathcal{H}_k$ to guess it. The total running time is

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

2.2 Convex Hulls in k-dimensions, $k > 3$

For convex hulls of higher dimensions, a recent result by Chazelle[13] 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 $\mathcal{H} = O(n^{\lfloor k/2 \rfloor})$[22]. One can also use *beneath-beyond* method[41] of adding points one at a time in ascending order along one of the coordinate axis[1]. We compute the convex hull $CH(S_{i-1})$ for points $S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$. For each added point p_i we update $CH(S_{i-1})$ to get $CH(S_i)$ for $i = 2, 3, \ldots, n$ by deleting those t-faces, $t = 0, 1, \ldots, k-1$, that are internal to $CH(S_{i-1} \cup \{p_i\})$. It is shown by Seidel[22] that $O(n^2 + \mathcal{H} \log h)$ time is sufficient, where h is the number of extreme points. Most recently Chan[10] obtained an algorithm based on gift-wrapping method 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 $\mathcal{H} = o(n^{1-\epsilon})$ for some $0 < \epsilon < 1$.

We conclude this section with the following theorem[10].

Theorem 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, $i = 0, 1, \ldots, k-1$.*

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 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[11] that runs in optimal $O(n \log n)$ time.

As described in Section 2.1, each convex layer of $\mathcal{C}(S)$ can be decomposed into two convex polygonal chains, called upper and lower hulls (Figure 5).

Let l and r denote the points with the minimum and maximum x-coordinate respectively in a convex layer. The upper (resp. lower) hull of this layer runs clockwise (resp. counterclockwise) from l to r. The upper and lower hulls are the same if the convex layer

[1] 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 internal, just discard it.

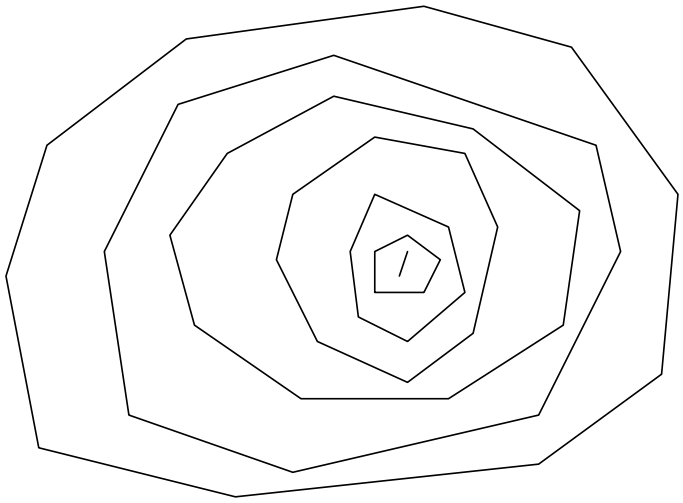

Figure 4: Convex layers of a point set.

has one or two points. Assume that the set S of points $p_0, p_1, \ldots, p_{n-1}$ are ordered in non-descending 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*[11]. (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, v_r are the left and right children of node v. Figures 6(a) and (b) illustrate 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

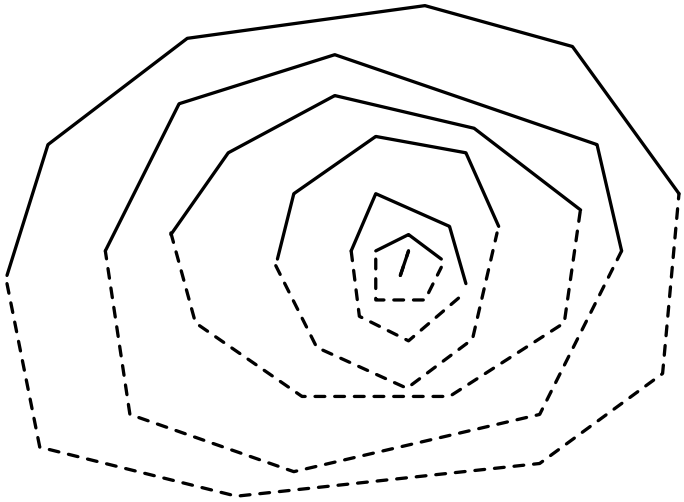

Figure 5: Decomposition of each convex layer into upper and lower hulls.

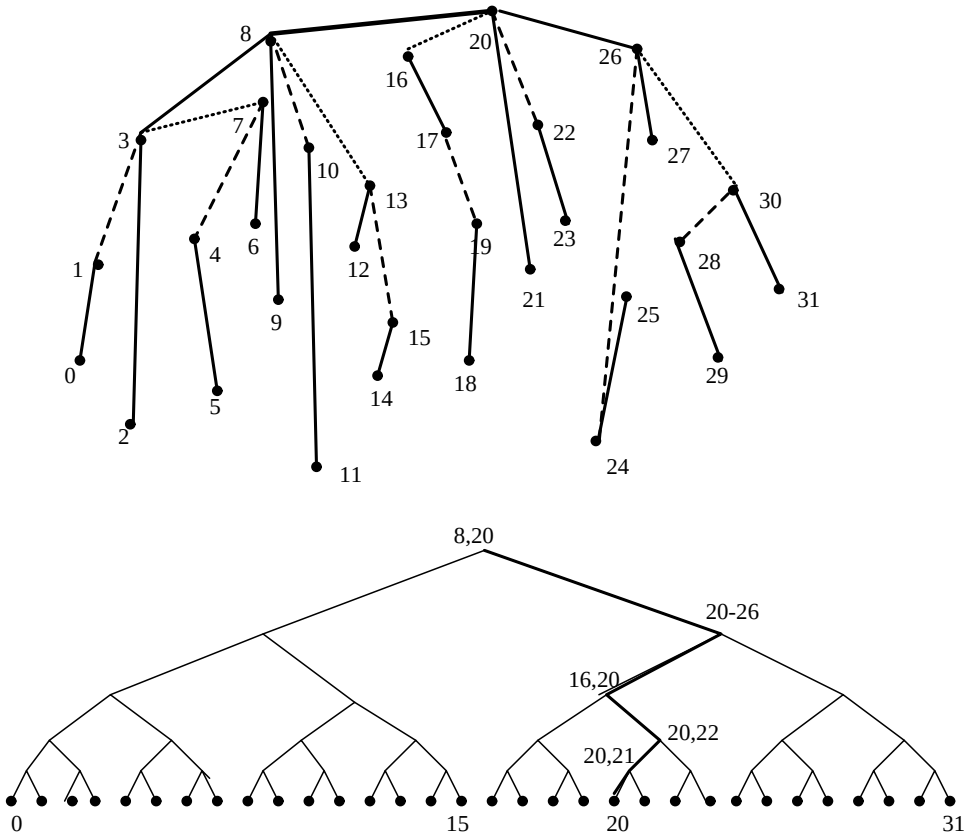

Figure 6: A complete binary tree and hull graph of upper hulls.

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 6(b) shows the leaf-to-root path when vertex p_{20} is deleted. Figs. 7(a)-(f) show the updates of bridges when p_{20} is deleted and Fig. 7(g) 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[11].

Theorem 2 *The convex layers of a set of n points in the plane can be computed in $O(n \log n)$ time.*

2.4 Applications of Convex Hulls

Convex hulls have applications in clustering, linear regression, and Voronoi diagrams (see the next Chapter.) 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 2-dimensions $O(n \log n)$ time is both sufficient and necessary in the worst case[41]. 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 pair in 2-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 3-dimensions. Most recently Ramos[42] obtained an $O(n \log^2 n)$ time deterministic algorithm, which is an $O(\log n)$ factor off from the best randomized algorithm due to Clarkson and Shor[17].

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[37] in 2-dimensions. In k-dimensions Houle et al.[30] 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.

3 Maxima Finding

In this section we discuss a problem concerned with *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 $\Re^k$.

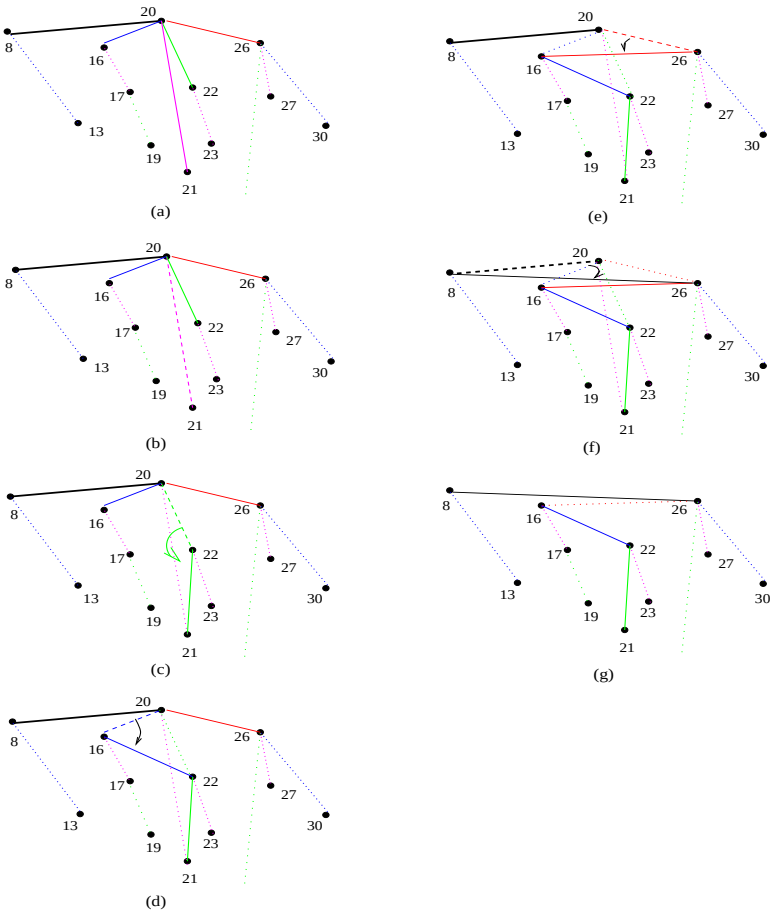

Figure 7: Update of hull graph.

3.1 Maxima in 2- and 3-Dimensions

In 2-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.* [35] or Section 5.1 below.) 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.*, $x(p_1) \leq x(p_2) \leq \ldots \leq x(p_n)$. We shall scan the points from right to left. 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 3-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^x_{i+1} of the points in S_{i+1} to $\mathcal{P}$ with $x = x(p_i)$. We now have an instance of an *on-line* 2-dimensional maximal problem, *i.e.*, for point p_i, if $p^x_i \preceq p^x_j$ for some $p^x_j \in S^x_{i+1}$, then it is not a maximal element, otherwise it is. (p^x_i denotes the projection of p_i onto $\mathcal{P}$.) If we maintain the points in $\mathcal{M}(S^x_{i+1})$ 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 $\mathcal{M}(S^x_{i+1})$, then it is ignored. Otherwise, it is in $\mathcal{M}(S^x_{i+1})$ (and also in $\mathcal{M}(S_{i+1})$; $\mathcal{M}(S^x_{i+1})$ will then be updated to be $\mathcal{M}(S^x_i)$ accordingly. The update may involve deleting points in $\mathcal{M}(S^x_{i+1})$ that are no longer maximal because they are dominated by p^x_i. Fig. 8 shows the effect of adding a maximal element p^x_i to the set $\mathcal{M}(S^x_{i+1})$ of maximal elements. Points in the shaded area will be deleted. Thus after the initial sorting, the set of maxima in 3-dimensions can be computed in $O(n \log \mathcal{H})$ time, as the *on-line* 2-dimensional maximal problem takes $O(\log \mathcal{H})$ time to maintain $\mathcal{M}(S^x_i)$ for each point p_i, where $\mathcal{H}$ denotes the size of $\mathcal{M}(S)$.

Since the total number of points deleted is at most n, we conclude that:

Lemma 1 *Given a set of n points in 2- and 3-dimensions, the set of maxima can be computed in $O(n \log n)$ time.*

For 2- and 3-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 2-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 3-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

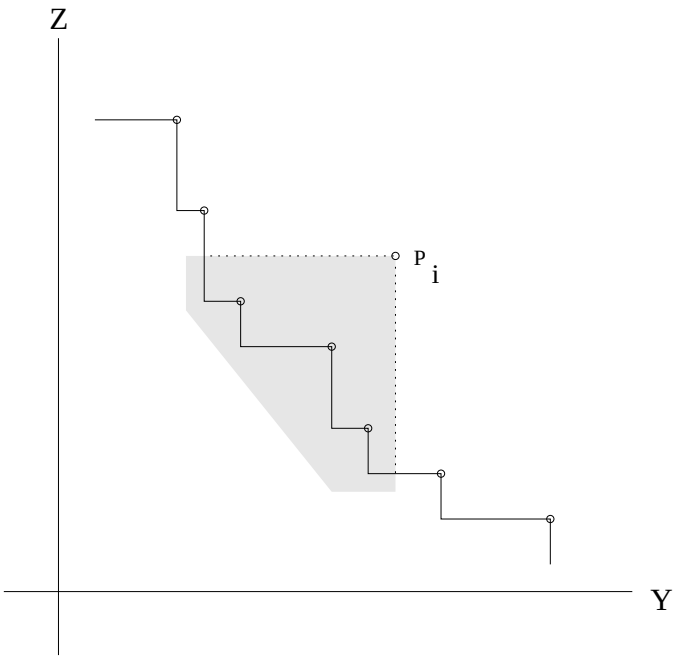

Figure 8: Update of maximal elements.

in Section 2.1 to derive an output-sensitive algorithm.

Theorem 3 *Given a set S of n points in 2- and 3-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 $\mathcal{M}(S)$.*

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 that one can compute the set of maxima in $O(n \log^{k-2} n)$ time, for $k > 3$ by divide-and-conquer.[2]

Let us first consider a *bi-chromatic maxima-finding* problem. Consider a set of n *red* and a set of m *blue* points, denoted R and B respectively. The bi-chromatic 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 3-dimensions, this problem can be solved by plane-sweep 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 $\mathcal{M}(R^x_{i+1})$ and $\mathcal{M}(B^x_{j+1})$, 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 $\mathcal{M}(B^x_{j+1})$. $r_i \in \mathcal{M}(R, B)$,

[2]This was improved to $O(n \log^{k-3} n \log \log n)$ time by Gabow et al.[27].

if r_i^x is not dominated by any points in $\mathcal{M}(B_{j+1}^x)$. We then update the set of maxima for $R_i^x = R_{i+1}^x \cup \{r_i^x\}$. That is, if $r_i^x \preceq q$ for $q \in \mathcal{M}(R_{i+1}^x)$, then $\mathcal{M}(R_i^x) = \mathcal{M}(R_{i+1}^x)$. Otherwise, the subset of $\mathcal{M}(R_{i+1}^x)$ dominated by r_i^x is removed, and r_i^x is included in $\mathcal{M}(R_i^x)$. 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 2 *The bi-chromatic maxima-finding problem for a set of n red and m blue points in 3-dimensions can be solved in $O(N \log N)$ time, where $N = m + n$.*

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

Lemma 3 *The bi-chromatic 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 bi-chromatic 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 3 we know that this bi-chromatic 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 that

Theorem 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 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.*

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 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[6] 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: 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 $p_{j_1} \preceq p_{j_2} \preceq \ldots \preceq p_{j_\lambda} = p_i$. 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 [6].[3]

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 $r \in R$. $leader_R(p), p \in R$ is the point $\mathcal{H}_p$ in $\mathcal{D}_R(p)$ with the largest y-coordinate. $Strip_L(p, R), p \in R$ is the subset of points of $\mathcal{D}_L(p)$ dominated by p but with y-coordinates greater than $leader_R(p)$, *i.e.*, $Strip_L(p, R) = \{q \in \mathcal{D}_L(p) | y(q) > y(\mathcal{H}_p)\}$ for $p \in R$. $Left_L(p, R), p \in R$, is defined to be the largest $l_S(q)$ over all $q \in Strip_L(p, R)$ if $Strip_L(p, R)$ 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 $Strip_L(p, R)$. Assume that the points in $S = \{p_1, p_2, \ldots, p_n\}$ have been sorted as $x(p_1) < x(p_2) < \ldots < x(p_n)$. We shall present a divide-and-conquer algorithm which can be called with $R = S$ and $Left_\emptyset(p, S) = -\infty$ 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.

Lemma 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 $p \in R$, $Left_L(p, R)$, 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 $w(p_r)$+ $Left_L(p_r, R)$, if $Left_L(p_r, R) \neq -\infty$ and to $w(p_r)$ if $Left_L(p_r, R) = -\infty$, 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 $Left_L(p, R_1)$ equals $Left_L(p, R)$, this call will return the labels for all $q \in R_1$ which are the final labels for $q \in R$.

[3] Some of the notation is slightly modified. In [6] min is used in the *label* function, instead of max. See [6] for details.

4. Compute $Left_{R_1}(p, R_2)$.

5. Compute $Left_{L \cup R_1}(p, R_2)$, given $Left_{R_1}(p, R_2)$ and $Left_L(p, R)$. That is, for each $p \in R_2$, set $Left_{L \cup R_1}(p, R_2)$ to be $\max\{Left_{R_1}(p, R_2),\ Left_L(p, R)\}$.

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 $Left_{R_1}(p, R_2)$ 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. (Fig. 9) Suppose we have computed the labels $l_L(p), p \in L$. We compute $Left_L(p, R)$ by using a plane sweep technique scanning points in $L \cup R$ is ascending y-coordinate. We will maintain for each point $r \in R$ $Strip_L(r, R)$ along with the highest and rightmost points in the subset, denoted $1st_L(r, R)$ and $last_L(r, R)$ respectively, and $leader_R(r)$. For each point $p \in L \cap Strip_L(r, R)$ for some $r \in R$ we maintain a label $max\_l(p)$, which is equal to $\max\{l_L(q) | q \in Strip_L(r, R) \text{ and } y(q) < y(p)\}$.

A stack ST_R will be used to store $leader_R(r_i)$ of $r_i \in R$ such that any element r_t in ST_R is $leader_R(r_{t+1})$ 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 Fig. 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 $Strip_L(r, R)$ for a yet-to-be-scanned point $r \in R$. (The staircase above r_4 in Fig. 9 is stored in ST_L. The solid staircases indicate $Strip_L(r_i, R)$ 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 $max\_l(q)$ to be the larger of $l_L(q)$ and $max\_l(q')$. 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 $Strip_L(q, R)$ is initialized to be the entire contents of ST_L and let $1st_L(q, R)$ be the top element of ST_L and $last_L(q, R)$ be the bottom element of ST_L.

If the top element of ST_R is equal to $leader_R(q)$, we set $Left_L(q, R)$ to $max\_l(q')$, where q' is $1st_L(q, R)$, 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 $leader_R(q)$. As shown in Fig. 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, $Strip_L(r_i, R)$ is *concatenated* with $Strip_L(q, R)$ to maintain its maximality. That is, the points in $Strip_L(r_i, R)$ are scanned from $1st_L(r_i, R)$ to $last_L(r_i, R)$ until a point, if any, α_i is encountered such that $x(\alpha_i) > x(last_L(q, R))$. $max\_l(q')$, $q' = 1st_L(q, R)$, is set to be the larger of $max\_l(q')$ and $max\_l(\alpha)$ and $last_L(q, R)$ is temporarily set to be $last_L(r_i, R)$. If no such α_i exists, then the entire $Strip_L(r_i, R)$ is ignored. This process repeats until $leader_R(q)$ of q is on top of ST_R. At that point, we would have computed $Strip_L(q, R)$ and $Left_L(q, R)$ is $max\_l(q')$, where $q' = 1st_L(q, R)$. We initialize ST_L to be empty and continue to scan the next point.

It has been shown in [6] that this scanning operation takes linear time (with path compression), so the overall algorithm takes $O(m \log m)$ time.

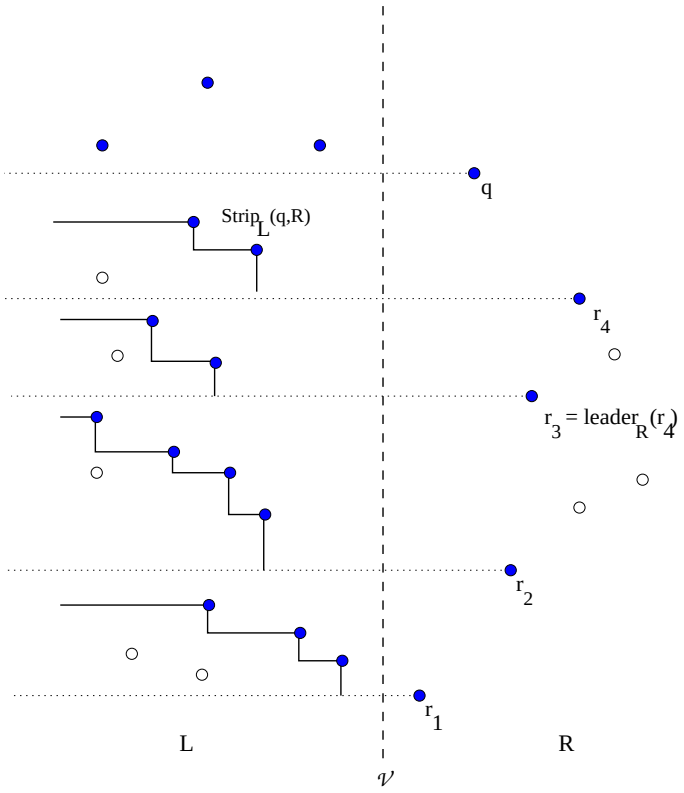

Figure 9: Computation of $Left_L(p, R)$

.

Theorem 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 $x(p_1) < x(p_2) < \ldots < x(p_n)$.

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 $r \in R$ $Strip_L(r, R)$ 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 $Strip_L(r, R)$ 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 7 *The maxdominance problem of a set S of n points can be solved in $O(n \log n + \mathcal{F})$ time, where $\mathcal{F} = \Sigma_{p \in S} |\mathcal{M}(\mathcal{D}_S(p))|$.*

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.*[1], 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'll 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 i-th row, where $i = \lceil \frac{n}{2} \rceil$.

2. Recursively solve the row maxima-searching problem for the submatrices $M[1, \ldots, i-1; 1, \ldots, j]$ 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 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 be 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.[1]

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 a more recent algorithm, the above problems can be solved in $O(n)$ time[29]. It is also used in solving the largest empty rectangle problem discussed in the section on geometric optimization in the next chapter.

5 Decomposition

Polygon decomposition arises in pattern recognition[47] 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.

5.1 Trapezoidalization

We'll 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 $\overline{v_l, v_r}$, 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 $\overline{v, v_l}$ or $\overline{v, v_r}$ lies totally in the interior of P. If v is a V-cusp or Λ-cusp, then $\overline{v_l, v_r}$ either lies totally in the interior of P or degenerates to v itself.

Consider only the segments $\overline{v_l, v_r}$ 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 (Fig. 10(a)).

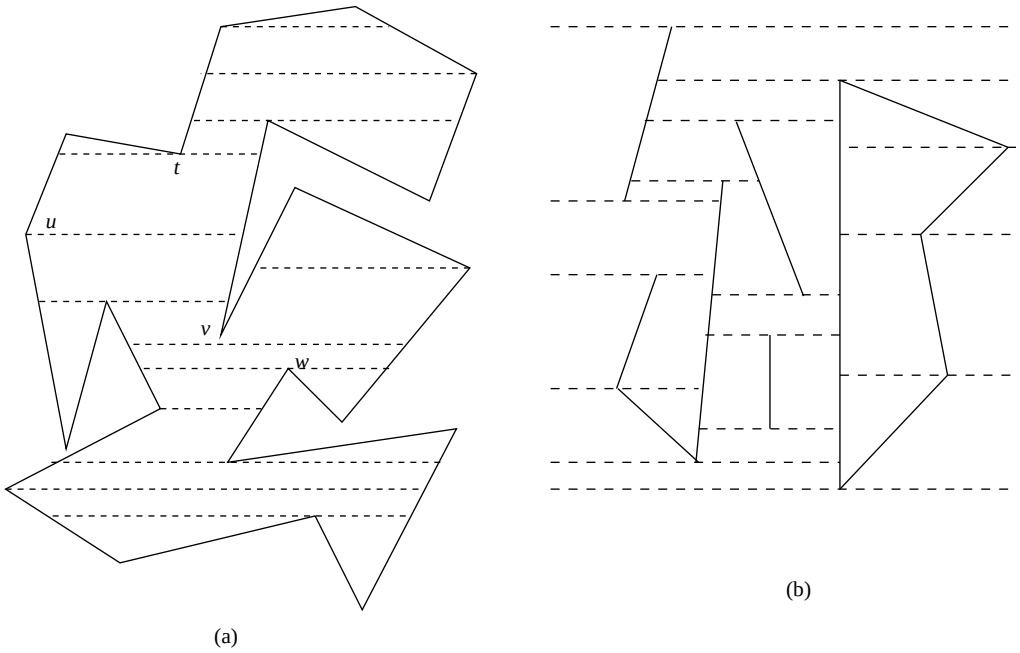

Figure 10: Trapezoidalization of a polygon.

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) (Fig. 10(b)). 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 $n = |V|$[41], while the trapezoidalization of a simple polygon can be found in linear time[12].

The plane-sweep algorithm works as follows. The vertices of the graph $G(V, E)$ are sorted in descending y-coordinates. We'll 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.

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_l and v_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 the 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 $E_l(v_i)$ and $E_r(v_i)$ respectively. The left projection v_{il} of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the left of $E_l(v_i)$ in the sweep-line status. Similarly the right projection v_{ir} of v_i is the intersection of $\mathcal{H}_{v_i}$ and the edge to the right of $E_r(v_i)$ in the sweep-line status. All the edges between $E_l(v_i)$ and $E_r(v_i)$ 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 $E_l(v_i)$ and $E_r(v_i)$ such that v_i lies in between. The left projection v_{il} of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_l(v_i)$ and the right projection v_{ir} of v_i is the intersection of $\mathcal{H}_{v_i}$ and $E_r(v_i)$. All the edges incident on v_i are then inserted in an order-preserving manner between $E_l(v_i)$ and $E_r(v_i)$ in the sweep-line status.

Fig. 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 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.*

5.2 Triangulation

In this section we consider triangulating a planar straight-line graph by introducing non-crossing 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

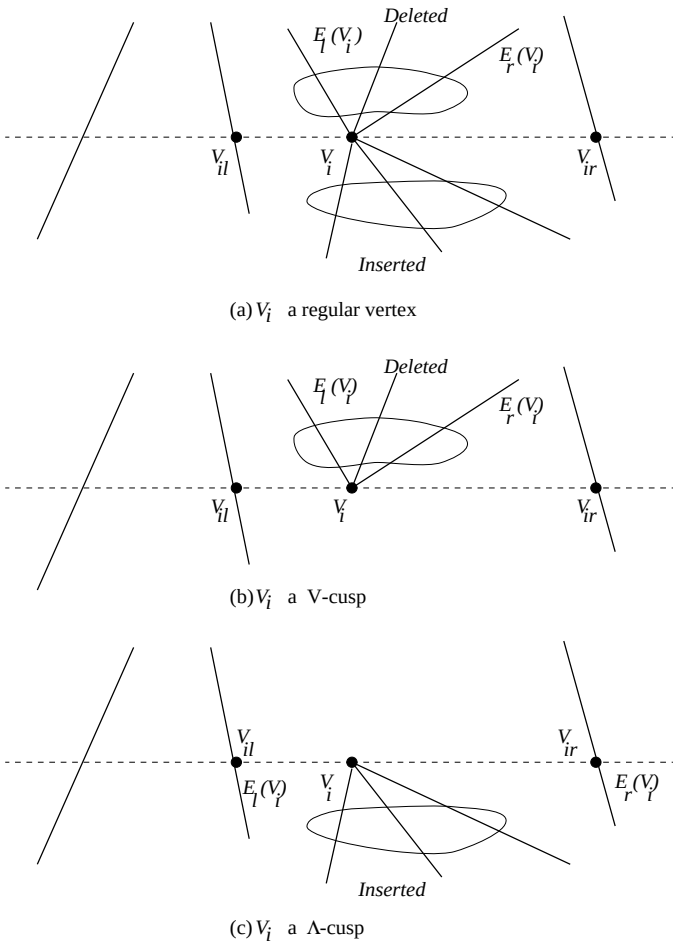

Figure 11: Updates of sweep-line status.

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.

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. 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 Fig. 12(a) is a monotone polygon.

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 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 Fig. 10(a) 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*[41]. Fig. 12 shows a decomposition of the simple polygon in Fig. 10(a) into a collection of monotone polygons.

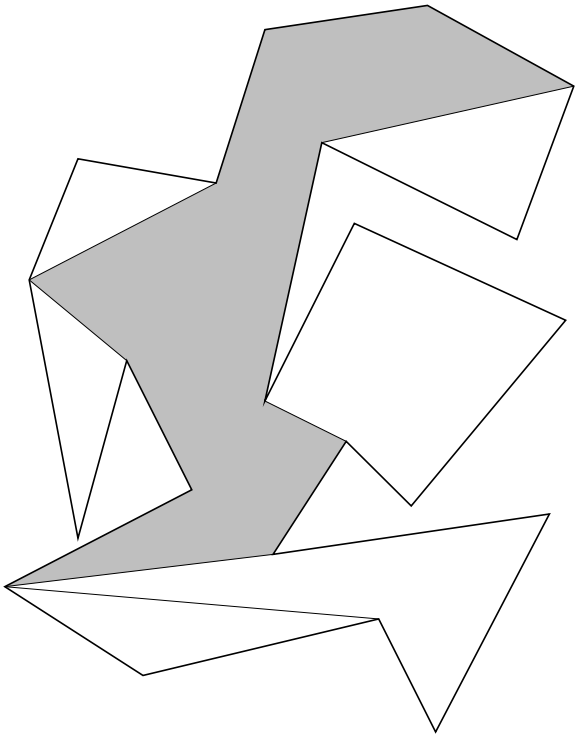

Figure 12: Decomposition of a simple polygon into monotone subpolygons.

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. $v_l \in \mathcal{L}$. Let v_{i_j} be the last vertex on $\mathcal{C}$ that is visible from v_l. That is, the internal angle $\angle(v_{i_j}, v_{i_{j'}}, v_l)$, where $j < j' \leq k$, is less than π, and either $v_{i_j} = v_{i_0}$ or the internal angle $\angle(v_{i_{j-1}}, v_{i_j}, v_l)$ is greater than π. Add diagonals $\overline{v_l, v_{i_{j'}}}$, for $j \leq j' < k$. Update $\mathcal{C}$ to be composed of vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_j}, v_l$.

2. $v_l \in \mathcal{R}$. In this case we add diagonals $\overline{v_l, v_{i_{j'}}}$, 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.

Figs. 13(a) and (b) illustrate these two cases respectively, in which the shaded portion has been triangulated.

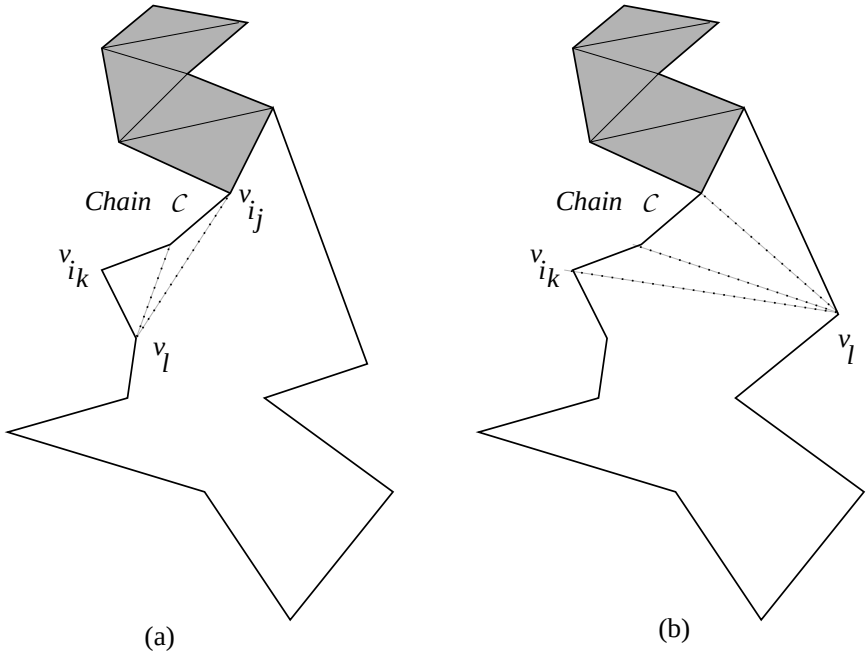

Figure 13: Triangulation of a monotone polygon.

Fournier and Montuno[26] and independently Chazelle and Incerpi[14] showed that triangulation of a polygon is linear-time equivalent to computing the horizontal visibility map. Based on this result Tarjan and Van Wyk 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 simpler algorithm of the same complexity was give in [33]. A Las Vegas algorithm with $O(n \log^* n)$ expected time was given in [18]. A recent breakthrough result of Chazelle[12] 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[4].

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 horizontal visibility 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.)

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*. 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'd 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 [36]. As the generalized Delaunay triangulation (also known as *constrained* Delaunay triangulation) is of fundamental importance, we describe in Section 5.2.4 an optimal algorithm due to Chew[16] 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 [24] were also studied. But if the constraints are on the measure of the triangles, for instance, each triangle in the triangulation must be non-obtuse, then

Steiner points must be introduced. See Bern and Eppstein (in [21], pp. 23-90) for a survey of triangulations satisfying different criteria and discussions of triangulations in 2- and 3-dimensions. Bern and Eppstein gave an $O(n \log n + \mathcal{F})$ algorithm for constructing a non-obtuse triangulation of polygons using $\mathcal{F}$ triangles. Bern et al.[9] 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.

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 $\mathcal{F}$ non-overlapping 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. See [21] for more references on 3-dimensional triangulations and Delaunay triangulations in higher dimensions.

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 non-intersecting 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 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 its 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 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 $G_l(V_l, E_l)$ and to the right graph $G_r(V_r, E_r)$, which are associated respectively with bounding rectangle R_{v_l} and R_{V_r} whose x-intervals are $\mathcal{X}_{V_l} = (x(p_1), m)$ and $\mathcal{X}_{V_r} = (m, x(p_n))$, respectively. The edges $\overline{p, q} \in E$ that are intersected by the dividing line and that do not *span*[4] 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*, $\overline{p, \epsilon(p, q)} \in E_l$ and $\overline{\epsilon(p, q), q} \in E_r$ are created. Fig. 14(a) 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 $\mathcal{L}$.

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

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

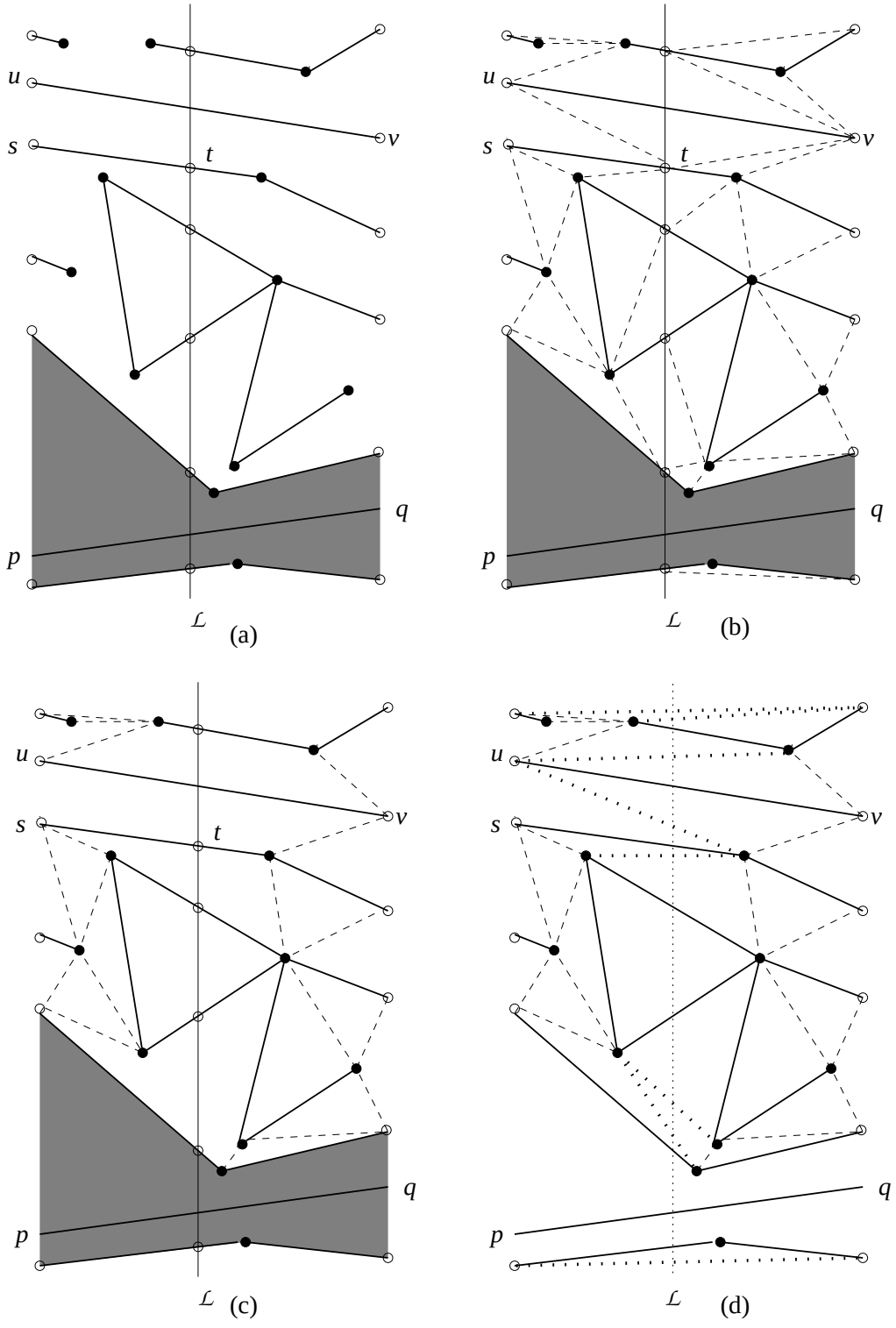

Figure 14: Computation of constrained Delaunay triangulation for subgraphs in adjacent bounding rectangles.

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. Fig. 14(b) 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. Fig. 14(c) 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 Fig. 14(b).

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 is constructing unconstrained Delaunay triangulation given in [36] and can be accomplished in time linear in the number of edges in the combined bounding rectangle. The dotted lines in Fig. 14(d) are the Delaunay edges introduced in Step 2. We therefore conclude with the following theorem.

Theorem 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.ece.nwu.edu/~theory.

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 polygons with n vertices. But if a partition of a simple polygon into a minimum number of star-shaped polygons is desired, Keil[32] gave an $O(n^5 N^2 \log n)$ 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**.[40] 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[32]. 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 quadrilaterization 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 quadrilaterization or positioning at most $\lfloor n/4 \rfloor$ guards is known (see [40] for more information). The minimum covering problem by star-shaped polygons for rectilinear polygons is still *open*. For variations and results of art gallery problems the reader is referred to [40, 45]. Polynomial time algorithms for computing the minimum partition of a simple polygon into simpler parts while allowing Steiner points can be found in [4].

The minimum partition or covering problem for simple polygons becomes NP-hard when the polygons are allowed to have *holes* [32]. Asano et al.[3] 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 polygons with holes into a minimum number of rectangles (allowing Steiner points) arises in VLSI artwork data. Imai and Asano[31] gave an $O(n^{3/2} \log n)$ time and $O(n \log n)$ space algorithm for partitioning a rectilinear polygons 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, which is a special case of a *Boolean basis problem*, however, is NP-hard[38].

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[15] 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[8]. The problem of minimum partition into convex part and the problem of determining if a nonconvex polyhedron can be partitioned into tetrahedra without introducing Steiner points are NP-hard[43].

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 represent the state of the art of this field, are presented. More topics will be covered in the next chapter.

In Section 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 2-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. It is known that finding a minimum covering of a simple polygon by star-shaped polygons is NP-hard, But the problem for rectilinear polygons, however, is still *open*. Although the triangulation problem of a simple polygon has been solved by Chazelle[12], 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.

7 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 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.
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 $x_k = \sum_{j=1}^{k-1} \mu_j x_j - \mu_k$; for a hyperplane $\mathcal{H} : x_k = \sum_{j=1}^{k-1} \mu_j x_j + \mu_k$, its dual $\mathcal{D}(\mathcal{H})$ is a point denoted by $(\mu_1, \mu_2, \ldots, -\mu_k)$. See [22, 19] for more information.
Divide-and-Marriage-before-Conquest: A problem solving paradigm derived from divide-and-conquer. A term coined by the Kirkpatrick and Seidel[34], 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 $\sum_{j=1}^{n} \alpha_i p_i$, where $\alpha_i, \forall i$ is nonnegative, and $\sum_{j=1}^{n} \alpha_i = 1$.
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 *on-line*, 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 on-line is similar to the off-line algorithms that work incrementally, *i.e.,* compute 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 boundary 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.

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] Aggarwal, A., M. M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric Applications of a matrix-Searching Algorithm," *Algorithmica*, 2,2, (1987), 195-208.

[2] N. Amato and F. P. Preparata, "The Parallel 3D Convex Hull Problem Revisited," *Int'l J. Comput. Geom. & Applications*, 2,2 (June 1992), 163-173.

[3] Ta. Asano, Te. Asano and H. Imai, "Partitioning a Polygonal Region into Trapezoids," *J. ACM* 33,2 (April 1986), 290-312.

[4] Ta. Asano, Te. Asano and R. Y. Pinter, "Polygon Triangulation: Efficiency and Minimality," *J. Algorithms*, 7 (1986), 221-231.

[5] M. J. Atallah, "Parallel Techniques for Computational Geometry," *Proceedings of IEEE*, 80,9 (Sept. 1992), 1435-1448.

[6] M. J. Atallah and S. R. Kosaraju, "An Efficient Algorithm for Maxdominance with Applications," *Algorithmica*, 4 (1989), 221-236.

[7] D. Avis, D. Bremner and R. Seidel, "How Good are Convex Hull Algorithms," *Computational Geometry: Theory and Applications*, 7,5/6 (April 1997), 265- 301.

[8] C. Bajaj and T. K. Dey, "Convex Decomposition of Polyhedra and Robustness," *SIAM J. Comput.*, 21 (1992), 339-364.

[9] M. Bern, S. Mitchell and J. Ruppert, "Linear-Size Nonobtuse Triangulation of Polygons," *Proc. 10th Annual ACM Symp. Comput. Geometry*, June 1994, 221-230.

[10] T. M. Chan, "Output-Sensitive Results on Convex Hulls, Extreme Points, and Related Problems," *Proc. 11th ACM Annual Symp. on Computational Geometry,* June 1995, 10-19.

[11] B. Chazelle, "On the Convex Layers of a Planar Set," *IEEE Trans. Inform. Theory*, IT-31 (1985), 509–517.

[12] B. Chazelle, "Triangulating a Simple Polygon in Linear Time," *Discrete & Comput. Geometry*, 6 (1991), 485-524.

[13] B. Chazelle, "An Optimal Convex Hull Algorithm for Point Sets in Any Fixed Dimension," *Discrete & Computational Geometry*, 8 (1993), 145-158.

[14] B. Chazelle and J. Incerpi, "Triangulation and shape-complexity," *ACM Trans. Graphics*, 3,2 (1984), 135-152.

[15] B. Chazelle and L. Palios, "Triangulating a Non-Convex Polytope," *Discrete & Comput. Geometry*, 5 (1990) 505-526.

[16] L. P. Chew, "Constrained Delaunay Triangulations," *Algorithmica*, 4,1 (1989), 97-108.

[17] K. L. Clarkson and P. W. Shor, "Applications of Random Sampling in Computational Geometry, II," *Discrete & Comput. Geometry*, 4 (1989), 387-421.

[18] K. L. Clarkson, R. E. Tarjan and C. J. Van Wyk, "A Fast Las Vegas Algorithm for Triangulating a Simple Polygon," *Discrete & Comput. Geometry*, 4 (1989), 423-432.

[19] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, Computational Geometry Algorithms and Applications, Springer-Verlag, 1997.

[20] D. P. Dobkin and D. G. Kirkpatrick, "Fast Detection of Polyhedral Intersection," *Theoret. Comput. Sci.*, 27 (1983), 241-253.

[21] D. Z. Du and F. K. Hwang, Computing in Euclidean Geometry, Eds., World Scientific Publishing Co., Singapore, 1992.

[22] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer-Verlag, 1987.

[23] H. Edelsbrunner and W. Shi, "An $O(n \log^2 h)$ Time Algorithm for the Three-Dimensional Convex Hull Problem," *SIAM J. Comput.,* 20,2, (April 1991), 259-269.

[24] H. Edelsbrunner and T. S. Tan, "A Quadratic Time Algorithm for the Minmax Length Triangulation," *SIAM J. Comput.*, 22 (1993), 527-551.

[25] S. Fortune, "Computational Geometry," in Directions in Computational Geometry, R. Martin, Ed., Information Geometers, 1993.

[26] A. Fournier and D. Y. Montuno, "Triangulating simple polygons and equivalent problems," *"ACM Trans. Graphics*, 3,2 (1984), 153-174.

[27] H. N. Gabow, J. L. Bentley and R. E. Tarjan, "Scaling and Related Techniques for Geometry Problems," *Proc. 16th Annu. ACM Sympos. Theory Comput.*, 1984, 135-143.

[28] J. E. Goodman and J. O'Rourke, (eds.) The Handbook of Discrete and Computational Geometry, CRC Press LLC, Boca Raton, FL, 1997.

[29] J. Hershberger and S. Suri, "Matrix Searching with the Shortest Path Metric," *Proc. 25th ACM Symp. Theory of Comput.*, (1993), 485-494; *SIAM J. Comput.,* to appear.

[30] M. E. Houle, H. Imai, K. Imai, J.-M. Robert and P. Yamamoto, "Orthogonal Weighted Linear L_1 and L_∞ Approximation and Applications". *Discrete Appl. Math.*, 43 (1993), 217-232.

[31] H. Imai and Ta. Asano, "Efficient Algorithms for Geometric Graph Search Problems," *SIAM J. Comput*, 15,2 (May 1986), 478-494.

[32] J. M. Keil, "Decomposing a Polygon into Simpler Components," *SIAM J. Comput.*, 14 (1985), 799-817.

[33] D. G. Kirkpatrick, M. M. Klawe and R. E. Tarjan, "Polygon Triangulation in $O(n \log\log n)$ Time with Simple Data Structures," *Proc. 6th Annual ACM Symp. Comput. Geometry*, 1990, 34-43.

[34] D. G. Kirkpatrick and R. Seidel, "The Ultimate Planar Convex Hull Algorithm?" *SIAM J. Comput.,* 15,1 (Feb. 1986), 287-299.

[35] D. T. Lee, "Computational Geometry," Computer Science and Engineering Handbook, Ed. A. Tucker, CRC Press, 1996, 111-140.

[36] D. T. Lee and A. K. Lin, "Generalized Delaunay Triangulation for Planar Graphs," *Discrete & Comput. Geometry*, 1 (1986), 201-217.

[37] D. T. Lee and Y. F. Wu, "Geometric Complexity of Some Location Problems," *Algorithmica,* 1 (1986), 193-211.

[38] A. Lubiw, "The Boolean Basis Problem and How to Cover Some Polygons with Rectangles," *SIAM J. Disc. Math.*, 3,1 (Feb. 1990), 98-115.

[39] J. Matoušek, and O. Schwarzkopf, "A Deterministic Algorithm for the Three-Dimensional Diameter Problem," *Proc. 25th ACM Symp. on Theory of Comput.*, May 1993, 478-484.

[40] J. O'Rourke, Art Gallery Theorems and Algorithms, Oxford University Press, New York, NY, 1987.

[41] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, 1988.

[42] E. A. Ramos, "Construction of 1-D Lower Envelopes and Applications," *Proc. 13th Annual ACM Symp. Comput. Geometry*, 1997, 57-66.

[43] J. Ruppert and R. Seidel, "On the Difficulty of Triangulating Three-Dimensional Nonconvex Polyhedra" *Discrete & Comput. Geometry*, 7 (1992), 227-253.

[44] , J. Sack and J. Urrutia, Handbook of Computational Geometry, Elsevier Sci. Publishers, Amsterdam, 1997.

[45] T. C. Shermer, "Recent Results in Art Galleries," *Proceedings IEEE*, 80,9 (Sept. 1992), 1384-1399.

[46] R. E. Tarjan and C. J. Van Wyk, "An $O(n \log\log n)$-time Algorithm for Triangulating a Simple Polygon," *SIAM J. Comput.*, 17,1 (Feb. 1988), 143-178; Erratum: 17,5 (1988), 1061.

[47] G. T. Toussaint, "New Results in Computational Geometry Relevant to Pattern Recognition in Practice," in Pattern Recognition in Practice II, E. S. Gelsema and L. N. Kanal, Eds., North-Holland, Amsterdam, Netherland, 1986, 135-146.

[48] F. F. Yao, "Computational Geometry," in Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, J. van Leeuwen, Ed., 1994, 343-389.

[49] Yap, C., "Towards Exact Geometric Computation," *Computational Geometry: Theory and Applications*, 7,3 (Feb. 1997), 3-23.

8 Further Information

For some problems we present efficient algorithms in pseudo code 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. A recent textbook by de Berg *et al.*[19] contains a very nice treatment of this topic. The reader who is interested in *parallel* computational geometry is referred to [5]. For current research results, the reader may consult the Proceedings of the Annual ACM symposium 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 [28, 35, 44, 48]. The **ftp** site /pub/geometry/geombib.tar.gz at ftp.cs.usask.ca contains close to ten thousand entries of bibliography in this field.

David Avis announced a convex hull/vertex enumeration code, *lrs*, based on reverse search and made it available at this site ftp://mutt.cs.mcgill.ca/pub/C/lrs.html. 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. For more details consult the user's manual found at the site. See [7] 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 computational geometry page by J. Erickson at http://www.cs.duke.edu/~jeffe/compgeom give a comprehensive description of research activities of computational geometry.