Topology Building and Random Polygon Generation

Chong Zhu

Technical Report 94-12
April 1994

Department of Computer Science
The University of British Columbia
Vancouver, B. C. V6T 1Z4
Canada

Abstract

In the island adoption problem from geographical information system we are asked to identify which islands are located in which lakes. This problem translates directly to polygon nesting in computational geometry: given a set of polygons, find their nesting structure. We present our research into a broader nesting problem, namely connected component nesting, beginning with the underlying concept of topology-building and a related issue of random polygon generation.

Topology building is a process of structuring data. We develop a plane sweep algorithm for building a quad-edge data structure that captures the topological structure of connected components of a set of line segments. The algorithm starts with a data structure representing a single edge then adds edges into the data structure at each step while sweeping across the connected components The algorithm's time complexity is determined by the time to sort the vertices of the line segments.

We develop two approaches for obtaining the nesting structure of polygons. The first adopts a basic idea of Bajaj and Dey [1], but introduces a new notch definition to simplify their algorithm. The second generalizes the nesting problem to a broader class including the nesting of connected components. We present a sweep algorithm, based on a union-find data structure, that computes the nesting of the connected components.

In order to test and verify the time complexity of our polygon nesting algorithm, we present an algorithm that generates x-monotone polygons uniformly at random over a vertex set of n points. This algorithm scans the point set to calculate the total number of monotone polygons that can be created, then reverses the scan to generate a random monotone polygon. This process generates a random polygon over the n vertices in $\mathsf{O}(K)$ time, where $n \leq K \leq n^2$ is the number edges of the visibility graph of the x-monotone chain whose vertices are the given n points. The space complexity of our algorithm is $\mathsf{O}(n)$.

Table of Contents

List of Tables

List of Figures

Acknowledgment

I sincerely thank my supervisor Dr. Jack Snoeyink. I am very grateful for the supervision Dr. Snoeyink gave me over the whole period of my Msc. program including taking courses, solving the proposed problem theoretically, doing experiments and proofreading this thesis. I thank him for being a exemplary researcher and a considerate mentor. I also thank him for the constant encouragement I received from him.

I thank Mr. Dan Lemkow of Essential Planning Systems for introducing the polygon nesting topic and supplying the real data to me.

I thank my wife, Dr. Ying Li, for her love and support.

Chapter 1

Introduction

In this thesis, we address three parts of our research work. In the first part, we survey the quad-edge data structure and propose an algorithm to build this data structure for capturing the topological structure of the connected components. In the second part, we study the nesting structure of both simple polygons and connected components, and developed algorithms to solve the nesting problems of both types of objects. In the final part, we give an algorithm to generate x-monotone polygon uniformly at random. In the rest of this chapter the motivation of the research work and the background of these problems are detailed.

1.1 Topology Building

Geometric algorithms involve the manipulation of objects that are not handled at the machine language level. We must therefore organize these complex objects by means of simpler data types directly representable by the computer. These organizations are universally referred to as data structures. The most important property of data structures is that they capture *structural relationships* between the data elements. When the data elements are organized in a data structure, we then have *structured data*. To retrieve structural information from structured data becomes much easier than from unstructured data. In the simplest form, a set of one-dimensional points, unstructured data is simply an amorphous mass of numerical values. By assigning an ordering structure on the point set, we are able to answer such questions as: which element is the smallest one? which is the largest one? which elements precede and follow a given one? etc. in an efficient way.

In more complicated situations, the data might be two-dimensional or higher. In order to capture the structural information of the data, the data elements should be organized appropriately, such as nesting structure among the simple polygons. Many applications require new algorithms to efficiently represent and manipulate the structural information presented within data. One such example, drawn from the geographical information system (GIS) domain, is to retrieve a collection of areas with a common property from a larger and more complex set with many properties.

Two-dimension structuring of data is called **topology building** in GIS. After structuring the input data, the traditional topological structures such as connected components, polygons, rings, and chains, can be retrieved through pointer lists. Topology building is an important technique in GIS, computational geometry, and computer graphics. By knowing the topological structure of the data one can perform various operations efficiently on the data, such as area intersection, union, difference, clipping, and parentage tracking in overlay process in GIS [16].

To capture the topological structure of connected components, we develop a sweep-based algorithm that builds a quad edge data structure of the components. Because the quad edge data structure [7] captures all the topological properties of vertices, edges, and faces of two-dimension graphs, this data structure is the ideal candidate for an intermediate data structure in topology building. The quad edge data structure of a set of line segments can be built by two simple operations from the local relations of the line segments: using only two primitives avoids subtle bugs due to pointer errors. A single topological operator, called `Splice`, and one primitive for the creation of isolated edges, called `Makeedge`, is sufficient to construct and modify this data structure.

The plane sweep method [14] is a widely used computational geometry paradigm that has been applied to solve many different problems involving spatial data [10, 16]. We have adopted this method in our algorithms for building quad edge data structure, extracting the nesting structure among simple polygons and connected components.

1.2 Nesting Structures

Typical GIS problems include: How can one retrieve a single area within a certain connected component? Here a *connected component* is a undirected planar graph, embedded in the Euclidean plane, in which there is a path that joins any two vertices. What is the parentage relation among these connected components? The parentage relation problem is to find the nesting structure among the connected components. The nesting structure problem is called the *topological relation* problem. Work by Egenhofer and Sharma [3] on topological relation problems is based upon knowing the topology structure of the data. Before we can extract the topological relation it is important to know the topological structure of the objects. But in many GIS applications the topological structure of the given data may not be known. Without knowing the topological structure of connected components, to extract the nesting structure of them will be a more practical and challenging problem.

A *region* of a connected component is the area enclosed by a set of edges of the component. For two nonintersecting connected components *M1* and *M2*, we say that *M1* is *nested* in *M2* if there exists a region R of $M2$ where every region of $M1$ is contained in R. If the connected components are simple nonintersecting polygons, the problem of finding the nesting structure of a set of connected components becomes the lake and islands problem in GIS. In this simplified case, $M2$ has only one region, that is, $R = M2$ and $M1 \subseteq R$.

Bajaj and Dey [1] extracted the nesting structure from a set of simple nonintersecting polygons. They did not discuss how to find the nesting structures from a set of connected components. Applications in GIS, graphics, and medical applications use the nesting structure from more complicated objects. Thus, an algorithm that extracts the nesting structure from a set of connected components becomes more practical than extracting nesting structure from simple polygons.

We develop an algorithm to extract the nesting structure from a set of connected components by a one pass sweep. We sweep a line L in the plane through all the connected components, while maintaining the ordering of edges that induced by L. Meanwhile disjoint region and subregion sets and nesting sets are created or united together according to the sets connection information discovered during the sweep. The final nesting sets capture the nesting structure of the connected components.

1.3 Random Polygon Generation

Interest in generating random geometric objects has increased recently, both in theory [8] and in applications [4, 13]. The generation of random geometric objects has applications which include testing and verifying the time complexity of computational geometry algorithms.

There are two ways to test computational geometry algorithms. The first involves the construction of geometric objects that the implementer considers difficult cases for the algorithm. For example, our polygon-nesting algorithm, based on a plane sweep, may require special case code for some polygons. It is important to make those polygons candidates for exposing errors of the algorithm. The second approach to testing involves executing the algorithm on a large set of geometric objects generated at random. We expect errors to be exposed if enough different valid inputs are applied to the algorithm.

To verify that an implementation of an algorithm achieves the stated algorithm time complexity is the problem we often have in implementation-oriented computational geometry research. This is done by timing the execution of the algorithm for various inputs of different sizes. There are many possible inputs of any given size, and the choice is important, since an algorithm may take more time on some inputs than others of the same size. If an average execution time is computed over a set of randomly generated objects of a given size, the relationship between time and problem size will typically follow a curve corresponding to its complexity. We can then check this complexity against the stated algorithm's complexity.

Some research has been done on generating geometric objects at random, such as Epstein [4]. In order to test and verify the time complexity of our polygon nesting algorithm we give an algorithm to generate monotone polygons uniformly at random.

1.4 Organization

This thesis proceeds as follows:

Chapter 2 describes the polygon nesting problem and the basic idea of the plane sweep process. An algorithm is proposed for extracting the nesting structure from a set of simple nonintersecting polygons.

Chapter 3 describes the basic concepts, the edge functions, and the topological operators of the quad edge data structure. Then an algorithm to build the quad edge data structure of connected components by a one pass sweep is proposed.

Chapter 4 adapts the plane sweep process and also gives an algorithm to extract the nesting structure from a set of connected components.

Chapter 5 describes how to generate random x-monotone polygons. Both a theoretical analysis and an algorithm are given in detail.

The conclusions and directions of future work follow in Chapter 6.

Chapter 2

Polygon Nesting

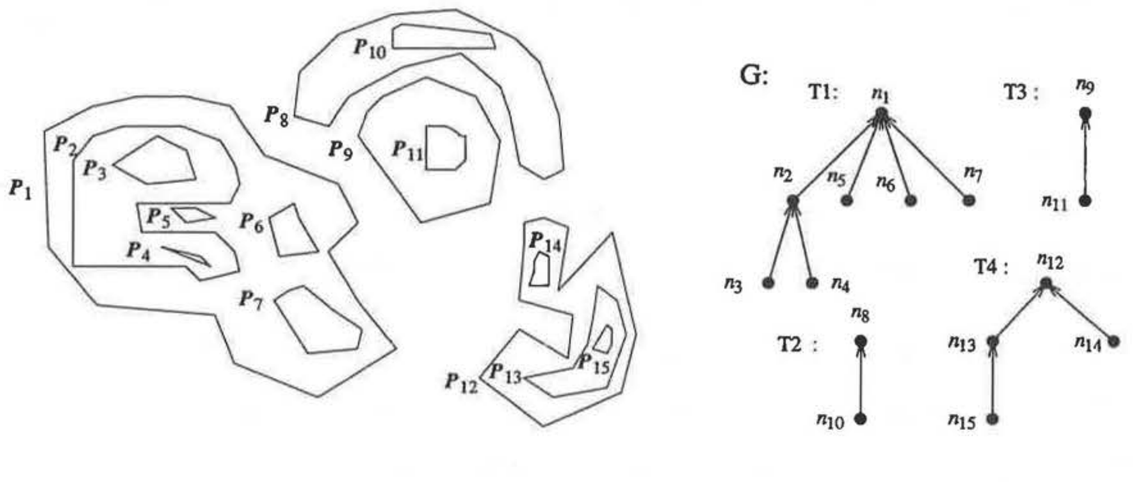
In this chapter we describe the polygon nesting problem, also known as the island adoption problem, and a plane sweep algorithm to extract the nesting structure of simple nonintersecting polygons.

2.1 The Background of the Polygon Nesting

To determine the area of a body of water, one must subtract the area of its islands from the body of water. For this, one must know which island belong to which body of water. Moreover, islands may contain lakes with islands; we would like to know the entire nesting structure of the shoreline polygons that separate water and land. The polygon nesting problem is to determine, for each polygon P in a set of disjoint polygons, which polygon directly contains P.

We have adapted a plane sweep algorithm (see Preparata and Shamos [14]) to compute polygon nesting. A sweep algorithm turns a static two dimension problem into a dynamic one dimension problem, in this case a dynamic interval containment problem. When the sweep encounters a polygon vertex, the intervals in which the sweep line intersects the polygon can be created or destroyed and can be split or merged. Keeping track of which polygons own which intervals allows us to compute nesting information with one pass through the data. In fact, if the data is clean we can compute topology within the same pass at the cost of additional bookkeeping.

The Problem. Let $\mathcal{P}$ be a set of m non-intersecting simple polygons P_i, $i = 1, 2, ..., m$. For each polygon P_i, we define $ancestor(P_i)$ as the set of polygons containing P_i. The polygon P_k in $ancestor(P_i)$ is called the *parent* of P_i if $ancestor(P_k) = ancestor(P_i) - P_k$. If $ancestor(P_i)$ is empty we say that the parent of P_i is *null.* Any polygon whose parent is P_k is called a *child* of P_k; see Figure 2.1. The *nesting structure* G of $\mathcal{P}$ is an acyclic directed graph (a forest) in which there is a node n_i, corresponding to each polygon P_i in $\mathcal{P}$, and there is a directed edge from a node n_i to n_j if and only if P_j is the parent of P_i. The polygon nesting problem is to compute the nesting structure of a set of simple non-intersecting polygons.



(a) A set of simple polygons (b) The answer trees of the polygons

Figure 2.1: The input and the answer of the polygon nesting problem.

In [1] Bajaj and Dey give an algorithm that computes the polygon nesting structure. We adapted their algorithm and introduced our notch definition (see definition 4.1) so that the number of notches is the same as the number of subchains. It reduces the number of subchains induced by Bajaj and Dey's notch definition.

2.2 Extracting Polygon Nesting Structure

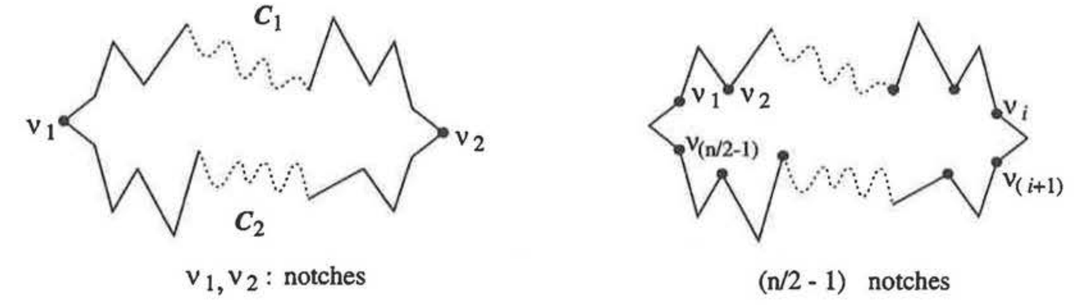
2.2.1 Definitions

Let P be a simple polygon with vertices $\nu_1, \nu_2, ..., \nu_n$ in clockwise order. A vertex ν_i is defined as a **notch** if the two edges that connect with vertex ν_i are either both at the left side of or both at the right side of the vertex ν_i. These left side and right side ordering relations can be defined with respect to the projection on any line, L, but the x coordinate is selected as the axis of projection and the precise definition of **notch** is given as follows.

Definition 2.1 Let $\nu_i \rightarrow x$ be the x-coordinate of vertex ν_i. A vertex ν_i is a **notch** of P if $\nu_{i-1} \rightarrow x \leq \nu_i \rightarrow x$ and $\nu_{i+1} \rightarrow x \leq \nu_i \rightarrow x$ or $\nu_{i-1} \rightarrow x \geq \nu_i \rightarrow x$ and $\nu_{i+1} \rightarrow x \geq \nu_i \rightarrow x$.

Between any two consecutive notches ν_i and ν_j in the clockwise order, the sequence of vertices $(\nu_i, \nu_{i+1}, ..., \nu_j)$ is called a *x-monotone polygonal line*, or *subchain*, of P. Figure 2.2 (a) shows the notches and the monotone polygonal lines in a simple polygon.

Our notch definition is different from that of Bajaj and Dey's. They defined a vertex ν_i to be a notch of P if the inner angle between the edge (ν_{i-1}, ν_i) and (ν_i, ν_{i+1}) is larger than 180°. From this definition the polygonal line between any two consecutive notches is a *convex polygonal line.* Then they partitioned the convex polygonal line into *convex chains* and further partitioned the convex chains into the *x-monotone polygonal lines.* But from our definition we can directly partition a polygon into *x-monotone polygonal lines* that make the definition clearer and reduce the total number of subchains. Figure 2.2 (a) shows that there are only μ_1 and μ_2 two notches, and C_1 and C_2 two subchains induced by our notch definition. But for the same polygon there are $(n/2 - 1)$ notches induced by Bajaj and Dey's notch definition, and all the edges become subchains, which is shown in Figure 2.2 (b).



(a) Two notches by our notch definition (b) Bajaj and Dey's notch definition

Figure 2.2: The comparison of two notch definitions.

From our notch definition, we get the following lemma.

Lemma 2.1 Let P be a simple polygon with N_p notches. The number of *subchains* S_p in P is N_p.

Proof. From Definition 2.1 we know that between any two consecutive notches ν_i and ν_j in the clockwise order, there is a subchain. So the number of *subchains* of P is N_p. $\square$

A vertex or an edge is said to lie inside a polygon if it lies completely inside the polygonal region enclosed by the boundary of the polygon. A vertex or an edge is said to be contained in a polygon if it lies on the boundary of the polygon.

Let L be any line drawn through a set of polygons. Let E be the set of edges that intersect L. The direction of an edge is always from left to right. If an edge e_k is parallel to L, the direction of e_k points up or down, whichever is consistent with edges connecting to e_k.

A vertex ν_i is said to be *above* ν_j if ν_i lies geometrically above ν_j. An edge e_1 is said to be *above* the edge e_2 in E if the point of intersection of L and e_1 lies above the point of intersection of L and e_2. If e_1 and e_2 have a common vertex through which L passes, e_1

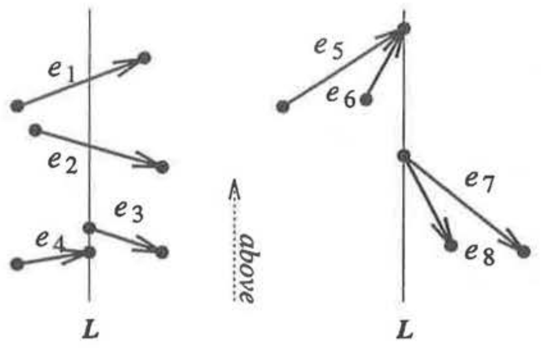


Figure 2.3: The *above* relation of the edges.

is "above" e_2 if e_1 is at the left of e_2 (see edges e_5 with e_6 and e_7 with e_8 in Figure 2.3). L induces a partial order R on the edges in E with respect to the "above" relation. If L passes through a vertex ν_i, we define $above(\nu_i)$ as the set of edges whose point of intersection with L is above ν_i. The lowest edge in $above(\nu_i)$ is called the *neighbor* of ν_i. Order R extends naturally to another order O of *subchains* associated with the edges in R. If edges e_1 and e_2 of subchains C_1 and C_2 intersect L, then C_1 is above C_2 if e_1 is above e_2.

2.2.2 The Plane Sweep Algorithm

A polygon P consists of subchains $C_1, C_2, .., C_k$. We sweep a vertical line L in the plane through all the polygons, while maintaining the ordering O of the subchains C induced by L. To maintain this order we stop only at the endpoints of the subchains, while sweeping from left to right.

First, we break the boundaries of all the polygons into subchains in $O(n)$ time where n is the total number of vertices of all polygons. Let N be the number of subchains. Clearly we have $2m \leq N \leq n$ where m is the number of polygons. We sort only the endpoints of all the N subchains according to their x coordinates. At each end point we update the ordering O and detect the parent of the polygon if required. The algorithm is as follows.

Algorithm

Input: A set of m simple, non-intersecting polygons.

Output: A directed acyclic graph G, called the nesting structure.

Step 1: Detect the endpoints of subchains in all polygons.

Step 2: Sort the x-coordinates of these endpoints in increasing order. If two points have the same x-coordinates, the one with lower y-coordinate is sorted before the other. Let this sorted sequence W be $\nu_1, \nu_2, ..., \nu_w$.

Step 3: Create a node in G for each polygon. Insert the two subchains connected with the leftmost vertex $\nu_1 \in W$ by inserting to the ordering O the two polygon edges connected to vertex ν_1. Then we sweep from left to right, taking steps at each vertex ν_a of W as follows.

Step 4: If vertex ν_a associated with polygon P_i is a left endpoint of a subchain C_1. We can find another subchain C_2 connected to ν_a.

(a) Insert the subchains C_1 and C_2 into ordering O on L by inserting the two polygon edges connected to ν_a in C_1 and C_2 by a simple binary search. The search is based on a procedure for determining the position of ν_a with respect to the edge inserted by L on a subchain C_i, already present in ordering O. Figure 2.4 shows the changes of the ordering O.

For the later purpose, we keep the last visited edge associated with each subchain C_i in O. This can be accomplished by the following simple bookkeeping.

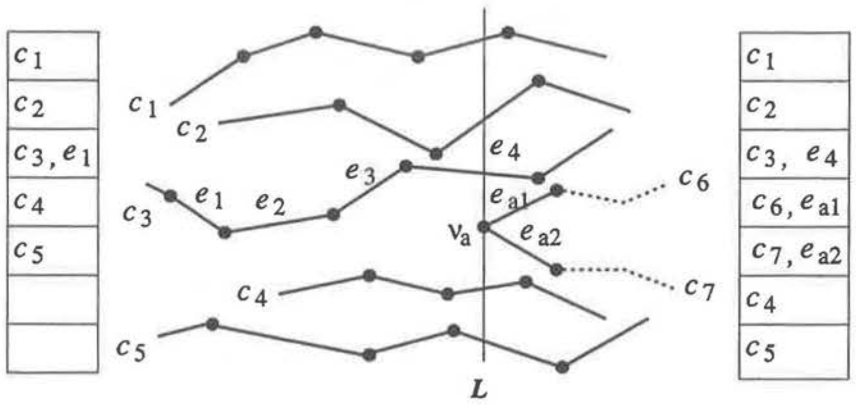


Figure 2.4: Update ordering O at a left endpoint.

Let the edge associated with C_i initially be e_1, the first edge of the subchain C_i. We visit the sequence of edges $e_1, e_2, ..., e_k$ of C_i, stopping at the first edge e_k that intersects L. We determine the position of ν_a with respect to e_k and associate edge e_k with C_i. Later, when we need to classify any other vertex with respect to C_i we start from edge e_k; Figure 2.4 shows the edge changes of C_3. Obviously the edges of C_i are visited only once without updating the ordering O throughout a sweep.

(b) Detect the parent of the polygon associated with ν_a.

Let e be the neighbor edge of ν_a found while inserting C_1 and C_2. Let P_j be the polygon containing e on the boundary. We determine k, the number of edges (or equivalently the number of subchains) of the polygon P_j that are in $above(v_a)$. Maintaining the ordering of subchains of each polygon separately, this number can be obtained in $O(\log S_i)$ time where S_i is the number of subchains in the polygon P_i. If k is odd and $P_j \neq P_i$, we set P_j to be the parent of P_i. Otherwise, we set the parent of P_j to be the parent of P_i. Certainly, parent determination at each update adds up to at most $O(\log S)$, where S is the total number of subchains. From our notch and subchain definitions we know that $S = N$, where N is the total number of notches. Thus, the total time for parent determination is $O(\log N)$.

Step 5: If vertex ν_a associated with polygon P_i is a right endpoint of subchains C_1 and C_2. We delete the subchains C_1 and C_2 from ordering O on L by a simple binary search.

Now we prove the correctness of detecting the parent of a polygon.

Lemma 2.2 Let L be any line passing through vertex ν_a of a polygon P_i. Let the edge e be the neighbor of ν_a. The parent of P_i is either the polygon P_j containing e or P_j's parent (possible *null*).

Proof. If the neighbor edge e of ν_a is an edge of P_j, which is a parent of P_i, the lemma holds trivially. Suppose P_j is not the parent of P_i. We claim that ν_a lies inside polygon P_l if and only if e lies inside it. Suppose e lies inside P_l and ν_a does not. Then the vertical segment between ν_a and e on L contains a part that is outside of P_l. This is impossible since e is the neighbor edge of ν_a. Similarly, we can argue that if ν_a lies inside polygon P_l, so does e. Hence e lies inside the same set of polygons, within which ν_a lies. Hence, if P_k is the parent of P_i it is a parent of P_j and vice versa. □

Lemma 2.3 Let L be any line passing through vertex ν_a of a polygon P_i. Vertex ν_a is contained in the polygon P_k, $k \neq i$ if and only if the number of edges of P_k that are in $above(\nu_a)$ is odd.

Proof. Since any edge demarks the region that is "inside polygon P" and "outside polygon P" on L the lemma is obvious. □

Theorem 2.4 Let L be any line passing through ν_a of P_i. Let edge e of polygon P_k be the neighbor of ν_a on L. If the number of edges of P_k in $above(\nu_a)$ is odd and $k \neq i$, then P_k is the parent of P_i. Otherwise, P_k's parent is the parent of P_i.

Proof. Combine lemmas 2.2 and 2.3. $\square$

Theorem 2.5 The problem of polygon nesting for m polygons can be solved in $O(n + N \log N)$ time where n is the total number of vertices in the polygons and N is the total number of notches of all polygons.

Proof. Detecting the endpoints of the subchains takes $O(n)$ time. Sorting these endpoints requires $O(S \log S)$ time. Updating and determining parent takes $O(n + S \log S)$ time. By Lemma 2.1, S, the total number of subchains, is N where N is the total number of notches. Hence, total time spent is $O(n + N \log N)$. $\square$

Chapter 3

Topology Building

3.1 Introduction

The topology building technique is especially useful in GIS, graphics, and computational geometry. GIS, CAD/CAM, and medical applications contain large amounts of spatial data which users want to analyze and from which they want to extract spatial information. An important criterion, fundamental to most spatial analysis, is the information about the relative spatial location among the objects. For instance, earth scientists monitoring global change want to retrieve all water area in a particular region; airplane designers are interested in all devices within reaching distance from the cockpit; or doctors, analyzing X-rays or tomographies, want to determine which organs are affected by a tumor.

We will focus topology building on vector-based geographical information systems. In GIS, maps are generally represented by separating areas having different properties by boundary arcs, which are made of line segments, e.g. forest, lakes, roads, agricultural areas, and some boundaries are joined together geographically. Joined boundaries form connected components. Retrieving the areas enclosed by the line segments is the basic step for retrieving the geographical information of the areas with the same property.

One may ask the questions as follows: how can one retrieve a single area within a certain connected component? What is the relative spatial location of the given line segments? We solve this problem by constructing the topological structure and building a pointer list for retrieving individual areas.

We assume that no topological information is given in the input data, because most data are supplied as sets of line segments, called edges. The only way for us to know the topological structure of the edges is to display them on a computer screen or draw them on a piece of paper, such as maps. For example, Figure 3.1 shows us the topological structure clearly. But without the help of this visual drawing we could not know anything about the topological structure except a set of edges $\{e_1, e_2, \ldots, e_n\}$.

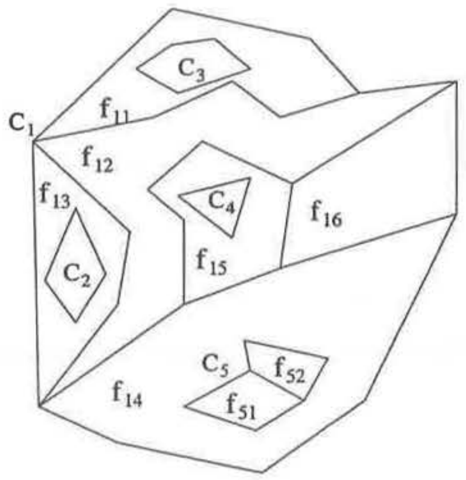


Figure 3.1: A set of nested connected components and faces.

Before we describe the problem we give the precise definitions of a connected component and faces of the connected component.

Definitions. Let G denote an arbitrary planar graph with vertex set $V = \{\nu_1, \nu_2, ..., \nu_n\}$ and edge set E. A (*planar*) *embedding* of G is a drawing of G in the plane which each vertex is represented as a point and each edge is represented as a simple curve joining its endpoints in such a way that no pair of edges intersect except (possibly) at their end points. An embedding $\mathbf{G}$ of G induces a partition of the plane into regions or faces bounded by the edges of $\mathbf{G}$. We call this the *planar subdivision* associated with $\mathbf{G}$. If each of the edges of **G** is straight line segment (respectively, a sequence of straight line segments) then **G** is said to be a *straight-edge* embedding and the associated subdivision is a straight-edge subdivision. It is well known that all planar graphs have straight-edge embeddings; in fact every planar embedding has a topologically equivalent straight-edge embedding [9]. Hereafter we will restrict our attention to linear embedding.

A *path* of *length* k from a vertex μ to a vertex μ' in a graph $G = (V,E)$ is a sequence $\langle \nu_0, \nu_1, ..., \nu_k \rangle$ of vertices such that $\mu = \nu_0$, $\mu' = \nu_k$, and $(\nu_{i-1}, \nu_i) \in E$ for $i = 1, 2, ..., k$. The path contains the vertices $\nu_0, \nu_1, ..., \nu_k$ and the edges $(\nu_0, \nu_1), (\nu_1, \nu_2), ..., (\nu_{k-1}, \nu_k)$. If there is a path p from μ to ν, we say that ν *is reachable from* μ via p. The **connected components** of a graph are the equivalence classes of the vertices under the "is reachable from" relation.

The graph in Figure 3.1 has five connected components: C_1, C_2, C_3, C_4, C_5 and the connected component C_1 has six faces: $\mathfrak{f}_{11}, \mathfrak{f}_{12}, \mathfrak{f}_{13}, \mathfrak{f}_{14}, \mathfrak{f}_{15}, \mathfrak{f}_{16}$.

The Problem. Let $\{e_1, e_2, \ldots, e_n\}$ be a set of n edges. Our topology building problem is to compute a quad-edge data structure [7] for the input edges as an intermediate data structure that contains the topological structure of the set of edges, meanwhile setting up pointer lists for retrieving the connected components and faces of the connected components.

We build the quad-edge data structure of the input data with a plane sweep method. The quad-edge data structure may be seen as a variant of the "winged edge" representation for polyhedral surfaces [2]. In the next section we will give the details of the quad edge data structure. A good property of the quad edge data structure is that the global topological structure can be built by simple operations from the local relations of the edges. This property ensures us that we can construct the data structure correctly by using the plane sweep method. To construct and modify this data structure a single topological operator, called `Splice`, together with a single primitive for the creation of isolated edges, called `MakeEdge`, is sufficient. This data structure is general enough to allow the representation of undirected graphs embedded in arbitrary two-dimensional manifolds. The definition of the data structure is supplied by Guibas and Stolfi [7].

In Section 3.2 and Section 3.3 we give the brief review of the quad edge data structure. If you are familiar with the data structure you may skip these two sections and continue reading from Section 3.4 because we use the same notations that are used by Guibas and Stolfi [7].

3.2 Edge Functions

In this section we give a precise definition for the informal concept of an embedding of an undirected graph on a surface. Special instances of this concept are sometimes referred to as a subdivision of the plane, a generalized polyhedron, a two-dimensional diagram, or other similar names. They have been extensively discussed in the solid modeling literature of computer graphics [2, 11]. We want a definition that accurately reflects the topological properties one would intuitively expect. (For instance, that every edge is on the boundary of two faces, every face is bounded by a closed chain of edges and vertices, every vertex is surrounded by a cyclic sequence of faces and edges, etc.). Our intuition leads the following definition.

Definition 3.1 A *subdivision* of a manifold M is a subset S of M partitioned into three finite collections of disjoint parts: the *vertices*, the *edges*, and *faces* (denoted, respectively, by $\mathcal{V}, \mathcal{E}$, and $\mathcal{F}$), with the following properties:

S1. Every vertex is a point of M.

S2. Every edge is a line segment of M.

S3. Every face is homotopic to (deformable to) an open disk of M.

S4. The boundary of every face is a closed path of edges and vertices.

The vertices, edges, and faces of a subdivision are called its *elements*.

In order to explain the quad edge data structure clearly, we first define the edge functions and describe the dual of a planar graph. Then we discuss the properties of edge functions.

3.2.1 Basic Edge Functions

The edge functions, their dual, and the properties of the functions are given by Guibas and Stolfi [7]. Here we only give a brief description in order to understand the quad edge data structure.

In Figure 3.2, on any disk D of a manifold there are exactly two ways of defining a local "clockwise" sense of rotation; these are called the two possible *orientations on D.* There are also exactly two consistent ways of defining a linear order among points of a line l; each of these orderings is called a *direction along l.* A *directed edge* of a subdivision P is an edge of P together with a direction along it. Since directions and orientations can be chosen independently, for every edge of a subdivision there are four directed and oriented edges. Observe that this is true even if the edge is a loop or is incident twice to the same face of P. Because in many applications, that we are going to discuss including ours, all manifolds to be handled are orientable. This means we can assign a specific orientation to each edge, vertex, and face of the subdivision so that any two incident elements have compatible orientation. Because we can pre-orient the edges we give a simplified version of the edge functions. For the details for non-orientable manifolds, please see Guibas and Stolfi's article [7].

For any oriented and directed edge e we can define its vertex of *origin, eOrg*, its *destination, eDest*, its *left face, eLeft*, and its *right face eRight.* We define also the *symmetric* of e, called *eSym*, as being the same undirected edge with the *opposite direction* but the same orientation as e.

Traversing the boundary of a disk D around a vertex v in the counterclockwise direction establishes a cyclical ordering of the edges. We obtain what is called the *ring of edges out of v*, shown in Figure 3.2. Note that if e is a loop, it will occur twice in the ring of edges out of e. To be precise, both e and *eSym* will occur once each: since the manifold around v is like a disk, e will appear only once in each circuit.

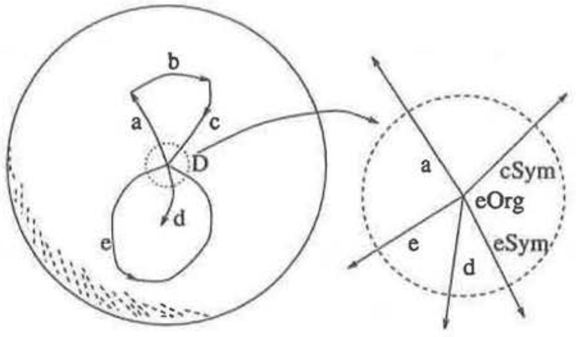


Figure 3.2: The ring of edges out of a vertex.

We can define the *next edge with same origin, eOnext*, as the edge immediately following e (counterclockwise) in this ring, such as d in Figure 3.2 is the *eOnext*. Similarly, given an edge e we define the *next counterclockwise edge with same left face*, denoted by *eLnext*, as being the first edge we encounter after e when moving along the boundary of the face F = *eLeft* in the counterclockwise sense as determined by the orientation of F. The edge *eLeft* is oriented and directed so that *eLnextLeft* = F (including orientation).

3.2.2 Duality

The dual of a planar graph G can be informally defined as a graph G^* obtained from G by interchanging vertices and faces while preserving the incidence relationships. The definition below extends the intuitive concept to arbitrary subdivision.

Definition 3.2 Two subdivisions S and S^* are said to be *dual* to each other if for every directed and oriented edge e of either subdivision there is another edge *eDual* of the other such that

D1. $(eDual)Dual = e$.

D2. $(eSym)Dual = (eDual)Sym$.

D3. $(eLNext)Dual = (eDual)Onext^{-1}$.

Equation D3 states that moving counterclockwise around the left face of e in one subdivision is the same as moving clockwise around the origin of *eDual* in other subdivision. To see why, note that the edges on the boundary of the face $F = eLeft$, in counterclockwise are

$$\{e, eLnext, eLnext^2, \ldots, eLnext^m = e\} \tag{3.1}$$

for some $m \geq 1$. This path maps through *Dual* to the sequence

$$\{eDual, (eDual)Onext^{-1}, (eDual)Onext^{-2}, \ldots, (eDual)Onext^{-m} = eDual\}:$$

all edges coming out of the vertex $v = (eDual)Org$ of S^*, in clockwise order around v.

We can therefore extend *Dual* to vertices and faces of the two subdivisions by defining *(eLeft)Dual = (eDual)Org* and *(eOrg)Dual = (eDual)Left*. Equations D2 and D3 imply that any two edges that differ only in direction will be mapped to two versions of the same undirected edge. The *Dual* establishes a correspondence between $\mathcal{E}$ and $\mathcal{E}^*$, between $\mathcal{V}$ and $\mathcal{V}^*$, and between $\mathcal{F}$ and $\mathcal{F}^*$, such that incident elements of S correspond to incident elements of S^* and vice versa.

The edge *eRot* is called the *rotated* version of e; it is the dual of e, directed from *eRight* to *eLeft* and oriented so that moving counterclockwise around the right face of

e corresponds to moving counterclockwise around the origin of $eRot$. Then we have $(eRot)Rot = eSym$ instead of e.

3.2.3 Properties of Edge Functions

The functions *Rot*, and *Onext* satisfy the following properties:

E1. $eRot^4 = e$.

E2. $eRotOnextRotOnext = e$.

E3. $eRot^2 = eSym \neq e$.

E4. $e \in \mathcal{E}S$ iff $eRot \in \mathcal{E}S^*$.

E5. $e \in \mathcal{E}S$ iff $eOnext \in \mathcal{E}S$.

A number of useful properties can be deduced from these, as for example

$$\begin{aligned} eRot^{-1} &= eRot^3 \\ eOnext^{-1} &= eRotOnextRot \end{aligned}$$

and so forth. For added convenience some derived functions are introduced. By analogy with *eLnext* and *eOnext*, for a given e we define the *next edge with same right face, eRight*, and *with same destination, eDnext*, as the first edges that we encounter when moving counterclockwise from e around *eRight* and *eDest*, respectively. These functions satisfy also the following equations:

$$\begin{aligned} eLnext &= eRot^{-1}OnextRot \\ eRnext &= eRotOnextRot^{-1} \\ eDnext &= eSymOnextSym. \end{aligned}$$

The direction of these edges is defined so that $eLnextLeft = eLeft$, $eRnextRight = eRight$, and $eDnextDest = eDest$. Note that $eRnextDest = eOrg$, rather than vice versa.

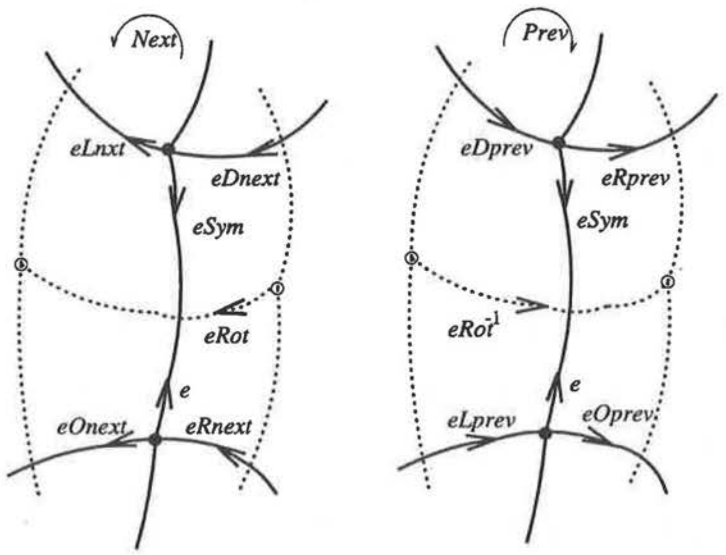


Figure 3.3: The edge functions.

By moving *clockwise* around a fixed endpoint or face, we get the inverse functions, defined by

$$\begin{aligned} eOprev &= eOnext^{-1} = eRotOnextRot \\ eLprev &= eLnext^{-1} = eOnextSym \\ eRprev &= eRnext^{-1} = eSymOnext \\ eDprev &= eDnext^{-1} = eRot^{-1}OnextRot^{-1} \end{aligned}$$

It is important to notice that every function defined so far can be expressed as the composition of a constant number of *Rot* and *Onext* operations, independently of the size or complexity of the subdivision. Figure 3.3 illustrates these various functions.

3.3 Quad Edge Data Structure and Topological Operators

Now we may have this question: do these edge functions accurately capture all the topological properties of a subdivision? The authors of the article [7] gave us the answer. In order to describe the result we first give a definition.

Definition 3.3 An *edge algebra* is an abstract algebra $(E, E^*, Onext, Rot)$ where E and E^* are arbitrary finite sets and $Onext$ and Rot are functions on E and E^* satisfying properties E1 – E5.

The result is that the topology of a subdivision is completely determined by its edge algebra, and vice versa.

3.3.1 Quad Edge Data Structure

We represent a subdivision S (and simultaneously a dual subdivision S^*) by means of the *quad edge data structure*, which is a natural computer implementation of the corresponding edge algebra. The edges of the algebra can be partitioned in groups of four: each group consists of the two directed versions of of an undirected edge of S plus the two versions of its dual edge. The group containing a particular edge e is therefore the orbit of e under the subalgebra generated by Rot. To build the data structure we select arbitrary *canonical representative.* from each edge group. Then any edge e can be written as $e_0 Rot^r$, where $r \in \{0, 1, 2, 3\}$ and e_0 is the canonical representative of the group containing e.

The group of edges containing e is represented in the data structure by one *edge record* e, divided into four parts $e[0]$ through $e[3]$. Part $e[r]$ corresponds to the edge $e_0 Rot^r$, see Figure 3.4.

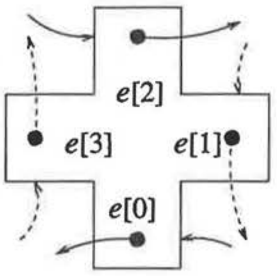


Figure 3.4: Edge record showing `Next` links.

An edge $e = e_0 Rot^r$ is represented by the tuple (e_0, r), called the *edge reference.* We may think of this tuple as a pointer to the "quarter-record" $e[r]$.

Each part $e[r]$ of an edge record contains two fields, `Data` and `Next`. The `Data` field is used to hold geometric and other nontopological information about the edge $e_0 Rot^r$. This field neither affects nor is affected by the topological operation that we will describe, so its contents and format are entirely dependent on the application.

The `Next` field of $e[r]$ contains a reference to the edge $e_0 Rot^r Onext$. Given an arbitrary edge reference (e, r), the two basic edge functions *Rot* and *Onext* are given by the formulas

$$\begin{aligned}(e,r)Rot &= (e, r+1),\\ (e,r)Onext &= e[r].\texttt{Next},\end{aligned} \tag{3.2}$$

Where the r is computed modulo 4. In the first expression in 3.2, this corresponds to rotate e 90° counterclockwise, we advance to the next part of the `Next` field of $e[r]$. The second expression says that applying the *Onext* function to $e[r]$ we get the `Next` field of $e[r]$. From these formulas it follows also that

$$\begin{aligned}(e,r)Sym &= (e, r+2),\\ (e,r)Rot^{-1} &= (e, r+3),\\ (e,r)Oprev &= (e[r+1].\texttt{Next})Rot,\end{aligned} \tag{3.3}$$

and so forth.

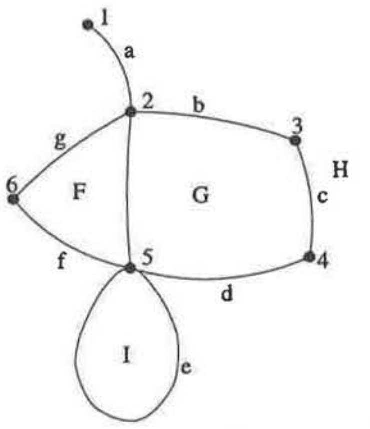
The actual quad edge data structure we use for each $e[r]$ is as follows.

```
typedef struct Qedge {
        Point *data;              /* data field associated with e.Org */
        struct Qedge *next;       /* reference to next edge of ring */
        int mark;                 /* marker for transversal */
        int tmp;                  /* reserved for future use */
} Qedge;
```

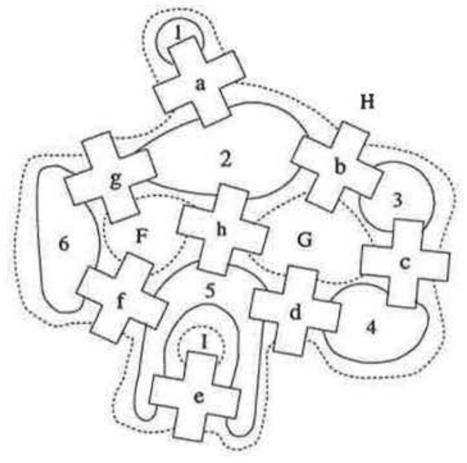
Where the field `data` is a structure `Point` that simply consists of the x and y coordinates of the vertex; the fields `mark` and `tmp` are used by our display routines to show the quad edge data structure.

Figure 3.5 illustrates a portion of a subdivision and its quad edge data structure. We may think of each record as belonging to four circular lists, corresponding to the two vertices and two faces incident to the edge.

The quad edge data structure contains no separate records for vertices or faces; a vertex is implicitly defined as ring of edges, and the standard way to refer to it is to specify one of its outgoing edges. This has the added advantage of specifying a reference point on its edge ring, which is frequently necessary when the vertex is used as a parameter to topological operations. Similarly, the standard way of referring to a connected component of the edge structure is by giving one of its directed edges. In this way, we are also specifying one of the two dual subdivisions and a "starting place" and "starting direction" on it. Therefore a subdivision referred to by the edge e can be "instantaneously" transformed into its dual by referring to $eRot$ instead.



(a) A subdivision of a plane



(b) The data structure for the subdivision (a)

Figure 3.5: The subdivision and its quad edge data structure.

3.3.2 Basic Topological Operators

Two basic topological operators are sufficient to construct and modify the quad edge data structure.

The first operator is denoted by $e \leftarrow$ `MakeEdge`$[org, dest]$. It takes two points *org* and *dest* as parameters, and returns an edge e of a newly created data structure representing a subdivision of the plane (see Figure 3.6), where *org* and *dest* are the end points of a line segment. By assigning *org* to the *eOrg* and *dest* to *eDest*, we set up the direction of the new quad edge e. Because e is the only edge of the subdivision its *left* and *right* faces are closed to be one face. Then the origin points of left and right faces are set to be null. Apart from the orientation and direction, e is the only edge of the subdivision and will not be a loop; then we only need to set up the *Onext* rings of e. Those are $eOnext = e$, $eSymOnext = eSym$, $eRotOnext = eRot^{-1}$, and $eRot^{-1}Onext = eRot$.

The second operator is denoted by `Splice`$[a,\ b]$, takes two edges a and b as input

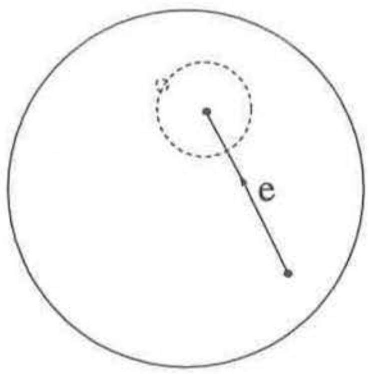


Figure 3.6: The result of `MakeEdge`.

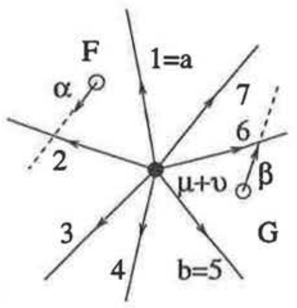
parameters and returns no value. This operation affects the two edge rings *aOrg* and *bOrg* and, simultaneously, the two edge rings *aLeft* and *bLeft*. These rings are defined in section 3.4.1. In each case,

(a) if the two rings are distinct, `Splice` will combine them into one;

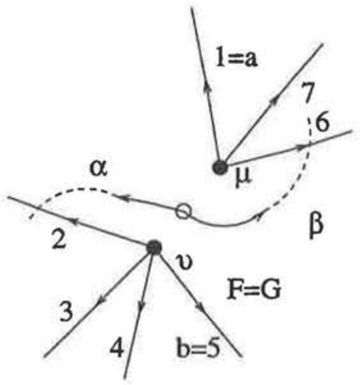
(b) if the two are exactly the same ring, `Splice` will break it into two separate pieces;

The parameters *a* and *b* determine the place where the edge rings will be cut and joined. For the rings *aOrg* and *bOrg*, the cuts will occur immediately *after* *a* and *b* (at *aDest* and *bDest*); for the rings *aLeft* and *bLeft*, the cut will occur immediately *before* *aRot* and *bRot*. Figure 3.7 illustrates this process for one of the simplest cases, when *a* and *b* have the same origin and distinct left faces. In this case `Splice`[*a*, *b*] splits the common origin of *a* and *b* in two separate vertices and joins their left faces. If the origins are distinct and the left faces are the same the effect will be precisely the opposite: the vertices are joined and the left faces are split. Indeed, `Splice` is its own inverse: if we perform `Splice`[*a*, *b*] twice in a row we will get back the same subdivision.

In the edge algebra, the *Org* and *Left* rings of an edge *e* are the orbits under *Onext*



(a) $aOrg = bOrg, aLeft \neq bLeft$.



(b) $aOrg \neq bOrg, aLeft = bLeft$.

Figure 3.7: The effect of `Splice`: Trading a vertex for a face.

of e and $eOnextRot$, respectively. The effect of `Splice` can be described as the construction of a new edge algebra $A' = (E, E^*, Onext', Rot)$ from an existing algebra $A = (E, E^*, Onext, Rot)$, where $Onext'$ is obtained from $Onext$ by redefining some of its values. The modifications needed to obtain the effect described above are quite simple. If we let $\alpha = aOnextRot$ and $\beta = bOnextRot$, basically all we have to do is to interchange the values of $aOnext$ with $bOnext$ and $\alpha Onext$ with $\beta Onext$. The apparently complex behavior of `Splice` now can be recognized as the familiar effect of interchanging the next links of two circular list nodes.

$$\begin{aligned} aOnext' &= bOnext, \\ bOnext' &= aOnext; \\ \alpha Onext' &= \beta Onext, \\ \beta Onext' &= \alpha Onext, \end{aligned} \tag{3.4}$$

Note that these equations reduce to $Onext' = Onext$ if $b = a$. Since $aOnext' =$

$bOnext$, to satisfy axiom E5 we must have $a \in E$ iff $bOnext \in E$, which is equivalent to $a \in E$ iff $b \in E$, where E is defined in Definition 3.3. We will take this as a precondition for validity of `Splice`$[a, b]$.

The following theorem proves that these manipulations are correct (for the proof please see article [7]).

Theorem 3.1 *If A is an edge algebra, a and b are both primal or both dual, then the algebra A' obtained by performing the operation* `Splice[a,b]` *on A is also an edge algebra.*

This theorem guarantees to build a valid quad edge data structure by adding one edge at a time. Based upon this theorem we have adapted a plane sweep algorithm [14] to compute a quad edge data structure from a set of line segments. When the sweep-line encounters a vertex, the edges associated with this vertex can be added or deleted from an array in a constant time and can be spliced according to the connectivity of the edges. Keeping track of this local connection information allows us to build up the quad edge data structure. In the next section we will describe how to construct the quad edge data structure by a one pass sweep.

3.4 Topology Building by a One Pass Sweep

The *Org* and *Left* rings of an edge e are the orbits of e under *Onext* and *eOnextRot*, respectively. The process of topology building is to construct a new edge algebra $A' = (E, E^*, Onext', Rot)$ from an existing algebra $A = (E, E^*, Onext, Rot)$, where $Onext'$ is obtained from *Onext* by applying the `Splice` operation on A in certain order. Since `Splice` does not affect *Rot* we only need to construct the *Org* ring of e correctly. The *Left* ring comes for free by duality. In another words, the *Org* ring of an edge e is a sequence of edges that satisfies the following equation for some $m \geq 1$.

$$\{e, eOnext, eOnext^2, ..., eOnext^m = e\} \tag{3.5}$$

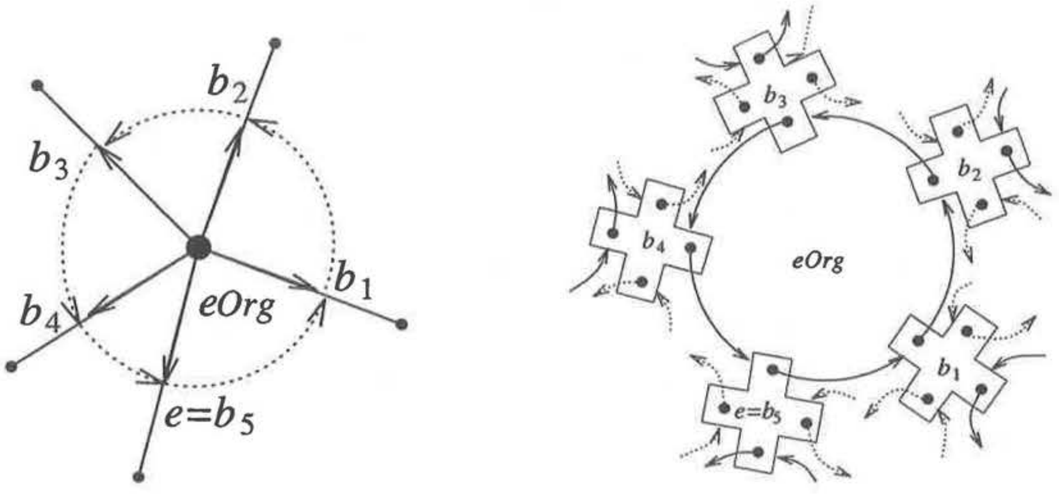
The basic idea to build the quad edge data structure for a connected component by a one pass sweep is to make use of the properties of the edge functions and the operators that can modify the *Org* ring of an existing quad data to build up the entire structure correctly. `Splice` gives us the tool to construct the quad edge data structure with only local information.

3.4.1 Constructing the *Org* Ring of an Edge

We start building an *Org* ring in the quad edge data structure by calling `MakeEdge[ ]` for e, the first edge we encounter. Now the *Org* ring of edge e consists of only itself, that is, $eOnext = e$, see Figure 3.6. From the edge functions (see Figure 3.3) we know that the edges in the *Org* ring of edge e are the edges that have the same origin vertex as edge e and different left faces, see Figure 3.8 (a). These edges are connected to be a cycle in their quad edge data structure; see Figure 3.8 (b).

Let the current *Org* ring of edge e be $\{eOnext, eOnext^2, ..., eOnext^i = e\}$, for $i \geq 1$. Let $a_j = eOnext^j$ for $j = 1, 2, ..., i$, then the *Org* ring of e is $\{a_1, a_2, ..., a_i = e\}$ with the same origin vertex $eOrg$. For a newly created edge b, if $eOrg = bOrg$, which means edges e and b are connected, we get the new *Org* ring of edge e by doing the operation `Splice`[e, b], see Figure 3.9. Since `Splice` does not affect the *Rot* function, we get the correct *Left* ring of edge e after we construct the *Org* ring of edge e.

In order to construct the *Org* rings of the edges consistently we assign the direction of the edges as follows. Let a and b be the two end points of an edge e. Let $(a.x, a.y)$ and $(b.x, b.y)$ be the x- and y- coordinates of the endpoints a and b. The direction of edge e



(a) The *Org* ring of edge *e*. (b) The quad edge data structure of (a).

Figure 3.8: The *Org* ring and its quad edge data structure.

is from a to b if $a.x < b.x$ or $a.y < b.y$ if the $a.x = b.x$. If the direction of edge e is from a to b, we say that endpoint a of edge e is the *origin* of edge e and endpoint b of edge e is the *destination* of edge e. According to this direction assignment there are three cases that need to be handled to build up *Org* rings; see Figure 3.10.

(a) The edge e and b have the same origin vertex, that is, $bOrg = eOrg$. By doing the operation `Splice`[e, b] we get the new *Org* ring of edge e, see Figure 3.10 (a).

(b) The destination vertex $eDest$ of edge e is connected with $bOrg$ of edge b, see Figure 3.10 (b). We know that the edge b is on the *Org* ring of edge $eSym$. We simply do the operation `Splice`[$eSym$, b] and the *Org* ring of edge $eSym$ is constructed correctly. By duality, the *Org* ring of edge e is constructed correctly.

(c) The edge b and edge e have the same destination vertex $eDest$ of edge e, see Figure 3.10 (c). In this case we find that both edge e and edge b are on the *Org* ring of edge $eSym$. By doing the `Splice` operation on $eSym$ and $bSym$, that is,

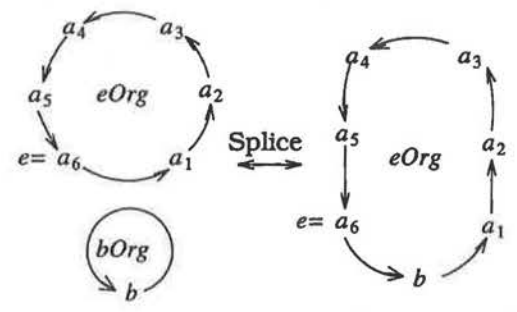


Figure 3.9: The result of `Splice`[e, b].

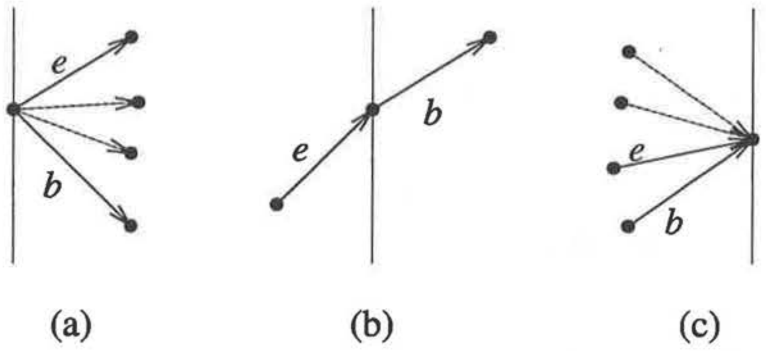


Figure 3.10: The connection cases.

`Splice`[$eSym$, $bSym$], we get the *Org* ring of edge $eSym$ at this vertex.

By constructing the *Org* ring of the edges step by step we get the entire quad edge data structure of the connected components. The following section will give the detailed description of the algorithm.

3.4.2 Algorithm

Any face of a connected component is enclosed by a chain of edges. In the quad edge data structure this chain of edges satisfies Equation 3.1. In another words, the edges of the chain are within the same *Left* ring. If we get an edge e on the chain we can retrieve the face by following the *Left* ring of edge e in the quad edge data structure. So we define a *handle* of a face of a connected component to be a pointer pointing to the quad edge data structure of an edge e that is on the face.

Note that there may be more than one handle for a face of a connected component because of the way we create the handles in **Step 3** (a)-(2). But this will not affect the correctness. We get the same face by retrieving the *Left* ring of these handles.

Input: A set of embedded line segments forms a connected component that has n vertices.

Output: A list of handles of faces of the connected component.

Step1: Sort the vertices of all edges and create an event list for sweeping, breaking ties according to the following rules.

(a) sort the x-coordinates of the vertices of edges

(b) If two vertices have the same x-coordinate, the one with higher y-coordinate is sorted before the other.

(c) If two vertices are the same, the one that is the *destination* vertex of an edge is sorted before the *origin* of another edge.

(d) If the two vertices are the same and have the same type, such as they are the *origin* points or the *destination* points of two edges, the vertex of the edge that is on the left side of another edge is sorted before the other.

Figure 3.11 shows us the order of the vertices associated with the edges. Copies of ν are vertices of edge e_1, e_2, e_3, e_4 and b_1, b_2, b_3, b_4. Their order after sorting is $(e_1, e_2, e_3, e_4, b_1,$

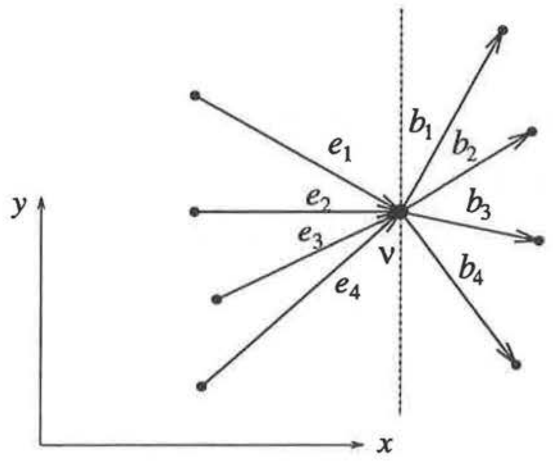


Figure 3.11: The order of the vertices: $(e_1, e_2, e_3, e_4, b_1, b_2, b_3, b_4)$.

$b_2, b_3, b_4)$. Let this sorted list W be $\nu_1, \nu_2, ..., \nu_n$, where n is the total number of the vertices.

The time of this sorting process is $O(n \log n)$.

Step2: Create the first quad edge data structure fe by applying `MakeEdge` to the edge e associated with vertex ν_1. Create the first handle of the face associated with edge e by setting a pointer pointing to fe and mark edge e to be added. Store fe and edge e in the record *SAVE*.

Step3: Sweep a line from left to right, taking steps at each vertex ν_i of W.

(a) If ν_i is the *origin* of edge b: We create the quad edge data structure bq of edge b by `MakeEdge`[b]. Mark edge b as added. Check if edge b is connected with edge e that is stored in the record *SAVE*.

(1) If edge b is connected with edge e.

If the *origin* of e is connected to ν_i, the *origin* of edge b, then we perform the operation `Splice`[e, b].

If the *destination* of e is connected with ν_i, the *origin* of edge b, we do operation `Splice`$[eSym, b]$.

(2) Else edge b is not connected with edge e. We create a handle of the quad edge data structure record of b and insert this handle into the handle list. Substitute e and fe with b and bq.

(b) If ν_i is the *destination* of edge b: We get the quad edge record bq of b that is created when sweeping at the *origin* of b. Check if edge b is connected with edge e that is stored in record *SAVE*.

If edge b is connected with edge e then by our sorting procedure **Step 1**, the *destination* of e is connected with ν_i, the *destination* of edge b. We do operation `Splice`$[eSym, bSym]$ to construct the Org ring of the edges.

After the `Splice` operation or if edge b is not connected with edge e that is stored in record *SAVE*, we substitute e and fe with b and bq in the record *SAVE* and mark the edge b as removed.

The time of this algorithm is mainly spending on sorting the vertices in $O(n \log n)$. Because we do not need to maintain the vertical relations of the edges, the whole sweeping process is linear $O(n)$. So the time complexity of this algorithm is $O(n \log n)$.

3.5 Conclusion

In this chapter we gave a brief review on the basic concepts, the edge functions, and the topological operators of the quad edge data structure. Based upon the edge functions and the topological operators we propose a plane sweep algorithm to construct the quad edge data structure for connected components of a set of line segments. The basic concept of the algorithm is to construct the quad edge data structure by adding an edge each step when sweeping across the connected components and applying the appropriate `Splice` operation on the connected edges. The time of the algorithm is $O(n \log n)$.

Chapter 4

Connected Component Nesting

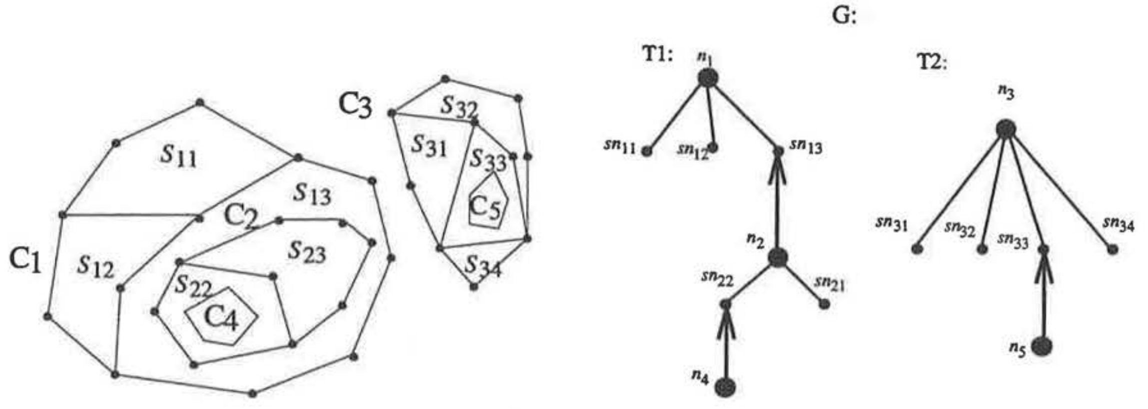
As we mentioned in section 3.1, many objects in GIS are connected components instead of simple polygons and there are many other applications that need the nesting structure among connected components. In Chapter 2 we have studied the polygon nesting problem. In this chapter we describe the connected component nesting problem, and propose an algorithm to extract the nesting structure of connected components.

4.1 Introduction

Unlike the polygon nesting problem for which the topological structure of the input data is given, we assume that no topological structure is given in the input data in the connected component nesting problem.

Because of our application background we restrict our attention to straight-line embeddings of planar graphs. Each connected component is called a *region* and each face that is bounded by a closed chain of edges and vertices, is called a *subregion* of the region.

Problem. Let $\mathcal{C}$ be a set of m connected components C_i in a straight-line planar graph, for $i = 1, 2, ..., m$; Let $\mathcal{SC}_i$ be the set of k_i subregions SC_{ij}, for $j = 1, 2, ..., k_i$; We define $ancestor(C_i)$ to be the set of connected components that have a subregion SC_{ij} containing C_i. The component C_k is called the *parent* of C_i if $ancestor(C_k) = ancestor(C_i) - C_k$. If



(a) The connected components (b) The nesting structure of the components

Figure 4.1: The connected components and the nesting structure.

ancestor$(C_i) = \emptyset$ we say that the parent of C_i is *null*. Any connected component whose parent is C_k is called the *child* of C_k; see Figure 4.1. As in the polygon nesting problem, the *nesting structure* G of $\mathcal{C}$ is an acyclic directed graph (a forest of trees) in which there is a node n_i, corresponding to each connected component C_i in $\mathcal{C}$, and there is a node sn_{ij}, corresponding to each subregion SC_{ij} in the connected component C_i. There is a directed edge from a node n_i to a node sn_{kj} if and only if a subregion SC_{kj} of C_k contains the region C_i and C_k is the parent of C_i. The undirected edges of Figure 4.1 (b) represent the regions with their subregions. The connected components nesting problem is to compute the nesting structure of a set of connected components. Figure 4.1 (b) shows the nesting structure forest of the connected components.

A region is represented by a region number and the leftmost point of the connected component. The subregion is represented by a subregion number and the leftmost point of the subregion. The region data structure is

```
typedef struct region {
```

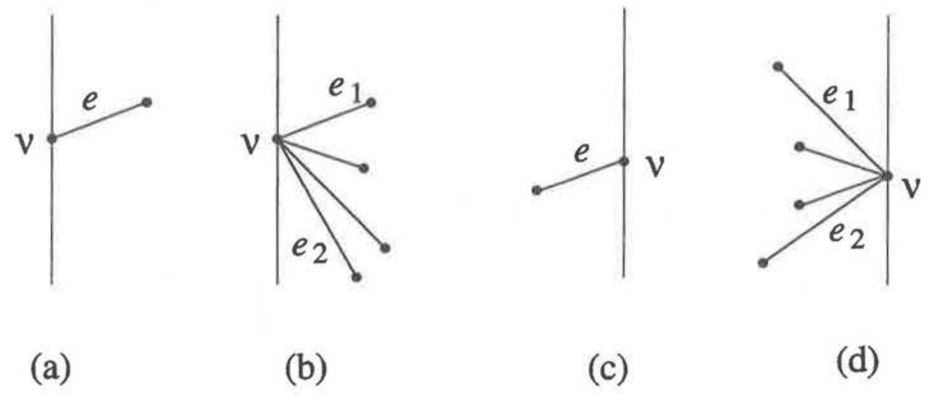


Figure 4.2: The *left-edge* e_1 and *right-edge* e_2.

```
        int *num;            /* the region number assigned */
        Point *leftmost;     /* the leftmost point of this region */
} Region;
```

The subregion data structures is

```
typedef struct subregion {
        int *num;            /* the subregion number assigned */
        Point *leftmost;     /* the leftmost point of this subregion */
} SubRegion;
```

Definitions Let ν_1 and ν_2 be the two vertices of an edge e. We say that ν_1 is the *begin point* of e if the x-coordinate of ν_1 is less than the x-coordinate of ν_2; or if ν_1 and ν_2 have the same x-coordinate but ν_1 has smaller y-coordinate. Then the other vertex ν_2 is called the *end point* of e. We say that an edge e is on the region R of a connected component C_i if e is an edge of which C_i consists. Each edge e is on the boundary of two regions or two subregions. We call these two regions or subregions the *left-region* and *right-region* of e according to the direction of e that is from *begin point* to *end point.* The edge data structure consists of two fields as follows.

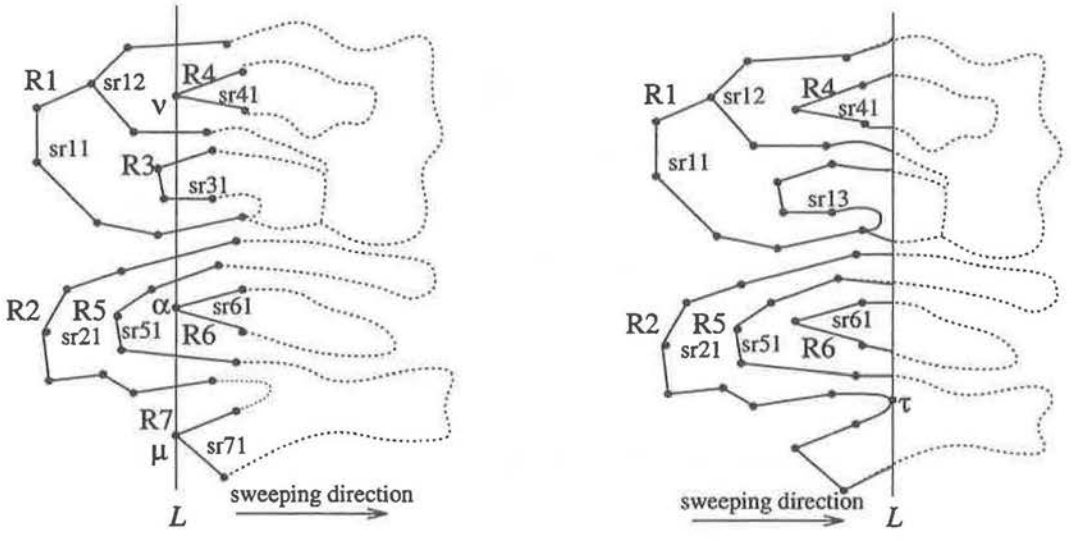
```
typedef struct edge {
        Point *begin;         /* the begin point of this edge */
        Point *end;           /* the end point of this edge */
} EDGE;
```

Let ν be a begin point of edge e. We say that edge e is a *left-edge* if no edge that takes ν as a begin point lies to the left of e. And edge e is a *right-edge* if no edge that takes ν as a begin point lies to the right of e. The edge e in Figure 4.2 (a) is both a left-edge and right-edge. The edge e_1 is left-edge and e_2 is a right-edge in Figure 4.2 (b). This definition is the same when the vertex ν is an end point of edge e, see Figure 4.2 (c) and (d).

4.2 Disjoint Sets and Notches

4.2.1 Disjoint Sets

Because the sweep algorithm transforms a static two dimension problem into a dynamic one dimension problem, we create *disjoint sets* of regions, subregions and, later, *nesting sets* before we know their connection information. When connection information is discovered during the sweep, these disjoint sets are united together to represent the connected components. In Figure 4.3 (a) we do not know any connection information of the regions and subregions, such as regions $R2$ and $R7$ and subregions $sr21$ and $sr71$, before the sweep line reaches the vertex τ. Before reaching vertex τ we create a set to represent each possible region R_i or subregion sr_{ij}. Depending on the information that we have at this stage, we also create a list of nesting sets, such as $nest_3 \subseteq nest_1$ which indicates that R_3 is nested within R_1 and $nest_5 \subseteq nest_2$ which indicates that R_2 nests R_5, etc. After the sweep line reaches vertex τ, we know that R_1 is connected with R_3 and R_2 is connected with R_7. According to the connection information R_1 is united with R_3 to be one region and R_2 is united with R_7 to be one region. The associated subregions and the nesting sets of the united regions are united accordingly. After the union operation, the regions, subregions, and nesting sets are shown in Figure 4.3 (b).



(a) The region and subregion sets at point ν (b) The region and subregion sets at point τ

Figure 4.3: The idea of using disjoint sets to extract the nesting structure.

The nesting sets created with the region and subregion sets are not shown in Figure 4.3 because the nesting sets are described in detail following. The nesting set is represented by a tree with pointers from child to parent, and each nesting set consists of the region and subregion information of the connected component and has a pointer *parent* pointing to its parent nesting set, and a flag indicating the type of this set because the union operation of the nesting set may change the set's type. If the region or the subregion that the set represents is truly a parent of its children sets, the flag is set to be 0, which means this set is the region representative set, or to 2, which means this set is a subregion representative set. Otherwise is set to be 1, indicating that this set is united with other set and the parent of this set will be the parent of this set's children. The `nest` data structure is

```
typedef struct nest {
        int *flag;              /* the set property flag = 0, 1, or 2 */
        struct nest *parent;    /* the parent of this set */
        Region *region;         /* the region on which this set is */
        SubRegion *subregion;   /* the subregion on which this set is */
} Nest;
```

Two disjoint-set operations are used. **Make_set** creates the disjoint region, subregion, and nesting sets. **Union** unites two region set, or subregion sets, or nesting sets and make them consistent with the connection information among these sets.

How to extract the nesting structure from connected components by the two disjoint-set operations will be discussed in detail in Section 4.3.3.

4.2.2 Notches

Because connected components have more complicated structure than simple polygons, a new notch definition is needed.

Definition 4.1 A vertex ν_i is a **notch** of C

(a) if vertex ν_i is taken as a begin point by all the edges that intersect at vertex ν_i;

(b) if vertex ν_i is taken as an end point by all the edges that intersect at vertex ν_i;

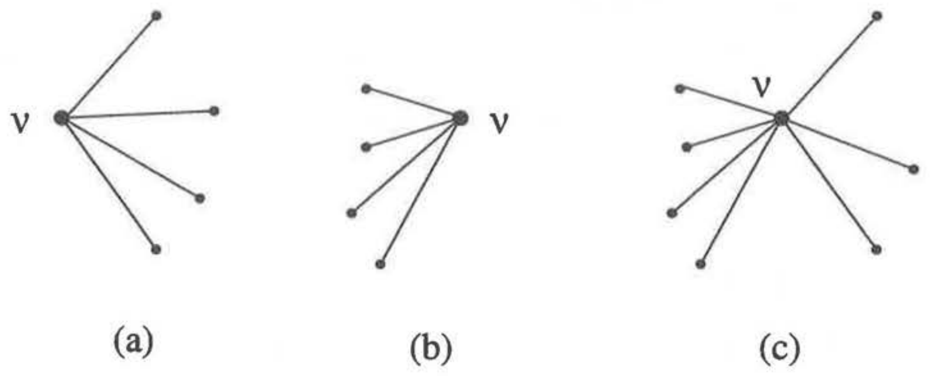


Figure 4.4: Vertex ν is a notch.

(c) if there are at least three edges are connected through ν_i.

In all these parts of Figure 4.4, ν is a notch. Notice that in cases (a) and (b) the number of edges may be only one. We can see that between any two neighboring notches (ν_i, ν_j), the sequence of vertices $(\nu_i, \nu_{i+1}, ..., \nu_j)$ is a x-monotone chain, we call these x-monotone chains *simple lines*; see Figure 4.5.

The simple line is represented by a list of edges. The data structure is

```
typedef struct line {
        EDGE *edge;                 /* one edge of this simple line */
        struct line *next;          /* the pointer to the next edge */
} LINE;
```

We do not stop only at the notches such as in Section 2.2.2 in our sweep algorithm. Because we have assumed that the input data does not give us any connection information of the edges, we get nesting information simultaneously with the connection information. If by having the edge connection information we can spend less than $O(N \log N)$ time to get the notches and simple lines, where N is the number of notches of connected

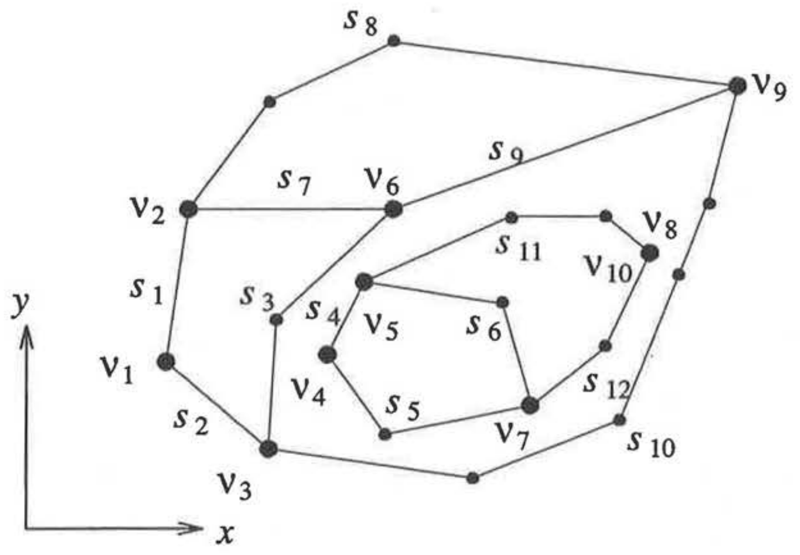


Figure 4.5: $\nu_1, ..., \nu_9$ are notches and $s_1, ..., s_{12}$ are simple lines.

components, it maybe is worthwhile to preprocess the input data and use notches instead of all vertices in the plane sweeping algorithm.

Because all the edges of a simple line have the same region, subregion, and nesting set, one can make simple modifications to our algorithm to substitute vertices and edges with notches and simple lines while sweeping to increase the efficiency of the algorithm because the all the edges of a simple line have the same region, subregion, and nesting set. For example, after we build the quad-edge data structure for the connected components, we can separate each face and reduce the connected component nesting problem to the polygon nesting problem. But we have more efficient way to extract the nesting structure from the connected components. We can get the nesting structure of the connected components while we build the quad edge data structure from them. Because no edge connection information is available at this moment, our algorithm can work with only vertices and edges.

4.3 Plane sweep

As in Section 2.2.2, we sweep a line L through all the connected components, while maintaining the ordering O of the edges induced by L. The difference is that we also create and maintain a list of disjoint dynamic sets of regions, subregions, and nesting sets, for extracting the nesting structure of the connected components. To maintain this ordering and the disjoint sets we stop at endpoints of each edge of the connected components, while sweeping from left to right.

When sweeping from left to right, the sweepline L encounters each vertex in the input data once. Depending on whether the vertex is a begin point or an end point of an edge, new region, subregion, and nesting sets may be created or existing region, subregion, and nesting sets may be united. For a begin point of an edge we update the ordering O, region, subregion, and nesting sets in the following ways.

4.3.1 Update at the Begin Point ν_i of Edge e

There are six cases to be handled when the sweep algorithm encounters the *begin point* ν_i of edge e.

Case (a): ν_i is a vertex such that no edge connected to e through ν_i has been encountered.

A new region R_e is created for e. If more than one edge takes ν_i as the begin point and e is not a right-edge, (which means that there is at least one edge that takes ν_i as a begin point and lies at the right of edge e), a subregion SR_e is created for e. We insert e into ordering O on L by a simple binary search. Simultaneously, we get the above and below neighbor edges of e, say a and b. (The "above neighbor" of e is the lowest edge that is above e in $above(\nu_i)$ and the "below neighbor" of e is the highest edge that is below e in $above(\nu_i)$, see Figure 4.6 (a).) A new nesting set $nest_e$ is created by assigning

R_e to $nest_e$.

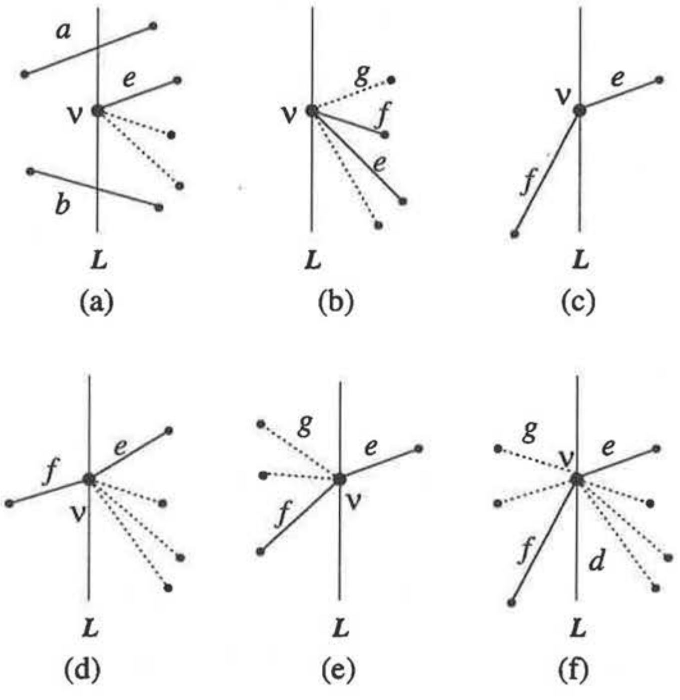


Figure 4.6: Vertex ν is a begin point of edge e.

If a's region $=$ b's region then we know that a's right-region must equal b's left-region. We take the nesting set $nest_a$, representing the subregion SR_a, to be the parent of R_e by creating a pointer from $nest_e$ to $nest_a$. We also assign SR_e or *null* to the subregion field of $nest_e$, and set the flag of $nest_e$ to be 0. The left-region of edge e is set to be the right-region of edge a and the right-region of edge e is set to be the left-region of edge b.

Now we store $nest_e$, R_e, along with the edge e in the ordering O.

If a's region $\neq$ b's region, we detect the parent nesting set of $nest_e$ in the following way. Let x_a be the x-coordinate of the leftmost point of R_a and x_b be the x-coordinate of the leftmost point of R_b. We assume that $x_a \leq x_b$. We take the parent nesting set of the $nest_b$ as the parent of $nest_e$. Because there are no other edges between a and e and

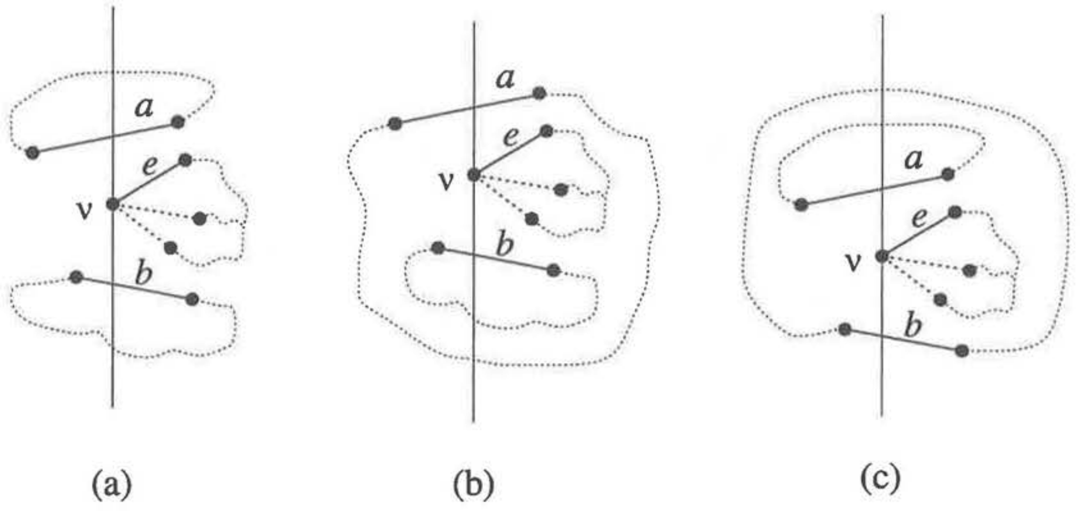


Figure 4.7: The nesting cases.

b and e there are only three situations under this condition. The first one is that $nest_a$ and $nest_b$ have the same parent. The second one is that $nest_a$ nests $nest_b$. The third one is that $nest_b$ nests $nest_a$. Figure 4.7 shows these three situations.

After detecting the parent nesting set we assign the appropriate region and subregion data to the edge and nesting set of e. Then we continue sweeping forward.

Case (b): ν_i is a vertex such that edge e is connected with its above neighbor edge f that takes ν_i as a begin point, see Figure 4.6 (b).

(1) If e is not a *right-edge*. Then e inherits the region and the parent information from f. Because edge e is on the boundary of a new subregion we create a new subregion SR_e for e and set SR_e to be the right region of e. According to their region, subregion, and nesting set information we create a new nesting set $nest_e$ for e and the set flag is set to be 2.

(2) If e is a *right-edge* then e inherits the region and the parent information from f.

Because e is on the boundary of left region of *left-edge* g we set e's right-region = g's left-region. With these region, subregion, and nesting set data we create a new nesting set $nest_e$ for e but no new subregion set is created.

Case (c): ν_i is a vertex such that edge e is connected to only one edge f and f takes ν_i as its end point; see Figure 4.6 (c). This is the simplest case. The edge e simply inherits all the region, subregions, and nesting set from f. There is no new set created.

Case (d): ν_i is a vertex such that edge e is connected with an edge f that takes ν_i as its end point; edge e is a left edge, and edge f is the only edge that takes ν as its end point, see Figure 4.6 (d). Edge e inherits the region and the nesting set information from edge f. Now edge e and edge f have the same left-region, but different right-regions because e is on a new subregion boundary. Then we create a new subregion for e and set the left-region of e to be the new subregion. From the region, subregion, and the nesting set we create a new nesting set $nest_e$ for e and set the flag of $nest_e$ to be 2.

Case (e): ν_i is a vertex such that edge e is connected with an edge f that takes ν_i as its end point; edge f is a right edge, and edge e is the only edge that takes ν as its begin point, see Figure 4.6 (e). Edge e inherits the region and the nesting set from f. But e and f have different left-regions because e has the same left-region as that g has. Because no new region and subregion are encountered we simply let edge e carry region, left-region, right-region, and nesting set data.

Case (f): ν_i is a vertex such that edge e is connected with a edge f that takes ν_i as its end point; edge f is a right-edge, and edge e is a left-edge, see Figure 4.6 (f). In this case, edge e inherits only the region and nesting set from f because the e's left-region and right-region are different from those of f's. Edge e's left-region is the left-region of g, and right-region is a new subregion. So we create a new subregion for e and with these data we make a new nesting set $nest_e$ for edge e.

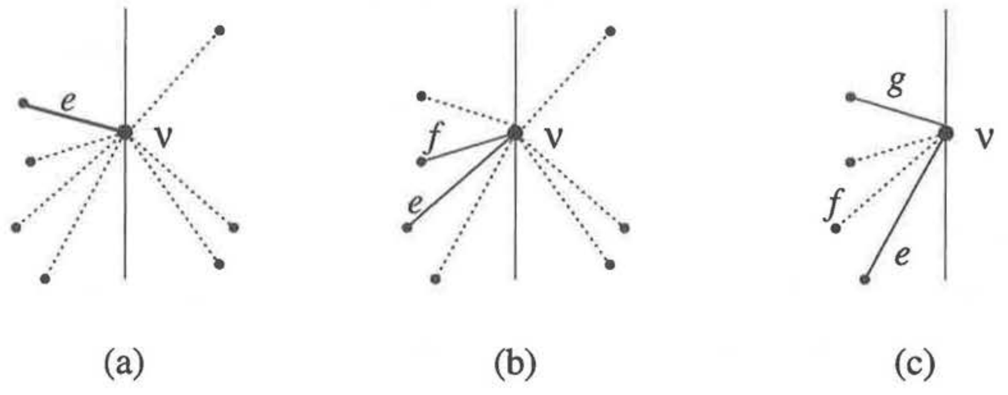


Figure 4.8: The cases of a notch ν as the end point of edge e.

In order to get the four left-edges and right-edges at vertex ν_i, which is currently encountered while sweeping, we simply keep four extra records for vertex ν_i and update these records while sweeping.

4.3.2 Update at the End Point ν_i of Edge e

When the sweep encounters the end point ν_i of an edge e, there are two major situations. In the first one, there are edges that take ν_i as begin point; see Figure 4.8 (a) and (b). In the second one, no edge takes ν_i as its begin point, see Figure 4.8 (c). In these two situations we have three cases that we handle differently. But in common we delete edge e from the ordering O on L by a simple binary search.

Case (a): ν_i is a vertex such that no edge connected to e through ν_i has been encountered. Because e is a left-edge we get e's region, subregion, and nesting set data; then delete e from ordering O on L by a simple binary search.

Case (b): ν_i is a vertex such that edge e is connected with its above neighbor edge f that takes ν_i as a end point, see Figure 4.8 (b).

Now we know edge e and f are connected and they should have a consistent region, subregion, and nesting set. If the data of edge e is not consistent with the data of edge f, we unite these sets and make them consistent, such that e and f have the same region data and consistent left-region of f and right-region of e.

The union policies are as follows.

(1) Suppose edge e and edge f are in the same region but the right-region of edge f and the left-region of edge e are different. Let L_e be the left-region of e and R_f be the right-region of edge f. Let x_e be the x-coordinate of the leftmost point of L_e and x_f be the x-coordinate of the leftmost point of R_f. Without losing generality we assume that $x_f \leq x_e$. Then we substitute L_e with R_f. Meanwhile all the nesting sets that have taken the nesting set of L_e as their parent are changed to the nesting set of R_f as their parent. We implement this by simply changing the data in the records that the pointer points to.

(2) Otherwise edge e and f have different region data. We unite the regions of e and f to be one region. Let R_e be the region of e and R_f be the region of f. Let x_e and x_f be the x-coordinate of the leftmost point of R_e and R_f respectively, and $x_f \leq x_e$. Then we substitute R_e with R_f. Meanwhile all the nesting sets that take the nesting set of region R_e as their parent are changed to the nesting set of R_f as their parent. Then we do a union operation on the left-region of e and right-region of f with the same policy as that in (1), if the left-region of e and right-region of f are different.

Case (c): ν_i is a vertex such that edge e is connected with its above neighbor edge f that takes ν_i as a end point, and there is no edge that takes ν_i as a begin point. see Figure 4.8 (c).

In this case, we not only need to make the region, nesting sets, and subregions of the edges e and f consistent, but also need to ensure that the right-region of e must be consistent with the left-region of the left-edge g, You may notice that g may equal f if there is no other edge between g and e. After we unite f and e as in **Case (b)**, we unite the left-region of g with the right-region of e by the same policy used in **Case (b)**. Then the nesting sets are united according to the region and the subregion data.

4.3.3 Getting Nesting Structure

The correct number of regions, subregions, and the nesting structure of the connected components are determined by the topological structure of the input data. For instance, the region number should be equal to the number of connected components. In this section, we prove that our algorithm determines these quantities correctly.

Lemma 4.1 *The two disjoint-set operations,* **Make_set** *and* **Union** *preserve the number of connected components of the input data.*

Proof: The **Make_set** operation creates at least one region set for each connected component. Let C be a connected component. Let lp be the left-most point of C. Suppose that k ($k \geq 1$) edges take lp as a begin point. When sweepline L encounters lp, it is the case (a) in Section 4.3.1 to be handled. Then a region set is created.

Let R_1, R_2, ..., R_t, ($t \geq 1$), be the region sets created by **Make_set** for connected component C, while sweeping crossing C. For $t = 1$, no **Union** operation is applied to the region set R_1 from the analysis in Section 4.3.1 and Section 4.3.2. Then the lemma holds.

For $t > 1$, there is at least one path from R_i to R_j, $i \neq j$. On the path there is one vertex ν that the two region sets R_i and R_j meet. The vertex ν must be in the case (b) and (c) in Section 4.3.2. While sweepline encounters vertex ν the region sets, R_i and R_j, are united to be one region set according the union policies in Section 4.3.2. Finally, all the region sets, R_1, R_2, ..., R_t, are united to be one region set.

Because no path connects the region sets of different connected components, no **Union** operation is applied to them. So the region sets of different connected components will not be changed. Hence, there is only one region set for each connected component to represent the connected component. Then the lemma holds. □

Lemma 4.2 *The two disjoint-set operations,* **Make_set** *and* **Union** *preserve the correct number of the faces of connected components of the input data.*

Proof: Let f_1, ..., f_k, $(k \geq 1)$, be the faces of connected component C. The same as the proof of lemma 4.1, **Make_set** creates at least one subregion set for each face in a connected component.

Let S_1, ..., S_t, $(t \geq k)$, be the subregion sets created for connected component C. These subregions may belong to different region sets created by **Make_set** for connected component C. Each face is enclosed by a chain of edges and vertices. If more than one subregion sets are created for a face, there is a path that connects them. Suppose that two subregions S_i and S_j meet at vertex ν. The vertex ν must be in the case (b) and (c) in Section 4.3.2. If S_i and S_j are in the same region set, then S_i and S_j are united to be one subregion set, say S_i, directly. If S_i and S_j are in different region sets, the region sets are united first, then S_i and S_j are united to be one subregion set. Hence the subregion sets on the same face are united to be one subregion set. For the subregion sets of the different faces, no **Union** operation is applied on them because the **Union** consistent policies. For the subregion sets of different connected components, no **Union** operation is applied to them because no path connects these subregion sets. Finally, we have one subregion set for each face of connected components. □

Lemma 4.3 *The two disjoint-set operations,* **Make_set** *and* **Union**, *preserve the correct structure of each connected component and its faces.*

Proof: From the analysis in Section 4.3.1 and Section 4.3.2, a nesting set with flag 0 is created when a region set is created and a nesting set with flag 2 is created when a subregion set is created. By combining lemma 4.1 and lemma 4.2, the nesting sets with flag 0 are united to be one nesting set when the region sets are united to be one set. Like the subregion sets, the nesting sets with flag 2, which represent the faces of the connected component, are united to be the nesting sets that each set represents a face of the connected component. Finally, we have one nesting set with flag 0 to represent the connected component and exactly the same number of nesting sets with flag 2 to represent the faces of the connected component. □

From the analysis in Section 4.3.2, the union of two nesting sets n_i and n_j to be one nesting set n_i is accomplished by marking the flag of nesting set n_j to be 1 and setting the parent to nesting set n_i. We call a parent pointer of n_j that points to n_i a *direct edge* from n_j to n_i. Then between a parent nesting set and a child nesting set is a sequence of direct edges. We call this sequence of direct edges is an *edge chain.* The edge chain may have one direct edge. We call the nesting set between two consecutive direct edges *intermediate node.* The flag of a nesting set between two consecutive direct edges is 1 because intermediate nodes have been united. The flags of the two end nesting sets of the sequence of direct edges are marked either 0 or 2.

Lemma 4.4 Let E be an edge chain that connects two nesting sets s_i and s_j. Let sn_1, ..., sn_t, $(t \geq 1)$ be the intermediate nodes. Let f be the face associated with s_i. Let C_1, ..., C_t and C_j be the connected components associated with nesting sets sn_1, ..., sn_t, and s_j. Then the connected components C_1, ..., C_t and C_j are nested in the face f.

Proof: Suppose this lemma is not true. Now let us check the relation between s_i and sn_1. From the assumption we have that C_1 is outside of f but the parent nesting set of sn_1 is s_i. According to the region and subregion consistency policy in Section 4.3.1, the region or subregion associated with s_i is outside of f. Then the subregion associated with f is united with its subregion representative and similarly s_i is united with the nesting set of its subregion representative. Thus, the flag of s_i is set to be 2 which contradicts the given condition 1. This contradiction proves that C_1 is nested in f.

Let C_1, ..., C_{k-1} be the connected components nested in f and C_k be the first connected component not nested in f in the edge chain. Clearly, the edges of f separate C_k from C_1, ..., C_{k-1}. Let μ be the left most point of C_k. When sweepline encounters vertex μ, it is not possible to detect any of the connected components C_1, ..., C_{k-1} as the parent of C_k because the parent detecting is based on the neighbor edges of μ as in Section 4.3.1. Similarly, it is impossible to detect any faces of the connected components C_1, ..., C_{k-1} as the parent of the subregions in C_k. So it is impossible to set a direct edge from sn_k to sn_{k-1}. Hence, the connected component C_k must be nested by the face f. The same kind of analysis can be applied to the connected component C_j associated with nesting set s_j. Then all the connected components associated with sn_1, ..., sn_t, and s_j are nested in the face f. □

Theorem 4.5 The tree of nesting sets records the nesting structure of the connected components correctly.

Proof: Let C_1, C_2, ..., C_m, for $m \geq 2$, be a set of nested connected components. According to lemma 4.3, there are m nesting sets to represent the m connected components. Let n_1, n_2, ..., n_m be the nesting sets that represent the connected components in the nesting tree. Let sn_{i1}, sn_{i2}, ..., sn_{it_i}, for $i = 1, ..., m$, be the nesting sets that represent the faces of the connected components.

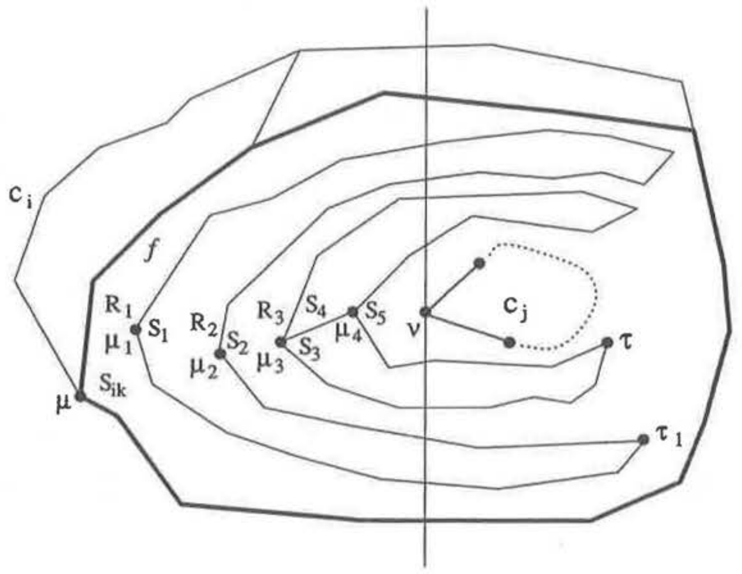


Figure 4.9: The parent of C_j can not be detected directly.

First, we prove that if C_i is the parent of C_j, $i \neq j$, then there is a nesting set sn_{ik} and an edge chain from n_j to sn_{ik} in the nesting tree, where sn_{ik} represents the face that subregion set S_{ik} represents in the connected component C_i.

Because C_j is nested in C_i, there is a face f of C_i that nests C_j. Let the subregion set of f be S_{ik} and the nesting set of f be sn_{ik}. Let lp_j be the left most point of C_j. When sweepline L encounters lp_j, the nesting set n_j is created. According to the parent detecting policies in Section 4.3.1, if we detect the parent of C_j directly, then the parent pointer is set from n_j to sn_{ik}.

Otherwise, the parent of C_j detected at vertex lp_j is not the subregion S_{ik} of C_i, but another connected component, say a subregion S_i of a region R_l. Let sn_i and n_l be the nesting sets associated with S_i and R_l. Because R_l and S_i are not the parent of C_j, the subregion set S_i, and the nesting sets sn_i will be united with another subregion set and nesting set when sweepline encounters a certain vertex, say τ. For example, in

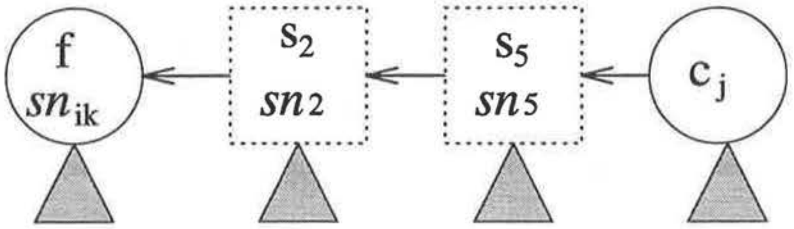


Figure 4.10: The direct edge chain in the tree of nesting sets.

Figure 4.9, the subregion set S_i is S_5, the region set R_l is R_3. For the region set R_l may be united with another region set or may not, depending on the connecting structure of the connected component. In Figure 4.9, there is no region-set union operation at vertex τ but the region set R_2 will be united to region set R_1 at vertex τ_1. Subregion sets that are detected nesting C_j at vertex lp_j will be united with another subregion set according to the region and subregion consistency policies in Section 4.3.2.

Let S_k and R_k be the subregion set and region set that S_i and R_l are united with. After the **Union** operation on S_k and S_i and R_k and R_l, the nesting sets are united accordingly. After nesting set sn_i is united with sn_k, nesting set sn_i has flag set to be 1 and the parent pointer is set pointing to nesting set sn_k associated with subregion S_k. If the region set R_l is united with region set R_k, the nesting set n_l is united with nesting set n_k. After the **Union** operation, nesting set n_l has flag set to be 1 and the parent pointer pointing nesting set n_k. If the parent set of sn_k and n_k is the true parent of C_j, we are done. Otherwise, we can use the same analysis as above to find the direct edge chain that connects C_j with C_i in the tree of nesting sets. The final edge chain found in the tree of nesting sets shows in Figure 4.10.

Now we prove that if an edge chain in the tree of the nesting sets that connects nesting sets from n_j to sn_{ik}, then the subregion S_{ik} that sn_{ik} is associated with nests with the region R_j that n_j is associated with. By the lemma 4.4 it is clear. $\square$

The edge chain of the Figure 4.9 is given in Figure 4.10, where the square boxes are intermediate nodes, the circles are end nodes of the edge chain, and the triangles are the subtrees under the nodes. For example, the subtrees under the intermediate nodes take the face f as their parent; the subtree under the end nodes take the end nodes as their parents.

Corollary 4.6 The forest of the nesting sets records the nesting structure of the connected components correctly.

4.4 Algorithm

In section 4.2 we need the vertex type to indicate whether a vertex is a begin or an end point of an edge, the edge that associated with the vertex, and the number of edges that take the vertex as begin point and the number of edges that take the vertex as an end point when we sweep crossing the data. In order to have these data, we create an event data structure to store these data and linked through a list. The vertex type is indicated by 0 or 1 if the vertex is a begin point or an end point. The structure is

```
typedef struct element {
        int vertype;       /* the vertex type */
        EDGE *edge;        /* the simple line associated with the notch */
        int left;          /* the number of edges that take the notch
                           as a begin point */
        int right;         /* the number of edges that take the notch
                           as a end point */
} ELEMENT;
```

These elements are linked by a list. In the following algorithm one can substitute the vertices and edges with notches and simple lines if the time of getting the notches and simple lines is less than $O(N \log N)$, where N is the total notches in the connected components.

Algorithm

Input: A set of embedded line segments that form a set of connected components that have total n vertices.

Output: A directed acyclic graph G, called the nesting structure.

Step1: Sort all the vertex of the edges and create the event list for sweeping as described in section 3.4.2 **Step1.**

Edge order in the event list are $(e_1, e_2, e_3, e_4, b_1, b_2, b_3, b_4)$, (see Figure 3.11). Let this sorted element list SW be $s_1, s_2, ..., s_n$, where n is the total number of the vertices.

Step2: Create the first region, subregion, and nesting set with parent equal to *null* for the leftmost vertex s_1 of SW, and insert the edge e associated with s_1 into ordering O. Notice O is initially empty.

Step3: Sweep a line from left to right, taking steps at each event element s_i of SW.

(a) If s_i is a begin point of edge e. We update the region, subregion, and nesting set according to the policies in section 4.3.1.

If using notches and simple lines we need to detect the position of s_i with respect to the simple lines intersected by sweep line. For this, carry out a binary search in the ordering O of these simple lines. To detect the position of s_i with respect to a simple line S_j during binary search, find the edge e_1 of this simple line kept in O and then follow the linked list of edges $e_1, e_2, ..., e_k$

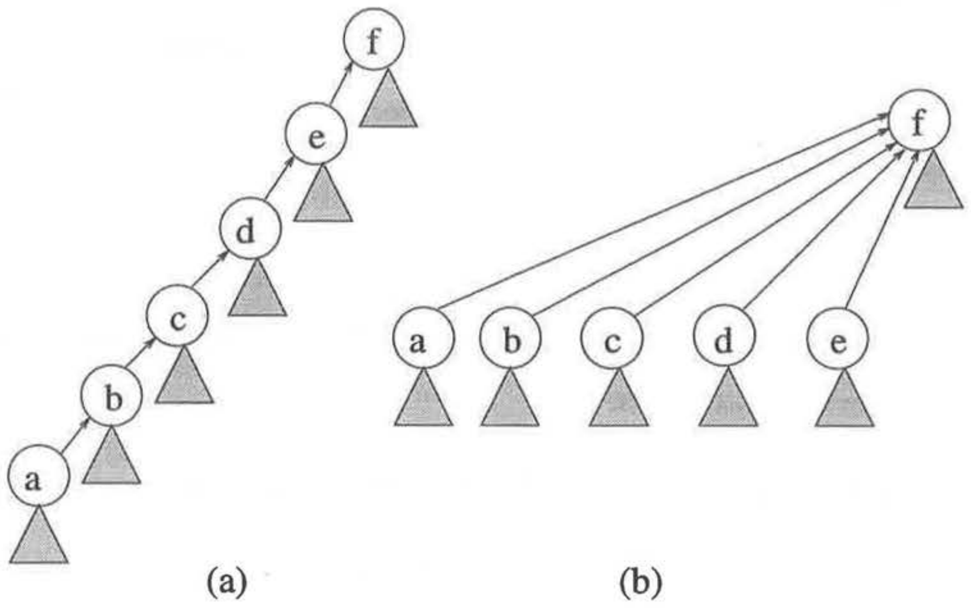


Figure 4.11: Path compression during the operation find parent.

until the edge e_k is found which intersects L. Insert the simple line associated with s_i in O.

(b) If s_i is a end point of the last edge e associated with the simple line that s_i is on. We do the union operation on the region, subregion, and nesting sets according to the policies in section 4.3.2. Then we delete the edge (or simple line associated with s_i) from ordering O by a simple binary search.

Step4: Retrieve the nesting structure from the nesting sets. To retrieve the nesting structure is to find the parent of each nesting set. Notice this may repeatedly report one nesting relation among some nesting sets. But this will not affect the correct results. We represent the nesting sets by a linked-list. During find parent operation we use **path compression**, that is, make each nesting set pointing directly to its parent nesting set on the way of finding a set's parent.

This operation is shown in Figure 4.11. Figure 4.11 (a) shows a tree representing a set prior to find parent. Triangles represent subtrees whose parents are the node shown. Each node represents a nesting set that has a pointer to its parent. The nodes *b, c, d, e* are united sets and their set flag is 1. Figure 4.11 (b) shows the same sets after the operation find parent. Each node on the find path now points directly to the root. From the results of Tarjan [15], the total time of n find parent operations on the nesting sets is $O(n\alpha(n,n))$, where $\alpha(n,n)$ is a very slowly growing inverse of Ackermann's function.

Theorem 4.7 The problem of connected components nesting for m connected components can be solved in $O(n \log n)$ time where n is the total number of vertices in m connected components.

Proof. We sorting the n vertices is $O(n \log n)$ time. Updating at the n vertices takes maximum $O(n \log n)$ time. Retrieving the nesting structure from the nesting sets needs $O(n\alpha(n,n)) \leq O(n \log n)$. Hence, total time spent is $O(n \log n)$. □

Corollary 4.8 If the notches and simple lines of the connected components can be obtained in $O(n)$ time, the problem of connected components nesting for m connected components can be solved in $O(N \log N + N\alpha(N,N) + n) = O(N \log N + n)$ time, where N is the total number of notches of the m connected components.

4.5 Conclusion

In this chapter we discussed the connected components nesting problem, and proposed one algorithm to solve it. Clearly our algorithm for the connected components nesting problem can solve the simple polygon nesting problem with the time of $O(N \log N + n)$.

The simple polygon nesting problem is only a simplified case in the problem we solved. We have generalized the nesting problem to a broader class inducing the nesting of connected components.

Chapter 5

Generating Random Monotone Polygons

5.1 Introduction

This chapter details some results that we have obtained in our study of generating random polygons. In particular, we describe an algorithm for generating x-monotone polygons uniformly at random. The remainder of this section provides motivation for this research and a detailed description of this problem. In Section 5.2, we give the general notation and definitions of our algorithm. In Section 5.3, we present our monotone polygon generating algorithm with the counting procedure and generating procedures. In Section 5.4 we prove that our algorithm can generate monotone polygons uniformly at random. In Section 5.5, we give the visibility computing procedures with the correctness proofs. In Section 5.6 we analyze the time and space complexity of our algorithm. A summary of our results and related open problems are presented in Section 5.7.

5.1.1 Motivation

As well as being of theoretical interest, the generation of random geometric objects has applications that include the testing and verifying the time complexity for computational geometry algorithms.

Algorithm Testing: The most direct use for a stream of geometric objects generated at random is for testing computational geometry algorithms. We can test such algorithms in two ways. The first involves the construction of geometric objects that the implementer considers difficult cases for the algorithm. For example, our polygon-nesting algorithm, based on a plane sweep, may require special case code for some polygons. It is important to make those polygons candidates for exposing errors of the algorithm. The second approach to testing involves executing the algorithm on a large set of geometric objects generated at random. Intuitively, we expect errors to be exposed if enough different valid inputs are applied to the algorithm.

Verification of Average Time Complexity: In implementation-oriented computational geometry research, we are often given the problem of verifying that an implementation of an algorithm achieves the stated algorithm time complexity. This is done by timing the execution of the algorithm for various inputs of different sizes. There are many possible inputs of any given size, and the choice is important, since an algorithm may take more time on some inputs than others of the same size. If an average execution time is computed over a set of randomly generated objects of a given size, the relationship between time and problem size will typically follow a curve corresponding to its average time complexity. We can then check this complexity against the stated algorithm's complexity.

Research, such as Epstein [4], has been done on generating geometric objects at random. In this thesis we give an algorithm that generates random monotone polygons uniformly at random.

5.1.2 Problem

Let $S_n = \{s_1, s_2, ..., s_n\}$ be a set of n arbitrary points. In this chapter, we assume that the x-coordinates of the points in the point set S_n are distinct. We want to generate a simple polygon with vertex set S_n at random. At this beginning stage we only consider generating a monotone polygon from S_n. Figure 5.1 shows a monotone polygon generated from a set of 12 points.

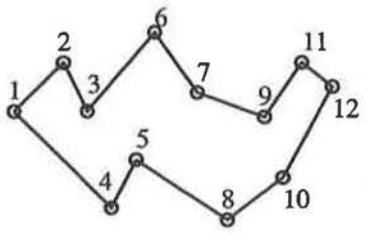


Figure 5.1: A monotone polygon generated from S_{12}

In [4] Epstein gives an $O(n^4)$ algorithm to generate triangulations of a given simple polygon at random. His algorithm, although it does not generate simple polygons at random, inspires us in constructing our algorithms for generating monotone polygons at random.

In Section 5.3, we will give an algorithm that generates a monotone polygon randomly on a set of n points in $O(K)$ time and in $O(n)$ space, where $n \leq K \leq n^2$ is the total number of *above-visible* and *below-visible* point-pairs (see Section 5.2 for definitions) in the point set.

In related work, Meijer and Rappaport [12] study monotone traveling salesmen tours and show that the number of x-monotone polygons on n vertices is between $(2+\sqrt{5})^{(n-3)/2}$ and $(\sqrt{5})^{(n-2)}$. Mitchell and Sundaram [13] have independently developed a routine to generate random monotone polygons in $O(n)$ space and $O(n^2)$ time.

5.2 Preliminaries

Notation. We refer to a *probability space* as (Ω, E, P_r), where Ω is the *sample space*, E is the *event space*, and P_r is the *probability function.* The sample space Ω is the set of all *elementary events* that are the possible outcomes of the experiment being described. The event space E is the set of all subsets of Ω that are assigned a probability. The function $P_r : E \rightarrow \Re_0^+$ defines the probability of events.

A geometric object generator is an algorithm that produces a stream of geometric objects of a given type. We say that a generator is *complete* if it can produce every object in a given sample space Ω.

The Uniform Probability Distributions. Probability theory defines both *discrete* and *continuous* uniform probability distributions. We are interested only in the discrete case: the *discrete uniform probability space* for a finite sample space Ω is defined as (Ω_U, E_U, P_{rU}), where E_U is the set of all subsets of Ω_U, and $P_{rU}(A) = 1/|\Omega_U|$ for all $A \in \Omega_U$. In other words, in a finite sample space, a uniform distribution is one in which each elementary event is equally likely.

Since the sample space we deal with is finite, we use the discrete uniform probability distribution.

A monotone polygon generator is called *uniform* if each of the monotone polygons has the same probability of being generated.

Definitions. Let $S_n = \{s_1, s_2, ..., s_n\}$ be a set of n arbitrary points and if $i < j$ then $s_i.x < s_j.x$. Hereafter S_n is referred to as $\{1, 2, ..., n\}$. Let $S_i = \{1, 2, ..., i\}$ for $1 \leq i \leq n$. The total number of monotone polygons that can be generated with vertex set S_i is denoted as $N(i)$.

Any monotone polygon constructed from S_i can be divided into two monotone chains of which the leftmost vertex is 1 and rightmost vertex is i. In Figure 5.2 the top monotone chain is $\{1, 2, 3, 6, 7, 11, 12\}$ and bottom monotone chain is $\{1, 4, 5, 8, 10, 12\}$. Any point in S_i is either on the top or bottom chain, except 1 and i are on both chains because they are the beginning and ending points of the chains.

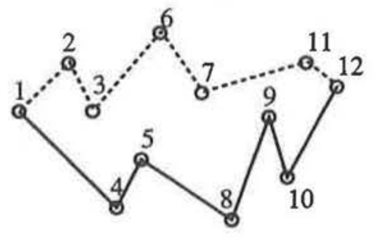


Figure 5.2: The top and bottom monotone chains

Let $T(i)$ be the set of monotone polygons that are generated from S_i with the edge $(i-1, i)$ on their top chains. Let $B(i)$ be the set of monotone polygons that are generated from S_i with the edge $(i-1, i)$ on their bottom chains. We define $TN(i) = |T(i)|$ to be the total number of monotone polygons included in $T(i)$ and $BN(i) = |B(i)|$ to be the total number of monotone polygons included in $B(i)$.

Let $l(j, i)$ be the line determined by j and i. Now we define *above-visible* or *below-visible* for a point. We say that a point k is *above-visible* from i if k is above all $l(j, i)$, for $j = i-1, ..., k-1$. And a point k is *below-visible* from i if k is below all $l(j, i)$, for $j = i-1, ..., k-1$. For example, in Figure 5.3, 10 is *above-visible* from 12, and {9, 7} are *below-visible* from 12.

Let $V_t(i)$ be the set of all the points that are *above-visible* from point i. Let $V_b(i)$ be the set of all the points that are *below-visible* from point i. For example, in Figure 5.3, the $V_t(12) = \{10\}$ and $V_b(12) = \{9, 7\}$.

5.3 Generating Monotone Polygons at Random

We have two steps to generate monotone polygons randomly from S_n. The first one is to calculate the number of monotone polygons that can be generated from S_n. Then we

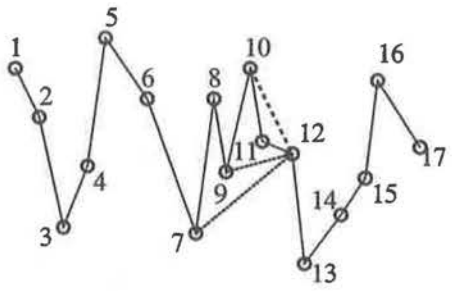


Figure 5.3: The *above-visible* and *below-visible* points from point {12}

scan S_n backward to generate monotone polygons.

5.3.1 Counting Monotone Polygons

Before we give the procedure to count monotone polygons we prove several theorems to build up the theoretical background.

Lemma 5.1 *The set of monotone polygons that are generated from S_k with edge $(k-1, k)$ on their top chains is disjoint from the set of monotone polygons that are generated from S_k with edge $(k-1, k)$ on their bottom chains. That is*

$$T(k) \bigcap B(k) = \emptyset.$$

Proof. Clearly, there is no polygon in $T(k)$ that could include edge $(k-1, k)$ in its bottom chain. And there is no polygon in $B(k)$ that could include edge $(k-1, k)$ in its top chain. So $T(k) \bigcap B(k) = \emptyset$. □

From this lemma we get the following result.

Theorem 5.2 *For any vertex set S_k, with $k > 2$, the number of monotone polygons generated with vertex set S_k is*

$$N(k) = TN(k) + BN(k) \tag{5.1}$$

Proof. Let P be any monotone polygon that is generated from S_k. Then we know that the edge $(k-1, k)$ is either on the top chain of P, which means $P \in T(k)$, or on the bottom chain of P which means $P \in B(k)$. Thus, P is counted by either $TN(k)$ or $BN(k)$. According to Lemma 5.1, we have $N(k) = TN(k) + BN(k)$. □

For any simple monotone polygon generated from S_k, its top chain and bottom chain are paths from 1 to k. The edge $(k-1, k)$ is either on the top chain or on the bottom chain of the monotone polygon. For the chain that does not contain edge $(k-1, k)$, there exists a point j, for $j < k-1$, that connects to k. For the point j we have the following results.

Lemma 5.3 *Let P be any simple monotone polygon that is generated from S_k.*
(1) If the edge $(k-1, k)$ is on the top chain of P and j, for $j < k-1$, is the point that connects to k, then j is below-visible from k.
(2) If the edge $(k-1, k)$ is on the bottom chain of P and j, for $j < k-1$, is the point of the top chain that connects to k, then j is above-visible from k.

Proof. We prove (1). If j is below-visible from k then the lemma holds. If j is not below-visible from k, there exists a line $l(i, k)$ such that j is above $l(i, k)$, where $j < i < k-1$. Because P is a monotone polygon, i is on the top chain of P. But i is below $l(j, k)$. Hence P can not be a simple monotone polygon. This contradiction proves that (1) is true. □

The proof for (2) is the same as that for (1).

Let $P(j,k)$ be a subset of the polygon set $T(k)$ in which the edge (j,k) is on the bottom chain, for $j \in V_b(k)$. That is, $P(j,k) = T(k) \bigcap \{$edge (j,k) is on the bottom chain$\}$ for $j \in V_b(k)$. Now we have lemma as follows.

Lemma 5.4 *The number of monotone polygons in the set $P(j,k)$ is $BN(j+1)$.*

Proof. For the monotone polygons in $P(j,k)$, we know that points j and k are on the bottom chains, and $j+1, \ldots, k$ are on the top chains. So the path of $j, k, k-1, \rightsquigarrow j+1$ is fixed. We can treat the path $j, k, k-1, \ldots, j+1$ as an edge $(j, j+1)$ that is on the bottom chain. Figure 5.4 shows an example. Now we know that the number of monotone polygons in the set of $P(j,k)$ equals the number of monotone polygons generated from S_{j+1} with the edge $(j, j+1)$ on the bottom chains. Hence the lemma is true. We call the set of $B(j+1)$ the *equivalent set* for $P(j,k)$. □

Using a similar proof we have the following result.

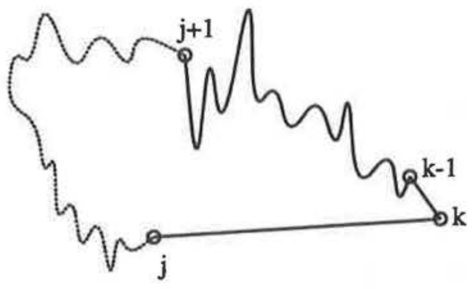
Lemma 5.5 *The number of polygons in $B(k) \bigcap \{$edge (j,k) is on the top chain$\}$ for $j \in V_t(k)$ is $TN(j+1)$.*

Theorem 5.6 *For any point set S_k, we have*

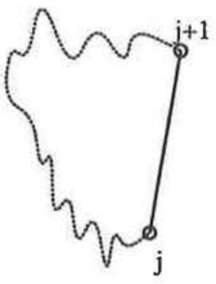
$$TN(k) = \sum_{j \in V_b(k)} BN(j+1) \tag{5.2}$$

$$BN(k) = \sum_{j \in V_t(k)} TN(j+1) \tag{5.3}$$

Proof. We prove formula (5.2). According to lemma 5.3, for any $P \in T(k)$, its bottom chain must use one of the points of $V_b(k)$. Let j be the point. Obviously



(a) The original monotone polygon in $P(k)$



(b) The equivalent set of monotone polygons $B(j+1)$

Figure 5.4: The original set and its equivalent set.

$P \in P(j,k)$. From lemma 5.4, we know that the number of monotone polygons in $P(j,k)$ is $BN(j+1)$. Then the total number of different monotone polygons is $\sum_{j \in V_b(k)} BN(j+1)$. So (5.2) holds. $\Box$

The proof for formula (5.3) is the same as that for formula (5.2).

According to this theorem we can calculate TN and BN if we know $V_b(k)$ and $V_t(k)$. Now we assume that we have $V_b(k)$ and $V_t(k)$ then we can have the procedure that calculates TN and BN. The procedure is shown in Table 5.1.

```
getTNandBN($n$)
    $TN(2) = 1;$
    $BN(2) = 1;$
    FOR $i = 3$, TO $n$
        $TN(i) = 0;$
        $BN(i) = 0;$
        FOR ALL $j \in V_b(i)$
            $TN(i) = TN(i) + BN(j+1);$
        FOR ALL $j \in V_t(i)$
            $BN(i) = BN(i) + TN(j+1);$
    $N(n) = TN(n) + BN(n);$
```

Table 5.1: The Procedure for calculating TN and BN.

After we get $TN(i)$ and $BN(i)$ for $i = 2, \ldots, n$, we start to generate a monotone polygon on $S(n)$ at random, under the uniform distribution. The following section gives the details.

5.3.2 Generating Monotone Polygons

For the general case, we give an algorithm to generate monotone polygons from S_n at random. Again we assume that we have $V_b(k)$ and $V_t(k)$, the *below-visible* and *above-visible* vertices. The algorithm scans the point set S_n backward from the right to the left to generate monotone polygons. Table 5.2 shows the procedure for generating monotone polygons.

Generate_Top, shown in Table 5.3, and **Generate_Bottom**, shown in Table 5.4, deal with two cases. **Generate_Top** deals with the case in which $k-1$ is on the bottom

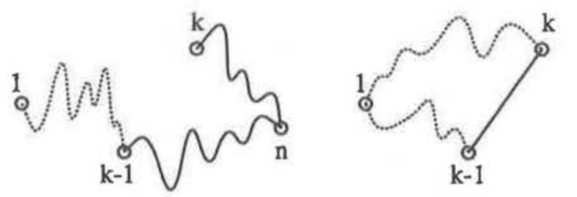
```
Generate
    PICK AN $x$ WITHIN $[1, N(n)]$ UNIFORMLY AT RANDOM;
    ADD $n$ TO top_chain; ADD $n$ TO bottom_chain;
    IF $x \leq TN(n)$
        ADD $n-1$ TO top_chain;
        Generate_Top$(n, x)$;
        ADD 1 TO bottom_chain;
    ELSE
        $x = x - TN(n)$;
        ADD $n-1$ TO bottom_chain;
        Generate_Bottom$(n, x)$;
        ADD 1 TO top_chain;
    END IF
```

Table 5.2: The Procedure for generating monotone polygons.

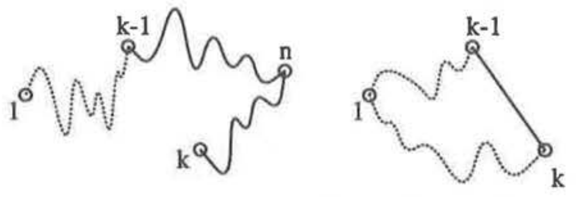
chain and k is on the top chain of the monotone polygon. In this case the undetermined points are $\{1, \ldots, k-2\}$. Then the set of all monotone polygons that can be generated from the original set is equivalent to that from the subset $S(k)$ with edge $(k-1, k)$ on the bottom chains; that is $B(k)$. **Generate_Bottom** deals with the case in which k is on the bottom chain and $k-1$ is on the top chain. In this case the set of all monotone polygons that can be generated is equivalent to $T(k)$. These two cases are shown in Figure 5.5.

Our generating algorithm combines **getTNandBN** and **Generate** together.

```
Polygon_Generator
    getTNandBN$(n)$
    Generate
```



(a) k is on the top chain and its equivalent set $B(k)$



(b) k is on the bottom chain and its equivalent set $T(k)$

Figure 5.5: The generating process.

The following section will show us that our **Polygon_Generator** can generate monotone polygons uniformly at random.

5.4 The Analysis of Polygon_Generator

Let $\Omega(n) = \{P_1, P_2, \ldots, P_{N(n)}\}$ be the sample space of all monotone polygons with vertex set S_n. Then $\Omega(n)$ is a sample space. Each event in $\Omega(n)$ is an unique monotone polygon P_i that can be generated from S_n. We map $\Omega(n)$ to an integer set of $[1, N(n)]$. Each $x \in [1, N(n)]$ corresponds to an unique monotone polygon $P_x \in \Omega(n)$. Now we have the following results.

```
Generate_Top($k$, $x$)
1.  IF $k \leq 2$ RETURN;
2.  FIND THE SMALLEST $i$ SUCH THAT $i$ SATISFIES:
            $x \leq \sum_{j \in V_b(k) \wedge j \leq i} BN(j+1)$;
3.      ADD POINT $i$ TO bottom_chain;
4.      ADD ALL THE POINTS $k-2, k-3, \ldots, i+1$ TO top_chain;
5.      $k = i + 1$;
6.      $x = x - \sum_{j \in V_b(k) \wedge j < i} BN(j+1)$;
7.  Generate_Bottom($k$, $x$)
```

Table 5.3: The Procedure of generating top chains.

Lemma 5.7 *For $n \geq 2$ and $\forall x \in [1, TN(n)]$,* **Generate_Top** *generates an unique monotone polygon $P_x \in T(n) \subseteq \Omega(n)$; For $n \geq 2$ and $\forall x' \in [1, BN(n)]$,* **Generate_Bottom** *generates an unique monotone polygon $P_{x'} \in B(n) \subseteq \Omega(n)$.*

Proof We use induction on n (the size of the point set). Our base case is $n = 2$. Because of $TN(2) = 1$ and $BN(2) = 1$, we know that x is 1. From the procedure **Generate** the input of **Generate_Top** is that 1 and 2 are on the top chain and $x = 1$, and the input of **Generate_Bottom** is that 1 and 2 are on the bottom chain and $x = 1$. For this trivial base case **Generate_Top** and **Generate_Bottom** generate the correct trivial monotone polygon by simply returning to **Generate**.

Now for all $k < n$, we assume that $\forall x \in [1, TN(n)]$ **Generate_Top** generates an unique monotone polygon $P_x \in T(k)$ and $\forall x' \in [1, BN(n)]$, **Generate_Bottom** generates an unique monotone polygon $P_{x'} \in B(k)$.

```
Generate_Bottom(k, x)
1.  IF $k \leq 2$ RETURN;
2.  FIND THE SMALLEST $i$ SUCH THAT $i$ SATISFIES:
            $x \leq \sum_{j \in V_t(k) \wedge j \leq i} TN(j+1)$;
3.      ADD POINT $i$ TO top_chain;
4.      ADD ALL THE POINTS $k-2, k-3, \ldots, i+1$ TO bottom_chain;
5.      $k = i + 1$;
6.      $x = x - \sum_{j \in V_b(k) \wedge j < i} TN(j+1)$;
    END IF
7.  Generate_Top(k, x);
```

Table 5.4: The Procedure of generating bottom chains.

For $k = n$, let $x_1, x_2 \in [1, TN(n)]$ and $P_{x_1}, P_{x_2} \in T(n)$ be the monotone polygons that are generated by **Generate_Top** according to x_1 and x_2. Now we prove that if $x_1 \neq x_2$, then $P_{x_1} \neq P_{x_2}$.

From **Generate** we know that $n-1$ and n are on the top chains of both P_{x_1} and P_{x_2}. Let $i_1 \geq 1$ and $i_2 \geq 1$ be the below_visible points found in **Generate_Top**. There are two cases in this situation.

Case 1: $i_1 \neq i_2$. Without loss of generality, let $i_1 < i_2$. From **Generate_Top** we know that for P_{x_1}, point i_1 is on the bottom chain and point i_2 is on the top chain. For P_{x_2}, point i_2 is on the bottom chain. This proves $P_{x_1} \neq P_{x_2}$.

Case 2: $i_1 = i_2$. From **Generate_Top** we know that $k_1' = k_2' = i_1 + 1$, Since $x_1 \leq TN(n)$ and $x_2 \leq TN(n)$, we have

$$x_1' = x_1 - \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1) \leq BN(i_1 + 1)$$

and

$$x_2' = x_2 - \sum_{j \in V_b(k) \wedge j < i_2} BN(j+1) \leq BN(i_2+1).$$

Because of $x_1 > \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1)$ and $x_2 > \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1)$, we have $x_1' \geq 1$. Then we have $x_1' \neq x_2'$, and $x_1' \in [1, BN(k_1')]$ and $x_2' \in [1, BN(k_2')]$. From our assumption, **Generate_Bottom** generates two different monotone polygons $P_{x_1'}$ and $P_{x_2'}$ with edge (i_1, i_1+1) on the bottom chains. From lemma 5.4 and lemma 5.5, we know that $P_{x_1'}, P_{x_2'} \in B(k_1')$ and that $B(k_1')$ is the *equivalent set* of $P(k_1', k)$. Then we know that the part of polygons of $P_{x_1'}$ and $P_{x_2'}$ without edge (i_1, i_1+1) are on the monotone polygons of P_{x_1} and P_{x_2}. Hence $P_{x_1} \neq P_{x_2}$. □

Using the similar proof, we can prove that for $\forall x' \in [1, BN(n)]$, **Generate_Bottom** generates an unique monotone polygon $P_{x'} \in B(k)$.

From this lemma we immediately get the following result.

Theorem 5.8 *For $n \geq 2$* **Generate** *generates monotone polygons from $\Omega(n)$ uniformly at random.*

Proof **Generate** picks an $x \in [1, N(n)]$ uniformly at random. If $x \leq TN(n)$ **Generate** calls **Generate_top**. If $x > TN(n)$ **Generate** calls **Generate_bottom**. From lemma 5.7 **Generate** generates an unique monotone polygon $P_x \in \Omega(n)$. Thus, **Generate** is an uniform monotone polygon generator. □

Corollary 5.9 **Generate** is *complete.*

5.5 Computing Visibility

The algorithms of the previous section assumed that the *above-visible* and *below-visible* sets, $V_t(i)$ and $V_b(i)$ for $i = 1, \ldots, n$, were available. A closer look, however, shows that these sets are only needed for one index i at a time: algorithm **getTNandBN** needs the sets in increasing order and algorithms **Generate_top** and **Generate_top** need them in decreasing order.

In this section, we show how to calculate each of the sets $V_t(i)$ incrementally as i increases (In Section 5.5.1) and as i decreases (In Section 5.5.2), using time proportional to $|V_t(i)|$ and $O(n)$ space to compute $V_t(i)$ from $V_t(i-1)$ or $V_t(i+1)$.

The idea is the following. Let S_k denote the monotone chain with vertices s_1, s_2, $\ldots$, s_k. If we think of S_k as a fence and compute the shortest paths in the plane above S_k from s_k to each s_i with $i \leq k$, then we obtain a tree that is known as the *shortest path tree rooted at* s_k [5, 6]. The *above-visible* set $V_t(i)$ is exactly the set of children of s_k in the shortest path tree rooted at s_k. Thus, we will incrementally compute shortest path trees rooted at s_1, s_2, $\ldots$, s_k to get the *above-visible* sets.

We represent shortest path trees (in which a node may have many children) by binary trees in which each node has pointers to its uppermost child and next sibling. Section 5.5.1 gives the details for computing these trees in the forward direction: computing $V_t(i)$ from $V_t(i-1)$. Section 5.5.2 gives the details for the reverse direction: computing $V_t(i)$ from $V_t(i+1)$.

5.5.1 Computing Visibility Forward

We use a tree data structure to calculate $V_t(k)$ and $V_b(k)$ recursively. Assuming $V_t(k-1)$ and $V_b(k-1)$ have been calculated, we calculate $V_t(k)$ and $V_b(k)$ according to the results of $V_t(k-1)$ and $V_b(k-1)$. The data structure that we use in the calculation is the tree of the shortest paths rooted at vertex k.

We store *top_tree*(i) and *bot_tree*(i) using child and sibling pointers. For each vertex $j \in [1, n]$, we have a record for *top_tree*

j:	*ptr*	*ptr* stores the coordinates of vertex j
	upc	*upc* is a pointer pointing the upper child of j in *top_tree*(k)
	sib	*sib* is a pointer pointing the sibling of j in *top_tree*(k)

and a record for *bot_tree*

j:	*ptr*	*ptr* stores the coordinates of vertex j
	lwc	*lwc* is a pointer pointing the lower child of j in *bot_tree*(k)
	sib	*sib* is a pointer pointing the sibling of j in *bot_tree*(k)

The initial value of *top_tree* for the recursive calculation is $1.ptr = 1$, $1.upc = nil$ and $1.sib = nil$. We assume that *top_tree*$(i-1)$ has been completed. Then we call $\mathbf{Make\_V_t}$ to calculate the *above-visible* set, V_t. In order to get V_t, the procedure $\mathbf{Make\_V_t}$ calls the procedure **Make_top** to calculate the *top_tree*(i).

```
$\mathbf{Make\_V}_t(i)$
    $t = tmp;$
    $\mathbf{Make\_top}(i-1, i, \mathbf{Var}: t);$
    $i.upc = tmp.sib;$
```

Procedure **Make_top**($i-1, i, lastsib$) makes the tree edge from k to j in $top\_tree(k)$, and puts it as the sibling of $lastsib$ and updates $lastsib$. Then it recursively builds the $top\_tree(k)$. One example is shown in Figure 5.6.

```
Make_top($j, k$, Var : $lastsib$)
    WHILE $j.upc \neq nil$ and $k$ is above $l(j.upc, j)$
        Make_top($j.upc, k$, Var : $lastsib$);
        /* make subtree for this child of $j$, which can be seen by $k$. */
        $j.upc = j.upc.sib$; /* consider next child of $j$ */
    END WHILE
    $lastsib.sib = j$; /* make the connection to $j$, one of the children of $k$ */
    $lastsib = j$;
```

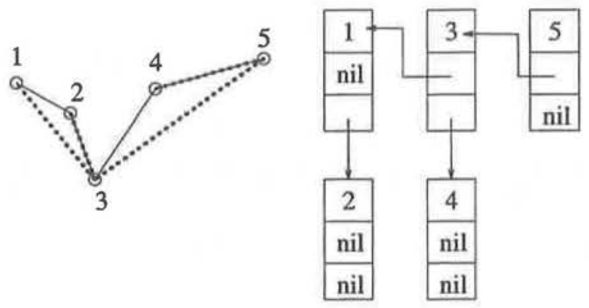


Figure 5.6: A point set S_5 and the data of $top\_tree(5)$.

To compute the bot_tree is similar to computing the top_tree. We need only change upc and 'above' in procedure **Make_V**$_t(i)$ and **Make_top** into lwc and 'below' to get the procedures **Make_V**$_b(i)$ and **Make_bot**. We use **Make_V**$_b(i)$ and **Make_bot** to compute $bot\_tree(i)$ from $bot\_tree(i-1)$. One example is shown in Figure 5.7.

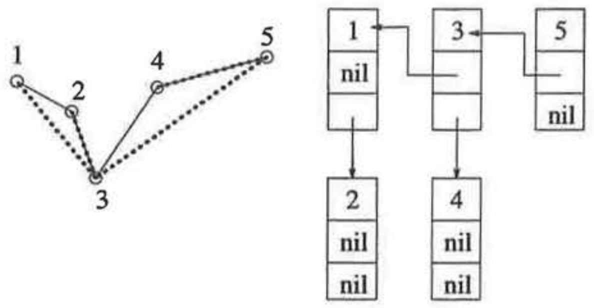


Figure 5.7: A point set S_5 and the data of *bot_tree*(5).

Knowing *top_tree*(k) and *bot_tree*(k), we know the *above-visible* and *below-visible* point sets, $V_t(k)$ and $V_b(k)$ of vertex k. Now we give the theorem to show us how to get $V_t(k)$ and $V_b(k)$ from *top_tree*(k) and *bot_tree*(k).

Let r be a record in the *top_tree* or *bot_tree*. We define $r.sib^i = r.sib^{i-1}.sib$, for any integer $i > 0$, and $r.sib^0 = r$. Now we claim that the upper child of k and its siblings are the vertices visible from k, and any vertex that is visible from k is either the upper child of k or its sibling. This is proved in the next theorem.

Theorem 5.10

Let $CT(k) = \{k.upc.(sib)^i \mid \forall i \geq 0\}$. *Let* $CB(k) = \{k.lwc.(sib)^i \mid \forall i \geq 0\}$. *We have* $V_t(k) = CT(k) - \{k-1\}$ *and* $V_b(k) = CB(k) - \{k-1\}$.

Proof First we prove $V_t(k) = CT(k) - \{k-1\}$. If $V_t(k) = \emptyset$, there is no point that is above line $l(k-1,k)$. This means that there is no $l(i,k-1)$ that is below k. From **Make_top**, we know that $CT(k) = \{k-1\}$. Hence $V_t(k) = CT(k) - \{k-1\}$. If $CT(k) = \{k-1\}$ there is no $l(i,k-1)$ that is below k for $i = 1,\ldots,k-2$. So there exists no point that is above $l(k-1,k)$. Hence $V_t(k) = \emptyset = CT(k) - \{k-1\}$.

For the general situation, $\forall j \in V_t(k)$, we have j is above all $l(i,k)$ for $i = j+1,\ldots,k-1$

```
Back_top(k + 1, k)
1.  FIND $i$, SUCH THAT $(k+1).upc.sib^i = k$;
2.  FOR $j = 0$ TO $i$
3.      $a_{i-j} = (k+1).upc.sib^j$;
4.  IF $i = 0$ RETURN;
    ELSE IF $i \geq 1$
5.          Granham-Scan-Top($i, a_0, \ldots, a_i$);
    END IF
```

Table 5.5: The Procedure of computing *top_tree*(k) from *top_tree*$(k+1)$.

that implies that k is above all $l(j,i)$ for $i = j+1, \ldots, k-1$. Now we prove $j = k.upc.(sib)^i$, $i \geq 0$. If there is no $j' \in V_t(k)$ and $j' < j$ such that k is above $l(j',j)$ then $j = k.upc$. Otherwise, $j = j'.sib$. Similarly this induction can be applied to j', that is, $j' = k.upc.(sib)^{i'}$. Then we have $j = k.upc.(sib)^{i'+1}$. So $V_t(k) \subseteq CT(k) - \{k-1\}$.

$\forall j \in CT(k) - \{k-1\}$, we know $j = k.upc.(sib)^i$. Then j is above all $l(i,k)$, for $i = j+1, \ldots, k-1$. Otherwise, there exists a point, say j', such that $j' > j$ and j is below $l(j',k)$. Then $l(j,k)$ is below $l(j',k)$ that means k is below $l(j,j')$. From **Make_top** we know that j can not be the format of $k.upc.(sib)^i$, $i \geq 0$. This contradiction proves that j is above all $l(i,k)$, for $i = j+1, \ldots, k-1$. Then we have that $j \in V_t(k)$ that implies $V_t(k) \supseteq CT(k) - \{k-1\}$. Now we have $V_t(k) = CT(k) - \{k-1\}$. □

The proof for $V_b(k) = CB(k) - \{k-1\}$ is similar to the proof above.

5.5.2 Computing Visibility backward

In procedure **Generate_Top** and **Generate_Bottom**, we need to find the smallest i satisfying the expression in line 2. Here we assume that *top_tree*$(k+1)$ and *bot_tree*$(k+$

```
Granham-Scan-Top($i, a_0, \ldots, a_i$)
1.  Push($a_0$,S);                    /* S is a stack */
2.  $a_1 = a_0.upc$;
3   $a_0.upc = a_1$;
4.  Push($a_1$,S);
5.  FOR $j = 2$ TO $i$
6.      WHILE the angle formed by points NEXT-TO-Top(S), Top(S),
                  and $a_j$ makes nonleft turn
7.          Pop(S);
        END WHILE
8.      $a_j = Top(S).upc$;
9.      Push(S,$a_j$);
```

Table 5.6: The Procedure of **Graham-Scan-Top**.

1) have been completed, and define procedures **Back_top** and **Back_bot** to generate *top_tree*(k) and *bot_tree*(k). Let $a_{i-j} = (k+1).upc.sib^j$, for $j = 0, 1, \ldots, i$. Then $a_i = (k+1).upc$ and $a_0 = k$. Let $Q = \{a_j, j = 0, \ldots, i\}$. From theorem 5.10, we know $Q = V_t(k+1) - \{k\}$. If we take a_0 as the origin of of coordinates, according to the *above-visible* definition, the points in Q are sorted lexicographically by polar angle and distance from a_0. Then from Graham-Scan we can get the correct *top_tree*(k). This is similar for calculating *bot_tree*(k). Table 5.5 and Table 5.6 show the procedures.

One example to calculate *top_tree*(k) from *top_tree*$(k+1)$ is shown in Figure 5.8.

Similarly we have the procedure to compute *bot_tree*(k) from *bot_tree*$(k+1)$. They are called **Back_bot** and **Graham-Scan-Bot**(i, Q). We get them simply by changing *upc* and 'nonleft turn' of **Back_top** and **Graham-Scan-Top**(i, Q) into *lwc* and 'nonright turn'.

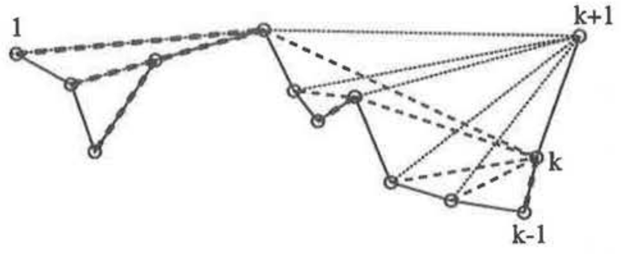


Figure 5.8: *top_tree*(k) is generated from *top_tree*$(k+1)$.

Now we prove that these procedures compute correct results.

Theorem 5.11 **Back_top** *and* **Graham-Scan-Top**(i, Q) *correctly compute top_tree*(k) *from top_tree*$(k+1)$. **Back_bot** *and* **Graham-Scan-Bot**(i, Q) *correctly compute bot_tree*(k) *from bot_tree*$(k+1)$.

Proof We prove that **Back_top** and **Graham-Scan-Top**(i, Q) correctly compute *top_tree*(k) from *top_tree*$(k+1)$. In **Back_bot** we first find the upper child of $k+1$ and its siblings. In order to get *top_tree*(k) from *top_tree*$(k+1)$ we must cut the edges of these vertices with $k+1$ and reconnect them with appropriate vertices. These points are the only points that need to be reconnected.

In **Graham-Scan-Top**(i, Q), point k is always kept in the bottom of the stack S. For any vertex visible from $k+1$, there two cases. Case 1 is that it is visible from k. Case 2 is that it is not visible.

Case 1: if point j is visible from k then all the points in the stack S are popped out but k. Now we output edge (j, k) and point j is pushed into S. Now there are at least two points in the stack S.

Case 2: otherwise point j is not visible from k. We know that j must be visible from a vertex in S, say j'. Then all the points on top of j' are popped out, and we output the edge (j, j') and j is pushed into S.

After we checked all the points visible from $k+1$, we reconnect the points correctly. □

Similarly we can that prove **Back_bot** and **Graham-Scan-Bot**(i, Q) correctly compute $bot\_tree(k)$ from $bot\_tree(k+1)$.

Now we have all the procedures to build up our algorithm. Next we give its time and space complexity.

5.6 Time and Space Complexity Analysis

Lemma 5.12 *The runtime of* **Make_top**$(k-1, k, \mathbf{Var}: t)$ *is* $O(|V_t(k)|)$. *And the runtime of* **Make_bot**$(k-1, k, \mathbf{Var}: t)$ *is* $O(|V_b(k)|)$.

Proof Because of the similarity, we only prove the runtime of **Make_top**$(k-1, k, \mathbf{Var}: t)$ is $O(|V_t(k)|)$.

Let us assign the following amortized costs:

WHILE checking	1
updating $j.upc$	1
updating $lastsib$	1

and each time we encounter the upper child, $k.upc$ or its sibling $k.upc.sib^i$, but excluding $k-1$, we get 3 credits. Clearly from theorem 5.10, we know that the total number of

$k.upc$ and $k.upc.(sib)^i$, excluding $k-1$, is $|V_t(k)|$.

We shall now show that we can pay any operation costs by charging the amortized costs. We start from **Make_top**$(k-1, k, \textbf{Var}: t)$ and we have 3 credits. Clearly if j is visible from k, WHILE checking succeeds. From this we get 3 more credits to pass to the next call to **Make_top**$(j, k, \textbf{Var}: \textit{lastsib})$. Then this call receives 3 credits to pay for its own checking and updating costs. If j is not visible from k, WHILE checking fails. Then the current call to **Make_top** saves 1 credit for upper level **Make_top** to pay another WHILE checking. We know that **Make_top**$(j, k, \textbf{Var}: \textit{lastsib})$ with 3 credits can pay their own costs and the number of total recursive calling for **Make_top**$(j, k, \textbf{Var}: \textit{lastsib})$ is $|V_t(k)|$. Then $3 * |V_t(k)|$ will pay all the costs. So the runtime of **Make_top**$(k-1, k, \textbf{Var}: t)$ is $O(|V_t(k)|)$. □

Theorem 5.13 Polygon_Generator *has time complexity of* $O(K)$ *and space in* $O(n)$*, where* K *is the total number of above-visible and below-visible points of the points in the point set.*

Proof From lemma 5.12 the runtime of **getTNandBN** is, for some constant c,

$$\sum_{k=3}^{n} c * (|V_t(k)| + |V_b(k)|) \leq cK = O(K).$$

Clearly the runtime of **Back_top** is $O(|V_t(k)|)$ and the runtime of **Back_bot** is $O(|V_b(k)|)$.

The time complexity of **Generate** depends on the time complexity of **Generate_Top** and **Generate_Bottom**. Because they have a similar structure the time complexity of **Generate_Top** and **Generate_Bottom** is the same. Let t_k be the run time of **Generate_Top**(k, x) From line 2 to 6, the time depends on the number of *above-visible* and *below-visible* points of k. Then we have, for some constant c

$$t_k = \sum_{j=i+1}^{k} c * (|V_t(k)| + |V_t(k)| + k - i) + t_{i+1}.$$

So

$$t_n \leq \sum_{k=1}^{n} c * (|V_t(k)| + |V_b(k)|) + n \leq c * (K + n).$$

Hence the run time of **Generate** is $O(n + K)$. Obviously, $n \leq K \leq n^2$. The time complexity of our **Polygon_Generator** is $O(n + K) + O(K) = O(K)$.

In the process of generating we need only to store the point set S_n, *top_tree*(n), *bot_tree*(n), and $TN(i)$ with $BN(i)$, for $i = 2, \ldots, n$. Since each of the data structures use no more than $O(n)$ memory space, we have that the memory space of **Polygon_Generator** is $O(n)$. □

5.7 Conclusion

We have presented an algorithm to generate monotone polygons uniformly at random. The time complexity of our algorithm is $O(K)$, where $n \leq K \leq n^2$ is the number edges of the visibility graph of the x-monotone chain whose vertices are the given n points. The space complexity of our algorithm is $O(n)$. We have given the detail analysis of the algorithm and the proof of its correctness. A random monotone polygon generator is useful for testing the many algorithms that accept a simple polygon or a group of simple polygons as input.

We are also interested in finding a polynomial algorithm to generate general simple polygons randomly from an arbitrary set of points. There is, to our knowledge, no efficient enumeration of the simple polygons on a given vertex set.

Chapter 6

Conclusions

Summary

In this thesis, we have briefly reviewed the basic concepts, the edge functions, and the topological operators of the quad edge data structure. Based upon the edge functions and the topological operators, we presented a plane sweep algorithm that extracts connected components of a set of line segments and captures the topology in a quad edge data structure. We have described that to construct the quad edge data structure correctly is to construct the *Org* rings of the edges correctly. The validity and feasibility of our sweep algorithm method are proved mathematically and tested by implementation. The time of the sweep process is linear in the total number of vertices n. The sorting time dominates the algorithm's complexity, so its time complexity is $O(n \log n)$.

For the polygon nesting problem, we reported results along two fronts. First, we redefined the idea of a notch in Bajaj and Dey's polygon nesting algorithm to provide a direct correspondence between notches and subchains. Second, we presented a plane sweep algorithm to solve the nesting problem on a set of connected components. In the connected components nesting problem the objects are more general than Bajaj and Dey's restriction to simple polygons. The correctness of our sweep algorithm is proved mathematically. The time complexity of our sweep algorithm is $O(n \log n)$, where n is the total number of vertices of the connected components. If the notches and simple lines of connected components can be obtained in $O(N \log N)$ time, we find the nesting structure of the connected components in $O(N \log N + n)$ time, where N is the number of notches and n is the total number of vertices. The nesting structure for simple polygons is an instance of the connected components problem, so our sweep algorithm can identify the structure in $O(N \log N + n)$ time.

The algorithms for constructing quad edge data structure, polygon nesting, and connected components nesting are implemented in C on Sun-workstation and Silicon Graphics machines.

We examined methods for generating random polygons and presented an algorithm to generate monotone polygons uniformly at random. We are able to generate a random monotone polygon in $O(K)$ and $O(n)$ space, where n is the total number of vertices in the point set and $n \leq K \leq n^2$ is the number edges of the visibility graph of the x-monotone chain whose vertices are the given n points. A detail analysis of the algorithm and a proof of its correctness are also given. A random monotone polygon generator is useful for testing many of the algorithms that accept a simple polygon or a group of simple polygons as input, such as the polygon nesting algorithm.

Future Directions

It is interesting to apply the algorithms to large GIS data set. Although we have not done this testing yet, we think that it is an important step in exploring more practical problems in the GIS application area.

A second interesting direction is to combine, the quad edge data structure building algorithm, the polygon nesting algorithm, and the connected component nesting algorithm, with a GIS database system to increase the tool power and the efficiency of the GIS. There is a great potential for many interesting problems to arise in this combining process.

Finally, because there is no efficient enumeration of the simple polygons on a given vertex set, we are interested in finding a polynomial algorithm to generate general simple polygons randomly from an arbitrary set of points.

Bibliography

[1] C. L. Bajaj and T. Dey. Polygon nesting and robustness. *Information Processing Letters*, 35(1):23–32, June 1990.

[2] B. G. Baumgart. A polyhedron representation for computer vision. In *1975 National Computer Conference*, volume 41 of *AFIPS Conference Proceedings*, pages 589–569. AFIPS Press, Arlington, Va., 1975.

[3] M. J. Egenhofer and J. Sharma. Topological consistency. In *5th International Symposium on Spatial Data Handling*, pages 335 – 343. IGU Commission on GIS, August 1992.

[4] P. Epstein and J. Sack. Generating triangulation at random. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, pages 305 – 310, 1992.

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

[6] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, October 1989.

[7] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[8] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[9] D. G. Kirkpatrick. Establishing order in planar subdivisions. *Discrete Comput. Geom.*, 3:267–280, 1988.

[10] H.-P. Kriegel, T. Brinkhoff, and R. Schneider. The combination of spatial access methods and computational geometry in geographic database systems. In *Data structures and efficient algorithms*, number 594 in Lecture Notes in Computer Science, pages 70–86. Springer-Verlag, 1992.

[11] M. J. Mantyla and R. Sulonen. GWB: A solid modeler with Euler operators. *IEEE Computer Graphics and Applications*, 2(5):17–31, Sept. 1982.

[12] H. Meijer and D. Rappaport. Upper and lower bounds for the number of monotone crossing free Hamiltonian cycles from a set of points. *ARS Combinatoria*, 30:203–208, 1990.

[13] J. S. B. Mitchell and G. Sundaram. Generating random geometric objects. *Unpublished manuscript*, 1993.

[14] F. P. Preparata and M. Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[15] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, 1975.

[16] J. W. van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems*, 18(1):49–67, January 1991.