Geometric Data Structures

Michael T. Goodrich

Center for Geometric Computing
Dept. of Computer Science
Johns Hopkins Univ.
Baltimore, MD 21218
goodrich@cs.jhu.edu

Kumar Ramaiyer

Informix Software, Inc.
1111, Broadway,
Suite 2000
Oakland, CA 94607
rk@informix.com

Chapter 1

Geometric Data Structures

1.1   Introduction

Computational geometry problems often require preprocessing geometric objects into a simple and space-efficient structure so that the operations on the geometric objects can be performed repeatedly in an efficient manner. We refer to this as the "geometric data structuring" approach. This approach has been widely used by several researchers to design very elegant data structures to solve a number of geometry problems [15, 22, 26, 32, 36, 43, 44, 45, 56, 62].

Classic data structures like lists, trees, and graphs are by themselves not sufficient to represent geometric objects as either they are generally one dimensional in nature or do not capture the rich structural properties of the geometric objects in the domain. For example, in a planar subdivision the clockwise and counter-clockwise orderings of edges around a vertex are often useful for solving many problems involving subdivisions (e.g, see Guibas and Stolfi [41]). Similarly, facial ordering and connectivity information of subdivisions is often needed and requires special representation. If one is given a collection of horizontal segments in $\mathbb{R}^2$, for example, one may like to represent the endpoints, and also some representation of the "aboveness" partial order (e.g., see Edesbrunner [34]). Higher dimensional geometric objects define even richer relationships and likewise cannot be easily represented by the classical data structures, and require careful study. Even with this short list of examples one can see that geometric data requires the representation of relationships that cannot be represented using strictly numeric or combinatoric data structures. Indeed, it is the interplay of nu-

1

meric and combinatoric data that makes the design of efficient geometric
data structures an interesting and challenging research domain.

1.1.1 Problem classification and goals

Data structuring problems involving geometric objects vary and are often
classified as follows:

Static: In this case all the geometric objects in the problem domain are given
as part of the input.

Online: In this case new geometric objects are allowed to be added to the
problem domain, but cannot be deleted.

Dynamic: In this most general case new geometric objects are allowed to be added
and some existing objects are allowed to be deleted from the problem
domain.

In addition, data structures used for storing geometric objects should ideally
achieve all of the following goals:

- capture structural information,

- allow for efficient query processing,

- allow for efficient updates,

- optimize the space required, and

- store objects efficiently so as to minimize the number of I/O accesses,
when the input size is very large.

1.1.2 Chapter outline

In this chapter we review and highlight research on geometric data struc-
tures, describing important examples for each of the above problem classific-
ations. In each case we review we sketch how well it achieves the basic goals
of geometric data structure design. In the next section we describe methods
for representing embedded straight line graphs, which arise in a number of
computational geometry contexts, including the construction and mainten-
ance of fundamental geometric structures, including convex hulls, Voronoi
diagrams, Delaunay triangulations, and arrangements. In Section 1.3 we re-
view several methods for performing an important search operation in such

subdivisions—the point location search, and we give a short review of dynamic methods for solving this problem in Section 1.4. In Section 1.5 we discuss some methods for representing convexity, and in Section 1.6 we describe some data structures for representing data that is *rectilinear* (i.e., aligned with the coordinate axes). Finally, in Section 1.7 we discuss some general techniques for designing geometric data structures.

Since geometric data structures are fundamental in the design and implementation of geometric computations, there are a necessarily a number of interesting geometric data structures that we will not be discussing in this chapter. Fortunately, many of these are covered in chapters by others in this collection. In particular, Nievergelt and Widmayer discuss a number of spatial and rectilinear structures for higher-dimensional spaces in Chapter **??**. In addition, general techniques for designing randomized geometric data structures are covered in the chapters by Matoušek (Chap. **??**) and Mulmuley (Chap. **??**). Data structures for shortest paths and ray shooting are discussed by Hershberger and Suri in Chapter **??** as well as by Maheshwari and Sack in Chapter **??**. In addition, Urrutia discusses the important and related visibility graph structure in Chapter **??**. An interesting variation of the data structuring problem is covered by Smid in Chapter **??** and by Mitchell in Chapter **??**, where one is allowed to answer queries *approximately*. There are also a host of interesting data structures that are based upon ϵ-nets and spanning trees with low stabbing numbers, which are topics covered by Matoušek in Chapter **??** and by Agarwal and Sharir in Chapter **??**. Finally, fundamental to the issue of geometric representations is the issue of numeric stability, which is covered by Yap in Chapter **??**.

We highlight in the following sections various geometric data structures and we also discern some of the general principles behind data structure design for geometric structures.

1.2 Embedded Planar Graphs

A graph $G = (V, E)$ is said to be *embedded* in a surface S when it is drawn on S so that no two edges intersect. A graph is *planar* if it can be embedded in the plane; a *plane graph* has already been embedded in the plane [42], in which case it makes sense to define the set F of faces of G. A planar graph can always be embedded in the plane so that all its edges are straight-line segments [37] and such an embedded graph is called *planar straight line graph* (PSLG).

The planar graphs play an important role in many two-dimensional computational geometry problems, for an embedded planar graph represents a planar subdivision, which is a structure that arises in several useful applications, including include arrangements of lines, Voronoi diagrams, Delaunay triangulations, and general triangulations.

A measure of the usefulness of an embedded graph representation is that such a representation should allow for efficient traversal of edges around a vertex (in clockwise and counter-clockwise direction), and it should allow for the efficient access of all edges bounding a face and all the faces incident on a vertex. In addition, it is very important for such a representation to preserve the topology of the embedding of the planar graph, as a given planar graph may have several embeddings.

Once the embedding of a planar graph is given in the form of a planar straight line graph, one of the simplest representation is to represent the graph as a collection of simple polygons. This representation is not flexible enough for traversal, however. Representations for embedded planar graphs that do allow for efficient traversals include the doubly connected edge list or DCEL [57], the winged-edge structure [3], and the quad-edge structure [41]. Let us therefore review each of these representations.

1.2.1   The Doubly Connected Edge List (DCEL)

Muller and Preparata [50, 57] designed a PSLG representation, which they called the doubly-connected edge list (or DCEL). The DCEL for a PSLG $G = (V, E, F)$ has a collection of edge nodes. This representation treats each edge as a directed edge; hence, it imposes an orientation on each edge. Each edge node $e = (v_a, v_b)$ is a structure consisting of six fields:

- V_o, representing the origin vertex (v_a),

- V_d, representing the destination vertex (v_b),

- F_l, representing the left face as we traverse on e from V_o to V_d,

- F_r, representing the right face as we traverse on e from V_o to V_d,

- CCW_o, representing the counter-clockwise successor of e around V_o, and

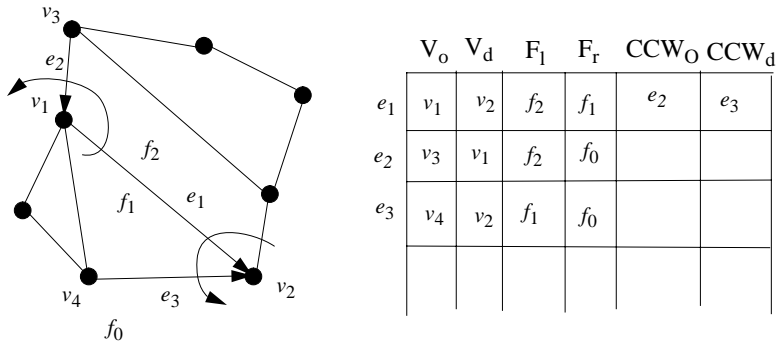- CCW_d, representing the counter-clockwise successor of e around V_d.

	V_o	V_d	F_l	F_r	CCW_O	CCW_d
e_1	v_1	v_2	f_2	f_1	e_2	e_3
e_2	v_3	v_1	f_2	f_0		
e_3	v_4	v_2	f_1	f_0		

Figure 1.1: A DCEL Representation of an Embedded Planar Graph.

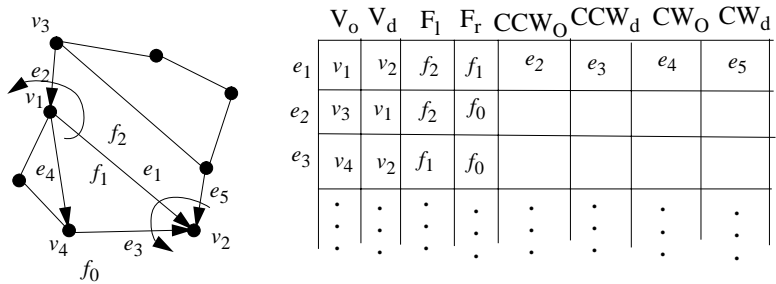|	V_o	V_d	F_l	F_r	CCW_O	CCW_d	CW_O	CW_d
e_1	v_1	v_2	f_2	f_1	e_2	e_3	e_4	e_5
e_2	v_3	v_1	f_2	f_0				
e_3	v_4	v_2	f_1	f_0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.2: A Winged-edge Representation of an Embedded Planar Graph.

The figure 1.1 shows the DCEL representation of the edge e_1 in the subdivision given.

From the DCEL representation in linear time one can easily extract the edges around faces (in clockwise-order) and the edges around a vertex (in counter-clockwise order). To get the other ordering information efficiently, however, one needs to duplicate the edges of the PSLG with opposite orientations from their original orientations and store a DCEL for this orientation as well. Once edges are duplicated and oriented in the opposite direction, one can access the edges bounding a face in counter-clockwise direction and edges around a vertex in clockwise direction in linear time. The total size of the DCEL representation is $O(|V| + |F| + |E|)$.

1.2.2   The Winged-Edge Representation

The winged-edge representation was proposed by Baumgart [3] and is similar to the DCEL. Given a PSLG $G = (V, E, F)$, the winged-edge representation

stores an array for the vertices. This array stores for each vertex an arbitrary edge incident on that vertex. The winged-edge representation also stores an array for faces, which stores for each face an arbitrary edge bounding that face. For each edge $e = (v_a, v_b)$ it stores the following information:

- V_o, representing the origin vertex (v_a),

- V_d, representing the destination vertex (v_b),

- F_l, representing the left face as seen from V_o to V_d,

- F_r, representing the right face as seen from V_o to V_d,

- CW_o, representing the clockwise successor of e around V_o,

- CCW_o, representing the counter-clockwise successor of e around V_o,

- CW_d, representing the clockwise successor of e around V_d, and

- CCW_d, representing the counter-clockwise successor of e around V_d.

Thus four successor edges are stored for each edge. This allows one to do all the accesses we outlined earlier as efficiently as in a (double-orientation) DCEL. Figure 1.2 shows the winged-edge representation for a PSLG. The total storage needed for the winged-edge is $O(|V| + |F| + |E|)$, but the constant factor is slightly better than for a double-orientation DCEL.

1.2.3   The Quad-edge Representation

Guibas and Stolfi [41] proposed the *quad-edge representation for the embedded planar graphs*. Their structure is isomorphic to the winged-edge structure, but it is given semantics general enough to represent an undirected graph embedded in an arbitrary two-dimensional manifold. Their structure also simultaneously represents the graph-theoretic primal and the dual of a planar graph. Figure 1.3 shows an example representation. The total space required for the quad-edge structure is $O(|V| + |F| + |E|)$, with the constants being essentially the same is for the winged-edge represenation.

1.2.4   Well Known PSLGs

As a motivation for the use of these subdivision representation, let us briefly review some of the well-known geometric structures that are special cases of PSLGs. These structures are treated in greater detail in other chapters of this handbook.
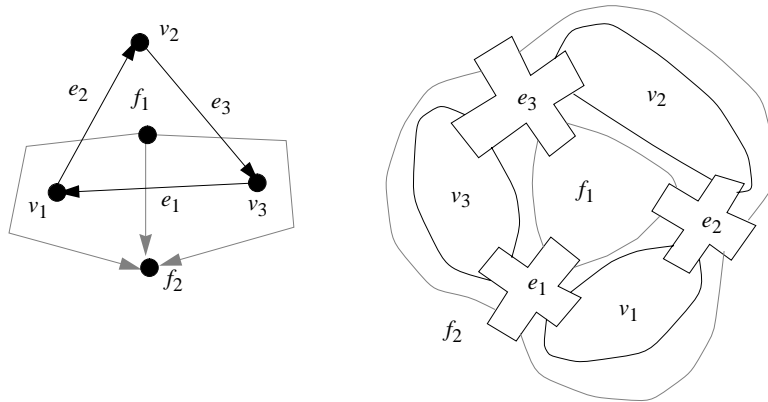
Figure 1.3: A Quad-Edge Representation of an Embedded Planar Graph. The graph and the representation are shown. The thick edges are the edges of the graph and the gray-dotted edges are the edges of the dual graph.

Line arrangements

Given a set of lines in the plane, the intersection of lines form a structure that is referred to as the *arrangement*. This structure is a planar subdivision. This is a very useful structure and has number of applications.

Since any two non-parallel lines intersect, if the given set of n lines does not contain any pair of parallel lines, then the number of intersections is $O(n^2)$ and hence the size of arrangement is $O(n^2)$. There are algorithms for computing the arrangement of lines in $O(n^2)$ time, which is of course optimal. Figure 1.4 shows an example arrangement of lines. The arrangement is a PSLG and can be represented using one of the data structures discussed in the previous section.

Voronoi Diagrams and Delaunay Triangulations

Given a collection of points and a metric, say L_2, one can define a geometric structure called the **Voronoi diagram**. This is a very useful geometric structure for answering number of useful questions one can ask about a collection of points, including closest pairs and nearest neighbors. The Voronoi diagram for a set of points is a PSLG. Each face in the graph contains an unique point from the given set. Each face is a locus of points in the plane which are closer to the point inside it, than any other point in the given set.

A related structure, the Delaunay triangulation, is the graph-theoretic
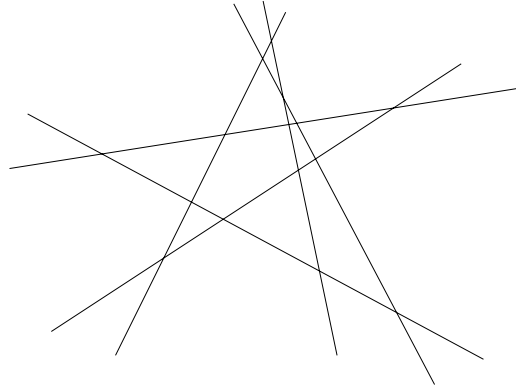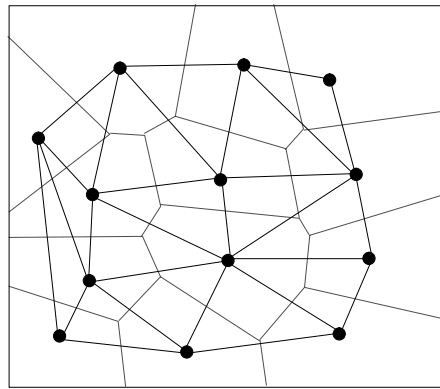
Figure 1.4: Arrangements of Lines in a Plane.

Figure 1.5: The Voronoi Diagram and Delaunay Triangulation of a Point Set. The thick edges represent the triangulation and the thin edges represent the Voronoi diagram.

planar dual of the Voronoi diagram in which the faces and vertices are inter-changed while preserving the incidence relationships (which forms a trian-gulation if the original points are in general position). Figure 1.5 shows the Voronoi diagram Both the Delaunay triangulation and the Voronoi diagram have a wide variety of uses; and they are PSLG's. We can therefore use one of the above representations (DCEL, quad-edge, or winged-edge) to store Voronoi diagrams and Delaunay triangulations. Interestingly, the quad-edge representation has the added advantage of being able to simultaneously rep-resent both the Voronoi diagram and the Delaunay triangulation of a point set using a single representation.

In the next section we study how to perform searches like point loca-tion or ray shooting in PSLG structures such as arrangements and Voronoi diagrams.

1.3 Planar Point Location and Ray Shooting

The planar point location problem is one of the fundamental computational geometry problems and has several applications. This problem has been studied by several researchers [22, 31, 32, 36, 44, 45, 56, 62], and there are a number of efficient solutions.

The problem in its widely studied form is stated as follows:

> Given a planar subdivision in the form of a PSLG (using one of the representations discussed in the previous section), preprocess the subdivision and store it in a data structure so as to answer queries of the form "given a query point p find the face of the subdivision containing p".

This query is typically answered by performing a vertical ray shooting query from p, where one determines the first segment(s) in the PSLG hit by vertical rays emanating out of p.

The important criteria for judging solutions to the point location problem are the *space* occupied by the data structure and the *query* time it allows. The preprocessing time is also an important criterion, but is generally not considered as critical as the others, since it amounts to a one-time cost.

Variations of the problem as stated above include methods for special types of subdivision i.e., introducing constraints on the shapes of the faces of the subdivision and the connectivity of the underlying planar graph. Dif-ferent subdivisions that have been studied over the years include general
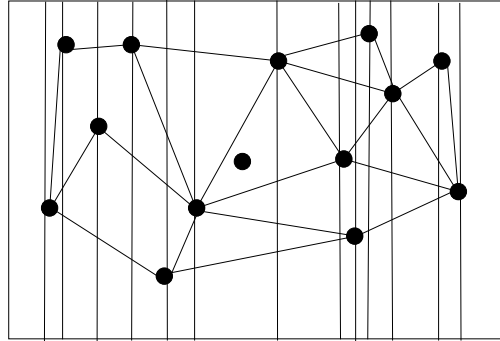
Figure 1.6: The Slab Method: Partitioning the Subdivision into Vertical Slabs Consisting of Trapezoids.

subdivisions (which may not even be connected), connected subdivisions, monotone and convex subdivisions (where each face is respectively a monotone polygon or convex polygon), and rectilinear subdivisions (where only horizontal and vertical edges are used). We outline below the various data structures used for solving the point location problem.

1.3.1   The Slab Method

The "slab" method proposed by Dobkin and Lipton [31] is historically the first non-trivial method to solve the point location problem, and it is suitable for the most general types of subdivisions. The idea is very simple. Since general subdivisions can be of an arbitrary nature, their basic idea is to partition the subdivision into a collection of vertical slabs so that each slab contains only triangles and trapezoids.

The partition is done as follows: one draws vertical lines through each of the vertices of the subdivision. This partitions the subdivision into $O(n)$ slabs. The slabs have the property that none of the edges inside the slab cross each other and the edges either cross the vertical boundaries of the slab or two or more edges meet at a vertex through which the vertical boundary line of the slab passes through. Thus, each slab contains a collection of triangles and trapezoids with vertical boundaries.

Dobkin and Lipton store this collection of slabs in a data structure to perform point location as follows: the slabs are totally ordered left-to-right by the x-coordinates of the vertices and hence can be stored using any balanced binary search tree structure. Call this search tree A. The edges within

each slab are totally ordered by the "above" relationship, i.e., given a point and the supporting line of an edge inside the slab, one can find out whether the point is above or below the edge[1]. This total order relationship (in each slab) can also be stored using a balanced binary search tree. Let us call the collection of search trees for all the slabs as B.

The point location is done as follows: given a query point $p = (x_0, y_0)$, we first use the x-coordinate of p i.e., x_0 to identify the slab in which the point lies in by doing a binary search in A. Once the slab is identified, the corresponding binary tree in B storing the trapezoids within the slab is searched by performing "aboveness" comparisons.

Summary:

Space: Since $O(n)$ binary search trees (each of size $O(n)$) need to be stored in B, the total space requirement is $O(n^2)$.

Query Time: To locate a point we need to perform two binary searches, and hence the query time complexity is $O(\log n)$.

1.3.2 The Trapezoid Method

The slab method was improved by Preparata [56] to achieve an $O(\log n)$ query time method with only $O(n \log n)$ space. His method is commonly referred to as the trapezoid method, and is in principle the same as slab method as it also partitions the subdivisions into simple trapezoids. But rather than drawing $O(n)$ vertical lines, a recursive structure is built.

The method for constructing a point location data structure using the trapezoid method is as follows. One inductively assumes one is given a trapezoid τ with vertical sides containing the subdivision (initially we can use a bounding rectangle). Identify in τ the "spanning" edges i.e., the edges which intersect both the vertical boundaries of τ. If there are no spanning edges, then partition τ again vertically at the median vertex and recurse on each side. If there are spanning edges, then parittion the slab into a number of trapezoids using the spanning edges, i.e., each trapezoid has a spanning edge as its top and bottom boundaries and the vertical boundaries of the slab as its two side boundaries. Then order these trapezoids by the "above"

---

[1]This is the unit-time operation that is used in the complexity measure. This operation can be implemented either by checking the point against the equation of the supporting line or by checking whether the vertices of the edge $e = (v_1, v_2)$ and the query point v_3 make a "left turn" or "right turn" in the order $v_1, v_2,$ and v_3.
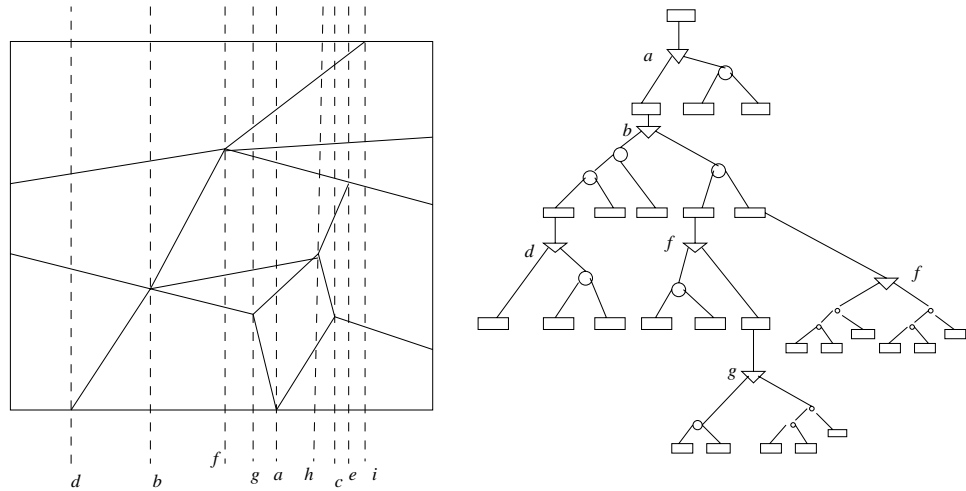
Figure 1.7: Trapezoid Method: The vertical cuts are shown in the tree as triangular nodes and the spanning cuts are shown as circular nodes.

relationship discussed in the previous section and recursively partition each such non-trivial trapezoid.

Thus, one recursively partitions the subdivision, first vertically and then using the spanning edges. The vertical cuts are global and can be organized using a single binary search tree. The trapezoids within a container trapezoid τ are organized using a *biased search tree* [4] which has the property that the depth of an item i with weight w_i is $O(\log W/w_i)$, where W is the sum of the weights of all the items in the tree. Each trapezoid is assigned a weight proportional to the number of vertices within the trapezoid. Hence the resulting structure storing the subdivision is a compound structure in which the primary tree is a balanced binary search tree organizing the vertical cuts and the secondary structure is a biased search tree storing the trapezoids. The figure 1.7 shows an example subdivision and the resulting data structure.

The worst-case depth of a leaf node u in this compound structure is calculated as follows: the depth in the primary structure is $O(\log n)$, since vertical cuts are always at median x-coordinates. Suppose the depths in the different levels of secondary structures from the leaf u (weight of $u = W_0 = 1$) to the root are $O(\log W_1/W_0), O(\log W_2/W_1), \cdots, O(\log W/W_k)$. These values form a telescoping sum that reduces to $O(\log W)$. But the total weight W at the root is $O(n \log n)$; hence, the worst-case depth of a leaf node in the compound structure is $O(\log n)$.

Summary:

Space: $O(n \log n)$ as there are $O(\log n)$ levels and in each level, structures of total size $O(n)$ are stored.

Query Time: To locate a point we need to perform alternate binary searches in the primary tree and the second tree. As argued above this is bounded by $O(\log n)$.

1.3.3 The Chain Method

Lee and Preparata [45] introduced an alternative approach. Unlike the slab method and the trapezoid method, which partition the subdivision into trapezoids, their method partitions the subdivision into regions separated by "chains". A *chain* is a sequence of edges that either forms a cycle or a path such that the end vertices belong to the boundary of the unbounded region. This way the chain, whether it is a cycle or a path, partitions the subdivision into two parts.

The method then is to first find a "median" chain so as to partition the subdivision into roughly two equal parts. Then recursively one finds chains in each of the subdivision to construct a "balanced tree" of chains. The point location method then proceeds by discriminating the point against the chain at the root to find the partition containing the point and then recurses at the appropriate child in the tree. The important work involved in searching such a data structure is therefore the discrimination of a point against a chain.

It is easy to see that discrimination of a point against a general chain is as difficult as point location in a simple polygon and hence is not really any simpler. So Lee and Preparata restricted each chain to be monotone, thus restricting their method to monotone subdivisions (which really is not a big restriction, since we can convert a general subdivision to a monotone subdivision by a vertical decomposition construction). For any monotone chain there exists a straight line such that the line orthogonal to the line intersects the chain at most once, and this property can be exploited for point location.

The figure 1.8 shows the partition of a subdivision into monotone chains. The edges are shared by different chains, but the data structure stores only one copy of each edge. Lee and Preparata's point location data structure is a compound data structure that has as its primary data structure a binary tree with a separating chain associated with each vertex and the secondary data
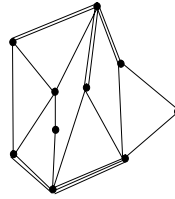
Figure 1.8: Chain Method: Parition of the subdivision into monotone chains (with respect to y-axis).

structure storing the description of each chain. The primary and secondary data structure are both balanced binary search trees, actually. During a point location discrimination of a point versus a monotone chain is done using the description of the chain in the secondary data structure. The discrimination of point versus monotone chain (say with respect to vertical line) is done in the obvious way. The projection of y-coordinates of the vertices of the chain on the vertical line partition the line into intervals which enables a binary search and the query point is located within one of the intervals (so that the point can then be compared with the straight line supporting the edge corresponding to the interval to find out which side of the chain the point lies in). Thus, each chain descrimination can be done in $O(\log n)$ time.

Summary:

Space: Each edge belongs potentially to more than one chain. But by storing each edge to the highest chain in the primary data structure to which the edge belongs, the space required for the data structure can be made to be $O(n)$.

Query Time: The discrimination of a point with respect to a monotone chain takes $O(\log n)$, and the depth of the primary data structure is $O(\log n)$. Thus, point location takes $O(\log^2 n)$ time using the chain method.

1.3.4 Improving the Chain Method via Fractional Cascading

Edelsbrunner, Guibas, and Stolfi [36] propose an improvement to the chain method using a technique called *fractional cascading*. This technique is applicable in any general situation where there are repeated similar searches along the nodes of a path in a directed graph in which the degree of each

node is bounded by constant and the set of items searched in each node are drawn from the same universe. Under these conditions one can do better than performing several independent binary searches. The list stored in each node of the graph is augmented with extra elements from the lists in the successors so as to correlate the searches in a node and its successors.

We now describe the fractional cascading technique and show how to improve the chain method. In chain method, a sequence ($O(\log n)$) of point-versus-monotone chain discriminations are performed. In each such discrimination, a search is performed using the same query point, but against different chains. Moreover, the set of y-coordinates of the vertices of the edges against with the comparison is made is a fixed set i.e., the y-coordinates of the vertices of the subdivision. Hence the conditions are favorable to apply the fractional cascading technique.

In order to be concrete let us therefore review in detail how the fractional cascading technique can be applied in this case. Consider a node u in the primary structure with children v and w. Let C_u, C_v, and C_w be the chain lists in the nodes u, v, and w respectively before augmentation. The problem is to compute appropriate augmented lists T_u, T_v, and T_w lists in the nodes u, v, and w respectively. We perform this augmentation bottom up. Assume T_v and T_w are already constructed. For an appropriate constant d ($d = 4$ works, for example), we select every d-th element from T_v and T_w, respectively, and copy the elements to the list C_u to form the augmented list T_u. These copied elements are referred to as *bridge* elements in T_u. Moreover each element i in T_u stores additional pointers as follows:

Left Bridge (P_l): Pointer to the closest bridge element (not to the left of i) copied from left child.

Right Bridge (P_r): Pointer to the closest bridge element (not to the left of i) copied from right child.

Proper (P_p): Pointer to the closest element from C_u not to the left of i.

Predecessor: Pointer to the predecessor element in T_u.

Bridge (P_b): Pointer to the corresponding element in T_v or T_w (only for elements copied into C_u).

The distance between two bridge elements in T_u is at most d. To perform the search, we first do a binary search using the the given query point in the list stored in the root to identify which child to search. Suppose we
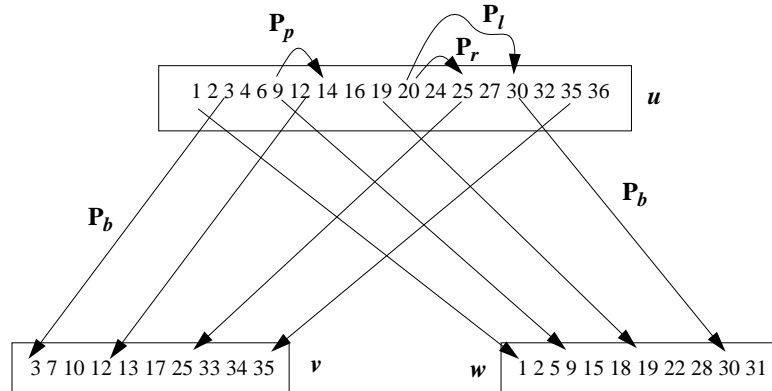
Figure 1.9: Fractional Cascading: Augmentation of list in a parent node with elements from the lists in two children. The *left bridge* and *right bridge* pointers for element 20 are shown. Similarly the *proper* pointer for element 9 is shown. Also the *bridge* pointers are shown.

need to search the left child. We then follow the *Left Bridge* pointer of the successor element of the query point in the T list of root. From the *Left Bridge* we follow the *Bridge* pointer to the T list of the left child. We then use the *Predecessor* pointers to find the two elements in the T list of left child which encompass the given query point. We select the successor and then use the *Proper* pointer to identify the element in the C list so as to perform comparisons for branching to next level.

The figure 1.9 shows an example where the list at a parent node is augmented with elements from the lists in two children. Every fourth element is copied from the child to the parent. For sake of exposition, we use lists of integers.

The total number of pointers traversed for crossing one level is $d + 3$ pointers i.e., one *Left Branch* pointer, one *Bridge* pointer, at most d *Predecessor* pointers, and one *Proper* pointer. Initial search takes $O(\log n)$ time and the subsequence searches together take $O(d \log n)$ time . If d is chosen as constant, then the total search time is $O(\log n)$.

Summary:

Space: Edelsbrunner, Guibas, and Stolfi [36] show the total space requirement is $O(n)$ for an appropriate constant value of d.

Query Time: $O(\log n)$, as argued above.

The preprocessing time is $O(n \log n)$.

Subdivision Hierarchies

Historically, the method of Edelsbrunner, Guibas, and Stolfi is not the first
to simultaneously achieve $O(n)$ space and an $O(\log n)$ query time. Kirk-
patrick [44] discovered earlier an elegant method, based upon a technique
we call the *subdivision hierarchy* method, for performing point location. This
method is also applicable for searching in higher-dimensional structures and
is even amenable to parallelization [23, 24, 25].

The method requires that the subdivision is triangulated and also that
the outer face is a triangle. The triangulation, however, can be done in linear
time using Chazelle's method [14] if the original subdivision is connected.
The method then proceeds as follows:

1. Identify a *maximal* independent set in the PSLG representing the sub-
   division using a greedy heuristic with the condition that the degree of
   vertices in the independent set is bounded by a constant c. Also the
   independent set should not include any vertices of the outer face.

2. Remove the vertices of the independent set and the edges attached
   to them. Retriangulate each of the *star* polygons which contain the
   vertices of the independent set.

3. Repeat the process until only the outer face remains.

The figure 1.10 shows the process of removal of independent set of vertices
and retriangulation for an example subdivision.

Now the point location data structure is constructed as follows: We have
a layered directed acylic graph in which one layer represents an intermediate
triangulation of the subdivision in the above recursive process. We assign
one node of the dag to each triangle. After removing the independent set
and retriangulating, some of the triangles are destroyed and new triangles
are introduced inside each star polygon. We introduce a node for each new
triangle and introduce pointers from the node representing the new triangle
to all the nodes representing the triangles which are destroyed within the star
polygon. In the limiting case we have one node representing the outer face
and three nodes representing the three triangles destroyed in the previous
step. The resulting data structure thus is a *layered dag* as shown in the
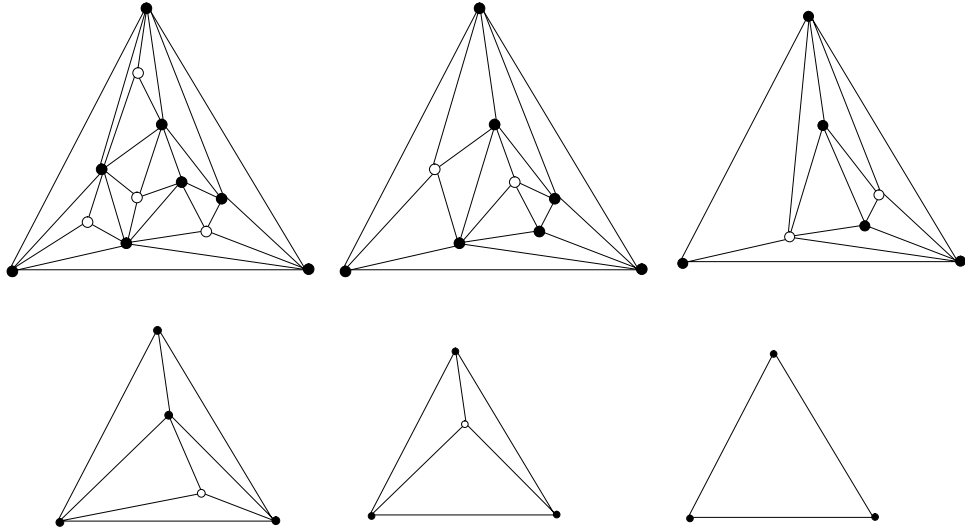figure 1.11.

Kirkpatrick shows that in the layered dag:

Figure 1.10: A Subdivision Hierarchy: The figure shows removal of independent set and retriangulation. The indepedent set vertices selected at a step are shown as hollow circles.
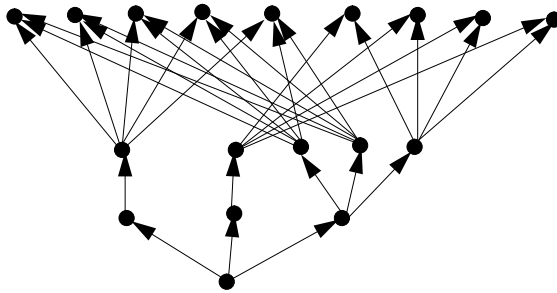

Figure 1.11: A Subdivision Hierarchy: Organizing triangles in a *dag* for searching. The last three levels of a hierarchy are shown.

- The degree of each node is constant, and

- A fraction of the vertices are removed at each step. Hence the total depth of the dag is $O(\log n)$.

This follows from a theorem in graph theory which states that there are "large" independent sets of constant degree vertices in a planar graph, which allows one to find maximal independent sets of size which is a fraction of the current number of vertices.

The point location algorithm proceeds by first locating the point inside the outer triangle. At any step in the search algorithm one has located the query point in a triangle t on some level in the dag. One then follows the pointers in the dag to search all the triangles in the next level that were destroyed to form t. The query point can be located within the unique triangle from among these candidates in $O(1)$ time, and this defines the invariant for the next level. Since there are only $O(\log n)$ levels in the *dag*, the point location takes $O(\log n)$ time.

Summary:

Space: The space occupied by the *dag* is $O(n)$.

Query Time: $O(\log n)$ as argued above.

The preprocessing time is $O(n)$ given the triangulated subdivision, as it takes only constant time to retriangulate each star polygon. Indeed, using Chazelle's triangulation method [14], the preprocessing for Kirkpatrick's method can be implemented in linear time any time the original subdivision is connected.

The Sweep Method and Persistence

There is actually one more well-known method for achieving $O(\log n)$-time queries and $O(n)$ space for planar point location. In particular, Sarnak and Tarjan [62] use the idea of *persistence* to build such a space-efficient point location data structure. Intuitively, they combine the techniques of the slab method, plane sweeping, and persistence to build a very elegant point location data structure. A similar method was also discovered by Cole [22].

A *persistent* data structure allows one to perform updates and queries. Updates must be performed on the most recent version of the data structure (in so-called *partial* persistence), but the queries can be done on past

versions, i.e., any previous version of the data structure that was modified
by updates.

Sarnak and Tarjan [62] modified the slab method as follows: given a
subdivision construct slabs by dropping vertical lines through each of the
vertices. Now perform a plane sweeping from the left most slab to the right
most. The event points for the sweep are the vertices of the subdivision. A
search structure is built for the edges within each slab which is maintained
during the sweep. The structure changes at each event point with the inser-
tion and deletion of edges. Sarnak and Tarjan [62] interpreted these changes
at the event points as persistent updates. As a result they maintain a single
persistent structure during the sweep which is updated at every event point.

This persistent structure need be only partially persistent as updates oc-
cur only on the latest version during the sweep. The point location query is
then performed as follows: using the x-coordinate of the query point the ap-
propriate version of the persistent structure (slab) is identified for searching.
Then a persistent search is performed on the corresponding "past" version
to identify the face. By implementing the sweep using a partially-persistent
red-black tree, Sarnak and Tarjan show how to construct an $O(n)$ space
data structure that allows $O(\log n)$ time persistent searches and updates.
Moreover, they show that each update adds only $O(1)$ amortized-space to
the data structure.

Summary:

Space: The space occupied by the persistent structure is $O(n)$, since there
are $O(n)$ updates and each requires $O(1)$ amortized space.

Query Time: $O(\log n)$, as mentioned above.

1.4   Dynamic Point Location

An important variant of the point location problem is to allow for environ-
ments that request incremental updates to the subdivision. In this variant
one studies how to best reflect the changes to the subdivision (e.g., deletion
and insertion of edges or vertices) in the data structure storing the subdivi-
sion. The goal, of course, is to continue to efficiently support point location
queries while also also performing updates in an efficient manner. In addi-
tion, the space required by the data structure should be kept as small as
possible. In this section we review some dynamic data structures used for
performing dynamic planar point location, where edges are allowed to be

inserted or deleted from a subdivision. The goal here is to efficiently maintain the data structure under the update operations and allow for fast point location queries.

The basic problems in performing updates are the following:

1. Propagation of new information or modification of old information efficiently. Suppose one is deleting an edge. If the edge is represented in several nodes of the data structure, then one needs to remove that information from all the nodes. For example in the slab method, if an edge spans multiple slabs, then one needs to remove that information from each of the binary trees storing that edge. A reverse problem occurs when such a long edge is inserted into the subdivision.

2. Restructuring of the data structure to maintain the invariants assumed by the algorithm. For example in the trapezoid method, we have the invariant that each node represents a trapezoid and adjacent nodes are separated by a "spanning edge". Suppose a spanning edge is deleted. Then the two adjacent trapezoids (which are represented as two different nodes) must now be collapsed into a single trapezoid and represented by a single node. Similarly if a new spanning edge is inserted, then the corresponding node must be split into two different nodes.

3. Dynamization of the techniques (fractional cascading, persistence, etc.) used to achieve good space bounds.

Preparata and Tamassia [59] present a method for dynamic point location in a convex subdivision. They construct the data structure by dynamizing the structure used in the static trapezoid method. They put a restriction that the vertices lie on a fixed set of N horizontal lines, however. The data structure in this case uses space $O(N \log N)$. Still, they achieve an impressive time of $O(\log n + \log N)$ for queries and an update time of $O(\log n \log N)$. The data structure consists of a primary component that is any balanced binary search tree and the secondary component that is a biased binary search trees [4].

Preparata and Tamassia [58] also dynamized the chain-method for performing point location on a monotone subdivision. They construct a data structure that allows for insertion and deletion of vertices, insertion and deletion of monotone chain of edges, and point location queries. The update operation is required to leave the subdivision monotone. They achieve an $O(\log^2 n)$ query time, $O(\log n)$ time for inserting or deleting a vertex, and

$O(\log^2 n + k)$ time for inserting or deleting a monotone chain of k edges. The space requirement is $O(n)$.

Chiang and Tamassia [21] later improved the above results for dynamic point location in a monotone subdivision. They further dynamized the trapezoid method to achieve $O(\log n)$ time for queries and $O(\log^2 n)$ time for updates. The space required for the data structure is $O(n \log n)$. Their data structure is a compound structure with the primary component being a BB[α] tree [10, 48, 51] and the secondary component being biased binary search trees [4].

Mehlhorn and Näher [49] dynamize fractional cascading to support insertions and deletions in $O(\log \log n)$ amortized time and queries in $O(\log n + k \log \log n)$, where k is the length of the path traversed. Hence their dynamization adds a $O(\log \log n)$ overhead to the static method. Dietz and Raman [27] improve the update time from amortized to worst-case.

Cheng and Janardan [18] present methods for dynamic point location for any connected subdivision. They have two schemes. In one scheme, they achieve an $O(\log^2 n)$ query time, $O(\log n)$ time for insertion and deletion of vertices, and $O(k \log(n + k))$ and $O(k \log n)$ times for insertion and deletion, respectively, of an arbitrary k-edge chain inside a region. The space requirement is $O(n)$. In their other scheme they speedup the insertion and deletion of k-edge monotone chain to $O(\log^2 n \log \log n + k)$, but increase the other bounds slightly. Their general approach is based on the a new search strategy using priority search trees [47] taken together with the technique of *dynamization* as proposed by Willard and Lueker [46, 65] and Overmars [53]. The main idea of this technique is that rather than updating the data structure immediately with each update request they perform only local updates and spread the restructuring over a sequence of future operations. They perform global rebuilding of the entire data structure periodically so that the structure does not go out balance too much. They also make use of BB[α] trees.

Goodrich and Tamassia [38] present a method for dynamic method for point location in monotone subdivisions. They improve the update times by paying a penalty on the query time. In particular, they achieve $O(\log n)$ time for insertion and deletion of vertices, $O(\log n + k)$ for the insertion and deletion of a monotone chain of k edges, and $O(\log^2 n)$ time for queries. The space requirement is $O(n)$. Their data structure consists of two inter-laced spanning trees, each of which is represented using link-cut trees [64]. In order to be concrete about one dynamic point location method, we briefly review their method.

1.4.1 The Inter-laced Trees Technique

Let S be a PSLG that is connected and monotone (and will remain that way throughout the update process). Goodrich and Tamassia [38] first construct a spanning tree T for the triangulation with the property that its root-to-leaf paths are monotone with respect to the y-axis. Then they construct a graph-theoritic dual of the triangulation such that it excludes the edges dual to the monotone spanning tree constructed above. This defines the spanning tree D for the dual graph of S, and these two trees "inter-lace."

Each node of D represents a triangle of S and each edge of D corresponds to non-tree edge of S, with respect to T (since edges dual to edges of T are ignored while constructing D), and hence determines a unique cycle in S. Moreover this cycle partitions S into two regions, one inside the cycle and the other outside the cycle. This property allows one to perform searching the subdivision where we need to do point-versus-cycle discrimination.

The main idea behind point location is as follows: since each edge of D represents a cycle of S, it partitions S into two regions. Given a query point, we can compare it against the edges of the cycle to determine whether it is inside or outside the cycle in $O(\log n)$ time, since T is monotone. Depending on this test, we proceed to an appropriate edge of D which is either inside or outside the cycle of S for further discrimination.

To perform such a search efficiently, one needs to balance the tree D. The authors use the link-cut tree data structure [64] to be able to implement a recursive *centroid* search [13, 40] in D to eliminate a constant fraction of the triangular faces with each cycle test. This allows one to perform searches in $O(\log^2 n)$ time since the depth of the centroid decomposition tree is $O(\log n)$ and in each step one needs to perform an $O(\log n)$ -time point-versus-chain discrimination.

Summary:

Space: The space occupied by the structure is $O(n)$ (again from fractional cascading method).

Query Time: $O(\log^2 n)$, as mentioned above.

Update Time: The authors show how to use link-cut tree primitives to implement updates in $O(\log n)$ time.

1.5   Convex Hulls and Convex Polytopes

Convex hulls and convex polytopes are fundamental geometric data structures and have been well studied. In this section we discuss the data structure representations of convex hulls and polytopes and also how they are maintained dynamically when the points are inserted and deleted.

1.5.1   d-Dimensional Representations

Given a collection of points in d-dimension, where d is a fixed constant, there are algorithms (both randomized and deterministic) for computing the convex hull of the points in this collection. Alternately, some of the algorithms exploit the duality between a point and a hyperplane in d dimensions, and compute the intersection of a collection of halfspaces determined by the origin and a set of hyperplanes, which by this duality directly gives the information about the convex hull of the input set of (primal) points.

A convex polytope is represented by the information about faces, edges, and vertices and the relationship between them. Each face of the convex polytope is a convex set. The $(d-1)$-dimensional faces of a d-dimensional polytope are called *facets*, its $(d-2)$- and lower dimensional faces are called *subfacets*, its one-dimensional faces are edges, and its zero-dimensional faces are vertices. In a convex polytope arises from a convex hull computation, then each vertex is a point in the input set.

A d-dimensional convex polytope is represented generally using an *incidence graph* [35]. Dobkin and Laszlo [30] define such a representation for 3-dimensional convex polytopes and Brisson [11] extends this to d-dimensional convex polytopes, for fixed $d \geq 2$. In addition to the above defitions, we refer to the $(d-2)$-dimensional face as a *ridge*. In the incidence graph, for $-1 \leq k \leq d-1$, a k-face f and a $(k+1)$-face g are incident upon each other if f belongs to the boundary of g. In this case, f is called the subface of g and g is called the super face of f.

1.5.2   2-dimensional Dynamic Maintenance

Let us now consider the problem of maintaining a convex hull of points in the plane when the underlying point set changes. The *online* problem, where the points are only allowed to be inserted, is easier than the dynamic problem, where one allows deletion of points as well. In the online case the convex hull can only expand in area. If a new point is determined to fall inside the

existing convex hull, then one does not need to do any additional work. But if the point falls outside, then one must compute the tangets from the new point to the current convex hull. These tangents are added to the convex hull and the chain of points on the convex hull between the two tangents are deleted. This can be done in amortized $O(\log n)$ time, where n is the number of points on the convex hull, as shown by Preparata [55].

When the deletion of points is allowed things get more complicated, since one needs to maintain convexity information about points that are not currently on the convex hull. Overmars and van Leeuwen [54] present an elegant solution to this problem. They maintain the convex hull as an union of two monotone chains—the upper and lower hulls—partitioned at the point with largest and smallest x-coordinate, respectively. Each hull is then maintained as a compound tree structure, where each internal node stores the convex hull of the points in the subtree and the parent node adds the supporting tangent to the convex hull stored at its two children to maintain the convex hull of all the points in its subtree. The insert and delete operations work to modify these lists at the nodes appropriately. They show that the update operations take $O(\log^2 n)$ time (where n is the current number of points in the set), while the query operation of asking for the current convex hull involvex just reading the list from the root of the tree. In addition, they show that one can still perform tangent queries in $O(\log n)$ time.

1.5.3   3-dimensional Subdivision Hierarchies

Representing 3-dimensional convex polytopes is considerably harder. We know of no efficient dynamic schemes, for example. Still, Dobkin and Kirkpatrick [28, 29] present an beautiful static data structure for representing 3-dimensional convex polyhedra so as to answer tangent and intersection queries quickly. Their structure is based upon the subdivision hierarchies technique introduced earlier (in Section 1.3.4). They form a hierarchy by first identifying a relatively large independent set of vertices of at most constant degree (viewing the edges of the polyedron as a graph). They then remove these vertices and form the convex hull of those that remain, while forming pointers between the new facets formed and the vertex in the previous level that was deleted to give rise to these new facets. They then recursively repeat this process, terminating this construction when the polyhedron has constant size. Interestingly, they show that this simple approach can be used to answer a number of types of tangent and intersection quer-

ies on the original polyhedron in $O(\log n)$ time, where n is the number of vertices.

1.6   Rectilinear Data Structures

Having reviewed some data structures for maintaining convexity inform-ation, let us now consider the organization of geometric objects so as to enable "rectilinear" types of searching. In particular, we briefly review in this section methods that partition and query the space occupied by the underlying geometric objects using axis-parellel hyperplanes. For a more-complete description of these techniques, the reader is referred to the chapter by Nievergelt and Widmayer **??**.

1.6.1 k-D Trees and Quad Trees

First we consider two rectilinear data structures namely, the k-D tree [5, 8, 7] and the quad-tree [60, 61]. We now briefly describe the two types of partitions used to build these structures. We discuss the partitioning for point sets in d-dimensions. The method can be extended for other types of geometric objects in a straightforward way.

In the case of the k-D tree[2], we first compute the median of the point set in one of the dimensions, say D_0 and paritition the point set into two sets based on the median point, i.e., all points having coordinates less than the median point along D_0 are placed in one set and the remaining points are placed in the other set. This process is then recursively continued along the other dimensions in the two resulting regions. Once the partitioning is completed along all dimensions, it is repeated starting from D_0 in the resulting regions. If the number of points in a particular region falls below a certain constant, the process is terminated for that region. These regions with boundaries parallel to the axes are organized in the form of tree with partitioning of a region into two smaller regions along an axis representing the parent-child relationship.

In the case of the quad-tree, the bounding box of the point set is par-titioned into 2^d regions by using axis-parellel hyperplanes passing through the mid point of each of the sides of the bounding box. The partitioning is continued recursively in each of the resulting regions until the number of

---

[2]The phrase means k-dimensional or multidimensional binary search tree, but we use d to denote the dimension to be consistent with other sections.

points falls below a certain constant. These regions are then organized in the form of a multi-ary tree.

The k-D tree and quad tree occupy linear space and the performance of the search operations depends on the application, but is in general not optimal.

The search algorithms proceed by intersecting the search volume with the bounding box of the region at the root node and recursively searching the regions in the children nodes whose bounding boxes intersect the search volume.

1.6.2   Segment Trees

Given a set of n segments in the plane, a segment tree [6] allows for efficient storage and searching operations on the underlying set. The x-coordinates of the segment endpoints are projected onto the real line so as to partition the line into several intervals (if the endpoints are in general position, there will be $2n + 1$ intervals). These intervals are then organized in the form of a tree structure. The intervals represent the leaf nodes of the tree. Each internal node represents an interval that is the union of all intervals in the leaf nodes in the subtree. We store a "cover list" of segments at each internal node (typically sorted by the "above" relationship). Formally we say a segment covers a node u if it spans interval at u and does not span the interval at the parent of u. One can show that a segment is stored in the cover lists of at most two nodes in each level and also in at most $O(\log n)$ different nodes, where n is the number of segments. A query operation, such as finding all the segments stabbed by a vertical query ray, can be answered by searching the cover list at each level and proceeding down the tree. The segment tree requires $O(n \log n)$ space and the vertical ray-intersection query can be answered in $O(\log n + k)$ time, where k is the output size.

1.6.3   Range Trees

Range trees allow for efficient storage of point sets for rectangle range searching. Given a set of n points in plane, for example, the 2-dimensional range tree is constructed as a compound tree structure. The primary tree structure is constructed as a balanced binary search tree structure on the x-coordinates of the points. Each node in the primary tree structure represents an interval in the x-axis. We associate with each internal node all the points within its interval and organize those points in the form of a search tree, but ordered

using their y-coordinates. To perform a range search, one first searches the primary tree and locates the two intervals in the leaf nodes containing the bounding x-values of the query range. Then we walk up the tree to the least common ancestor of the two leaves and along the path we search the secondary structure stored in the nodes which are siblings (nodes not on the path) for the points which are within the range in the y-axis.

The space occupied by the range tree is $O(n \log n)$ since there $O(\log n)$ levels each containing $O(n)$ nodes. The complexity of range search is $O(\log^2 n + k)$, where k is the output size. This can be improved to $O(\log n + k)$ by using techniques like fractional cascading.

1.7   General Techniques

In this section we briefly outline the some of the transformation and construction techniques used for improving the performance of the searches and updates on a data structure.

1.7.1   Fractional Cascading

As mentioned earlier, fractional cascading [16, 17] is a very powerful data structure transformation technique that can improve the query performance of the data structures. We outlined the method in detail in Section 1.3.4. Given a graph-based data structure consisting of nodes of bounded degree in which search operations proceed along a path in the graph and compare the information stored in each node of the path with the "same" key, one can improve the performance by augmenting the information stored in each node. This will eliminate the need for independent searches in each node and will make the searches dependent. One important restriction is that all the information stored must be from the same universal set.

1.7.2   Persistence

The data structures we study normally are ephemeral in nature, i.e., once the updates are done on the structures the previous information is lost. The persistent data structure maintains information about the past versions. This allows one to perform queries in the past. Such structures are called *partially persistent structures.* Sometimes one would like to allow for updates in the past versions. This becomes quite complicated as an update to one of the past version creates a new chain of data structures. We refer the

reader to an excellent paper by Driscoll *et. al.* [33] for complete details for how to make structures persistent. Such structures are referred to as *fully persistent structures.* As showed in Section 1.3.4, the persistent structures can maintain the information during a plane sweep in a simple way which provides for efficient planar point location algorithm.

1.7.3 Static to Dynamic Conversions

Bentley and Saxe [63, 9] propose general techniques for coverting static data structures to be dynamic. They considered a class of problems called *decomposable searching* problems and presented general techniques for converting static data structures to dynamic. The decomposable searching problems have the property that one can decompose a query about the complete set of objects into queries involving subset of objects and combining the results in a certain way to obtain the solution for the original query. These have applications to problems like membership querying, nearest neighbor querying, fathest point querying, and intersection querying.

The transformations to online structures (i.e., ones that allow only insertions) is to maintain a collection of static structures of appropriate size and merge them to build large structures periodically. The size and time at which new structures are built are determined typically based on geometric progressions, such as powers-of-twos or a Fibonacci series. When deletions are allowed they advocate the use of a shadow structure where the deleted elements are maintained. One can then answer queries by searching both the shadow and the actual structures.

Overmars [53] introduced another class called *order decomposable set* problems and he presented general techniques for dynamizations. This is a generalization of the method of Overmars and van Leeuwen [54] for dynamic maintenance of convex hull of planar point set. This technique has applications in maintenance of the contour of maximal elements of a two-dimensional point set, maintenance of the intersection of a set of halfspaces in the plane, etc.

1.7.4 Internal-Memory to External-Memory Conversions

When the input data to a given problem is huge, one would like to design algorithms that optimize the number of I/O accesses. Because of the order of magnitude difference in the access times between disks and internal memory, algorithms dealing with large inputs should pay more attention

to the organization of the underlying data structure so as to minimize the number of disk accesses. The main task in external memory organization of a data structure involves determining which substructures share the same block or page in the disk. The blocking of the nodes of the internal memory data structure is very crucial and different blocking schemes lead to different space requirement and different I/O performance of the query algorithms.

Goodrich *et. al.* [39] present external memory techniques for solving computational geometry problems dealing with large inputs. They presented four general techniques and showed how they can be applied to obtain efficient external memory algorithms for problems, like computing the pairwise intersection of orthogonal segments, constructing the 2-d and 3-d convex hull of points, answering batched range queries on points, point location queries on the planar subdivision, finding all nearest neighbors, etc.

There are several other works related to external memory computational geometry. We refer the reader to some of the recent papers on this topic [1, 12, 19, 52, 20]. In addition, we highlight a recent paper by Arge [2], where he introduces a general technique, based on a data structure he calls the *buffer tree*, for converting certain types of internal memory computations into efficient external memory computations.

Bibliography

[1] L. Arge. External-storage data structures for plane-sweep algorithms. Technical Report RS-94-16, BRICS, Aarhus Univ., Denmark, 1994.

[2] L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Workshop Algorithms Data Struct.*, number 955 in Lect. Notes in Comp. Science, pages 334–345, 1995.

[3] B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596, 1975.

[4] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14:545–568, 1985.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[6] J. L. Bentley. Solutions to Klee's rectangle problems. Report ??, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.

[7] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.*, SE-5:333–340, 1979.

[8] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[9] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

[10] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[11] E. Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete Comput. Geom.*, 9:387–426, 1993.

[12] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 1995.

[13] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.

[14] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[15] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. In *Proc. 18th Internat. Colloq. Automata Lang. Program.*, volume 510 of *Lecture Notes in Computer Science*, pages 661–673. Springer-Verlag, 1991.

[16] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[17] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[18] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992.

[19] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes in Computer Science*, pages 346–357. Springer-Verlag, 1995.

[20] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 139–149, 1995.

[21] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Internat. J. Comput. Geom. Appl.*, 2(3):311–333, 1992.

[22] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.

[23] N. Dadoun and D. G. Kirkpatrick. Parallel processing for efficient subdivision search. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 205–214, 1987.

[24] N. Dadoun and D. G. Kirkpatrick. Cooperative subdivision search algorithms with applications. In *Proc. 27th Allerton Conf. Commun. Control Comput.*, pages 538–547, 1989.

[25] N. Dadoun and D. G. Kirkpatrick. Parallel construction of subdivision hierarchies. *J. Comput. Syst. Sci.*, 39:153–165, 1989.

[26] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1993.

[27] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symp. Discrete Algorithms*, pages 78–88, 1991.

[28] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[29] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes in Computer Science*, pages 400–413. Springer-Verlag, 1990.

[30] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[31] D. P. Dobkin and R. J. Lipton. The complexity of searching lines in the plane. Report ??, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1976.

[32] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[33] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[34] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[35] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[36] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[37] I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged.*, 11:229–233, 1948.

[38] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[39] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS 93)*, pages 714–723, 1993.

[40] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[41] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[42] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1972.

[43] J. Hershberger and S. Suri. Offline maintenance of planar configurations. In *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, pages 32–41, 1991.

[44] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[45] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[46] G. S. Lueker and D. E. Willard. A data structure for dynamic range searching. *Inform. Process. Lett.*, 15:209–213, 1982.

[47] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.

[48] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Heidelberg, West Germany, 1984.

[49] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[50] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[51] J. Nievergelt and E. Reingold. Binary search trees of bounded balanced. *SIAM J. Computing*, 2:33–43, 1973.

[52] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proc. 12th Annu. ACM Sympos. Principles Database Syst. (PODS '93)*, pages 222–232, 1993.

[53] M. H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms*, 2:245–260, 1981. Corrigendum in 4(1983), 301.

[54] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[55] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Commun. ACM*, 22:402–405, 1979.

[56] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[57] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[58] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830, 1989.

[59] F. P. Preparata and R. Tamassia. Dynamic planar point location with optimal query time. *Theoret. Comput. Sci.*, 74:95–114, 1990.

[60] H. Samet. *Applications of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

[61] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[62] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[63] J. B. Saxe and J. L. Bentley. Transforming static data structures to dynamic structures. In *Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 148–168, 1979.

[64] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.

[65] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985.