CMSC 754
Computational Geometry[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2002

Lecture 1: Introduction

What is Computational Geometry? Computational geometry is a term claimed by a number of different groups. The term was coined perhaps first by Marvin Minsky in his book "Perceptrons", which was about pattern recognition, and it has also been used often to describe algorithms for manipulating curves and surfaces in solid modeling. Its most widely recognized use, however, is to describe the subfield of algorithm theory that involves the design and analysis of efficient algorithms for problems involving geometric input and output.

The field of computational geometry developed rapidly in the late 70's and through the 80's and 90's, and it still continues to develop. Historically, computational geometry developed as a generalization of the study of algorithms for sorting and searching in 1-dimensional space to problems involving multi-dimensional inputs. It also developed to some extent as a restriction of computational graph theory by considering graphs that arise naturally in geometric settings (e.g., networks of roads or wires in the plane).

Because of its history, the field of computational geometry has focused mostly on problems in 2-dimensional space and to a lesser extent in 3-dimensional space. When problems are considered in multi-dimensional spaces, it is usually assumed that the dimension of the space is a small constant (say, 10 or lower). Because the field was developed by researchers whose training was in discrete algorithms (as opposed to numerical analysis) the field has also focused more on the discrete nature of geometric problems, as opposed to continuous issues. The field primarily deals with straight or flat objects (lines, line segments, polygons, planes, and polyhedra) or simple curved objects such as circles. This is in contrast, say, to fields such as solid modeling, which focus on problems involving more complex curves and surfaces.

A Typical Problem in Computational Geometry: Here is an example of a typical problem, called the *shortest path problem*. Given a set polygonal obstacles in the plane, find the shortest obstacle-avoiding path from some given start point to a given goal point. Although it is possible to reduce this to a shortest path problem on a graph (called the *visibility graph*, which we will discuss later this semester), and then apply a nongeometric algorithm such as Dijkstra's algorithm, it seems that by solving the problem in its geometric domain it should be possible to devise more efficient solutions. This is one of the main reasons for the growth of interest in geometric algorithms.

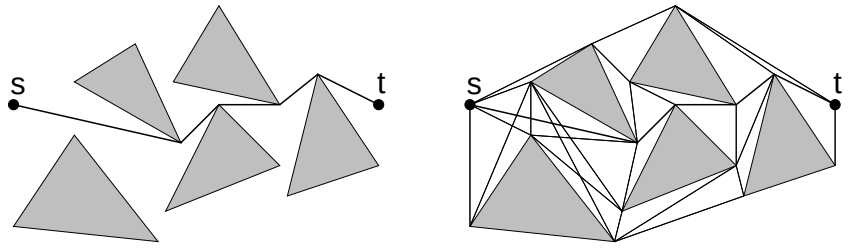


Figure 1: Shortest path problem.

The measure of the quality of an algorithm in computational geometry has traditionally been its *asymptotic worst-case running time*. Thus, an algorithm running in $O(n)$ time is better than one running in $O(n \log n)$ time which is better than one running in $O(n^2)$ time. (This particular problem can be solved in $O(n^2 \log n)$ time by a fairly simple algorithm, in $O(n \log n)$ by a relatively complex algorithm, and it can be approximated quite well by an algorithm whose running time is $O(n \log n)$.) In some cases *average case* running time is considered instead. However, for many types of geometric inputs it is difficult to define input distributions that are both easy to analyze and representative of typical inputs.

There are many fields of computer science that deal with solving problems of a geometric nature. These include computer graphics, computer vision and image processing, robotics, computer-aided design and manufacturing, computational fluid-dynamics, and geographic information systems, to name a few. One of the goals of computational geometry is to provide the basic geometric tools needed from which application areas can then build

their programs. There has been significant progress made towards this goal, but it is still far from being fully realized.

Limitations of Computational Geometry: There are some fairly natural reasons why computational geometry may never fully address the needs of all these applications areas, and these limitations should be understood before undertaking this course. One is the *discrete nature* of computational geometry. In some sense any problem that is solved on digital computers must be expressed in a discrete form, but many applications areas deal with discrete approximations to continuous phenomenon. For example, in image processing the image may be a discretization of a continuous 2-dimensional gray-scale function, and in robotics issues of vibration, oscillation in dynamic control systems are of a continuous nature. Nonetheless, there are many applications in which objects are of a very discrete nature. For example, in geographic information systems, road networks are discretized into collections of line segments.

The other limitation is the fact that computational geometry deals primarily with straight or flat objects. To a large extent, this is a result of the fact that computational geometers were not trained in geometry, but in discrete algorithm design. So they chose problems for which geometry and numerical computation plays a fairly small role. Much of solid modeling, fluid dynamics, and robotics, deals with objects that are modeled with curved surfaces. However, it is possible to approximate curved objects with piecewise planar polygons or polyhedra. This assumption has freed computational geometry to deal with the combinatorial elements of most of the problems, as opposed to dealing with numerical issues. This is one of the things that makes computational geometry fun to study, you do not have to learn a lot of analytic or differential geometry to do it. But, it does limit the applications of computational geometry.

One more limitation is that computational geometry has focused primarily on 2-dimensional problems, and 3-dimensional problems to a limited extent. The nice thing about 2-dimensional problems is that they are easy to visualize and easy to understand. But many of the daunting applications problems reside in 3-dimensional and higher dimensional spaces. Furthermore, issues related to topology are much cleaner in 2- and 3-dimensional spaces than in higher dimensional spaces.

Trends in CG in the 80's and 90's: In spite of these limitations, there is still a remarkable array of interesting problems that computational geometry has succeeded in addressing. Throughout the 80's the field developed many techniques for the design of efficient geometric algorithms. These include well-known methods such as divide-and-conquer and dynamic programming, along with a number of newly discovered methods that seem to be particularly well suited to geometric algorithm. These include plane-sweep, randomized incremental constructions, duality-transformations, and fractional cascading.

One of the major focuses of this course will be on understanding technique for designing efficient geometric algorithms. A major part of the assignments in this class will consist of designing and/or analyzing the efficiency of problems of a discrete geometric nature.

However throughout the 80's there a nagging gap was growing between the "theory" and "practice" of designing geometric algorithms. The 80's and early 90's saw many of the open problems of computational geometry solved in the sense that theoretically optimal algorithms were developed for them. However, many of these algorithms were nightmares to implement because of the complexity of the algorithms and the data structures that they required. Furthermore, implementations that did exist were often sensitive to geometric degeneracies that caused them to produce erroneous results or abort. For example, a programmer designing an algorithm that computes the intersections of a set of line segments may not consider the situation when three line segments intersect in a single point. In this rare situation, the data structure being used may be corrupted, and the algorithm aborts.

Much of the recent work in computational geometry has dealt with trying to make the theoretical results of computational geometry accessible to practitioners. This has been done by simplifying existing algorithms, dealing with geometric degeneracies, and producing libraries of geometric procedures. This process is still underway. Whenever possible, we will discuss the simplest known algorithm for solving a problem. Often these algorithms will be randomized algorithms. We will also discuss (hopefully without getting too bogged down in

details) some of the techniques for dealing with degenerate situations in order to produce clean and yet robust geometric software.

Overview of the Semester: Here are some of the topics that we will discuss this semester.

Convex Hulls: Convexity is a very important geometric property. A geometric set is *convex* if for every two points in the set, the line segment joining them is also in the set. One of the first problems identified in the field of computational geometry is that of computing the smallest convex shape, called the *convex hull*, that encloses a set of points.

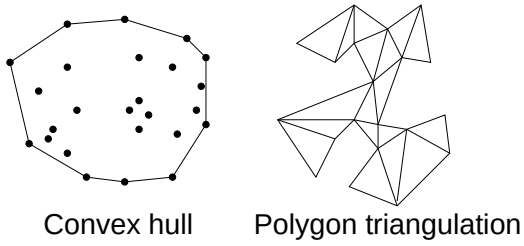


Figure 2: Convex hulls and polygon triangulation.

Intersections: One of the most basic geometric problems is that of determining when two sets of objects intersect one another. Determining whether complex objects intersect often reduces to determining which individual pairs of primitive entities (e.g., line segments) intersect. We will discuss efficient algorithms for computing the intersections of a set of line segments.

Triangulation and Partitioning: Triangulation is a catchword for the more general problem of subdividing a complex domain into a disjoint collection of "simple" objects. The simplest region into which one can decompose a planar object is a triangle (a *tetrahedron* in 3-d and *simplex* in general). We will discuss how to subdivide a polygon into triangles and later in the semester discuss more general subdivisions into trapezoids.

Low-dimensional Linear Programming: Many optimization problems in computational geometry can be stated in the form of a linear programming problem, namely, find the extreme points (e.g. highest or lowest) that satisfies a collection of linear inequalities. Linear programming is an important problem in the combinatorial optimization, and people often need to solve such problems in hundred to perhaps thousand dimensional spaces. However there are many interesting problems (e.g. find the smallest disc enclosing a set of points) that can be posed as low dimensional linear programming problems. In low-dimensional spaces, very simple efficient solutions exist.

Line arrangements and duality: Perhaps one of the most important mathematical structures in computational geometry is that of an arrangement of lines (or generally the arrangement of curves and surfaces). Given n lines in the plane, an arrangement is just the graph formed by considering the intersection points as vertices and line segments joining them as edges. We will show that such a structure can be constructed in $O(n^2)$ time. These reason that this structure is so important is that many problems involving points can be transformed into problems involving lines by a method of duality. For example, suppose that you want to determine whether any three points of a planar point set are collinear. This could be determines in $O(n^3)$ time by brute-force checking of each triple. However, if the points are dualized into lines, then (as we will see later this semester) this reduces to the question of whether there is a vertex of degree greater than 4 in the arrangement.

Voronoi Diagrams and Delaunay Triangulations: Given a set S of points in space, one of the most important problems is the nearest neighbor problem. Given a point that is not in S which point of S is closest to it? One of the techniques used for solving this problem is to subdivide space into regions, according to which point is closest. This gives rise to a geometric partition of space called a *Voronoi diagram*. This geometric

structure arises in many applications of geometry. The dual structure, called a *Delaunay triangulation* also has many interesting properties.

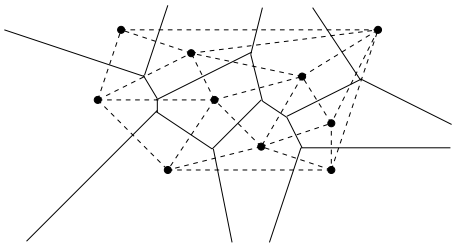


Figure 3: Voronoi diagram and Delaunay triangulation.

Search: Geometric search problems are of the following general form. Given a data set (e.g. points, lines, polygons) which will not change, preprocess this data set into a data structure so that some type of query can be answered as efficiently as possible. For example, a *nearest neighbor* search query is: determine the point of the data set that is closest to a given query point. A *range query* is: determine the set of points (or count the number of points) from the data set that lie within a given region. The region may be a rectangle, disc, or polygonal shape, like a triangle.

Lecture 2: Fixed-Radius Near Neighbors and Geometric Basics

Reading: The material on the Fixed-Radius Near Neighbor problem is taken from the paper: "The complexity of finding fixed-radius near neighbors," by J. L. Bentley, D. F. Stanat, and E. H. Williams, *Information Processing Letters*, 6(6), 1977, 209–212. The material on affine and Euclidean geometry is covered in many textbooks on basic geometry and computer graphics.

Fixed-Radius Near Neighbor Problem: As a warm-up exercise for the course, we begin by considering one of the oldest results in computational geometry. This problem was considered back in the mid 70's, and is a fundamental problem involving a set of points in dimension d. We will consider the problem in the plane, but the generalization to higher dimensions will be straightforward.

We assume that we are given a set P of n points in the plane. As will be our usual practice, we assume that each point p is represented by its (x, y) coordinates, denoted (p_x, p_y). The Euclidean distance between two points p and q, denoted $|pq|$ is

$$|pq| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Given the set P and a distance $r > 0$, our goal is to report all pairs of distinct points $p, q \in P$ such that $|pq| \leq r$. This is called the *fixed-radius near neighbor (reporting) problem*.

Reporting versus Counting: We note that this is a *reporting* problem, which means that our objective is to report all such pairs. This is in contrast to the corresponding *counting* problem, in which the objective is to return a count of the number of pairs satisfying the distance condition.

It is usually easier to solve reporting problems optimally than counting problems. This may seem counterintuitive at first (after all, if you can report, then you can certainly count). The reason is that we know that any algorithm that reports some number k of pairs must take at least $\Omega(k)$ time. Thus if k is large, a reporting algorithm has the luxury of being able to run for a longer time and still claim to be optimal. In contrast, we cannot apply such a lower bound to a counting algorithm.

Simple Observations: To begin, let us make a few simple observations. This problem can easily be solved in $O(n^2)$ time, by simply enumerating all pairs of distinct points and computing the distance between each pair. The number of distinct pairs of n points is

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Letting k denote the number of pairs that reported, our goal will be to find an algorithm whose running time is (nearly) linear in n and k, ideally $O(n+k)$. This will be optimal, since any algorithm must take the time to read all the input and print all the results. (This assumes a naive representation of the output. Perhaps there are more clever ways in which to encode the output, which would require less than $O(k)$ space.)

To gain some insight to the problem, let us consider how to solve the 1-dimensional version, where we are just given a set of n points on the line, say, $x_1, x_2, \ldots, x_n$. In this case, one solution would be to first sort the values in increasing order. Let suppose we have already done this, and so:

$$x_1 < x_2 < \ldots < x_n.$$

Now, for i running from 1 to n, we consider the successive points x_{i+1}, x_{i+2}, x_{i+3}, and so on, until we first find a point whose distance exceeds r. We report x_i together with all succeeding points that are within distance r.

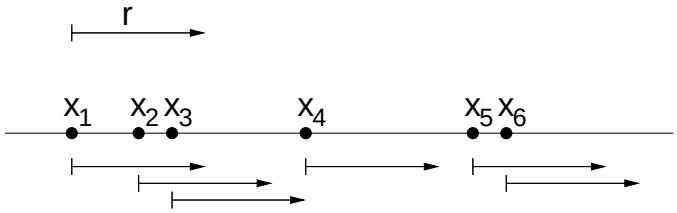


Figure 4: Fixed radius nearest neighbor on the line.

The running time of this algorithm involves the $O(n \log n)$ time needed to sort the points and the time required for distance computations. Let k_i denote the number of pairs generated when we visit p_i. Observe that the processing of p_i involves k_i+1 distance computations (one additional computation for the points whose distance exceeds r). Thus, up to constant factors, the total running time is:

$$\begin{aligned} T(n,k) &= n \log n + \sum_{i=1}^{n} (k_i + 1) \;=\; n \log n + n + \sum_{i=1}^{n} k_i \\ &= n \log n + n + k \;=\; O(k + n \log n). \end{aligned}$$

This is close to the $O(k+n)$ time we were hoping for. It seems that any approach based on sorting is doomed to take at least $\Omega(n \log n)$ time. So, if we are to improve upon this, we cannot sort. But is sorting really necessary? Let us consider an approach based on bucketing.

1-dimensional Solution with Bucketing: Rather than sorting the points, suppose that we subdivide the line into intervals of length r. In particular, we can take the line to be composed of an infinite collection of half-open intervals:

$$\ldots, [-3r, -2r),\ [-2r, -r),\ [-r, 0),\ [0, r),\ [r, 2r),\ [2r, 3r), \ldots$$

In general, interval b is $[br, (b+1)r)$.

A point x lies in the interval $b = \lfloor x/r \rfloor$. Thus, in $O(n)$ time we can associate the n points of P with a set of n integer bucket indices b_i. Although there are an infinite number of intervals, at most n will be *occupied*, meaning that they contain a point of P.

There are a number of ways to organize the occupied buckets. They could be sorted, but then we are back to $O(n \log n)$ time. Since bucket indices are integers, a better approach is to store the occupied bucket indices

in a hash table. Thus in $O(1)$ expected time, we can determine which bucket contains a given point and look this bucket up in the hash table. We can store all the points lying with any given bucket in an (unsorted) list associated with its bucket, in $O(n)$ total time.

The fact that the running time is in the expected case, rather than worst case is a bit unpleasant. However, it can be shown that by using a good randomized hash function, the probability that the total running time is worse than $O(n)$ can be made arbitrarily small. If the algorithm performs significantly more than the expected number of computations, we can simply chose a different random hash function and try again. This will lead to a very practical solution.

How does bucketing help? Observe that if point x lies in bucket b, then any successors that lie within distance r must lie either in bucket b or in $b+1$. This suggests the straightforward solution shown below.

Fixed-Radius Near Neighbor on the Line by Bucketing

(1) Store the points of P into buckets of size r, stored in a hash table.

(2) For each $x \in P$ do the following:

  (a) Let b be the bucket containing x.

  (b) Search all the points of buckets b and $b+1$, report x along with all those points x' that lie within distance r of x.

To avoid duplicates, we need only report pairs (x, x') where $x' > x$. The key question is determining the time complexity of this algorithm is how many distance computations are performed in step (2b). We compare each point in bucket b with all the points in buckets b and $b+1$. However, not all of these distance computations will result in a pair of points whose distance is within r. Might it be that we waste a great deal of time in performing computations for which we receive no benefit? The lemma below shows that we perform no more than a constant factor times as many distance computations and pairs that are produced.

It will simplify things considerably if, rather than counting distinct pairs of points, we simply count all (ordered) pairs of points that lie within distance r of each other. Thus each pair of points will be counted twice, (p, q) and (q, p). Note that this includes reporting each point as a pair (p, p) as well, since each point is within distance r of itself. This does not affect the asymptotic bounds, since the number of distinct pairs is smaller by a factor of roughly $1/2$.

Lemma: Let k denote the number of (not necessarily distinct) pairs of points of P that are within distance r of each other. Let D denote the total number distance computations made in step (2b) of the above algorithm. Then $D \leq 2k$.

Proof: We will make use of the following inequality in the proof:

$$xy \leq \frac{x^2 + y^2}{2}.$$

This follows by expanding the obvious inequality $(x - y)^2 \geq 0$.

Let B denote the (infinite) set of buckets. For any bucket $b \in B$, let $b+1$ denote its successor bucket on the line, and let n_b denote the number of points of P in b. Define

$$S = \sum_{b \in B} n_b^2.$$

First we bound the total number of distance computations D as a function of S. Each point in bucket b computes the distance to every other point in bucket b and every point in bucket $b+1$, and hence

$$D \;=\; \sum_{b \in B} n_b(n_b + n_{b+1}) \;=\; \sum_{b \in B} n_b^2 + n_b n_{b+1} \;=\; \sum_{b \in B} n_b^2 + \sum_{b \in B} n_b n_{b+1}$$

$$\leq \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2 + n_{b+1}^2}{2}$$

$$= \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2}{2} + \sum_{b\in B} \frac{n_{b+1}^2}{2} = S + \frac{S}{2} + \frac{S}{2} = 2S.$$

Next we bound the number of pairs reported from below as a function of S. Since each pair of points lying in bucket b is within distance r of every other, there are n_b^2 pairs in bucket b alone that are within distance r of each other, and hence (considering just the pairs generated within each bucket) we have $k \geq S$.

Therefore we have

$$D \leq 2S \leq 2k,$$

which completes the proof.

By combining this with the $O(n)$ expected time needed to bucket the points, it follows that the total expected running time is $O(n + k)$.

Generalization to d dimensions: This bucketing algorithm is easy to extend to multiple dimensions. For example, in dimension 2, we bucket points into a square grid in which each grid square is of side length r. The bucket index of a point (x, y) is a pair $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$. We apply a hash function that accepts two arguments. To generalize the algorithm, for each point we consider the points in its surrounding 3×3 subgrid of buckets. By generalizing the above arguements, it can be shown that the algorithm's expected running time is $O(n + k)$. The details are left as an exercise.

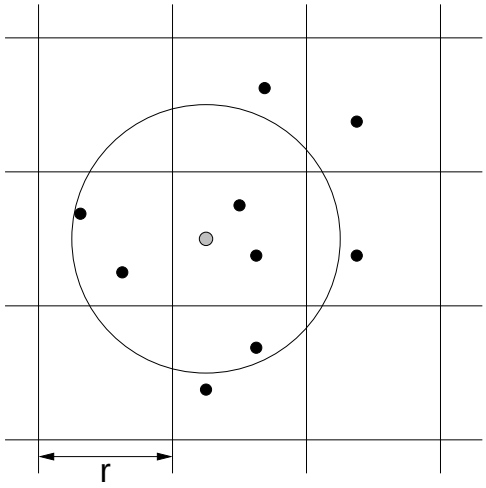


Figure 5: Fixed radius nearest neighbor on the plane.

This example problem serves to illustrate some of the typical elements of computational geometry. Geometry itself did not play a significant role in the problem, other than the relatively easy task of computing distances. We will see examples later this semester where geometry plays a much more important role. The major emphasis was on accounting for the algorithm's running time. Also note that, although we discussed the possibility of generalizing the algorithm to higher dimensions, we did not treat the dimension as an asymptotic quantity. In fact, a more careful analysis reveals that this algorithm's running time increases exponentially with the dimension. (Can you see why?)

Geometry Basics: As we go through the semester, we will introduce much of the geometric facts and computational primitives that we will be needing. For the most part, we will assume that any geometric primitive involving a

constant number of elements of constant complexity can be computed in $O(1)$ time, and we will not concern ourselves with how this computation is done. (For example, given three noncolinear points in the plane, compute the unique circle passing through these points.) Nonetheless, for a bit of completeness, let us begin with a quick review of the basic elements of affine and Euclidean geometry.

There are a number of different geometric systems that can be used to express geometric algorithms: affine geometry, Euclidean geometry, and projective geometry, for example. This semester we will be working almost exclusively with affine and Euclidean geometry. Before getting to Euclidean geometry we will first define a somewhat more basic geometry called *affine geometry*. Later we will add one operation, called an inner product, which extends affine geometry to Euclidean geometry.

Affine Geometry: An affine geometry consists of a set of *scalars* (the real numbers), a set of *points*, and a set of *free vectors* (or simply *vectors*). Points are used to specify position. Free vectors are used to specify direction and magnitude, but have no fixed position in space. (This is in contrast to linear algebra where there is no real distinction between points and vectors. However this distinction is useful, since the two are really quite different.)

The following are the operations that can be performed on scalars, points, and vectors. Vector operations are just the familiar ones from linear algebra. It is possible to subtract two points. The difference $p - q$ of two points results in a free vector directed from q to p. It is also possible to add a point to a vector. In point-vector addition $p + v$ results in the point which is translated by v from p. Letting S denote an generic scalar, V a generic vector and P a generic point, the following are the legal operations in affine geometry:

$$\begin{array}{rcll} S \cdot V & \rightarrow & V & \text{scalar-vector multiplication} \\ V + V & \rightarrow & V & \text{vector addition} \\ P - P & \rightarrow & V & \text{point subtraction} \\ P + V & \rightarrow & P & \text{point-vector addition} \end{array}$$

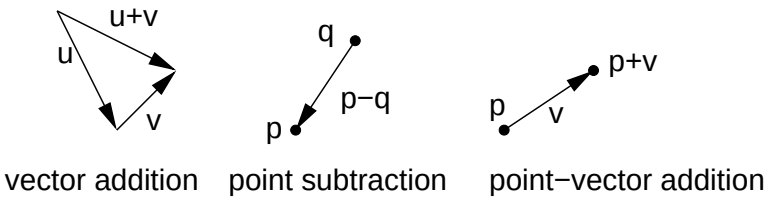


Figure 6: Affine operations.

A number of operations can be derived from these. For example, we can define the subtraction of two vectors $\vec{u} - \vec{v}$ as $\vec{u} + (-1) \cdot \vec{v}$ or scalar-vector division $\vec{v}/\alpha$ as $(1/\alpha) \cdot \vec{v}$ provided $\alpha \neq 0$. There is one special vector, called the *zero vector*, $\vec{0}$, which has no magnitude, such that $\vec{v} + \vec{0} = \vec{v}$.

Note that it is *not* possible to multiply a point times a scalar or to add two points together. However there is a special operation that combines these two elements, called an *affine combination*. Given two points p_0 and p_1 and two scalars α_0 and α_1, such that $\alpha_0 + \alpha_1 = 1$, we define the affine combination

$$\text{aff}(p_0, p_1; \alpha_0, \alpha_1) \;=\; \alpha_0 p_0 + \alpha_1 p_1 \;=\; p_0 + \alpha_1 (p_1 - p_0).$$

Note that the middle term of the above equation is not legal given our list of operations. But this is how the affine combination is typically expressed, namely as the weighted average of two points. The right-hand side (which is easily seen to be algebraically equivalent) is legal. An important observation is that, if $p_0 \neq p_1$, then

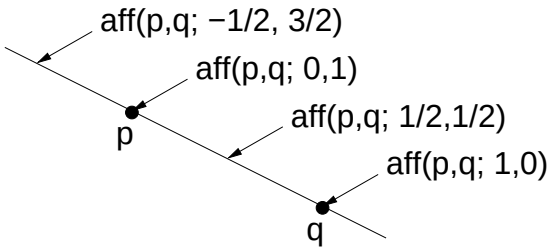


Figure 7: Affine combination.

the point $\mathrm{aff}(p_0, p_1; \alpha_0, \alpha_1)$ lies on the line joining p_0 and p_1. As α_1 varies from $-\infty$ to $+\infty$ it traces out all the points on this line.

In the special case where $0 \leq \alpha_0, \alpha_1 \leq 1$, $\mathrm{aff}(p_0, p_1; \alpha_0, \alpha_1)$ is a point that subdivides the line segment $\overline{p_0 p_1}$ into two subsegments of relative sizes α_1 to α_0. The resulting operation is called a *convex combination*, and the set of all convex combinations traces out the line segment $\overline{p_0 p_1}$.

It is easy to extend both types of combinations to more than two points, by adding the condition that the sum $\alpha_0 + \alpha_1 + \alpha_2 = 1$.

$$\mathrm{aff}(p_0, p_1, p_2; \alpha_0, \alpha_1, \alpha_2) \;=\; \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \;=\; p_0 + \alpha_1(p_1 - p_0) + \alpha_2(p_2 - p_0).$$

The set of all affine combinations of three (noncolinear) points generates a plane. The set of all convex combinations of three points generates all the points of the triangle defined by the points. These shapes are called the *affine span* or *affine closure*, and *convex closure* of the points, respectively.

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*, which maps two real vectors (not points) into a nonnegative real. One important example of an inner product is the *dot product*, defined as follows. Suppose that the d-dimensional vector $\vec{u}$ is represented by the (nonhomogeneous) coordinate vector $(u_1, u_2, \ldots, u_d)$. Then define

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{d-1} u_i v_i,$$

The dot product is useful in computing the following entities.

Length: of a vector $\vec{v}$ is defined to be $|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$.

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$. We will denote this by $\hat{v}$:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}.$$

Distance between points: Denoted either $\mathrm{dist}(p, q)$ or $|pq|$ is the length of the vector between them, $|p - q|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\mathrm{ang}(\vec{u}, \vec{v}) = \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right). = \cos^{-1}(\hat{u} \cdot \hat{v}).$$

This is easy to derive from the law of cosines.

Lecture 3: Orientations and Convex Hulls

Reading: Chapter 1 in the 4M's (de Berg, van Kreveld, Overmars, Schwarzkopf). The divide-and-conquer algorithm is given in Joseph O'Rourke's, "Computational Geometry in C." O'Rourke's book is also a good source for information about orientations and some other geometric primitives.

Orientation: In order to make discrete decisions, we would like a geometric operation that operates on points in a manner that is analogous to the relational operations $(<, =, >)$ with numbers. There does not seem to be any natural intrinsic way to compare two points in d-dimensional space, but there is a natural relation between ordered $(d+1)$-tuples of points in d-space, which extends the notion of binary relations in 1-space, called *orientation*.

Given an ordered triple of points $\langle p, q, r\rangle$ in the plane, we say that they have *positive orientation* if they define a counterclockwise oriented triangle, *negative orientation* if they define a clockwise oriented triangle, and *zero orientation* if they are collinear (which includes as well the case where two or more of the points are identical). Note that orientation depends on the order in which the points are given.

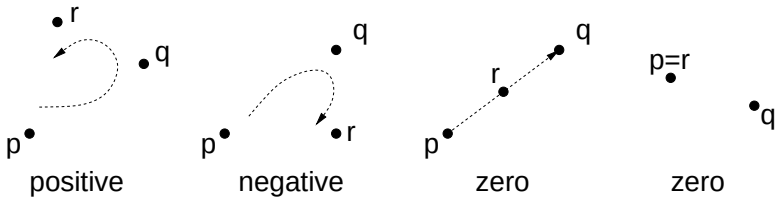


Figure 8: Orientations of the ordered triple (p, q, r).

Orientation is formally defined as the sign of the determinant of the points given in homogeneous coordinates, that is, by prepending a 1 to each coordinate. For example, in the plane, we define

$$\text{Orient}(p, q, r) = \det \begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}.$$

Observe that in the 1-dimensional case, Orient(p, q) is just $q - p$. Hence it is positive if $p < q$, zero if $p = q$, and negative if $p > q$. Thus orientation generalizes $<, =, >$ in 1-dimensional space. Also note that the sign of the orientation of an ordered triple is unchanged if the points are translated, rotated, or scaled (by a positive scale factor). A reflection transformation, e.g., $f(x, y) = (-x, y)$, reverses the sign of the orientation. In general, applying any affine transformation to the point alters the sign of the orientation according to the sign of the matrix used in the transformation.

In general, given an ordered 4-tuple points in 3-space, we can also define their orientation as being either positive (forming a right-handed screw), negative (a left-handed screw), or zero (coplanar). This can be generalized to any ordered $(d+1)$-tuple points in d-space.

Areas and Angles: The orientation determinant, together with the Euclidean norm can be used to compute angles in the plane. This determinant Orient(p, q, r) is equal to twice the signed area of the triangle $\triangle pqr$ (positive if CCW and negative otherwise). Thus the area of the triangle can be determined by dividing this quantity by 2. In general in dimension d the area of the simplex spanned by $d+1$ points can be determined by taking this determinant and dividing by $(d!)$. Once you know the area of a triangle, you can use this to compute the area of a polygon, by expressing it as the sum of triangle areas. (Although there are other methods that may be faster or easier to implement.)

Recall that the dot product returns the cosine of an angle. However, this is not helpful for distinguishing positive from negative angles. The sine of the angle $\theta = \angle pqr$ (the signed angled from vector $p - q$ to vector $r - q$) can

be computed as

$$\sin\theta = |p-q||r-q|\text{Orient}(q,p,r).$$

(Notice the order of the parameters.) By knowing both the sine and cosine of an angle we can unambiguously determine the angle.

Convexity: Now that we have discussed some of the basics, let us consider a fundamental structure in computational geometry, called the *convex hull*. We will give a more formal definition later, but the convex hull can be defined intuitively by surrounding a collection of points with a rubber band and letting the rubber band snap tightly around the points.

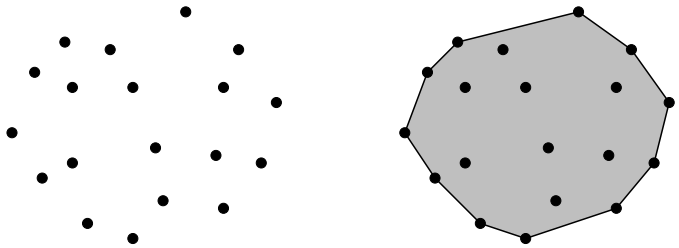


Figure 9: A point set and its convex hull.

There are a number of reasons that the convex hull of a point set is an important geometric structure. One is that it is one of the simplest shape approximations for a set of points. It can also be used for approximating more complex shapes. For example, the convex hull of a polygon in the plane or polyhedron in 3-space is the convex hull of its vertices. (Perhaps the most common shape approximation used in the minimum axis-parallel bounding box, but this is trivial to compute.)

Also many algorithms compute the convex hull as an initial stage in their execution or to filter out irrelevant points. For example, in order to find the smallest rectangle or triangle that encloses a set of points, it suffices to first compute the convex hull of the points, and then find the smallest rectangle or triangle enclosing the hull.

Convexity: A set S is *convex* if given any points $p, q \in S$ any convex combination of p and q is in S, or equivalently, the line segment $\overline{pq} \subseteq S$.

Convex hull: The *convex hull* of any set S is the intersection of all convex sets that contains S, or more intuitively, the smallest convex set that contains S. Following our book's notation, we will denote this $\mathcal{CH}(S)$.

An equivalent definition of convex hull is the set of points that can be expressed as convex combinations of the points in S. (A proof can be found in any book on convexity theory.) Recall that a convex combination of three or more points is an affine combination of the points in which the coefficients sum to 1 and all the coefficients are in the interval $[0, 1]$.

Some Terminology: Although we will not discuss topology with any degree of formalism, we will need to use some terminology from topology. These terms deserve formal definitions, but we are going to cheat and rely on intuitive definitions, which will suffice for the sort simple, well behaved geometry objects that we will be dealing with. Beware that these definitions are not fully general, and you are refered to a good text on topology for formal definitions. For our purposes, define a *neighborhood* of a point p to be the set of points whose distance to p is strictly less than some positive r, that is, it is the set of points lying within an open ball of radius r centered about p. Given a set S, a point p is an *interior point* of S if for some radius r the neighborhood about p of radius r is contained within S. A point is an *exterior point* if it an interior point for the complement of S.

Points that are neither interior or exterior are boundary points. A set is *open* if it contains none of its boundary points and *closed* if its complement is open. If p is in S but is not an interior point, we will call it a *boundary point*. We say that a geometric set is *bounded* if it can be enclosed in a ball of finite radius. A *compact set* is one that is both closed and bounded.

In general, convex sets may have either straight or curved boundaries and may be bounded or unbounded. Convex sets may be topologically open or closed. Some examples are shown in the figure below. The convex hull of a finite set of points in the plane is a bounded, closed, convex polygon.

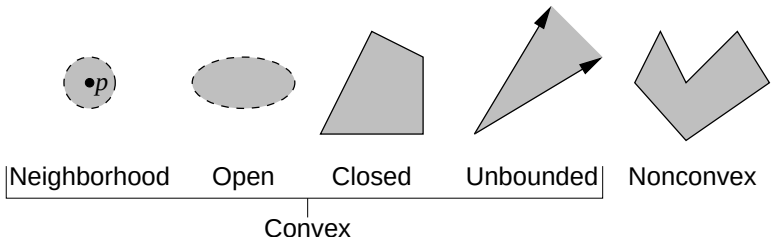


Figure 10: Terminology.

Convex hull problem: The (planar) *convex hull problem* is, given a set of n points P in the plane, output a representation of P's convex hull. The convex hull is a closed convex polygon, the simplest representation is a counterclockwise enumeration of the vertices of the convex hull. (A clockwise is also possible. We usually prefer counterclockwise enumerations, since they correspond to positive orientations, but obviously one representation is easily converted into the other.) Ideally, the hull should consist only of *extreme points*, in the sense that if three points lie on an edge of the boundary of the convex hull, then the middle point should not be output as part of the hull.

There is a simple $O(n^3)$ convex hull algorithm, which operates by considering each ordered pair of points (p, q), and the determining whether all the remaining points of the set lie within the half-plane lying to the right of the directed line from p to q. (Observe that this can be tested using the orientation test.) The question is, can we do better?

Graham's scan: We will present an $O(n \log n)$ algorithm for convex hulls. It is a simple variation of a famous algorithm for convex hulls, called *Graham's scan*. This algorithm dates back to the early 70's. The algorithm is based on an approach called *incremental construction*, in which points are added one at a time, and the hull is updated with each new insertion. If we were to add points in some arbitrary order, we would need some method of testing whether points are inside the existing hull or not. To avoid the need for this test, we will add points in increasing order of x-coordinate, thus guaranteeing that each newly added point is outside the current hull. (Note that Graham's original algorithm sorted points in a different way. It found the lowest point in the data set and then sorted points cyclically around this point.)

Since we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right. The convex hull is a cyclically ordered sets. Cyclically ordered sets are somewhat messier to work with than simple linearly ordered sets, so we will break the hull into two hulls, an *upper hull* and *lower hull*. The break points common to both hulls will be the leftmost and rightmost vertices of the convex hull. After building both, the two hulls can be concatenated into a single cyclic counterclockwise list.

Here is a brief presentation of the algorithm for computing the upper hull. We will store the hull vertices in a stack U, where the top of the stack corresponds to the most recently added point. Let first(U) and second(U) denote the top and second element from the top of U, respectively. Observe that as we read the stack from top to bottom, the points should make a (strict) left-hand turn, that is, they should have a positive orientation. Thus, after adding the last point, if the previous two points fail to have a positive orientation, we pop them off the stack. Since the orientations of remaining points on the stack are unaffected, there is no need to check any points other than the most recent point and its top two neighbors on the stack.

Let us consider the upper hull, since the lower hull is symmetric. Let $\langle p_1, p_2, \ldots, p_n \rangle$ denote the set of points, sorted by increase x-coordinates. As we walk around the upper hull from left to right, observe that each consecutive triple along the hull makes a right-hand turn. That is, if p, q, r are consecutive points along the upper hull, then Orient$(p, q, r) < 0$. When a new point p_i is added to the current hull, this may violate the right-hand turn

(1) Sort the points according to increasing order of their x-coordinates, denoted $\langle p_1, p_2, \ldots, p_n \rangle$.

(2) Push p_1 and then p_2 onto U.

(3) for $i = 3$ to n do:

  (a) while size$(U) \geq 2$ and Orient$(p_i, \text{first}(U), \text{second}(U)) \leq 0$, pop U.

  (b) Push p_i onto U.

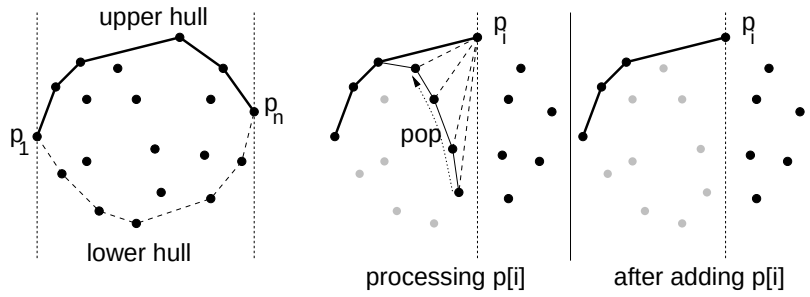


Figure 11: Convex hulls and Graham's scan.

invariant. So we check the last three points on the upper hull, including p_i. They fail to form a right-hand turn, then we delete the point prior to p_i. This is repeated until the number of points on the upper hull (including p_i) is less than three, or the right-hand turn condition is reestablished. See the text for a complete description of the code. We have ignored a number of special cases. We will consider these next time.

Analysis: Let us prove the main result about the running time of Graham's scan.

Theorem: Graham's scan runs in $O(n \log n)$ time.

Proof: Sorting the points according to x-coordinates can be done by any efficient sorting algorithm in $O(n \log n)$ time. Let D_i denote the number of points that are popped (deleted) on processing p_i. Because each orientation test takes $O(1)$ time, the amount of time spent processing p_i is $O(D_i + 1)$. (The extra $+1$ is for the last point tested, which is not deleted.) Thus, the total running time is proportional to

$$\sum_{i=1}^{n} (D_i + 1) \;=\; n + \sum_{i=1}^{n} D_i.$$

To bound $\sum_i D_i$, observe that each of the n points is pushed onto the stack once. Once a point is deleted it can never be deleted again. Since each of n points can be deleted at most once, $\sum_i D_i \leq n$. Thus after sorting, the total running time is $O(n)$. Since this is true for the lower hull as well, the total time is $O(2n) = O(n)$.

Convex Hull by Divide-and-Conquer: As with sorting, there are many different approaches to solving the convex hull problem for a planar point set P. Next we will consider another $O(n \log n)$ algorithm, which is based on the divide-and-conquer design technique. It can be viewed as a generalization of the famous MergeSort sorting algorithm (see Cormen, Leiserson, and Rivest). Here is an outline of the algorithm. It begins by sorting the points by their x-coordinate, in $O(n \log n)$ time.

The asymptotic running time of the algorithm can be expressed by a recurrence. Given an input of size n, consider the time needed to perform all the parts of the procedure, ignoring the recursive calls. This includes the time to partition the point set, compute the two tangents, and return the final result. Clearly the first and third of these steps can be performed in $O(n)$ time, assuming a linked list representation of the hull vertices. Below we

(1) If $|P| \leq 3$, then compute the convex hull by brute force in $O(1)$ time and return.

(2) Otherwise, partition the point set P into two sets A and B, where A consists of half the points with the lowest x-coordinates and B consists of half of the points with the highest x-coordinates.

(3) Recursively compute $H_A = \mathcal{CH}(A)$ and $H_B = \mathcal{CH}(B)$.

(4) Merge the two hulls into a common convex hull, H, by computing the upper and lower tangents for H_A and H_B and discarding all the points lying between these two tangents.

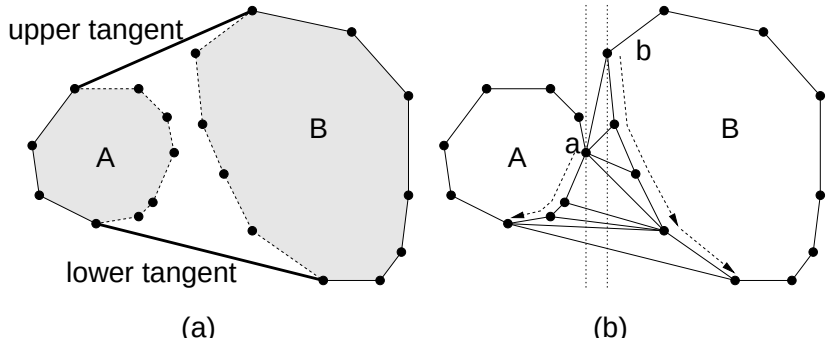


Figure 12: Computing the lower tangent.

will show that the tangents can be computed in $O(n)$ time. Thus, ignoring constant factors, we can describe the running time by the following recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ n + 2T(n/2) & \text{otherwise.} \end{cases}$$

This is the same recurrence that arises in Mergesort. It is easy to show that it solves to $T(n) \in O(n \log n)$ (see CLR). All that remains is showing how to compute the two tangents.

One thing that simplifies the process of computing the tangents is that the two point sets A and B are separated from each other by a vertical line (assuming no duplicate x-coordinates). Let's concentrate on the lower tangent, since the upper tangent is symmetric. The algorithm operates by a simple "walking" procedure. We initialize a to be the rightmost point of H_A and b is the leftmost point of H_B. (These can be found in linear time.) Lower tangency is a condition that can be tested locally by an orientation test of the two vertices and neighboring vertices on the hull. (This is a simple exercise.) We iterate the following two loops, which march a and b down, until they reach the points lower tangency.

Finding the Lower Tangent

LowerTangent(H_A, H_B) :

(1) Let a be the rightmost point of H_A.

(2) Let b be the leftmost point of H_B.

(3) While ab is not a lower tangent for H_A and H_B do

  (a) While ab is not a lower tangent to H_A do $a = a - 1$ (move a clockwise).

  (b) While ab is not a lower tangent to H_B do $b = b + 1$ (move b counterclockwise).

(4) Return ab.

Proving the correctness of this procedure is a little tricky, but not too bad. Check O'Rourke's book out for a careful proof. The important thing is that each vertex on each hull can be visited at most once by the search, and

hence its running time is $O(m)$, where $m = |H_A| + |H_B| \leq |A| + |B|$. This is exactly what we needed to get the overall $O(n \log n)$ running time.

Lecture 4: More Convex Hull Algorithms

Reading: Today's material is not covered in the 4M's book. It is covered in O'Rourke's book on Computational Geometry. Chan's algorithm can be found in T. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions", *Discrete and Computational Geometry*, 16, 1996, 361–368.

QuickHull: If the divide-and-conquer algorithm can be viewed as a sort of generalization of MergeSort, one might ask whether there is corresponding generalization of other sorting algorithm for computing convex hulls. In particular, the next algorithm that we will consider can be thought of as a generalization of the QuickSort sorting procedure. The resulting algorithm is called QuickHull.

Like QuickSort, this algorithm runs in $O(n \log n)$ time for favorable inputs but can take as long as $O(n^2)$ time for unfavorable inputs. However, unlike QuickSort, there is no obvious way to convert it into a randomized algorithm with $O(n \log n)$ expected running time. Nonetheless, QuickHull tends to perform very well in practice.

The intuition is that in many applications most of the points lie in the interior of the hull. For example, if the points are uniformly distributed in a unit square, then it can be shown that the expected number of points on the convex hull is $O(\log n)$.

The idea behind QuickHull is to discard points that are not on the hull as quickly as possible. QuickHull begins by computing the points with the maximum and minimum, x- and y-coordinates. Clearly these points must be on the hull. Horizontal and vertical lines passing through these points are support lines for the hull, and so define a bounding rectangle, within which the hull is contained. Furthermore, the convex quadrilateral defined by these four points lies within the convex hull, so the points lying within this quadrilateral can be eliminated from further consideration. All of this can be done in $O(n)$ time.

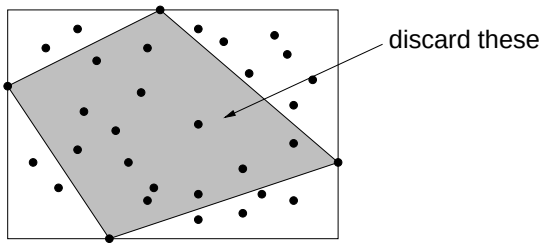


Figure 13: QuickHull's initial quadrilateral.

To continue the algorithm, we classify the remaining points into the four corner triangles that remain. In general, as this algorithm executes, we will have an inner convex polygon, and associated with each edge we have a set of points that lie "outside" of that edge. (More formally, these points are witnesses to the fact that this edge is not on the convex hull, because they lie outside the half-plane defined by this edge.) When this set of points is empty, the edge is a final edge of the hull. Consider some edge ab. Assume that the points that lie "outside" of this hull edge have been placed in a bucket that is associated with ab. Our job is to find a point c among these points that lies on the hull, discard the points in the triangle abc, and split the remaining points into two subsets, those that lie outside ac and those than lie outside of cb. We can classify each point by making two orientation tests.

How should c be selected? There are a number of possible selection criteria that one might think of. The method that is most often proposed is to let c be the point that maximizes the perpendicular distance from the line ab. (For example, another possible choice might be the point that maximizes the angle cba or cab. It turns out that

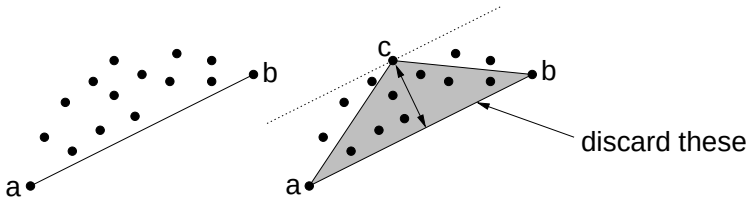


Figure 14: QuickHull elimination procedure.

these can be are very poor choices because they tend to produce imbalanced partitions of the remaining points.) We replace the edge ab with the two edges ac and cb, and classify the points as lying in one of three groups: those that lie in the triangle abc, which are discarded, those that lie outside of ac, and those that lie outside of cb. We put these points in buckets for these edges, and recurse. (We claim that it is not hard to classify each point p, by computing the orientations of the triples acp and cbp.)

The running time of Quickhull, as with QuickSort, depends on how evenly the points are split at each stage. Let $T(n)$ denote the running time on the algorithm assuming that n points remain outside of some edge. In $O(n)$ time we can select a candidate splitting point c and classify the points in the bucket in $O(n)$ time. Let n_1 and n_2 denote the number of remaining points, where $n_1 + n_2 \leq n$. Then the running time is given by the recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n_1) + T(n_2) & \text{where } n_1 + n_2 \leq n. \end{cases}$$

In order to solve this recurrence, it would be necessary to determine the "reasonable" values for n_1 and n_2. If we assume that the points are "evenly" distributed, in the sense that $\max(n_1, n_2) \leq \alpha n$ for some constant $\alpha < 1$, then by applying the same analysis as that used in QuickSort (see Cormen, Leiserson, Rivest) the running time will solve to $O(n \log n)$, where the constant factor depends on α. On the other hand, if the splits are not balanced, then the running time can easily increase to $O(n^2)$.

Does QuickHull outperform Graham's scan? This depends to a great extent on the distribution of the point set. There are variations of QuickHull that are designed for specific point distributions (e.g. points uniformly distributed in a square) and their authors claim that they manage to eliminate almost all of the points in a matter of only a few iterations.

Gift-Wrapping and Jarvis's March: The next algorithm that we will consider is a variant on an $O(n^2)$ sorting algorithm called SelectionSort. For sorting, this algorithm repeatedly finds the next element to add to the sorted order from the remaining items. The corresponding convex hull algorithm is called *Jarvis's march*. which builds the hull in $O(nh)$ time by a process called "gift-wrapping". The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the "next" edge on the hull in counterclockwise order. Assuming that p_k and p_{k-1} were the last two points added to the hull, compute the point q that maximizes the angle $\angle p_{k-1} p_k q$. Thus, we can find the point q in $O(n)$ time. After repeating this h times, we will return back to the starting point and we are done. Thus, the overall running time is $O(nh)$. Note that if h is $o(\log n)$ (asymptotically smaller than $\log n$) then this is a better method than Graham's algorithm.

One technical detail is that when we to find an edge from which to start. One easy way to do this is to let p_1 be the point with the lowest y-coordinate, and let p_0 be the point $(-\infty, 0)$, which is infinitely far to the right. The point p_0 is only used for computing the initial angles, after which it is discarded.

Output Sensitive Convex Hull Algorithms: It turns out that in the worst-case, convex hulls cannot be computed faster than in $\Omega(n \log n)$ time. One way to see this intuitively is to observe that the convex hull itself is sorted along its boundary, and hence if every point lies on the hull, then computing the hull requires sorting of some form. Yao proved the much harder result that determining which points are on the hull (without sorting them along the boundary) still requires $\Omega(n \log n)$ time. However both of these results rely on the fact that all (or at least a constant fraction) of the points lie on the convex hull. This is often not true in practice.

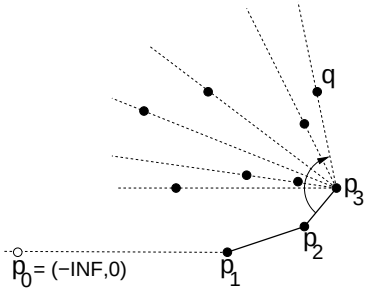


Figure 15: Jarvis's march.

The QuickHull and Jarvis's March algorithms that we saw last time suggest the question of how fast can convex hulls be computed if we allow the running time to be described in terms of both the input size n and the output size h. Many geometric algorithms have the property that the output size can be a widely varying function of the input size, and worst-case output size may not be a good indicator of what happens typically. An algorithm which attempts to be more efficient for small output sizes is called an *output sensitive algorithm*, and running time is described as a asymptotic function of both input size and output size.

Chan's Algorithm: Given than any convex hull algorithm must take at least $O(n)$ time, and given that "$\log n$" factor arises from the fact that you need to sort the at most n points on the hull, if you were told that there are only h points on the hull, then a reasonable target running time is $O(n \log h)$. (Below we will see that this is optimal.) Kirkpatrick and Seidel discovered a relatively complicated $O(n \log h)$ time algorithm, based on a clever pruning method in 1986. The problem was considered closed until around 10 years later when Timothy Chan came up with a much simpler algorithm with the same running time. One of the interesting aspects of Chan's algorithm is that it involves combining two slower algorithms (Graham's scan and Jarvis's March) together to form an algorithm that is faster than either one.

The problem with Graham's scan is that it sorts all the points, and hence is doomed to having an $\Omega(n \log n)$ running time, irrespective of the size of the hull. On the other hand, Jarvis's march can perform better if you have few vertices on the hull, but it takes $\Omega(n)$ time for each hull vertex.

Chan's idea was to partition the points into groups of equal size. There are m points in each group, and so the number of groups is $r = \lceil n/m \rceil$. For each group we compute its hull using Graham's scan, which takes $O(m \log m)$ time per group, for a total time of $O(rm \log m) = O(n \log m)$. Next, we run Jarvis's march on the groups. Here we take advantage of the fact that you can compute the tangent between a point and a convex m-gon in $O(\log m)$ time. (We will leave this as an exercise.) So, as before there are h steps of Jarvis's march, but because we are applying it to r convex hulls, each step takes only $O(r \log m)$ time, for a total of $O(hr \log m) = ((hn/m) \log m)$ time. Combining these two parts, we get a total of

$$O\left(\left(n + \frac{hn}{m}\right) \log m\right)$$

time. Observe that if we set $m = h$ then the total running time will be $O(n \log h)$, as desired.

There is only one small problem here. We do not know what h is in advance, and therefore we do not know what m should be when running the algorithm. We will see how to remedy this later. For now, let's imagine that someone tells us the value of m. The following algorithm works correctly as long as $m \geq h$. If we are wrong, it returns a special error status.

We assume that we store the convex hulls from step (2a) in an ordered array so that the step inside the for-loop of step (4a) can be solved in $O(\log m)$ time using binary search. Otherwise, the analysis follows directly from the comments made earlier.

Chan's Partial Convex Hull Algorithm (Given m)

PartialHull(P, m) :

(1) Let $r = \lceil n/m \rceil$. Partition P into disjoint subsets $P_1, P_2, \ldots, P_r$, each of size at most m.

(2) For $i = 1$ to r do:

  (a) Compute $\text{Hull}(P_i)$ using Graham's scan and store the vertices in an ordered array.

(3) Let $p_0 = (-\infty, 0)$ and let p_1 be the bottommost point of P.

(4) For $k = 1$ to m do:

  (a) For $i = 1$ to r do:

    - Compute point $q_i \in P_i$ that maximizes the angle $\angle p_{k-1} p_k q_i$.

  (b) Let p_{k+1} be the point $q \in \{q_1, \ldots, q_r\}$ that maximizes the angle $\angle p_{k-1} p_k q$.

  (c) If $p_{k+1} = p_1$ then return $\langle p_1, \ldots, p_k \rangle$.

(5) Return "m was too small, try again."

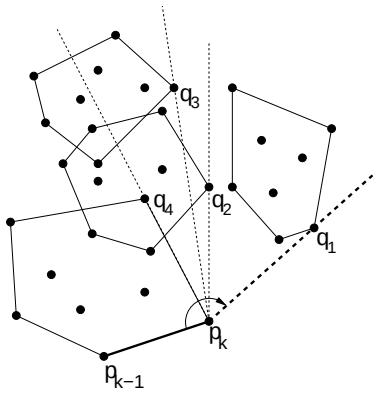


Figure 16: Chan's Convex Hull Algorithm.

The only question remaining is how do we know what value to give to m? We could try $m = 1, 2, 3, \ldots$, until we luck out and have $m \geq h$, but this would take too long. Binary search would be a more efficient option, but if we guess to large a value for m (e.g. $m = n/2$) then we are immediately stuck with $O(n \log n)$ time, and this is too slow.

Instead, the trick is to start with a small value of m and increase it rapidly. Since the dependence on m is only in the log term, as long as our value of m is within a polynomial of h, that is, $m = h^c$ for some constant c, then the running time will still be $O(n \log h)$. So, our approach will be to guess successively larger values of m, each time squaring the previous value, until the algorithm returns a successful result. This technique is often called *doubling search* (because the unknown parameter is successively doubled), but in this case we will be squaring rather than doubling.

Chan's Complete Convex Hull Algorithm

Hull(P) :

(1) For $t = 1, 2, \ldots$ do:
  (a) Let $m = \min(2^{2^t}, n)$.
  (b) Invoke PartialHull(P, m), returning the result in L.
  (c) If $L \neq$ "try again" then return L.

Note that 2^{2^t} has the effect of squaring the previous value of m. How long does this take? The t-th iteration takes $O(n \log 2^{2^t}) = O(n2^t)$ time. We know that it will stop as soon as $2^{2^t} \geq h$, that is if $t = \lceil \lg \lg n \rceil$. (We will use lg to denote logarithm base 2.) So the total running time (ignoring the constant factors) is

$$\sum_{t=1}^{\lg \lg h} n2^t \;=\; n \sum_{t=1}^{\lg \lg h} 2^t \;\leq\; n2^{1+\lg \lg h} \;=\; 2n \lg h \;=\; O(n \log h),$$

which is just what we want.

Lecture 5: Line Segment Intersection

Reading: Chapter 2 in the 4M's.

Geometric intersections: One of the most basic problems in computational geometry is that of computing intersections. Intersection computation in 2- and 3-space is basic to many different application areas.

- In solid modeling people often build up complex shapes by applying various boolean operations (intersection, union, and difference) to simple primitive shapes. The process in called *constructive solid geometry* (CSG). In order to perform these operations, the most basic step is determining the points where the boundaries of the two objects intersect.
- In robotics and motion planning it is important to know when two objects intersect for *collision detection* and *collision avoidance*.
- In geographic information systems it is often useful to *overlay* two subdivisions (e.g. a road network and county boundaries to determine where road maintenance responsibilities lie). Since these networks are formed from collections of line segments, this generates a problem of determining intersections of line segments.
- In computer graphics, *ray shooting* is an important method for rendering scenes. The computationally most intensive part of ray shooting is determining the intersection of the ray with other objects.

Most complex intersection problems are broken down to successively simpler and simpler intersection problems. Today, we will discuss the most basic algorithm, upon which most complex algorithms are based.

Line segment intersection: The problem that we will consider is, given n line segments in the plane, report all points where a pair of line segments intersect. We assume that each line segment is represented by giving the coordinates of its two endpoints.

Observe that n line segments can intersect in as few as 0 and as many as $\binom{n}{2} = O(n^2)$ different intersection points. We could settle for an $O(n^2)$ algorithm, claiming that it is worst-case asymptotically optimal, but it would not be very useful in practice, since in many instances of intersection problems intersections may be rare. Therefore it seems reasonable to look for an *output sensitive algorithm*, that is, one whose running time should be efficient both with respect to input and output size.

Let I denote the number of intersections. We will assume that the line segments are in general position, and so we will not be concerned with the issue of whether three or more lines intersect in a single point. However, generalizing the algorithm to handle such degeneracies efficiently is an interesting exercise. See our book for more discussion of this.

Complexity: We claim that best worst-case running time that one might hope for is $O(n \log n + I)$ time algorithm. Clearly we need $O(I)$ time to output the intersection points. What is not so obvious is that $O(n \log n)$ time is needed. This results from the fact that the following problem is known to require $\Omega(n \log n)$ time in the algebraic decision tree model of computation.

Element uniqueness: Given a list of n real numbers, are all of these numbers distinct? (That is, are there no duplicates.)

Given a list of n numbers, $(x_1, x_2, \ldots, x_n)$, in $O(n)$ time we can construct a set of n vertical line segments, all having the same y-coordinates. Observe that if the numbers are distinct, then there are no intersections and otherwise there is at least one intersection. Thus, if we could detect intersections in $o(n \log n)$ time (meaning strictly faster than $\Theta(n \log n)$ time) then we could solve element uniqueness in faster than $o(n \log n)$ time. However, this would contradict the lower bound on element uniqueness.

We will present a (not quite optimal) $O(n \log n + I \log n)$ time algorithm for the line segment intersection problem. A natural question is whether this is optimal. Later in the semester we will discuss an optimal radomized $O(n \log n + I)$ time algorithm for this problem.

Line segment intersection: Like many geometric primitives, computing the point at which two line segments intersect can be reduced to solving a linear system of equations. Let $\overline{ab}$ and $\overline{cd}$ be two line segments, given by their endpoints. First observe that it is possible to determine *whether* these line segments intersect, simply by applying an appropriate combination of orientation tests. (We will leave this as an exercise.)

One way to determine the point at which the segments intersect is to use an old trick from computer graphics. We represent the segments using a *parametric representation*. Recall that any point on the line segment $\overline{ab}$ can be written as a convex combination involving a real parameter s:

$$p(s) = (1 - s)a + sb \qquad \text{for } 0 \leq s \leq 1.$$

Similarly for $\overline{cd}$ we may introduce a parameter t:

$$q(t) = (1 - t)c + td \qquad \text{for } 0 \leq s \leq 1.$$

An intersection occurs if and only if we can find s and t in the desired ranges such that $p(s) = q(t)$. Thus we get the two equations:

$$\begin{aligned} (1 - s)a_x + sb_x &= (1 - t)c_x + td_x \\ (1 - s)a_y + sb_y &= (1 - t)c_y + td_y. \end{aligned}$$

The coordinates of the points are all known, so it is just a simple exercise in linear algebra to solve for s and t. The computation of s and t will involve a division. If the divisor is 0, this means that the line segments are

either parallel or collinear. These special cases should be dealt with carefully. If the divisor is nonzero, then we get values for s and t as rational numbers (the ratio of two integers). We can approximate them as floating point numbers, or if we want to perform exact computations it is possible to simulate rational number algebra exactly using high-precision integers (and multiplying through by least common multiples). Once the values of s and t have been computed all that is needed is to check that both are in the interval $[0, 1]$.

Plane Sweep Algorithm: Let $S = \{s_1, s_2, \ldots, s_n\}$ denote the line segments whose intersections we wish to compute. The method is called *plane sweep*. Here are the main elements of any plane sweep algorithm, and how we will apply them to this problem:

Sweep line: We will simulate the sweeping of a vertical line ℓ, called the *sweep line* from left to right. (Our text uses a horizontal line, but there is obviously no significant difference.) We will maintain the line segments that intersect the sweep line in sorted order (say from top to bottom).

Events: Although we might think of the sweep line as moving continuously, we only need to update data structures at points of some significant change in the sweep-line contents, called *event points*.

Different applications of plane sweep will have different notions of what event points are. For this application, event points will correspond to the following:

Endpoint events: where the sweep line encounters an endpoint of a line segment, and

Intersection events: where the sweep line encounters an intersection point of two line segments.

Note that endpoint events can be presorted before the sweep runs. In contrast, intersection events will be discovered as the sweep executes. For example, in the figure below, some of the intersection points lying to the right of the sweep line have not yet been "discovered" as events. However, we will show that every intersection point will be discovered as an event before the sweep line arrives here.

Event updates: When an event is encountered, we must update the data structures associated with the event. It is a good idea to be careful in specifying exactly what invariants you intend to maintain. For example, when we encounter an intersection point, we must interchange the order of the intersecting line segments along the sweep line.

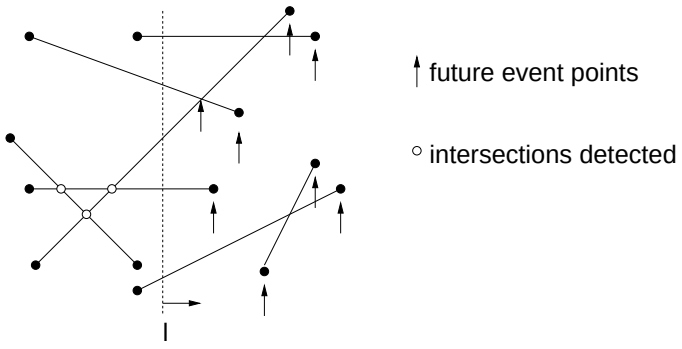


Figure 17: Plane sweep.

There are a great number of nasty special cases that complicate the algorithm and obscure the main points. We will make a number of simplifying assumptions. They can be overcome through a more careful handling of these cases.

(1) No line segment is vertical.

(2) If two segments intersect, then they intersect in a single point (that is, they are not collinear).

(3) No three lines intersect in a common point.

Detecting intersections: We mentioned that endpoint events are all known in advance. But how do we detect intersection events. It is important that each event be detected before the actual event occurs. Our strategy will be as follows. Whenever two line segments become adjacent along the sweep line, we will check whether they have an intersection occuring to the right of the sweep line. If so, we will add this new event (assuming that it has not already been added).

A natural question is whether this is sufficient. In particular, if two line segments do intersect, is there necessarily some prior placement of the sweep line such that they are adjacent. Happily, this is the case, but it requires a proof.

Lemma: Given two segments s_i and s_j, which intersect in a single point p (and assuming no other line segment passes through this point) there is a placement of the sweep line prior to this event, such that s_i and s_j are adjacent along the sweep line (and hence will be tested for intersection).

Proof: From our general position assumption it follows that no three lines intersect in a common point. Therefore if we consider a placement of the sweep line that is infinitessimally to the left of the intersection point, lines s_i and s_j will be adjacent along this sweepline. Consider the event point q with the largest x-coordinate that is strictly less than p_x. The order of lines along the sweep-line after processing q will be identical the order of the lines along the sweep line just prior p, and hence s_i and s_j will be adjacent at this point.

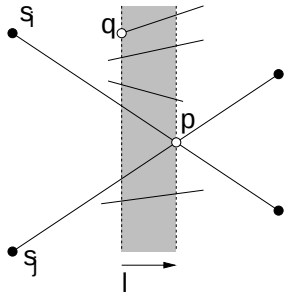


Figure 18: Correctness of plane sweep.

Data structures: In order to perform the sweep we will need two data structures.

Event queue: This holds the set of future events, sorted according to increasing x-coordinate. Each event contains the auxiliary information of what type of event this is (segment endpoint or intersection) and which segment(s) are involved. The operations that this data structure should support are inserting an event (if it is not already present in the queue) and extracting the minimum event.

It seems like a heap data structure would be ideal for this, since it supports insertions and extract-min in $O(\log M)$ time, where M is the number of entries in the queue. (See Cormen, Leiserson, and Rivest for details). However, a heap cannot support the operation of checking for duplicate events.

There are two ways to handle this. One is to use a more sophisticated data structure, such as a balanced binary tree or skip-list. This adds a small constant factor, but can check that there are no duplicates easily. The second is use the heap, but when an extraction is performed, you may have to perform many extractions to deal with multiple instances of the same event. (Our book recommends the prior solution.)

If events have the same x-coordinate, then we can handle this by sorting points lexicographically by (x, y). (This results in events be processed from bottom to top along the sweep line, and has the same geometric effect as imagining that the sweep line is rotated infinitesimally counterclockwise.)

Sweep line status: To store the sweep line status, we maintain a balanced binary tree or perhaps a skiplist whose entries are pointers to the line segments, stored in increasing order of y-coordinate along the current sweep line.

Normally when storing items in a tree, the key values are constants. Since the sweep line varies, we need "variable" keys. To do this, let us assume that each line segment computes a line equation $y = mx + b$ as part of its representation. The "key" value in each node of the tree is a pointer to a line segment. To compute the y-coordinate of some segment at the location of the current sweep line, we simply take the current x-coordinate of the sweep line, plug it into the line equation for this line, and use the resulting y-coordinate in our comparisons.

The operations that we need to support are those of deleting a line segment, inserting a line segment, swapping the position of two line segments, and determining the immediate predecessor and successor of any item. Assuming any balanced binary tree or a skiplist, these operations can be performed in $O(\log n)$ time each.

The complete plane-sweep algorithm is presented below. The various cases are illustrated in the following figure.

Plane-Sweep Algorithm for Line Segment Intersection

(1) Insert all of the endpoints of the line segments of S into the event queue. The initial sweep status is empty.

(2) While the event queue is nonempty, extract the next event in the queue. There are three cases, depending on the type of event:

Segment left endpoint: Insert this line segment into the sweep line status, based on the y-coordinate of this endpoint and the y-coordinates of the other segments currently along the sweep line. Test for intersections with the segment immediately above and below.

Segment right endpoint: Delete this line segment from the sweep line status. For the entries immediately preceding and succeeding this entry, test them for intersections.

Intersection point: Swap the two line segments in order along the sweep line. For the new upper segment, test it against its predecessor for an intersection. For the new lower segment, test it against its successor for an intersection.

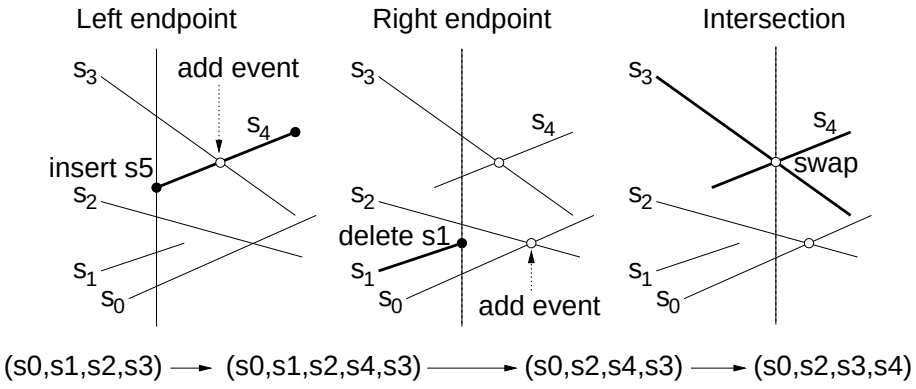


Figure 19: Plane-sweep algorithm events.

Analysis: The work done by the algorithm is dominated by the time spent updating the various data structures (since otherwise we spend only constant time per sweep event). We need to count two things: the number of operations applied to each data structure and the amount of time needed to process each operation.

For the sweep line status, there are at most n elements intersecting the sweep line at any time, and therefore the time needed to perform any single operation is $O(\log n)$, from standard results on balanced binary trees.

Since we do not allow duplicate events to exist in the event queue, the total number of elements in the queue at any time is at most $2n + I$. Since we use a balanced binary tree to store the event queue, each operation

takes time at most logarithmic in the size of the queue, which is $O(\log(2n + I))$. Since $I \leq n^2$, this is at most $O(\log n^2) = O(2 \log n) = O(\log n)$ time.

Each event involves a constant number of accesses or operations to the sweep status or the event queue, and since each such operation takes $O(\log n)$ time from the previous paragraph, it follows that the total time spent processing all the events from the sweep line is

$$O((2n + I) \log n) = O((n + I) \log n) = O(n \log n + I \log n).$$

Thus, this is the total running time of the plane sweep algorithm.

Lecture 6: Planar Graphs, Polygons and Art Galleries

Reading: Chapter 3 in the 4M's.

Topological Information: In many applications of segment intersection problems, we are not interested in just a listing of the segment intersections, but want to know how the segments are connected together. Typically, the plane has been subdivided into regions, and we want to store these regions in a way that allows us to reason about their properties efficiently.

This leads to the concept of a *planar straight line graph* (PSLG) or *planar subdivision* (or what might be called a *cell complex* in topology). A PSLG is a graph embedded in the plane with straight-line edges so that no two edges intersect, except possibly at their endpoints. (The condition that the edges be straight line segments may be relaxed to allow curved segments, but we will assume line segments here.) Such a graph naturally subdivides the plane into regions. The 0-dimensional *vertices*, 1-dimensional *edges*, and 2-dimensional *faces*. We consider these three types of objects to be disjoint, implying each edge is topologically open (it does not include it endpoints) and that each face is open (it does not include its boundary). There is always one unbounded face, that stretches to infinity. Note that the underlying planar graph need not be a connected graph. In particular, faces may contain holes (and these holes may contain other holes. A subdivision is called a *convex subdivision* if all the faces are convex.

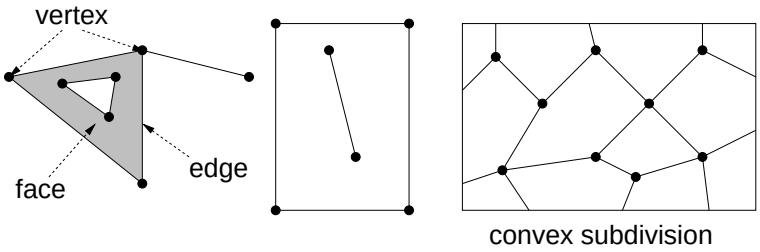


Figure 20: Planar straight-line subdivision.

Planar subdivisions form the basic objects of many different structures that we will discuss later this semester (triangulations and Voronoi diagrams in particular) so this is a good time to consider them in greater detail. The first question is how should we represent such structures so that they are easy to manipulate and reason about. For example, at a minimum we would like to be able to list the edges that bound each face of the subdivision in cyclic order, and we would like to be able to list the edges that surround each vertex.

Planar graphs: There are a number of important facts about planar graphs that we should discuss. Generally speaking, an (undirected) *graph* is just a finite set of vertices, and collection of unordered pairs of distinct vertices called *edges*. A graph is *planar* if it can be drawn in the plane (the edges need not be straight lines) so that no two distinct edges cross each other. An *embedding* of a planar graph is any such drawing. In fact, in specifying an embedding it is sufficient just to specify the counterclockwise cyclic list of the edges that are incident

to each vertex. Since we are interested in geometric graphs, our embeddings will contain complete geometric information (coordinates of vertices in particular).

There is an important relationship between the number of vertices, edges, and faces in a planar graph (or more generally an embedding of any graph on a topological 2-manifold, but we will stick to the plane). Let V denote the number of vertices, E the number of edges, F the number of faces in a connected planar graph. Euler's formula states that

$$V - E + F = 2.$$

The quantity $V - E + F$ is called the *Euler characteristic*, and is an invariant of the plane. In general, given a orientable topological 2-manifold with g handles (called the *genus*) we have

$$V - E + F = 2 - 2g.$$

Returning to planar graphs, if we allow the graph to be disconnected, and let C denote the number of connected components, then we have the somewhat more general formula

$$V - E + F - C = 1.$$

In our example above we have $V = 13$, $E = 12$, $F = 4$ and $C = 4$, which clearly satisfies this formula. An important fact about planar graphs follows from this.

Theorem: A planar graph with V vertices has at most $3(V - 2)$ edges and at most $2(V - 2)$ faces.

Proof: We assume (as is typical for graphs) that there are no multiple edges between the same pair of vertices and no self-loop edges.

We begin by *triangulating* the graph. For each face that is bounded by more than three edges (or whose boundary is not connected) we repeatedly insert new edges until every face in the graph is bounded by exactly three edges. (Note that this is not a "straight line" planar graph, but it is a planar graph, nonetheless.) An example is shown in the figure below in which the original graph edges are shown as solid lines.

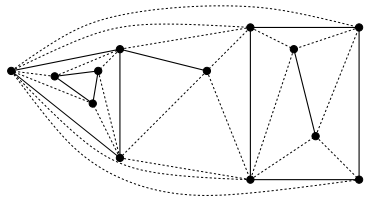


Figure 21: Triangulating a planar graph.

Let $E' \geq E$ and $F' \geq F$ denote the number edges and faces in the modified graph. The resulting graph has the property that it has one connected component, every face is bounded by exactly three edges, and each edge has a different face on either side of it. (The last claim may involve a little thought.)

If we count the number of faces and multiply by 3, then every edge will be counted exactly twice, once by the face on either side of the edge. Thus, $3F' = 2E'$, that is $E' = 3F'/2$. Euler's formula states that $V + E' - F' = 2$, and hence

$$V - \frac{3F'}{2} + F' = 2 \quad \Rightarrow \quad F \leq F' = 2(V - 2),$$

and using the face that $F' = 2E'/3$ we have

$$V - E' + \frac{2E'}{3} = 2 \quad \Rightarrow \quad E \leq E' = 3(V - 2).$$

This completes the proof.

The fact that the numbers of vertices, edges, and faces are related by constant factors seems to hold only in 2-dimensional space. For example, a polyhedral subdivision of 3-dimensional space that has n vertices can have as many as $\Theta(n^2)$ edges. (As a challenging exercise, you might try to create one.) In general, there are formulas, called the *Dehn-Sommerville equations* that relate the maximum numbers of vertices, edges, and faces of various dimensions.

There are a number of reasonable representations that for storing PSLGs. The most widely used on is the *winged-edge data structure*. Unfortunately, it is probably also the messiest. There is another called the *quad-edge data structure* which is quite elegant, and has the nice property of being self-dual. (We will discuss duality later in the semester.) We will not discuss any of these, but see our text for a presentation of the *doubly-connected edge list* (or *DCEL*) structure.

Simple Polygons: Now, let us change directions, and consider some interesting problems involving polygons in the plane. We begin study of the problem of triangulating polygons. We introduce this problem by way of a cute example in the field of combinatorial geometry.

We begin with some definitions. A *polygonal curve* is a finite sequence of line segments, called *edges* joined end-to-end. The endpoints of the edges are *vertices*. For example, let $v_0, v_2, \ldots, v_n$ denote the set of $n+1$ vertices, and let $e_1, e_2, \ldots, e_n$ denote a sequence of n edges, where $e_i = v_{i-1}v_i$. A polygonal curve is *closed* if the last endpoint equals the first $v_0 = v_n$. A polygonal curve is *simple* if it is not self-intersecting. More precisely this means that each edge e_i does not intersect any other edge, except for the endpoints it shares with its adjacent edges.

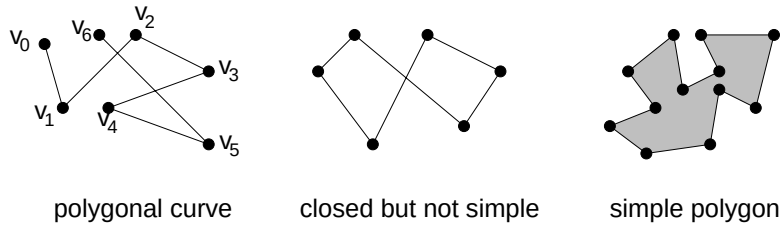


Figure 22: Polygonal curves

The famous *Jordan curve theorem* states that every simple closed plane curve divides the plane into two regions (the *interior* and the *exterior*). (Although the theorem seems intuitively obvious, it is quite difficult to prove.) We define a *polygon* to be the region of the plane bounded by a simple, closed polygonal curve. The term *simple polygon* is also often used to emphasize the simplicity of the polygonal curve. We will assume that the vertices are listed in counterclockwise order around the boundary of the polygon.

Art Gallery Problem: We say that two points x and y in a simple polygon can *see* each other (or x and y are *visible*) if the open line segment xy lies entirely within the interior of P. (Note that such a line segment can start and end on the boundary of the polygon, but it cannot pass through any vertices or edges.)

If we think of a polygon as the floor plan of an art gallery, consider the problem of where to place "guards", and how many guards to place, so that every point of the gallery can be seen by some guard. Victor Klee posed the following question: Suppose we have an art gallery whose floor plan can be modeled as a polygon with n vertices. As a function of n, what is the minimum number of guards that suffice to guard such a gallery? Observe that are you are told about the polygon is the number of sides, not its actual structure. We want to know the fewest number of guards that suffice to guard *all* polygons with n sides.

Before getting into a solution, let's consider some basic facts. Could there be polygons for which no finite number of guards suffice? It turns out that the answer is no, but the proof is not immediately obvious. You might consider placing a guard at each of the vertices. Such a set of guards will suffice in the plane. But to show how counterintuitive geometry can be, it is interesting to not that there are simple nonconvex polyhedra in

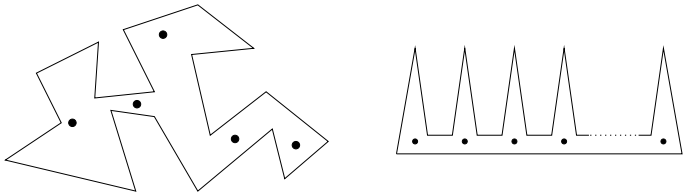


Figure 23: Guarding sets.

3-space, such that even if you place a guard at every vertex there would still be points in the polygon that are not visible to any guard. (As a challenge, try to come up with one with the fewest number of vertices.)

An interesting question in combinatorial geometry is how does the number of guards needed to guard any simple polygon with n sides grow as a function of n? If you play around with the problem for a while (trying polygons with $n = 3, 4, 5, 6 \ldots$ sides, for example) you will eventually come to the conclusion that $\lfloor n/3 \rfloor$ is the right value. The figure above shows a worst-case example, where $\lfloor n/3 \rfloor$ guards are required. A cute result from combinatorial geometry is that this number always suffices. The proof is based on three concepts: polygon triangulation, dual graphs, and graph coloring. The remarkably clever and simple proof was discovered by Fisk.

Theorem: (The Art-Gallery Theorem) Given a simple polygon with n vertices, there exists a guarding set with at most $\lfloor n/3 \rfloor$ guards.

Before giving the proof, we explore some aspects of polygon triangulations. We begin by introducing a triangulation of P. A *triangulation* of a simple polygon is a planar subdivision of (the interior of) P whose vertices are the vertices of P and whose faces are all triangles. An important concept in polygon triangulation is the notion of a *diagonal*, that is, a line segment between two vertices of P that are visible to one another. A triangulation can be viewed as the union of the edges of P and a maximal set of noncrossing diagonals.

Lemma: Every simple polygon with n vertices has a triangulation consisting of $n - 3$ diagonals and $n - 2$ triangles.

(We leave the proof as an exercise.) The proof is based on the fact that given any n-vertex polygon, with $n \geq 4$ it has a diagonal. (This may seem utterly trivial, but actually takes a little bit of work to prove. In fact it fails to hold for polyhedra in 3-space.) The addition of the diagonal breaks the polygon into two polygons, of say m_1 and m_2 vertices, such that $m_1 + m_2 = n + 2$ (since both share the vertices of the diagonal). Thus by induction, there are $(m_1 - 2) + (m_2 - 2) = n + 2 - 4 = n - 2$ triangles total. A similar argument holds for the case of diagonals.

It is a well known fact from graph theory that any planar graph can be colored with 4 colors. (The famous *4-color theorem*.) This means that we can assign a color to each of the vertices of the graph, from a collection of 4 different colors, so that no two adjacent vertices have the same color. However we can do even better for the graph we have just described.

Lemma: Let T be the triangulation graph of a triangulation of a simple polygon. Then T is 3-colorable.

Proof: For every planar graph G there is another planar graph G^* called its *dual*. The dual G^* is the graph whose vertices are the faces of G, and two vertices of G^* are connected by an edge if the two corresponding faces of G share a common edge.

Since a triangulation is a planar graph, it has a dual, shown in the figure below. (We do not include the external face in the dual.) Because each diagonal of the triangulation splits the polygon into two, it follows that each edge of the dual graph is a *cut edge*, meaning that its deletion would disconnect the graph. As a

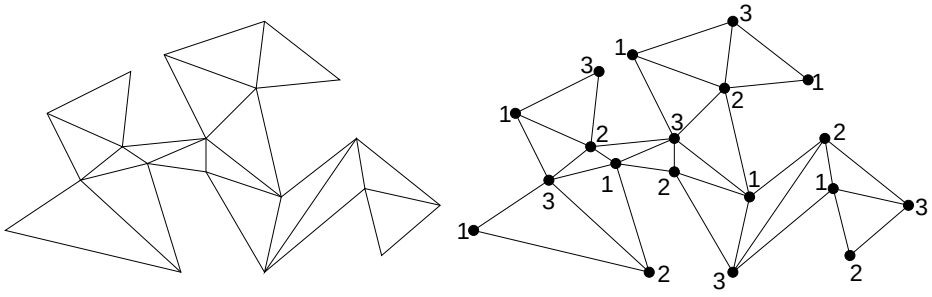


Figure 24: Polygon triangulation and a 3-coloring.

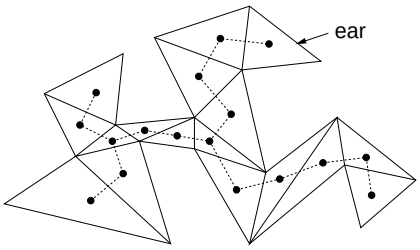


Figure 25: Dual graph of triangulation.

result it is easy to see that the dual graph is a *free tree* (that is, a connected, acyclic graph), and its maximum degree is 3. (This would not be true if the polygon had holes.)

The coloring will be performed inductively. If the polygon consists of a single triangle, then just assign any 3 colors to its vertices. An important fact about any free tree is that it has at least one leaf (in fact it has at least two). Remove this leaf from the tree. This corresponds to removing a triangle that is connected to the rest triangulation by a single edge. (Such a triangle is called an *ear*.) By induction 3-color the remaining triangulation. When you add back the deleted triangle, two of its vertices have already been colored, and the remaining vertex is adjacent to only these two vertices. Give it the remaining color. In this way the entire triangulation will be 3-colored.

We can now give the simple proof of the guarding theorem.

Proof: (of the Art-Gallery Theorem:) Consider any 3-coloring of the vertices of the polygon. At least one color occurs at most $\lfloor n/3 \rfloor$ time. (Otherwise we immediately get there are more than n vertices, a contradiction.) Place a guard at each vertex with this color. We use at most $\lfloor n/3 \rfloor$ guards. Observe that every triangle has at least one vertex of each of the tree colors (since you cannot use the same color twice on a triangle). Thus, every point in the interior of this triangle is guarded, implying that the interior of P is guarded. A somewhat messy detail is whether you allow guards placed at a vertex to see along the wall. However, it is not a difficult matter to push each guard infinitesimally out from his vertex, and so guard the entire polygon.

Lecture 7: Polygon Triangulation

Reading: Chapter 3 in the 4M's.

The Polygon Triangulation Problem: Triangulating simple polygons is an operation used in many other applications where complex shapes are to be decomposed into a set of disjoint simpler shapes. There are many applications

in which the shapes of the triangles is an important issue (e.g. skinny triangles should be avoided) but there are equally many in which the shape of the triangle is unimportant. We will consider the problem of, given an arbitrary simple polygon, compute any triangulation for the polygon.

This problem has been the focus of computational geometry for many years. There is a simple naive polynomial-time algorithm, based on adding successive diagonals, but it is not particularly efficient. There are very simple $O(n \log n)$ algorithms for this problem that have been known for many years. A longstanding open problem was whether there exists an $O(n)$ time algorithm. The problem was solved by Chazelle in 1991, but the algorithm is so amazingly intricate, it could never compete with the practical but asymptotically slower $O(n \log n)$ algorithms. In fact, there is no known algorithm that runs in less than $O(n \log n)$ time, that is really practical enough to replace the standard $O(n \log n)$ algorithm, which we will discuss.

We will present one of many known $O(n \log n)$ algorithms. The approach we present today is a two-step process (although with a little cleverness, both steps can be combined into one algorithm). The first is to consider the special case of triangulating a *monotone polygon*. After this we consider how to convert an arbitrary polygon into a collection of disjoint monotone polygons. Then we will apply the first algorithm to each of the monotone pieces. The former algorithm runs in $O(n)$ time. The latter algorithm runs in $O(n \log n)$ time.

Monotone Polygons: A polygonal chain C is said to be *strictly monotone* with respect to a given line L, if any line that is orthogonal to L intersects C in at most one point. A chain C is *monotone* with respect to L if each line that is orthogonal to L intersects C in a single connected component. Thus it may intersect, not at all, at a single point, or along a single line segment. A polygon P is said to be *monotone* with respect to a line L if its boundary, ∂P, can be split into two chains, each of which is monotone with respect to L.

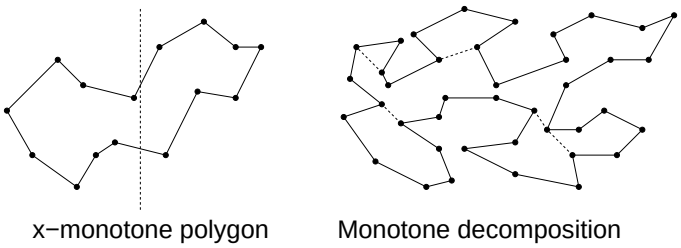


Figure 26: Monotonicity.

Henceforth, let us consider monotonicity with respect to the x-axis. We will call these polygons *horizontally monotone*. It is easy to test whether a polygon is horizontally monotone. How?

(a) Find the leftmost and rightmost vertices (min and max x-coordinate) in $O(n)$ time.

(b) These vertices split the polygon's boundary into two chains, an *upper chain* and a *lower chain*. Walk from left to right along each chain, verifying that the x-coordinates are nondecreasing. This takes $O(n)$ time.

As a challenge, consider the problem of determining whether a polygon is monotone in any (unspecified) direction. This can be done in $O(n)$ time, but is quite a bit harder.

Triangulation of Monotone Polygons: We can triangulate a monotone polygon by a simple variation of the plane-sweep method. We begin with the assumption that the vertices of the polygon have been sorted in increasing order of their x-coordinates. (For simplicity we assume no duplicate x-coordinates. Otherwise, break ties between the upper and lower chains arbitrarily, and within a chain break ties so that the chain order is preserved.) Observe that this does not require sorting. We can simply extract the upper and lower chain, and merge them (as done in mergesort) in $O(n)$ time.

The idea behind the triangulation algorithm is quite simple: Try to triangulate everything you can to the left of the current vertex by adding diagonals, and then remove the triangulated region from further consideration.

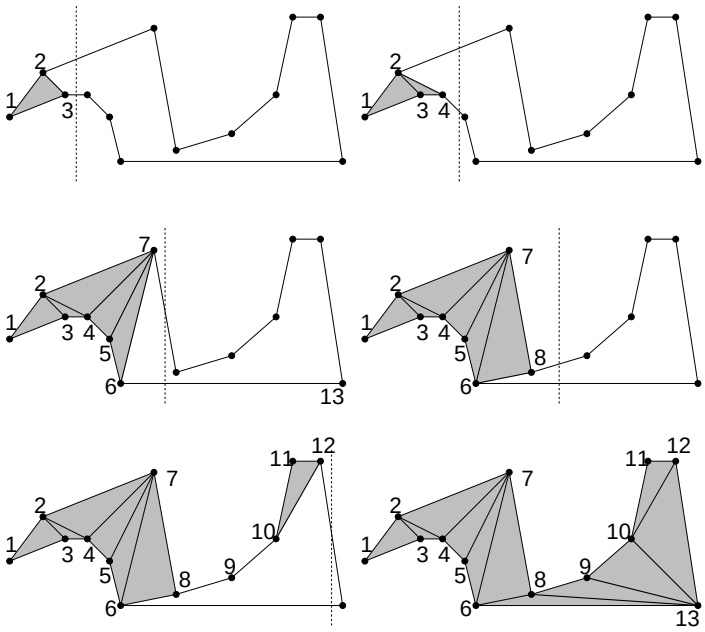


Figure 27: Triangulating a monotone polygon.

In the example, there is obviously nothing to do until we have at least 3 vertices. With vertex 3, it is possible to add the diagonal to vertex 2, and so we do this. In adding vertex 4, we can add the diagonal to vertex 2. However, vertices 5 and 6 are not visible to any other nonadjacent vertices so no new diagonals can be added. When we get to vertex 7, it can be connected to 4, 5, and 6. The process continues until reaching the final vertex.

The important thing that makes the algorithm efficient is the fact that when we arrive at a vertex the *untriangulated region* that lies to the left of this vertex always has a very simple structure. This structure allows us to determine in *constant time* whether it is possible to add another diagonal. And in general we can add each additional diagonal in constant time. Since any triangulation consists of $n - 3$ diagonals, the process runs in $O(n)$ total time. This structure is described in the lemma below.

Lemma: *(Main Invariant)* For $i \geq 2$, let v_i be the vertex just processed by the triangulation algorithm. The untriangulated region lying to the left of v_i consists of two x-monotone chains, a lower chain and an upper chain each containing at least one edge. If the chain from v_i to u has two or more edges, then these edges form a reflex chain (that is, a sequence of vertices with interior angles all at least 180 degrees). The other chain consists of a single edge whose left endpoint is u and whose right endpoint lies to the right of v_i.

We will prove the invariant by induction. As the basis case, consider the case of v_2. Here $u = v_1$, and one chain consists of the single edge v_2v_1 and the other chain consists of the other edge adjacent to v_1.

To prove the main invariant, we will give a case analysis of how to handle the next event, involving v_i, assuming that the invariant holds at v_{i-1}. and see that the invariant is satisfied after each event has been processed. There are the following cases that the algorithm needs to deal with.

Case 1: v_i lies on the opposite chain from v_{i-1}:

In this case we add diagonals joining v_i to all the vertices on the reflex chain, from v_{i-1} back to (but not including) u. Note that all of these vertices are visible from v_i. Certainly u is visible to v_i. Because the chain is reflex, x-monotone, and lies to the left of v_i it follows that the chain itself cannot block the visibility from v_i to some other vertex on the chain. Finally, the fact that the polygon is x-monotone

implies that the unprocessed portion of the polygon (lying to the right of v_i) cannot "sneak back" and block visibility to the chain.

After doing this, we set $u = v_{i-1}$. The invariant holds, and the reflex chain is trivial, consisting of the single edge $v_i v_{i-1}$.

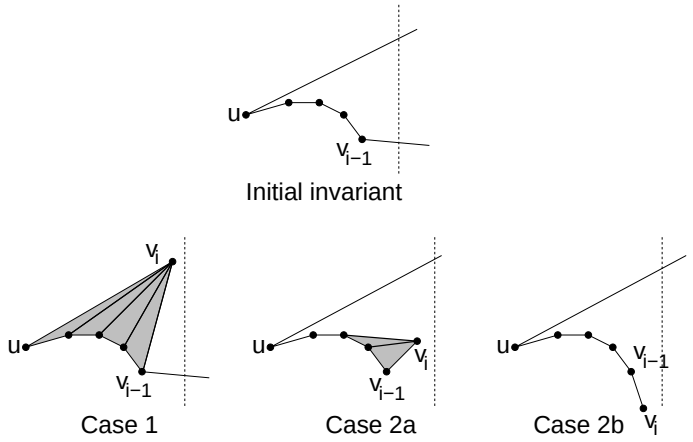


Figure 28: Triangulation cases.

Case 2: v is on the same chain as v_{i-1}:

We walk back along the reflex chain adding diagonals joining v_i to prior vertices until we find the first that is not visible to v_i. As can be seen in the figure, this may involve connecting v_i to one or more vertices (2a) or it may involve connecting v_i to no additional vertices (2b), depending on whether the first angle is less or greater than 180 degrees. In either case the vertices that were cut off by diagonals are no longer in the chain, and v_i becomes the new endpoint to the chain. Again, by x-monotonicity it follows that the unprocessed portion of the polygon cannot block visibility of v_i to the chain.

Note that when we are done the remaining chain from v_i to u is a reflex chain. (Note the similarity between this step and the main iteration in Graham's scan.)

How is this implemented? The vertices on the reflex chain can be stored in a stack. We keep a flag indicating whether the stack is on the upper chain or lower chain, and assume that with each new vertex we know which chain of the polygon it is on. Note that decisions about visibility can be based simply on orientation tests involving v_i and the top two entries on the stack. When we connect v_i by a diagonal, we just pop the stack.

Analysis: We claim that this algorithm runs in $O(n)$ time. As we mentioned earlier, the sorted list of vertices can be constructed in $O(n)$ time through merging. The reflex chain is stored on a stack. In $O(1)$ time per diagonal, we can perform an orientation test to determine whether to add the diagonal and (assuming a DCEL) the diagonal can be added in constant time. Since the number of diagonals is $n - 3$, the total time is $O(n)$.

Monotone Subdivision: In order to run the above triangulation algorithm, we first need to subdivide an arbitrary simple polygon P into monotone polygons. This is also done by a plane-sweep approach. We will add a set of nonintersecting diagonals that partition the polygon into monotone pieces.

Observe that the absence of x-monotonicity occurs only at vertices in which the interior angle is greater than 180 degrees and both edges lie either to the left of the vertex or both to the right. Following our book's notation, we call the first type a *merge vertex* (since as the sweep passes over this vertex the edges seem to be merging) and the latter type a *split vertex*.

Let's discuss the case of a split vertex first (both edges lie to the right of the vertex). When a split vertex v is encountered in the sweep, there will be an edge e_a of the polygon lying above and an edge e_b lying below. We

might consider attaching the split vertex to left endpoint of one of these two edges, but it might be that neither endpoint is visible to the split vertex. We need to maintain a vertex that is visible to any split vertex that may arise between e_a and e_b. To do this, imagine a fluorescent light shining down from every point on e_a (these are the white vertices in the figure below, left). Note that e_a left endpoint is considered in this set as well. We say that these vertices are *vertically visible below* e_a. Among all the vertices that are illuminated by these vertical light rays, let u be the rightmost. We claim that u is visible to every point along the sweep line between e_a and e_b. This suggests the following concept, which is defined for each edge e_a that intersects the sweep line, such that the polygon's interior lies locally below e_a.

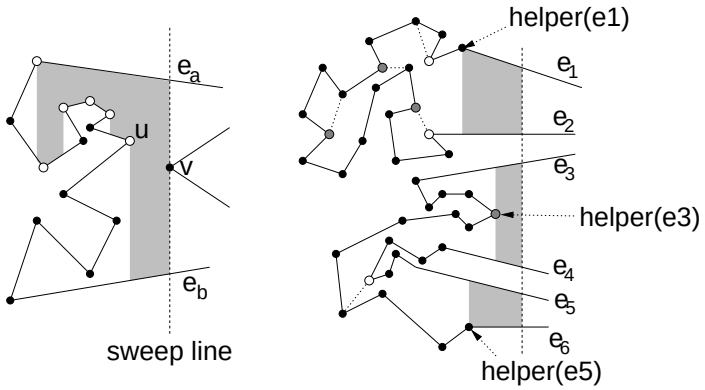


Figure 29: Split vertices, merge vertices, and helpers.

helper(e_a) : Let e_b be the edge of the polygon lying just below e_a on the sweep line. The helper is the rightmost vertically visible vertex below e_a on the polygonal chain between e_a and e_b.

We join each split vertex to *helper*(e_a), where e_a is the edge of P immediately above the split vertex. Note that it is possible that the helper is the left endpoint of e_a. Also note that *helper*(e_a) is defined with respect to the current location of the sweep line. As the sweep line moves, its value changes. Also, it is only defined for those edges intersected by the sweep line.

Events: The endpoints of the edges of the polygon. These are sorted by increasing order of x-coordinates. Since no new events are generated, the events may be stored in a simple sorted list (i.e., no priority queue is needed).

Sweep status: The sweep line status consists of the list of edges that intersect the sweep line, sorted from top to bottom. Our book notes that we actually only need to store edges such that the polygon lies just below this edge (since these are the only edges that we evaluate *helper*$()$ from).

These edges are stored in a dictionary (e.g., a balanced binary tree or a skip list), so that the operations of insert, delete, find, predecessor and successor can be evaluated in $O(\log n)$ time each.

Event processing: There are six event types based on a case analysis of the local structure of edges around each vertex. Let v be the current vertex encountered by the sweep.

Split vertex: Search the sweep line status to find the edge e lying immediately above v. Add a diagonal connecting v to *helper*(e). Add the two edges incident to v in the sweep line status, and make v the helper of the lower of these two edges and make v the new helper of e.

Merge vertex: Find the two edges incident to this vertex in the sweep line status (they must be adjacent). Delete them both. Let e be the edge lying immediately above them. Make v the new helper of e.

Start vertex: (Both edges lie to the right of v, but the interior angle is less than 180 degrees.) Insert this vertex's edges into the sweep line status. Set the helper of the upper edge to v.

End vertex: (Both edges lie to the left of v, but the interior angle is less than 180 degrees.) Delete both edges from the sweep line status.

Upper-chain vertex: (One edge is to the left, and one to the right, and the polygon interior is below.) Replace the left edge with the right edge in the sweep line status. Make v the helper of the new edge.

Lower-chain vertex: (One edge is to the left, and one to the right, and the polygon interior is above.) Replace the left edge with the right edge in the sweep line status. Let e be the edge lying above here. Make v the helper of e.

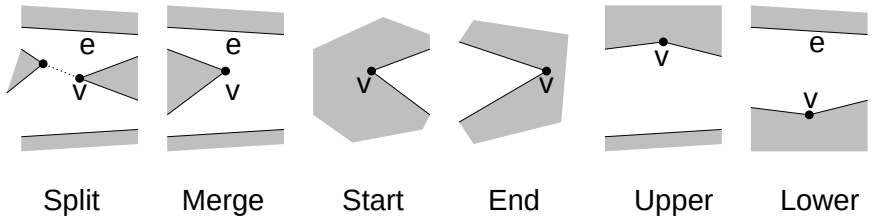


Figure 30: Plane sweep cases.

This only inserts diagonals to fix the split vertices. What about the merge vertices? This could be handled by applying essentially the same algorithm using a reverse (right to left) sweep. It can be shown that this will never introduce crossing diagonals, but it might attempt to insert the same diagonal twice. However, the book suggests a simpler approach. Whenever we change a helper vertex, check whether the original helper vertex is a merge vertex. If so, the new helper vertex is then connected to the merge vertex by a new diagonal. It is not hard to show that this essentially has the same effect as a reverse sweep, and it is easier to detect the possibility of a duplicate insertion (in case the new vertex happens to be a split vertex).

There are many special cases (what a pain!), but each one is fairly easy to deal with, so the algorithm is quite efficient. As with previous plane sweep algorithms, it is not hard to show that the running time is $O(\log n)$ times the number of events. In this case there is one event per vertex, so the total time is $O(n \log n)$. This gives us an $O(n \log n)$ algorithm for polygon triangulation.

Lecture 8: Halfplane Intersection

Reading: Chapter 4 in the 4M's, with some elements from Sections 8.2 and 11.4.

Halfplane Intersection: Today we begin studying another very fundamental topic in geometric computing, and along the way we will show a rather surprising connection between this topic the topic of convex hulls, which we discussed earlier. Any line in the plane splits the plane into two regions, called *halfplane*, one lying on either side of the line. We may refer to a halfplane as being either *closed* or *open* depending on whether it contains the line itself. For this lecture we will be interested in closed halfplanes.

How do we represent lines and halfplanes? For the cleanest and most general understanding of representing lines, it is good to study projective geometry and homogeneous coordinates. However, for the sake of time, we will skip this. Typically, it will suffice to represent lines in the plane using the following equation:

$$y = ax - b,$$

where a denotes the slope and b denotes the negation of the y-intercept. (We will see later why this representation is convenient.) Unfortunately, it is not fully general, since it cannot represent vertical lines. A more general line representation will generally involve three parameters, as in:

$$ax + by = c.$$

In this case, it is easy to see that the line has slope $-b/a$ and hence is perpendicular to the vector (a, b). The equation is unchanged by an scalar multiplication, and so if $c \neq 0$ (the line does not pass through the origin) we could express this more succinctly as $a'x + b'y = 1$, where $a' = a/c$ and $b' = b/c$.

To represent a closed halfplane, we convert either representation into an inequality:

$$y \leq ax - b \qquad \text{or} \qquad ax + by \leq c.$$

In the former case, this represents the halfplane lying below the line. The latter representation is more general, since we can represent halfplanes on either side of the line, simply by multiplying all the coefficients by -1.

Halfplane intersection problem: The *halfplane intersection problem* is, given a set of n closed halfplanes, $H = \{h_1, h_2, \ldots, h_n\}$ compute their intersection. A halfplane (closed or open) is a convex set, and hence the intersection of any number of halfplanes is also a convex set. Unlike the convex hull problem, the intersection of n halfplanes may generally be empty or even unbounded. A reasonable output representation might be to list the lines bounding the intersection in counterclockwise order, perhaps along with some annotation as to whether the final figure is bounded or unbounded.

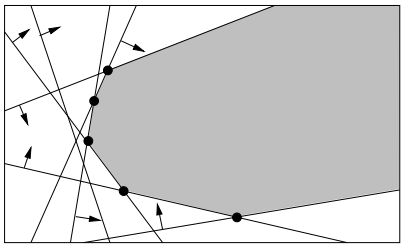


Figure 31: Halfplane intersection.

How many sides can bound the intersection of n halfplanes in the worst case? Observe that by convexity, each of the halfplanes can appear only once as a side, and hence the maximum number of sides is n. How fast can we compute the intersection of halfspaces? As with the convex hull problem, it can be shown through a suitable reduction from sorting that the problem has a lower bound of $\Omega(n \log n)$.

Who cares about this problem? Our book discusses a rather fanciful application in the area of casting. More realistically, halfplane intersection and halfspace intersection in higher dimensions are used as a method for generating convex shape approximations. In computer graphics for example, a bounding box is often used to approximate a complex multi-sided polyhedral shape. If the bounding box is not visible from a given viewpoint then the object within it is certainly not visible. Testing the visibility of a 6-sided bounding box is much easier than a multi-sided nonconvex polyhedron, and so this can be used as a filter for a more costly test. A bounding box is just the intersection of 6 axis-aligned halfspace in 3-space. If more accurate, but still convex approximations are desired, then we may compute the intersection of a larger number of tight bounding halfspaces, in various orientations, as the final approximation.

Solving the halfspace intersection problem in higher dimensions is quite a bit more challenging than in the plane. For example, just storing the output as a cyclic sequence of bounding planes is not sufficient. In general some sort of adjacency structure (a DCEL, for example) is needed.

We will discuss two algorithms for the halfplane intersection problem. The first is given in the text. For the other, we will consider somewhat simpler problem of computing something called the *lower envelope* of a set of lines, and show that it is closely related to the convex hull problem.

Divide-and-Conquer Algorithm: We begin by sketching a divide-and-conquer algorithm for computing the intersection of halfplanes. The basic approach is very simple:

(1) If $n = 1$, then just return this halfplane as the answer.

(2) Split the n halfplanes of H into subsets H_1 and H_2 of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, respectively.

(3) Compute the intersection of H_1 and H_2, each by calling this procedure recursively. Let C_1 and C_2 be the results.

(4) Intersect the convex polygons C_1 and C_2 (which might be unbounded) into a single convex polygon C, and return C.

The running time of the resulting algorithm is most easily described using a *recurrence*, that is, a recursively defined equation. If we ignore constant factors, and assume for simplicity that n is a power of 2, then the running time can be described as:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + S(n) & \text{if } n > 1, \end{cases}$$

where $S(n)$ is the time required to compute the intersection of two convex polygons whose total complexity is n. If we can show that $S(n) = O(n)$, then by standard results in recurrences it will follow that the overall running time $T(n)$ is $O(n \log n)$. (See CLR, for example.)

Intersecting Two Convex Polygons: The only nontrivial part of the process is implementing an algorithm that intersects two convex polygons, C_1 and C_2, into a single convex polygon. Note that these are somewhat special convex polygons because they may be empty or unbounded.

We know that it is possible to compute the intersection of line segments in $O((n + I) \log n)$ time, where I is the number of intersecting pairs. Two convex polygons cannot intersect in more than $I = O(n)$ pairs. (This follows from the observation that each edge of one polygon can intersect at most two edges of the other polygon by convexity.) This would given $O(n \log n)$ algorithm for computing the intersection and an $O(n \log^2 n)$ solution for $T(n)$, which is not as good as we would like.

There are two common approaches for intersecting convex polygons. Both essentially involve merging the two boundaries. One works by a plane-sweep approach. The other involves a simultaneous counterclockwise sweep around the two boundaries. The latter algorithm is described in O'Rourke's book. We'll discuss the plane-sweep algorithm.

We perform a left-to-right plane sweep to compute the intersection. We begin by breaking the boundaries of the convex polygons into their upper and lower chains. By convexity, the sweep line intersects each convex polygon C_i in at most two points, and hence, there are at most four points in the sweep line status at any time. Thus we do not need a dictionary for storing the sweep line status, a simple 4-element list suffices. Also, our event queue need only be of constant size. At any point there are at most 8 possible candidates for the next event, namely, the right endpoints of the four edges stabbed by the sweep line and the (up to four) intersection points of these upper and lower edges of C_1 with the upper and lower edges of C_2. Since there are only a constant number of possible events, and each can be handled in $O(1)$ time, the total running time is $O(n)$.

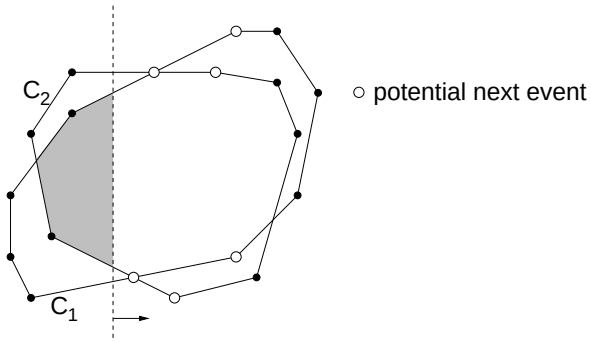


Figure 32: Convex polygon intersection.

Lower Envelopes and Duality: Next we consider a slight variant of this problem, to demonstrate some connections with convex hulls. These connections are very important to an understanding of computational geometry, and we see more about them in the future. These connections have to do with a concept called *point-line duality*. In a nutshell there is a remarkable similarity between how points interact with each other an how lines interact with each other. Sometimes it is possible to take a problem involving points and map it to an equivalent problem involving lines, and vice versa. In the process, new insights to the problem may become apparent.

The problem to consider is called the *lower envelope* problem, and it is a special case of the halfplane intersection problem. We are given a set of n lines $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ where ℓ_i is of the form $y = a_i x - b_i$. Think of these lines as defining n halfplanes, $y \leq a_i x - b_i$, each lying *below* one of the lines. The *lower envelope* of L is the boundary of the intersection of these half planes. (There is also an upper envelope, formed by considering the intersection of the halfplanes lying above the lines.)

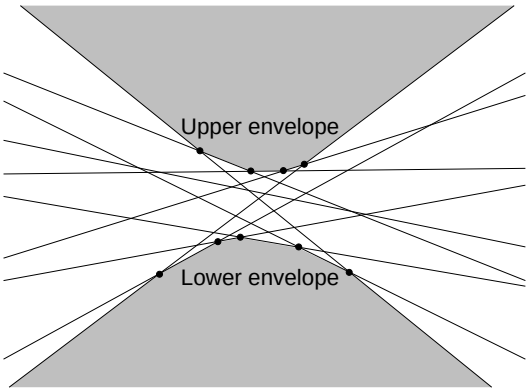


Figure 33: Lower and upper envelopes.

The lower envelope problem is a restriction of the halfplane intersection problem, but it an interesting restriction. Notice that any halfplane intersection problem that does not involve any vertical lines can be rephrased as the intersection of two envelopes, a lower envelope defined by the lower halfplanes and an upper envelope defined by the upward halfplanes.

I will show that solving the lower envelope problem is essentially equivalent to solving the upper convex hull problem. In fact, they are so equivalent that exactly the same algorithm will solve both problems, without changing even a single character of code. All that changes is the way in which you view the two problems.

Duality: Let us begin by considering lines in the plane. Each line can be represented in a number of ways, but for now, let us assume the representation $y = ax - b$, for some scalar values a and b. We cannot represent vertical lines in this way, and for now we will just ignore them. Later in the semester we will fix this up. Why did we subtract b? We'll see later that this is just a convenience.

Therefore, in order to describe a line in the plane, you need only give its two coordinates (a, b). In some sense, lines in the plane can be thought of as points in a new plane in which the coordinate axes are labeled (a, b), rather than (x, y). Thus the line $y = 7x - 4$ corresponds to the point $(7, 4)$ in this new plane. Each point in this new plane of "lines" corresponds to a nonvertical line in the original plane. We will call the original (x, y)-plane the *primal plane* and the new (a, b)-plane the *dual plane*.

What is the equation of a line in the dual plane? Since the coordinate system uses a and b, we might write a line in a symmetrical form, for example $b = 3a - 5$, where the values 3 and 5 could be replaced by any scalar values.

Consider a particular point $p = (p_x, p_y)$ in the primal plane, and consider the set of all nonvertical lines passing through this point. Any such line must satisfy the equation $p_y = ap_x - b$. The images of all these lines in the dual plane is a set of points:

$$\mathcal{L} \quad = \quad \{(a, b) \mid p_y = ap_x - b\}$$

$$= \quad \{(a, b) \mid b = p_x a - p_y\}.$$

Notice that this set is just the set of points that lie on a line in the dual (a, b)-plane. (And this is why we negated b.) Thus, not only do lines in the primal plane map to points in the dual plane, but there is a sense in which a point in the primal plane corresponds to a line in the dual plane.

To make this all more formal, we can define a function that maps points in the primal plane to lines in the dual plane, and lines in the primal plane to points in the dual plane. We denote it using a asterisk $(*)$ as a superscript. Thus, given point $p = (p_x, p_y)$ and line $\ell : (y = \ell_a x - \ell_b)$ in the primal plane we define ℓ^* and p^* to be a point and line respectively in the dual plane defined by:

$$\begin{aligned} \ell^* &= (\ell_a, \ell_b) \\ p^* &: (b = p_x a - p_y). \end{aligned}$$

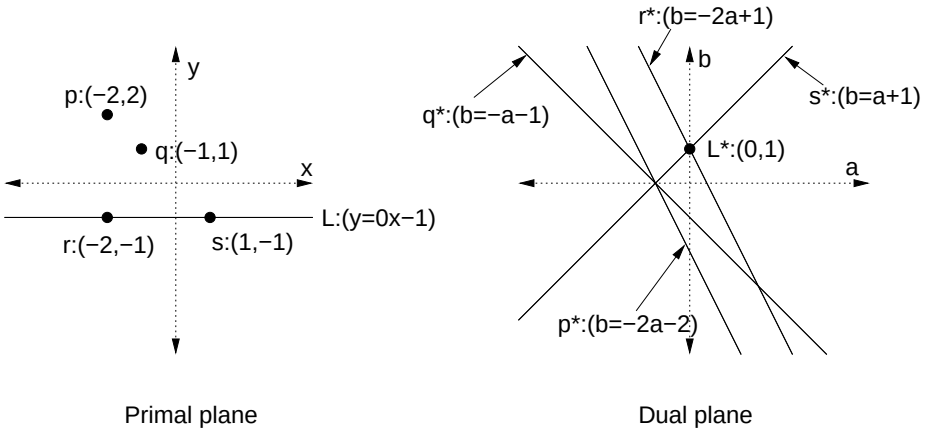


Figure 34: Dual transformation.

We can define the same mapping from dual to primal as well. Duality has a number of interesting properties, each of which is easy to verify by substituting the definition and a little algebra.

Self Inverse: $(p^*)^* = p$.

Order reversing: Point p lies above/on/below line ℓ in the primal plane if and only if line p^* passes below/on/above point ℓ^* in the dual plane, respectively.

Intersection preserving: Lines ℓ_1 and ℓ_2 intersect at point p if and only if line p^* passes through points ℓ_1^* and ℓ_2^* in the dual plane.

Collinearity/Coincidence: Three points are collinear in the primal plane if and only if their dual lines intersect in a common point.

For example, to verify the order reversing property, observe that p lies above ℓ if and only if

$$p_y > \ell_a p_x - \ell_b.$$

Rewriting this gives

$$\ell_b > p_x \ell_a - p_y,$$

which (in the dual plane) is equivalent to saying that ℓ^* lies above p^*, that is, p^* passes below ℓ^*. To finish things up, we need to make the connection between the upper convex hull of a set of points and the lower envelope of a set of lines.

Lemma: Let P be a set of points in the plane. The counterclockwise order of the points along the upper (lower) convex hull of P, is equal to the left-to-right order of the sequence of lines on the lower (upper) envelope of the dual P^*.

Proof: We will prove the result just for the upper hull and lower envelope, since the other case is symmetrical. For simplicity, let us assume that no three points are collinear. Observe that a necessary and sufficient condition for a pair of points p_ip_j to form an edge on the upper convex hull is that the line ℓ_{ij} that passes through both of these points has every other point in P strictly beneath it.

Consider the dual lines p_i^* and p_j^*. A necessary and sufficient condition that these lines are adjacent on the lower envelope is that the dual point at which they meet, ℓ_{ij}^* lies beneath all of the other dual lines in P^*.

The order reversing condition of duality assures us that the primal condition occurs if and only if the dual condition occurs. Therefore, the sequence of edges on the upper convex hull is identical to the sequence of vertices along the lower envelope.

As we move counterclockwise along the upper hull observe that the slopes of the edges increase monotonically. Since the slope of a line in the primal plane is the a-coordinate of the dual point, it follows that as we move counterclockwise along the upper hull, we visit the lower envelope from left to right.

One rather cryptical feature of this proof is that, although the upper and lower hulls appear to be connected, the upper and lower envelopes of a set of lines appears to consist of two disconnected sets. To make sense of this, we should interpret the primal and dual planes from the perspective of projective geometry, and think of the rightmost line of the lower envelope as "wrapping around" to the leftmost line of the upper envelope, and vice versa. We will discuss projective geometry later in the semester.

Another interesting question is that of orientation. We know the orientation of three points is positive if the points have a counterclockwise orientation. What does it mean for three lines to have a positive orientation? (The definition of line orientation is exactly the same, in terms of a determinant of the coefficients of the lines.)

Lecture 9: Linear Programming

Reading: Chapter 4 in the 4M's.

Linear Programming: Last time we considered the problem of computing the intersection of n halfplanes, and presented in optimal $O(n \log n)$ algorithm for this problem. In many applications it is not important to know the entire polygon (or generally the entire polytope in higher dimensions), but only to find one particular point of interest on the polygon.

One particularly important application is that of linear programming. In linear programming (LP) we are given a set of linear inequalities, or *constraints*, which we may think of as defining a (possibly empty, possibly unbounded) polyhedron in space, called the *feasible region*, and we are given a linear *objective function*, which is to be minimized or maximized subject to the given constraints. A typical description of a d-dimensional linear programming problem might be:

$$\begin{aligned} \text{Maximize:} \quad & c_1x_1 + c_2x_2 + \cdots + c_dx_d \\ \text{Subject to:} \quad & a_{1,1}x_1 + \cdots + a_{1,d}x_d \le b_1 \\ & a_{2,1}x_1 + \cdots + a_{2,d}x_d \le b_2 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \le b_n \end{aligned}$$

where $a_{i,j}$, c_i, and b_i are given real numbers. This can be also be expressed in matrix notation:

$$\begin{aligned} \text{Maximize:} \quad & c^Tx, \\ \text{Subject to:} \quad & Ax \le b. \end{aligned}$$

where c and x are d-vectors, b is an n-vector and A is an $n \times d$ matrix.

From a geometric perspective, the feasible region is the intersection of halfspaces, and hence is a (possibly unbounded) convex polyhedron in d-space. We can think of the objective function as a vector $\vec{c}$, and the problem is to find the point of the feasible region that is farthest in the direction $\vec{c}$, called the *optimal vertex*.

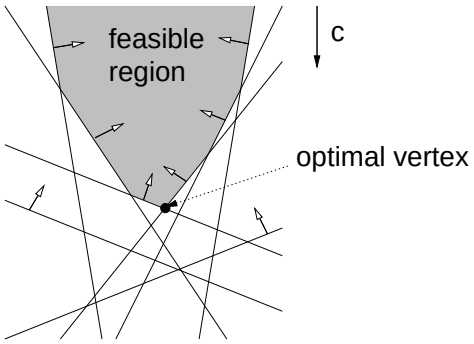


Figure 35: 2-dimensional linear programming.

Note that the magnitude of $\vec{c}$ is irrelevant, since the problem is unchanged for any positive scalar multiple of $\vec{c}$. In many of our examples, we will imagine that the vector $\vec{c}$ is pointing straight down (that is, $\vec{c} = (0, -1)$) and hence the problem is just that of finding the lowest point (minimum y-coordinate) of the feasible region. This involves no loss of generality, since it is always possible to transform the problem into one in which $\vec{c} = (0, -1)$ by an appropriate rotation.

Normally, this extreme point will be a vertex of the feasible region polyhedron, but there some other possibilities as well. The feasible region may be empty (in which case the linear programming problem is said to be *infeasible*) and there is no solution. It may be unbounded, and if $\vec{c}$ points in the direction of the unbounded part of the polyhedron, then there may solutions with infinitely large values of the objective function. In this case there is no (finite) solution, and the LP problem is said to be *unbounded*. Finally observe that in degenerate situations, it is possible to have an infinite number of finite optimum solutions, because an edge or face of the feasible region is perpendicular to the objective function vector. In such instances it is common to break ties by requiring that the solution be lexicographically maximal (e.g. among all maximal solutions, take the one with the lexicographically maximum vector).

Linear Programming in High Dimensional Spaces: Linear programming is a very important technique used in solving large optimization problems. Typical instances may involve hundreds to thousands of constraints in very high dimensional space. It is without doubt one of the most important formulations of general optimization problems.

The principal methods used for solving high-dimensional linear programming problems are the *simplex algorithm* and various *interior-point methods*. The simplex algorithm works by finding a vertex on the feasible polyhedron, then walking edge by edge downwards until reaching a local minimum. By convexity, the local minimum is the global minimum. It has been long known that there are instances where the simplex algorithm runs in exponential time. The question of whether linear programming was even solvable in polynomial time was unknown until Khachiyan's ellipsoid algorithm (late 70's) and Karmarkar's more practical interior-point algorithm (mid 80's).

2-dimensional LP: We will restrict ourselves to low dimensional instances of linear programming. There are a number of interesting optimization problems that can posed as a low-dimensional linear programming problem, or as closely related optimization problems. One which we will see later is the problem of finding a minimum radius circle that encloses a given set of n points.

Let us consider the problem just in the plane. Here we know that there is an $O(n \log n)$ algorithm based on just computing the feasible polyhedron, and finding its lowest vertex. However, since we are only interested in one

point on the polyhedron, it seems that we might hope to do better. Also, in d dimensional space, a polytope bounded by n halfspaces may have as many as $\Omega(n^{\lfloor d/2 \rfloor})$ complexity. We will show that 2-dimensional LP problems can be solved in $O(n)$ time. This algorithm can be applied in any dimension d, but the constant factor hidden by the asymptotic notation grows as $d!$, and so it is not practical except for very small dimensions.

Deterministic Incremental Construction: The two algorithms that we will discuss for linear programming are very simple, and are based on a method called *incremental construction*. Plane-sweep and incremental construction are the two pillars of computational geometry, and so this is another interesting reason for studying the linear programming problem.

Assume that we are given a set of n linear inequalities (halfplanes) $h_1, \ldots, h_n$ of the form:

$$h_i : a_{i,x}x + a_{i,y}y \leq b_i,$$

and a nonzero objective function given by vector $\vec{c} = (c_x, c_y)$. The problem is to find $\vec{p} = (p_x, p_y)$ that is feasible and maximizes the dot product $c_x p_x + c_y p_y$. Henceforth, we will assume that $\vec{c} = (0, -1)$. (Beware: As you decrease the y-coordinate of a point, you increase the objective function.)

Let us suppose for simplicity that the LP problem is bounded, and furthermore we can find two halfplanes whose intersection (a cone) is bounded with respect to the objective function. Our book explains how to overcome these assumptions in $O(n)$ additional time. Let us assume that the halfplanes are renumbered so that these are h_1 and h_2, and let v_2 denote this first optimum vertex, where the two lines associated with these halfplanes intersect.

We will then add halfplanes one by one, $h_3, h_4, \ldots$, and with each addition we update the current optimum vertex. Let v_i denote the optimal feasible vertex after the addition of $\{h_1, h_2, \ldots, h_i\}$. Our job is to update v_i with each new addition. Notice that with each new constraint, the feasible region generally becomes smaller, and hence the value of the objective function at optimum vertex can only decrease.

There are two cases that can arise when h_i is added (see the figure below). In the first case, v_{i-1} lies within the halfplane h_i, and so it satisfies this constraint. If so, then it is easy to see that the optimum vertex does not change, that is $v_i = v_{i-1}$. In the second case v_{i-1} violates constraint h_i. In this case we need to find a new optimum vertex. Let us consider this case in greater detail.

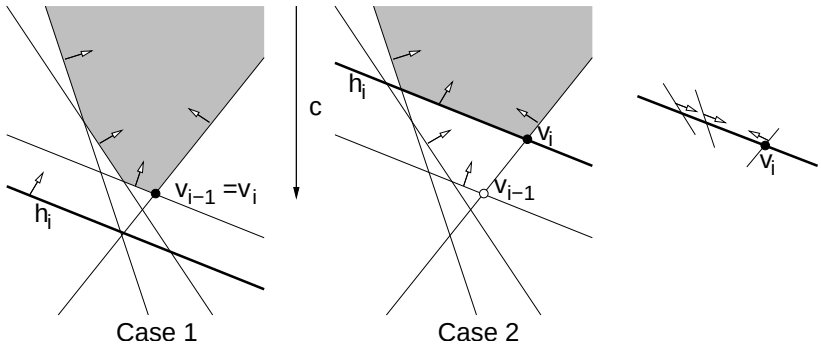


Figure 36: Incremental construction.

The important observation is that (assuming that the feasible region is not empty) the new optimum vertex must lie on the line that bounds h_i. Call this line ℓ_i. (The book proves this formally, but here is an intuitive argument. Suppose that the new optimum vertex does not lie on ℓ_i. Draw a line segment from v_{i-1} to the new optimum. Observe (1) that as you walk along this segment the value of the objective function is decreasing monotonically (by linearity), and (2) that this segment must cross ℓ_i (because it goes from being infeasible with respect to h_i to being feasible). Thus, it is maximized at the crossing point, which lies on ℓ_i.) Convexity and linearity are both very important for the proof.

So this leaves the question of how do we find the optimum vertex lying on line ℓ_i. This turns out to be a 1-dimensional LP problem. Simply intersect each of the halfplanes with this line. Each intersection will take the

form of a ray that lies on the line. We can think of each ray as representing an interval (unbounded to either the left or to the right). All we need to do is to intersect these intervals, and find the point that maximizes the objective function (that is, the lowest point). Computing the intersection of a collection of intervals, is very easy and can be solved in linear time. We just need to find the smallest upper bound and the largest lower bound. We select the point that maximizes the objective function. If this interval is empty, then it follows that the feasible region is empty, and we may terminate and report that there is no solution.

Notice that we have solved a 2-dimensional LP problem by a reduction to a 1-dimensional LP (which is easy to solve). This general framework can be applied to solving LP problems in any dimension, by repeatedly reducing to an LP problem in the next lower dimension whenever the current optimum vertex is not feasible.

Analysis: What is the worst-case running time of this algorithm? There are roughly n halfspace insertions. In step i, we may either find that the current optimum vertex is feasible, in which case the processing time is constant. On the other hand, if the current optimum vertex is infeasible, then we must solve a 1-dimensional linear program with $i-1$ constraints. In the worst case, this second step occurs all the time, and the overall running time is given by the summation:

$$\sum_{i=3}^{n}(i-1) \;\leq\; \sum_{i=1}^{n-1} i \;=\; \frac{n(n-1)}{2} \;=\; O(n^2).$$

This follows from standard results on summations.

However notice that this worst-case is based on the rather pessimistic assumption that the current vertex is almost always infeasible. Next time we will consider whether this assumption is reasonable, and how to better deal with this.

Randomized Algorithm: The $O(n^2)$ time is not very encouraging (considering that we could compute the entire feasible region in $O(n \log n)$ time). But we presented it because it leads to a very elegant randomized $O(n)$ time algorithm. The algorithm operates in exactly the same way, but we insert the halfplanes in random order, each time updating the answer based on the existing halfplanes. This is an example of a general class of algorithms called *randomized incremental algorithms*. For example, just prior to calling the algorithm given last time, we call a procedure that randomly permutes the initial input list. The code is shown below.

Randomized Incremental 2D Linear Programming

(0) Let H be a set of n halfplanes, and $\vec{c}$ be the objective function.

(1) Randomly permute the halfplanes of H, letting $\langle h_1, h_2, \ldots, h_n \rangle$ denote the resulting sequence. We assume that h_1 and h_2 define a bounded LP relative to $\vec{c}$. (See our text for the general case.) Let v_2 be the intersection point of the two associated lines.

(2) for $i = 3$ to n do:

  (a) if ($v_{i-1} \in h_i$) then $v_i \leftarrow v_{i-1}$.

  (b) else project $\{h_1, h_2, \ldots, h_{i-1}\}$ onto the line l_i that supports h_i. Solve the resulting 1-dimensional LP.

    (i) if the 1D LP is infeasible, then terminate and report that the LP is infeasible.

    (ii) else let v_i be the solution to the 1D LP.

(3) Return v_n as the final solution.

Probabilistic Analysis: We analyze the running time in the expected case where we average over all $n!$ possible permutations. Each permutation has an equal probability of $1/n!$ of occurring, and an associated running time. However, presenting the analysis as sum of $n!$ terms does not lead to something that we can easily simplify. We will apply a technique called *backward analysis*, which is quite useful.

To motivate how backward analysis works, let us consider a much simpler example, namely the problem of computing the minimum of a set of n distinct numbers. We permute the numbers and inspect them in this order.

We maintain a variable that holds the minimum value seen so far. If we see a value that is smaller than the current minimum, then we update the minimum. The question we will consider is, on average how many times is the minimum value updated? For example, in the following sequence, the minimum is updated 4 times.

$$\underline{5} \quad 9 \quad \underline{4} \quad \underline{2} \quad 6 \quad 8 \quad \underline{0} \quad 3 \quad 1 \quad 7$$

Let p_i denote the probability that the minimum value changes on inspecting the ith number of the random permutation. Thus, with probability p_i the minimum changes (1) and with probability $1 - p_i$ it does not (0). The total expected number of changes is

$$C(n) \;=\; \sum_{i=1}^{n} (p_i \cdot 1 + (1 - p_i) \cdot 0) \;=\; \sum_{i=1}^{n} p_i.$$

It suffices to compute p_i. We reason as follows. Let S_i be an arbitrary subset of i numbers from our initial set of n. (In theory, the probability is conditional on the fact that the elements of S_i represent the first i elements to be chosen, but since the analysis will not depend on the particular choice of S_i, it follows that the probability that we compute will hold unconditionally.) Among all $i!$ permutations of the elements of S_i, in how many of these does the minimum change when inspecting the ith value? The answer quite simply is that this only happens for those sequences in which the minimum element is the last (ith) element of the sequence. Since the minimum item appears with equal probability in each of the i positions of a random sequence, the probability that it appears last is $1/i$. Thus, $p_i = 1/i$. From this we have

$$C(n) \;=\; \sum_{i=1}^{n} p_i \;=\; \sum_{i=1}^{n} \frac{1}{i} \;=\; \ln n + O(1).$$

This summation is the *Harmonic series* and the fact that it is nearly equal to $\ln n$ is a well known fact.

This is called an *backwards analysis* because the analysis depends on the last item to be considered (as opposed to looking forward to the next). Let us try to apply this same approach to analyze the running time of the randomized incremental linear programming algorithm. This time, let p_i denote the probability that the insertion of the ith hyperplane in the random order resulted in a change in the optimum vertex. With probability $(1 - p_i)$ there is no change (Case 1), and it takes us $O(1)$ time to determine this. ($O(d)$ time in general in dimension d.) With probability p_i we need to invoke a 1-dimensional LP on a set of $i - 1$ halfplanes in dimension 1, with a running time of $O(i)$. (In general, if $T(n, d)$ is the expected running time for n halfspaces in dimension d, then this cost would be $T(i - 1, d - 1)$.) Combining this we have a total expected running time of

$$T(n) \;=\; \sum_{i=1}^{n} ((1 - p_i) \cdot 1 + p_i \cdot i)) \;\leq\; n + \sum_{i=1}^{n} p_i \cdot i$$

All that remains is to determine p_i. We will apply the same technique. Let S_i denote an arbitrary subset consisting of i of the original halfplanes. Among all $i!$ permutations of S_i, in how many does the optimum vertex change with the ith step? Let v_i denote the optimum vertex for these i halfplanes. It is important to note that v_i only depends on the set S_i and not on the order of their insertion. (If you do not see why this is important, think about it.)

Assuming general position, there are two halfplanes h' and h'' of S_i passing through v_i. If neither of these was the last to be inserted, then $v_i = v_{i-1}$, and there is no change. If either h' or h'' was the last to be inserted, then v_i did not exist yet, and hence the optimum must have changed as a result of this insertion. Thus, the optimum changes if and only if either h' or h'' was the last halfplane inserted. Since all of the i halfplanes are equally likely to be last, this happens with probability $2/i$. Therefore, $p_i = 2/i$.

To illustrate this, consider the example shown in the following figure. We have $i = 7$ random halfplanes that have been added, so far and $v_i = v$ is the current optimum vertex and it is defined by h_4 and h_8. Let's consider

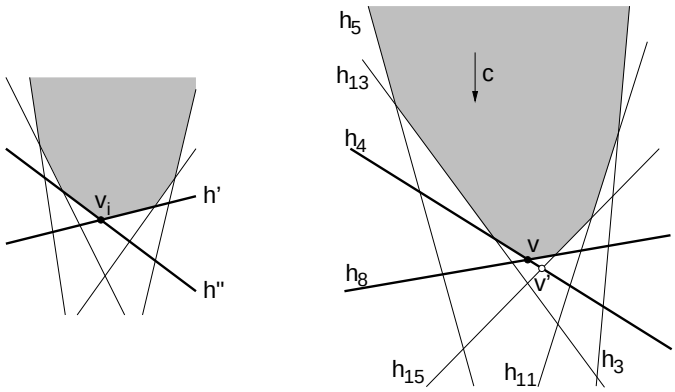


Figure 37: Backward analysis.

which halfplane was added last. If h_5 was the last to be added, imagine the picture without h_5. Prior to this v would already have been the optimum vertex. Therefore, if h_5 was added last, it would have been added with $O(1)$ cost.

On the other hand, if h_8 was added last, then prior to its addition, we would have had a different optimum vertex, namely v'. In summary, in 2 out of 7 cases (h_4 and h_8) on the ith insertion we need to solve a 1-dimensional LP and in 5 out of 7 instances $O(1)$ time suffices.

Returning to our analysis, since $p_i = 2/i$ we have

$$T(n) \;\leq\; n + \sum_{i=1}^{n} p_i \cdot i \;=\; n + \sum_{i=1}^{n} \frac{2i}{i} \;=\; n + 2n \;=\; O(n).$$

Therefore, the expected running time is linear in n. I'll leave the analysis for the d-dimensional case as an exercise. (Note that in dimension d the optimum vertex is defined by d halfspaces, not 2.)

Lecture 10: More Linear Programming and Smallest Enclosing Disk

Reading: Chapter 4 in the 4M's.

Low-Dimensional Linear Programming: Last time we presented an $O(n)$ time algorithm for linear programming in the plane. We begin by observing that the same algorithm can be generalized to any dimension d. In particular, let $\{h_1, h_2, \ldots, h_n\}$ be a set of n closed halfspaces in dimension d (each h_i is defined by one linear inequality of the form

$$h_i : a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,d}x_d \leq b_i.$$

and $\vec{c}$ is a vector in d-space. We begin by selecting d halfspaces whose feasible region is bounded with respect to c. This can be done by a generalization of the method used for the planar case. Next, we add halfspaces one by one in random order. As before if the current optimal vertex is feasible with respect to the latest halfspace, there is no change. This can be checked in $O(d)$ time. Otherwise, let ℓ_i denote the hyperplane that supports h_i (formed by changing the above inequality into an equality). we intersect all the halfspaces with ℓ_i (using Gauss elimination), and then solve the resulting $d-1$ dimensional LP problem, involving $i-1$ halfspaces recursively, and return the result.

The running time is derived in exactly the same way as for the 2-dimensional case. Let $T(d, n)$ be the expected running time of the algorithm for n halfspaces in dimension d. Let's consider the ith stage. Assuming general position, there are d halfspaces whose intersection (of the supporting hyperplanes) defines the optimum vertex.

If any of these halfspaces was the last to be added, then the running time of the last stage required a recursive call taking $T(d-1, i-1)$ time. This occurs with probability d/i since each of the existing i halfspaces is equally likely to be the last to be added. On the other hand, with probability $(i-d)/i$ the last halfspace was not one that defines the optimal vertex, and hence the running time is just $O(d)$ for the last stage. The total running time comes by summing over all the n stages. Ignoring the constant factor hidden in the $O(d)$, the expected running time is given by the recurrence

$$\begin{aligned} T(1,n) &= n \\ T(d,n) &= \sum_{i=1}^{n} \left(\frac{i-d}{i} d + \frac{d}{i} T(d-1, i-1) \right) \end{aligned}$$

We claim that $T(d,n) \in O(d!n)$. The proof is by induction on d. This is clearly true in the basis case ($d = 1$). In general we have

$$\begin{aligned} T(d,n) &= \sum_{i=1}^{n} \frac{i-d}{i} d + \frac{d}{i} T(d-1, i-1) \\ &\leq dn + \sum_{i=1}^{n} \frac{d}{i} (d-1)!(i-1) \\ &\leq dn + d! \sum_{i=1}^{n} \frac{i-1}{i} \;\leq\; dn + d!n. \end{aligned}$$

Oops, this is not quite what we wanted. We wanted $T(d,n) \leq d!n$. However, a more careful (but messier) induction proof will do the job. We leave this as an exercise.

Smallest Enclosing Disk: Although the vast majority of applications of linear programming are in relatively high dimensions, there are a number of interesting applications in low dimensions. We will present one such example, called the *smallest enclosing disk problem*. We are given n points in the plane and we are asked to find the closed circular disk of minimum radius that encloses all of these points. We will present a randomized algorithm for this problem that runs in $O(n)$ expected time.

We should say a bit about terminology. A *circle* is the set of points that are equidistant from some center point. A *disk* is the set of points lying within a circle. We can talk about *open* or *closed* disks to distinguish whether the bounding circle itself is part of the disk. In higher dimensions the generalization of a circle is a *sphere* in 3-space, or *hypersphere* in higher dimensions. The set of points lying within a sphere or hypersphere is called a *ball*.

Before discussing algorithms, we first observe that any circle is uniquely determined by three points (as the circumcenter of the triangle they define). We will not prove this, but it follows as an easy consequence of linearization, which we will discuss later in the lecture.

Claim: For any finite set of points in general position (no four cocircular), the smallest enclosing disk either has at least three points on its boundary, or it has two points, and these points form the diameter of the circle. If there are three points then they subdivide the circle bounding the disk into arcs of angle at most π.

Proof: Clearly if there are no points on the boundary the disk's radius could be decreased. If there is only one point on the boundary then this is also clearly true. If there are two points on the boundary, and they are separated by an arc of length strictly less than π, then observe that we can find a disk that passes through both points and has a slightly smaller radius. (By considering a disk whose center point is only the perpendicular bisector of the two points and lies a small distance closer to the line segment joining the points.)

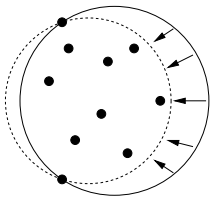


Figure 38: Contact points for a minimum enclosing disk.

Thus, none of these configurations could be a candidate for the minimum enclosing disk. Also observe that if there are three points that define the smallest enclosing disk they subdivide the circle into three arcs each of angle at most π (for otherwise we could apply the same operation above). Because points are in general position we may assume there cannot be four or more cocircular points.

This immediately suggests a simple $O(n^4)$ time algorithm. In $O(n^3)$ time we can enumerate all triples of points and then for each we generate the resulting circle and test whether it encloses all the points in $O(n)$ additional time, for an $O(n^4)$ time algorithm. You might make a few observations to improve this a bit (e.g. by using only triples of points on the convex hull). But even so a reduction from $O(n^4)$ to $O(n)$ is quite dramatic.

Linearization: We can "almost" reduce this problem to a linear programming problem in 3-space. Although the method does not work, it does illustrate the similarity between this problem and LP.

Recall that a point $p = (p_x, p_y)$ lies within a circle with center point $c = (c_x, c_y)$ and radius r if

$$(p_x - c_x)^2 + (p_y - c_y)^2 \leq r^2.$$

In our case we are given n such points p_i and are asked to determine whether there exists c_x, c_y and r satisfying the resulting n inequalities, with r as small as possible. The problem is that these inequalities clearly involve quantities like c_x^2 and r^2 and so are not linear inequalities in the parameters of interest.

The technique of *linearization* can be used to fix this. First let us expand the inequality above and rearrange the terms

$$\begin{aligned} p_x^2 - 2p_xc_x + c_x^2 + p_y^2 - 2p_yc_y + c_y^2 &\leq r^2 \\ 2p_xc_x + 2p_yc_y + (r^2 - c_x^2 - c_y^2) &\geq p_x^2 + p_y^2. \end{aligned}$$

Now, let us introduce a new parameter $R = r^2 - c_x^2 - c_y^2$. Now we have

$$(2p_x)c_x + (2p_y)c_y + R \geq (p_x^2 + p_y^2).$$

Observe that this is a linear inequality in c_x, c_y and R. If we let p_x and p_y range over all the coordinates of all the n points we generate n linear inequalities in 3-space, and so we can apply linear programming to find the solution, right? The only problem is that the previous objective function was to minimize r. However r is no longer a parameter in the new version of the problem. Since we $r^2 = R + c_x^2 + c_y^2$, and minimizing r is equivalent to minimizing r^2 (since we are only interested in positive r), we could say that the objective is to minimize $R + c_x^2 + c_y^2$. Unfortunately, this is not a linear function of the parameters c_x, c_y and R. Thus we are left with an optimization problem in 3-space with linear constraints and a nonlinear objective function.

This shows that LP is closely related, and so perhaps the same techniques can be applied.

Randomized Incremental Algorithm: Let us consider how we can modify the randomized incremental algorithm for LP directly to solve this problem. The algorithm will mimic each step of the randomized LP algorithm.

To start we randomly permute the points. We select any two points and compute the unique circle with these points as diameter. (We could have started with three just as easily.) Let D_{i-1} denote the minimum disk after

the insertion of the first $i-1$ points. For point p_i we determine in constant time whether the point lies within D_{i-1}. If so, then we set $D_i = D_{i-1}$ and go on to the next stage. If not, then we need to update the current disk to contain p_i, letting D_i denote the result. When the last point is inserted we output D_n.

How do we compute this updated disk? It might be tempting at first to say that we just need to compute the smallest disk that encloses p_i and the three points that define the current disk. However, it is not hard to construct examples in which doing so will cause previously interior points to fall outside the current disk. As with the LP problem we need to take all the existing points into consideration. But as in the LP algorithm we want some way to reduce the "dimensionality" of the problem. How do we do this?

The important claim is that if p_i is not in the minimum disk of the first $i-1$ points, then p_i does help constrain the problem, which we establish below.

Claim: If $p_i \notin D_{i-1}$ then p_i is on the boundary of the minimum enclosing disk for the first i points, D_i.

Proof: The proof makes use of the following geometric observation. Given a disk of radius r_1 and a circle of radius r_2, where $r_1 < r_2$, the intersection of the disk with the circle is an arc of angle less than π. This is because an arc of angle π or more contains two (diametrically opposite) points whose distance from each other is $2r_2$, but the disk of radius r_1 has diameter only $2r_1$ and hence could not simultaneously cover two such points.

Now, suppose to the contrary that p_i is not on the boundary of D_i. It is easy to see that because D_i covers a point not covered by D_{i-1} that D_i must have larger radius than D_{i-1}. If we let r_1 denote the radius of D_{i-1} and r_2 denote the radius of D_i, then by the above argument, the disk D_{i-1} intersects the circle bounding D_i in an arc of angle less than π. (Shown in a heavy line in the figure below.)

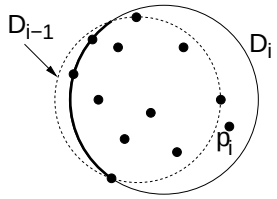


Figure 39: Why p_i must lie on the boundary of D_i.

Since p_i is not on the boundary of D_i, the points defining D_i must be chosen from among the first $i-1$ points, from which it follows that they all lie within this arc. However, this would imply that between two of the points is an arc of angle greater than π (the arc not shown with a heavy line) which, by the earlier claim could not be a minimum enclosing disk.

The algorithm is identical in structure to the LP algorithm. We will randomly permute the points and insert them one by one. For each new point p_i, if it lies within the current disk then there is nothing to update. Otherwise, we need to update the disk. We do this by computing the smallest enclosing disk that contains all the points $\{p_1, \ldots, p_{i-1}\}$ and is constrained to have p_i on its boundary. (The requirement that p_i be on the boundary is analogous to the constraint used in linear programming that optimum vertex lie on the line supporting the current halfplane.)

This will involve a slightly different recursion. In this recursion, when we encounter a point that lies outside the current disk, we will then recurse on a subproblem in which two points are constrained to lie on the boundary of the disk. Finally, if this subproblem requires a recursion, we will have a problem in which there are three points constrained to lie on a the boundary of the disk. But this problem is trivial, since there is only one circle passing through three points.

MinDisk(P) :

(1) If $|P| \leq 3$, then return the disk passing through these points. Otherwise, randomly permute the points in P yielding the sequence $\langle p_1, p_2, \ldots, p_n \rangle$.
(2) Let D_2 be the minimum disk enclosing $\{p_1, p_2\}$.
(3) for $i = 3$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ MinDiskWith1Pt$(P[1..i-1], p_i)$.

MinDiskWith1Pt(P, q) :

(1) Randomly permute the points in P. Let D_1 be the minimum disk enclosing $\{q, p_1\}$.
(2) for $i = 2$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ MinDiskWith2Pts$(P[1..i-1], q, p_i)$.

MinDiskWith2Pts(P, q_1, q_2) :

(1) Randomly permute the points in P. Let D_0 be the minimum disk enclosing $\{q_1, q_2\}$.
(2) for $i = 1$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ Disk(q_1, q_2, p_i).

Lecture 11: Orthogonal Range Searching and kd-Trees

Reading: Chapter 5 in the 4M's.

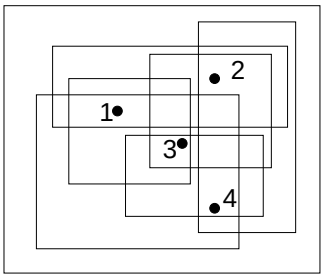
Range Queries: We will shift our focus from algorithm problems to data structures for the next few lectures. We will consider the following class of problems. Given a collection of objects, preprocess them (storing the results in a data structure of some variety) so that queries of a particular form can be answered efficiently. Generally we measure data structures in terms of two quantities, the time needed to answer a query and the amount of space needed by the data structure. Often there is a tradeoff between these two quantities, but most of the structures that we will be interested in will have either linear or near linear space. Preprocessing time is an issue of secondary importance, but most of the algorithms we will consider will have either linear of $O(n \log n)$ preprocessing time.

In a *range queries* we are given a set P of points and region R in space (e.g., a rectangle, polygon, halfspace, or disk) and are asked list (or count or compute some accumulation function of) the subset of P lying within the region. To get the best possible performance, the design of the data structure is tailored to the particular type of region, but there are some data structures that can be used for a wide variety of regions.

An important concept behind all geometric range searching is that the subsets that can be formed by simple geometric ranges is much smaller than the set of possible subsets (called the *power set*) of P. We can define any range search problem abstractly as follows. Given a particular class of ranges, a *range space* is a pair (P, R) consisting of the points P and the collection R of all subsets of P that be formed by ranges of this class. For example, the following figure shows the range space assuming rectangular ranges for a set of points in the plane. In particular, note that the sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be formed by rectangular ranges.

Today we consider *orthogonal rectangular range queries*, that is, ranges defined by rectangles whose sides are aligned with the coordinate axes. One of the nice things about rectangular ranges is that they can be decomposed into a collection of 1-dimensional searches.

Canonical Subsets: A common approach used in solving almost all range queries is to represent P as a collection of *canonical subsets* $\{S_1, S_2, \ldots, S_k\}$, each $S_i \subseteq S$ (where k is generally a function of n and the type of ranges),



R = { {},
{1}, {2}, {3}, {4},
{1,2}, {1,3}, {2,3},{2,4},{3,4},
{1,2,3}, {1,3,4}, {2,3,4} ,
{1,2,3,4} }

Figure 40: Rectangular range space.

such that any set can be formed as the disjoint union of canonical subsets. Note that these subsets may generally overlap each other.

There are many ways to select canonical subsets, and the choice affects the space and time complexities. For example, the canonical subsets might be chosen to consist of n singleton sets, each of the form $\{p_i\}$. This would be very space efficient, since we need only $O(n)$ total space to store all the canonical subsets, but in order to answer a query involving k objects we would need k sets. (This might not be bad for reporting queries, but it would be too long for counting queries.) At the other extreme, we might let the canonical subsets be the power set of P. Now, any query could be answered with a single canonical subset, but we would have 2^n different canonical subsets to store. (A more realistic solution would be to use the set of all ranges, but this would still be quite large for most interesting range spaces.) The goal of a good range data structure is to strike a balance between the total number of canonical subsets (space) and the number of canonical subsets needed to answer a query (time).

One-dimensional range queries: Before consider how to solve general range queries, let us consider how to answer 1-dimension range queries, or *interval queries*. Let us assume that we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ on the line, which we will preprocess into a data structure. Then, given an interval $[x_{lo}, x_{hi}]$, the goal is to report all the points lying within the interval. Ideally we would like to answer a query in time $O(\log n + k)$ time, where k is the number of points reported (an output sensitive result). Range counting queries can be answered in $O(\log n)$ time with minor modifications.

Clearly one way to do this is to simply sort the points, and apply binary search to find the first point of P that is greater than or equal to x_{lo}, and less than or equal to x_{hi}, and then list all the points between. This will not generalize to higher dimensions, however.

Instead, sort the points of P in increasing order and store them in the leaves of a balanced binary search tree. Each internal node of the tree is labeled with the largest key appearing in its left child. We can associate each node of this tree (implicitly or explicitly) with the subset of points stored in the leaves that are descendents of this node. This gives rise to the $O(n)$ *canonical subsets*. For now, these canonical subsets will not be stored explicitly as part of the data structure, but this will change later when we talk about range trees. This is illustrated in the figure below.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the leftmost leaf u whose key is greater than or equal to x_{lo} and the rightmost leaf v whose key is less than or equal to x_{hi}. Clearly all the leaves between u and v, together possibly with u and v, constitute the points that lie within the range. If $key(u) = x_{lo}$ then we include u's canonical (single point) subset and if $key(v) = x_{hi}$ then we do the same for v. To form the remaining canonical subsets, we take the subsets of all the *maximal subtrees* lying between u and v.

Here is how to compute these subtrees. The search paths to u and v may generally share some common subpath, starting at the root of the tree. Once the paths diverge, as we follow the left path to u, whenever the path goes to the left child of some node, we add the canonical subset associated with its right child. Similarly, as we follow

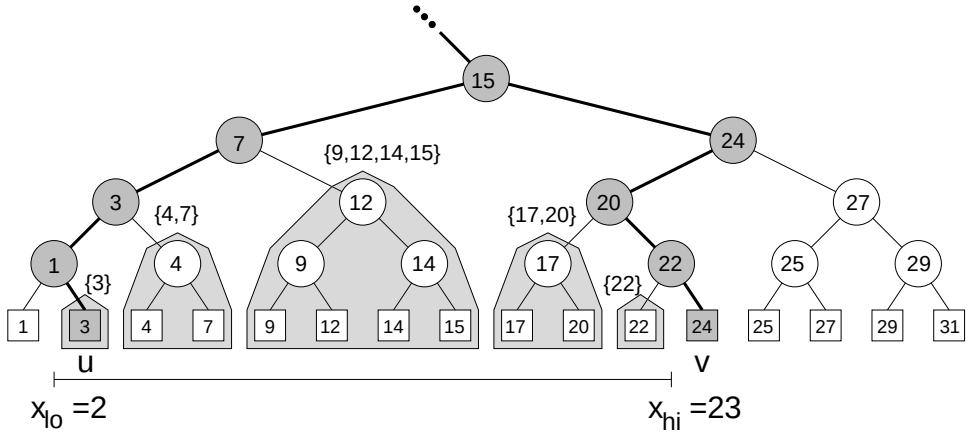


Figure 41: Canonical sets for interval queries.

the right path to v, whenever the path goes to the right child, we add the canonical subset associated with its left child.

To answer a range reporting query we simply traverse these canonical subtrees, reporting the points of their leaves. Each tree can be traversed in time proportional to the number of leaves in each subtree. To answer a range counting query we store the total number of points in each subtree (as part of the preprocessing) and then sum all of these over all the canonical subtrees.

Since the search paths are of length $O(\log n)$, it follows that $O(\log n)$ canonical subsets suffice to represent the answer to any query. Thus range counting queries can be answered in $O(\log n)$ time. For reporting queries, since the leaves of each subtree can be listed in time that is proportional to the number of leaves in the tree (a basic fact about binary trees), it follows that the total time in the search is $O(\log n + k)$, where k is the number of points reported.

In summary, 1-dimensional range queries can be answered in $O(\log n)$ time, using $O(n)$ storage. This concept of finding maximal subtrees that are contained within the range is fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Let see next how can we extend this to higher dimensional range queries.

Kd-trees: The natural question is how to extend 1-dimensional range searching to higher dimensions. First we will consider kd-trees. This data structure is easy to implement and quite practical and useful for many different types of searching problems (nearest neighbor searching for example). However it is not the asymptotically most efficient solution for the orthogonal range searching, as we will see later.

Our terminology is a bit nonstandard. The data structure was designed by Jon Bentley. In his notation, these were called "k-d trees," short for "k-dimensional trees". The value k was the dimension, and thus there are 2-d trees, 3-d trees, and so on. However, over time, the specific value of k was lost. Our text uses the somewhat nonstandard form "kd-tree" rather than "k-d tree." By the way, there are many variants of the kd-tree concept. We will describe the most commonly used one, which is quite similar to Bentley's original design. In our trees, points will be stored only at the leaves. There are variants in which points are stored at internal nodes as well.

The idea behind a kd-tree is to extend the notion of a one dimensional tree. But for each node we subdivide space either by splitting along x-coordinate of the points or along the y-coordinates. Each internal node t of the kd-tree is associated with the following quantities:

$t.cutDim$	the cutting dimension
$t.cutVal$	the cutting value
$t.size$	the number of points in t's subtree

In dimension d, the cutting dimension may be represented as in integer ranging from 0 to $d-1$. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to $t.cutVal$ are stored in the left subtree and the remaining points are stored in the right subtree. (See the figure below.) If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side. This is done to allow us to balance the number of points in the left and right subtrees if there are many equal coordinate values. When a single point remains (or more generally a small constant number of points), we store it in a leaf node, whose only field $t.point$ is this point (or generally a pointer to this point).

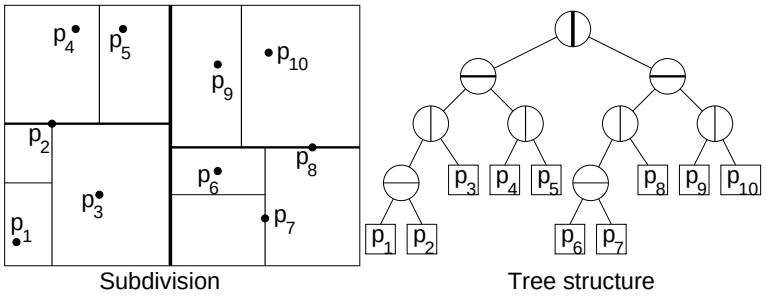


Figure 42: A kd-tree and the associated spatial subdivision.

The cutting process has a geometric interpretation. Each node of the tree is associated implicitly with a rectangular region of space, called a *cell*. (In general these rectangles may be unbounded, but in many applications it is common to restrict ourselves to some bounded rectangular region of space before splitting begins, and so all these rectangles are bounded.) The cells are nested in the sense that a child's cell is contained within its parent's cell. Hence, these cells define a *hierarchical decomposition* of space. This is illustrated in figure.

There are two key decisions in the design of the tree.

How is the cutting dimension chosen? The simplest method is to cycle through the dimensions one by one. (This method is shown in the above figure.) Since the cutting dimension depends only on the level of a node in the tree, one advantage of this rule is that the cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

One disadvantage of this splitting rule is that, depending on the data distribution, this simple cyclic rule may produce very skinny (elongated) cells, and such cells may adversely affect query times. Another method is to select the cutting dimension to be the one along which the points have the greatest *spread*, defined to be the difference between the largest and smallest coordinates. Bentley call the resulting tree an *optimized kd-tree*.

How is the cutting value chosen? To guarantee that the tree has height $O(\log n)$, the best method is to let the cutting value be the median coordinate along the cutting dimension. If there are an even number of points in the subtree, we may take either the upper or lower median, or we may simply take the midpoint between these two points. In our example, when there are an odd number of points, the median is associated with the left (or lower) subtree.

A kd-tree is a special case of a more general class of hierarchical spatial subdivisions, called *binary space partition trees* (or *BSP trees*) in which the splitting lines (or hyperplanes in general) may be oriented in any direction. In the case of BSP trees, the cells are convex polygons.

Constructing the kd-tree: It is possible to build a kd-tree in $O(n \log n)$ time by a simple top-down recursive procedure. The most costly step of the process is determining the median coordinate for splitting purposes. One way to do this is to maintain two lists of pointers to the points, one sorted by x-coordinate and the other containing pointers to the points sorted according to their y-coordinates. (In dimension d, d such arrays would be maintained.) Using these two lists, it is an easy matter to find the median at each step in constant time. In linear time

it is possible to split each list about this median element. For example, if $x = s$ is the cutting value, then all points with $p_x \leq s$ go into one list and those with $p_x > s$ go into the other. (In dimension d this generally takes $O(d)$ time per point.) This leads to a recurrence of the form $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$. Since there are n leaves and each internal node has two children, it follows that the number of internal nodes is $n - 1$. Hence the total space requirements are $O(n)$.

Theorem: A balanced kd-tree of n points can be constructed in $O(n \log n)$ time and has $O(n)$ space.

Range Searching in kd-trees: Let us consider how to answer orthogonal range counting queries. Range reporting queries are an easy extension. Let Q denote the desired range, t denote the current node in the kd-tree, and let C denote the rectangular cell associated with t. The search algorithm traverses the tree recursively. If it arrives at a leaf cell, we check to see whether the associated point, t.*point*, lies within Q, and if so we count it. Otherwise, t is an internal node. If t's cell C is disjoint from Q, then we know that no point in the subtree rooted at t is in the query range, and so there is nothing to count. If C is entirely contained within Q, then every point in the subtree rooted at t can be counted. (These points represent a canonical subset.) Otherwise, t's cell partially overlaps Q. In this case we recurse on t's two children and update the count accordingly.

kd-tree Range Counting Query

int rangeCount(Range Q, KDNode t, Rectangle C)

(1) if (t is a leaf)

  (a) if (Q contains t.*point*) return 1,

  (b) else return 0.

(2) if (t is not a leaf)

  (a) if ($C \cap Q = \emptyset$) return 0.

  (b) else if ($C \subseteq Q$) return t.*size*.

  (c) else, split C along t's cutting dimension and cutting value, letting C_1 and C_2 be the two rectangles. Return (rangeCount(Q, t.*left*, C_1) + rangeCount(Q, t.*right*, C_2)).

The figure below shows an example of a range search. White nodes have been visited in the search. Light shaded nodes were not visited because their cell is contained entirely within Q. Dark shaded nodes are not visited because their cell is disjoint from Q.

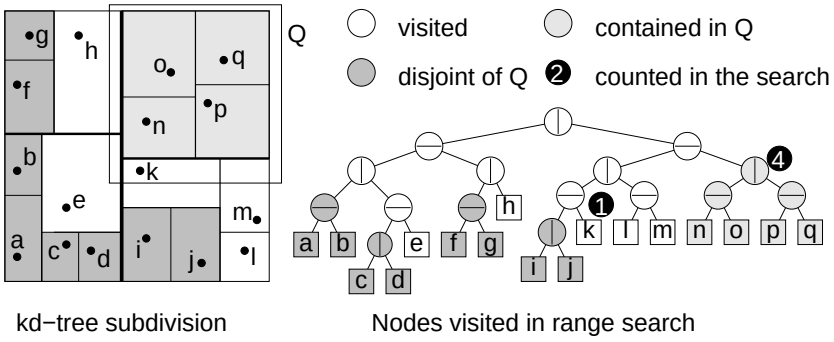


Figure 43: Range search in a kd-tree. (Note: This particular tree was not generated by the algorithm described above.)

Analysis of query time: How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree. Recall from the discussion above that a node is processed (both children visited) if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are processed in the search, we first bound the number of nodes that are stabbed.

Lemma: Given a balanced kd-tree with n points using the alternating splitting rule, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider a processed node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$.

Because we have an exponentially increasing number, the total sum is dominated by its last term. Thus is suffices to count the number of nodes stabbed at the lowest level of the tree. Since the tree is balanced, it has roughly $\lg n$ levels. Thus the number of leaf nodes processed at the bottommost level is $2^{(\lg n)/2} = 2^{\lg \sqrt{n}} = \sqrt{n}$. This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$. Since we only make recursive calls when a cell is stabbed, it follows that the total number of nodes visited by the search is $O(\sqrt{n})$.

Theorem: Given a balanced kd-tree with n points, orthogonal range counting queries can be answered in $O(\sqrt{n})$ time and reporting queries can be answered in $O(\sqrt{n} + k)$ time. The data structure uses space $O(n)$.

Lecture 12: Orthogonal Range Trees

Reading: Chapter 5 in the 4M's.

Orthogonal Range Trees: Last time we saw that kd-trees could be used to answer orthogonal range queries in the plane in $O(\sqrt{n} + k)$ time. Today we consider a better data structure, called *orthogonal range trees*.

An orthogonal range tree is a data structure which, in all dimensions $d \geq 2$, uses $O(n \log^{(d-1)} n)$ space, and can answer orthogonal rectangular range queries in $O(\log^{(d-1)} n + k)$ time, where k is the number of points reported. Preprocessing time is the same as the space bound. Thus, in the plane, we can answer range queries in time $O(\log n)$ and space $O(n \log n)$. We will present the data structure in two parts, the first is a version that can answer queries in $O(\log^2 n)$ time in the plane, and then we will show how to improve this in order to strip off a factor of $\log n$ from the query time.

Multi-level Search Trees: The data structure is based on the concept of a *multi-level search tree*. In this method, a complex search is decomposed into a constant number of simpler range searches. We cascade a number of search structures for simple ranges together to answer the complex range query. In this case we will reduce a d-dimensional range search to a series of 1-dimensional range searches.

Suppose you have a query which can be stated as the intersection of a small number of simpler queries. For example, a rectangular range query in the plane can be stated as two range queries: Find all the points whose x-coordinates are in the range $[Q.x_{lo}, Q.x_{hi}]$ and all the points whose y-coordinates are in the range $[Q.y_{lo}, Q.y_{hi}]$. Let us consider how to do this for 2-dimensional range queries, and then consider how to generalize the process. First, we assume that we have preprocessed the data by building a range tree for the first range query, which in this case is just a 1-dimensional range tree for the x-coordinates. Recall that this is just a balanced binary tree

on these points sorted by x-coordinates. Also recall that each node of this binary tree is implicitly associated with a *canonical subset* of points, that is, the points lying in the subtree rooted at this node.

Observe that the answer to any 1-dimensional range query can be represented as the disjoint union of a small collection of $m = O(\log n)$ canonical subsets, $\{S_1, S_2, \ldots, S_m\}$, where each subset corresponds to a node t in the range tree. This constitutes the first level of the search tree. To continue the preprocessing, for the second level, for each node t in this x-range tree, we build an *auxiliary tree*, t_{aux}, each of which is a y-coordinate range tree, which contains all the points in the canonical subset associated with t.

The final data structure consists of two levels: an x-range tree, such that each node in this tree points to auxiliary y-range tree. This notion of a tree of trees is basic to solving range queries by leveling. (For example, for d-dimensional range trees, we will have d-levels of trees.)

Query Processing: Given such a 2-level tree, let us consider how to answer a rectangular range query Q. First, we determine the nodes of the tree that satisfy the x portion of the range query. Each such node t is associated with a canonical set S_t, and the disjoint union of these sets $O(\log n)$ sets consistute all the points of the data set that lie within the x portion of the range. Thus to finish the query, we need to find out which points from each canonical subset lie within the range of y-coordinates. To do this, for each such node t, we access the auxiliary tree for this node, t_{aux} and perform a 1-dimensional range search on the y-range of the query. This process is illustrated in the following figure.

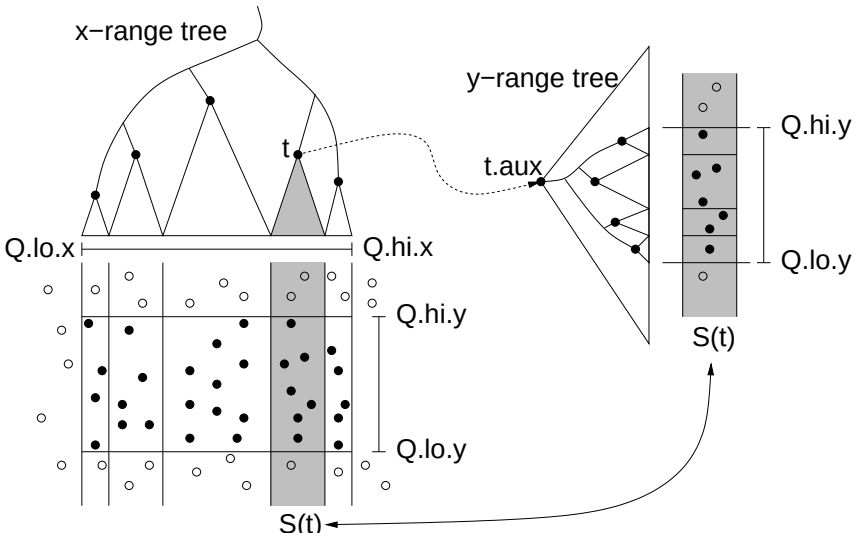


Figure 44: Orthogonal range tree search.

What is the query time for a range tree? Recall that it takes $O(\log n)$ time to locate the nodes representing the canonical subsets for the 1-dimensional range query. For each, we invoke a 1-dimensional range search. Thus there are $O(\log n)$ canonical sets, each invoking an $O(\log n)$ range search, for a total time of $O(\log^2 n)$. As before, listing the elements of these sets can be performed in additional k time by just traversing the trees. Counting queries can be answered by precomputing the subtree sizes, and then just adding them up.

Space: The space used by the data structure is $O(n \log n)$ in the plane (and $O(n \log^{(d-1)} n)$ in dimension d). The reason comes by summing the sizes of the two data structures. The tree for the x-coordinates requires only $O(n)$ storage. But we claim that the total storage in all the auxiliary trees is $O(n \log n)$. We want to count the total sizes of all these trees. The number of nodes in a tree is proportional to the number of leaves, and hence the number of points stored in this tree. Rather than count the number of points in each tree separately, instead let us count the number of trees in which each point appears. This will give the same total. Observe that a point appears in the auxiliary trees of each of its ancestors. Since the tree is balanced, each point has

$O(\log n)$ ancestors, and hence each point appears in $O(\log n)$ auxiliary trees. It follows that the total size of all the auxiliary trees is $O(n \log n)$. By the way, observe that because the auxiliary trees are just 1-dimensional trees, we could just store them as a sorted array.

We claim that it is possible to construct a 2-dimensional range tree in $O(n \log n)$ time. Constructing the 1-dimensional range tree for the x-coordinates is easy to do in $O(n \log n)$ time. However, we need to be careful in constructing the auxiliary trees, because if we were to sort each list of y-coordinates separately, the running time would be $O(n \log^2 n)$. Instead, the trick is to construct the auxiliary trees in a bottom-up manner. The leaves, which contain a single point are trivially sorted. Then we simply merge the two sorted lists for each child to form the sorted list for the parent. Since sorted lists can be merged in linear time, the set of all auxiliary trees can be constructed in time that is linear in their total since, or $O(n \log n)$. Once the lists have been sorted, then building a tree from the sorted list can be done in linear time.

Multilevel Search and Decomposable Queries: Summarizing, here is the basic idea to this (and many other query problems based on leveled searches). Let (S, R) denote the range space, consisting of points S and range sets R. Suppose that each range of R can be expressed as an intersection of simpler ranges $R = R_1 \cap R_2 \cap \ldots \cap R_c$, and assume that we know how to build data structures for each of these simpler ranges.

The multilevel search structure is built by first building a range search tree for query R_1. In this tree, the answer to any query can be represented as the disjoint union of some collection $\{S_1, S_2, \ldots, S_m\}$ of canonical subsets, each a subset of S. Each canonical set corresponds to a node of the range tree for R_1.

For each node t of this range tree, we build an auxiliary range tree for the associated canonical subset S_t for ranges of class R_2. This forms the second level. We can continue in this manner, with each node of the range tree for the ranges R_i being associated with an auxiliary range tree for ranges R_{i+1}.

To answer a range query, we solve the first range query, resulting in a collection of canonical subsets whose disjoint union is the answer to this query. We then invoke the second range query problem on each of these canonical subsets, and so on. Finally we take the union of all the answers to all these queries.

Fractional Cascading: Can we improve on the $O(\log^2 n)$ query time? We would like to reduce the query time to $O(\log n)$. As we descend the search the x-interval tree, for each node we visit, we need to search the corresponding y-interval tree. It is this combination that leads to the squaring of the logarithms. If we could search each y-interval in $O(1)$ time, then we could eliminate this second log factor. The trick to doing this is used in a number of places in computational geometry, and is generally a nice idea to remember. We are repeatedly searching different lists, but always with the same key. The idea is to merge all the different lists into a single massive list, do one search in this list in $O(\log n)$ time, and then use the information about the location of the key to answer all the remaining queries in $O(1)$ time each. This is a simplification of a more general search technique called *fractional cascading*.

In our case, the massive list on which we will do one search is the entire of points, sorted by y-coordinate. In particular, rather than store these points in a balanced binary tree, let us assume that they are just stored as sorted arrays. (The idea works for either trees or arrays, but the arrays are a little easier to visualize.) Call these the *auxiliary lists*. We will do one (expensive) search on the auxiliary list for the root, which takes $O(\log n)$ time. However, after this, we claim that we can keep track of the position of the y-range in each auxiliary list in constant time as we descend the tree of x-coordinates.

Here is how we do this. Let v be an arbitrary internal node in the range tree of x-coordinates, and let v_L and v_R be its left and right children. Let A_v be the sorted auxiliary list for v and let A_L and A_R be the sorted auxiliary lists for its respective children. Observe that A_v is the disjoint union of A_L and A_R (assuming no duplicate y-coordinates). For each element in A_v, we store two pointers, one to the item of equal or larger value in A_L and the other to the item of equal or larger value in A_R. (If there is no larger item, the pointer is null.) Observe that once we know the position of an item in A_v, then we can determine its position in either A_L or A_R in $O(1)$ additional time.

Here is a quick illustration of the general idea. Let v denote a node of the x-tree, and let v_L and v_R denote its left and right children. Suppose that (in bottom to top order) the associated nodes within this range are:

$\langle p_1, p_2, p_3, p_4, p_5, p_6 \rangle$, and suppose that in v_L we store the points $\langle p_2, p_3, p_5 \rangle$ and in v_R we store $\langle p_1, p_4, p_6 \rangle$. (This is shown in the figure below.) For each point in the auxiliary list for v, we store a pointer to the lists v_L and v_R, to the position this element would be inserted in the other list (assuming sorted by y-values). That is, we store a pointer to the largest element whose y-value is less than or equal to this point.

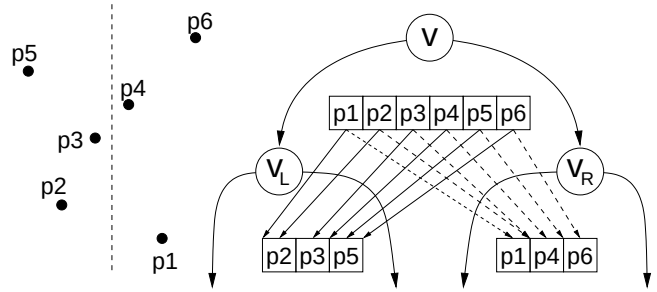


Figure 45: Cascaded search in range trees.

At the root of the tree, we need to perform a binary search against all the y-values to determine which points lie within this interval, for all subsequent levels, once we know where the y-interval falls with respect to the order points here, we can drop down to the next level in $O(1)$ time. Thus (as with fractional cascading) the running time is $O(2 \log n)$, rather than $O(log^2 n)$. It turns out that this trick can only be applied to the last level of the search structure, because all other levels need the full tree search to compute canonical sets.

Theorem: Given a set of n points in R^d, orthogonal rectangular range queries can be answered in $O(\log^{(d-1)} n + k)$ time, from a data structure of size $O(n \log^{(d-1)} n)$ which can be constructed in $O(n \log^{(d-1)} n)$ time.

Lecture 13: Hereditary Segment Trees and Red-Blue Intersection

Reading: Segment trees are presented in Chapt 10 of the 4M's. However, most of today's material has been taken from the paper, "Algorithms for Bichromatic Line-Segment Problems and Polyhedral Terrains," by B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, *Algorithmica*, 11, 1994 116–132.

Red-Blue Segment Intersection: We have been talking about the use of geometric data structures for solving query problems. Often data structures are used as intermediate structures for solving traditional input/output problems, which do not involve preprocessing and queries. (Another famous example of this is HeapSort, which introduces the heap data structure for sorting a list of numbers.) Today we will discuss a variant of a useful data structure, the *segment tree*. The particular variant is called a *hereditary segment tree*. It will be used to solve the following problem.

Red-Blue Segment Intersection: Given a set B of m pairwise disjoint "blue" segments in the plane and a set R of n pairwise disjoint "red" segments, count (or report) all *bichromatic pairs* of intersecting line segments (that is, intersections between red and blue segments).

It will make things simpler to think of the segments as being open (not including their endpoints). In this way, the pairwise disjoint segments might be the edges of a planar straight line graph (PSLG). Indeed, one of the most important application of red-blue segment intersection involves computing the overlay of two PSLG's (one red and the other blue) This is also called the *map overlay problem*, and is often used in geographic information systems. The most time consuming part of the map overlay problem is determining which pairs of segments overlap. See the figure below.

Let $N = n + m$ denote the total input size and let k denote the total number of bichromatic intersecting pairs. We will present an algorithm for this problem that runs in $O(k + N \log^2 N)$ time for the reporting problem and

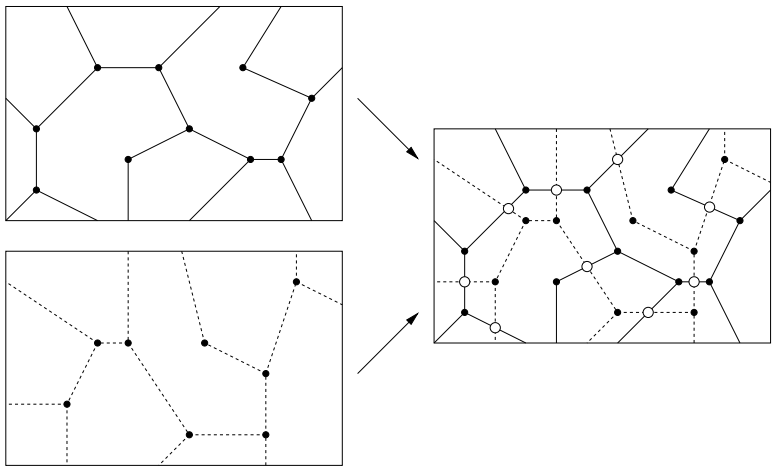


Figure 46: Red-blue line segment intersection. The algorithm outputs the white intersection points between segments of different colors. The segments of each color are pairwise disjoint (except possibly at their endpoints).

$O(N \log^2 N)$ time for the counting problem. Both algorithms use $O(N \log N)$ space. Although we will not discuss it (but the original paper does) it is possible to remove a factor of $\log n$ from both the running time and space, using a somewhat more sophisticated variant of the algorithm that we will present.

Because the set of red segments are each pairwise disjoint as are the blue segments, it follows that we could solve the reporting problem by our plane sweep algorithm for segment intersection (as discussed in an earlier lecture) in $O((N + k) \log N)$ time and $O(N)$ space. Thus, the more sophisticated algorithm is an improvement on this. However, plane sweep will not allow us to solve the counting problem.

The Hereditary Segment Tree: Recall that we are given two sets B and R, consisting of, respectively, m and n line segments in the plane, and let $N = m + n$. Let us make the general position assumption that the $2N$ endpoints of these line segments have distinct x-coordinates. The x-coordinates of these endpoints subdivide the x-axis into $2N + 1$ intervals, called *atomic intervals*. We construct a balanced binary tree whose leaves are in 1–1 correspondence with these intervals, ordered from left to right. Each internal node u of this tree is associated with an interval I_u of the x-axis, consisting of the union of the intervals of its descendent leaves. We can think of each such interval as a vertical slab S_u whose intersection with the x-axis is I_u. (See the figure below, left.)

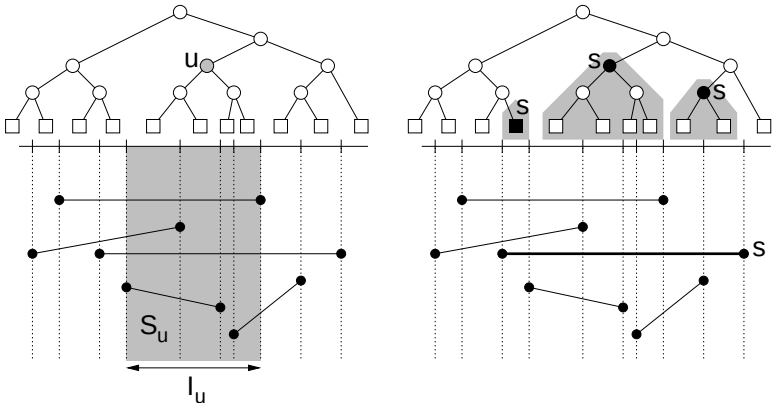


Figure 47: Hereditary Segment Tree: Intervals, slabs and the nodes associated with a segment.

We associate a segment s with a set of nodes of the tree. A segment is said to *span* interval I_u if its projection

covers this interval. We associate a segment s with a node u if s spans I_u but s does not span I_p, where p is u's parent. (See the figure above, right.)

Each node (internal or leaf) of this tree is associated with a list, called the *blue standard list*, B_u of all blue line segments whose vertical projection contains I_u but does not contain I_p, where p is the parent of u. Alternately, if we consider the nodes in whose standard list a segment is stored, the intervals corresponding to these nodes constitute a disjoint cover of the segment's vertical projection. The node is also associated with a red standard list, denoted R_u, which is defined analogously for the red segments. (See the figure below, left.)

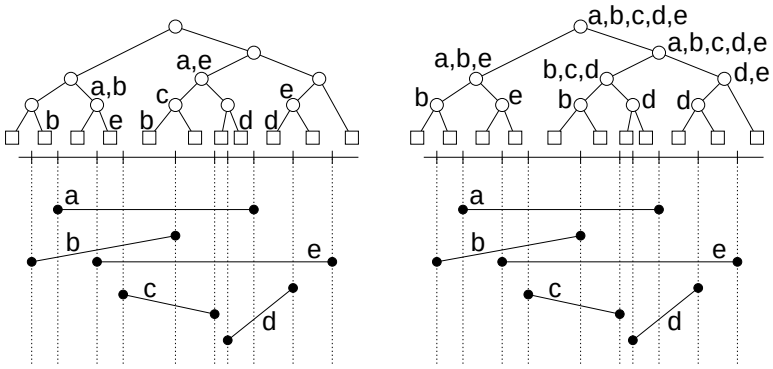


Figure 48: Hereditary Segment Tree with standard lists (left) and hereditary lists (right).

Each node u is also associated with a list B_u^*, called the *blue hereditary list*, which is the union of the B_v for all proper descendents v or u. The red hereditary list R_u^* is defined analogously. (Even though a segment may occur in the standard list for many descendents, there is only one copy of each segment in the hereditary lists.) The segments of R_u and B_u are called the *long segments*, since they span the entire interval. The segments of R_u^* and B_u^* are called the *short segments*, since they do not span the entire interval.

By the way, if we ignored the fact that we have two colors of segments and just considered the standard lists, the resulting tree is called a *segment tree*. The addition of the hereditary lists makes this a *hereditary segment tree*. Our particular data structure differs from the standard hereditary segment tree in that we have partitioned the various segment lists according to whether the segment is red or blue.

Time and Space Analysis: We claim that the total size of the hereditary segment tree is $O(N \log N)$. To see this observe that each segment is stored in the standard list of at most $2 \log N$ nodes. The argument is very similar to the analysis of the 1-dimensional range tree. If you locate the left and right endpoints of the segment among the atomic intervals, these define two paths in the tree. In the same manner as canonical sets for the 1-dimensional range tree, the segment will be stored in all the "inner" nodes between these two paths. (See the figure below.) The segment will also be stored in the hereditary lists for all the ancestors of these nodes. These ancestors lie along the two paths to the left and right, and hence there are at most $2 \log N$ of them. Thus, each segment appears in at most $4 \log N$ lists, for a total size of $O(N \log N)$.

The tree can be built in $O(N \log N)$ time. In $O(N \log N)$ time we can sort the $2N$ segment endpoints. Then for each segment, we search for its left and right endpoints and insert the segment into the standard and hereditary lists for the appropriate nodes and we descend each path in $O(1)$ time for each node visited. Since each segment appears in $O(\log N)$ lists, this will take $O(\log N)$ time per segment and $O(N \log N)$ time overall.

Computing Intersections: Let us consider how to use the hereditaray segment tree to count and report bichromatic intersections. We will do this on a node-by-node basis. Consider any node u. We classify the intersections into two types, *long-long intersections* are those between a segment of B_u and R_u, and *long-short intersections* are those between a segment of B_u^* and R_u or between R_u^* and B_u. Later we will show that by considering just these intersection cases, we will consider every intersection exactly once.

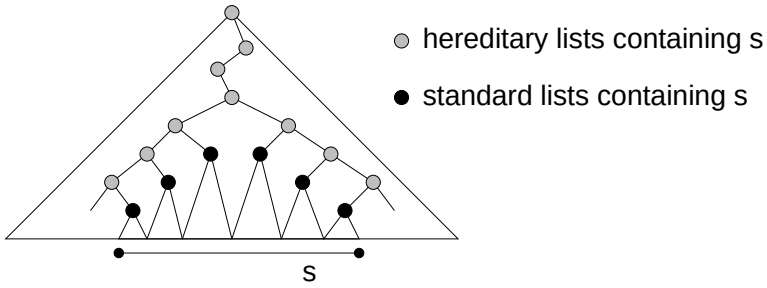


Figure 49: Standard and hereditary lists containing a segment s.

Long-long intersections: Sort each of the lists B_u and R_u of long segments in ascending order by y-coordinate. (Since the segments of each set are disjoint, this order is constant throughout the interval for each set.) Let $\langle b_1, b_2, \ldots, b_{m_u} \rangle$ and $\langle r_1, r_2, \ldots, r_{n_u} \rangle$ denote these ordered lists. Merge these lists twice, once according to their order along the left side of the slab and one according to their order along the right side of the slab. Observe that for each blue segment $b \in B_u$, this allows us to determine two indices i and j, such that b lies between r_i and r_{i+1} along the left boundary and between r_j and r_{j+1} along the right boundary. (For convenience, we can think of segment 0 as an imaginary segment at $y = -\infty$.)

It follows that if $i < j$ then b intersects the red segments $r_{i+1}, \ldots, r_j$. (See the figure below, (a)). On the other hand, if $i \geq j$ then b intersects the red segments $r_{j+1}, \ldots, r_i$. (See the figure below, (b)). We can count these intersections in $O(1)$ time or report them in time proportional to the number of intersections.

For example, consider the segment $b = b_2$ in the figure below, (c). On the left boundary it lies between r_3 and r_4, and hence $i = 3$. On the right boundary it lies between r_0 and r_1, and hence $j = 0$. (Recall that r_0 is at $y = -\infty$.) Thus, since $i \geq j$ it follows that b intersects the three red segments $\{r_1, r_2, r_3\}$.

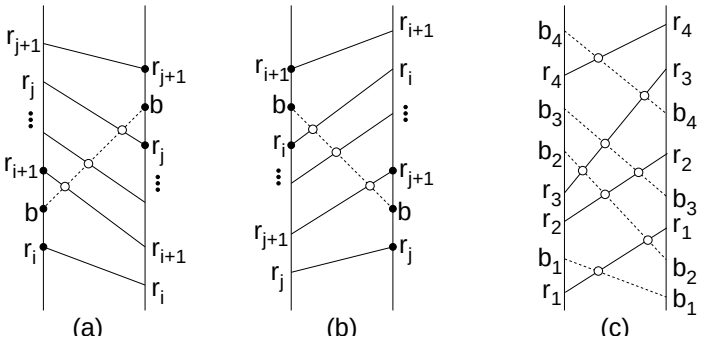


Figure 50: Red-blue intersection counting/reporting. Long-long intersections.

The total time to do this is dominated by the $O(m_u \log m_u + n_u \log n_u)$ time needed to sort both lists. The merging and counting only requires linear time.

Long-short intersections: There are two types of long-short intersections to consider. Long red and short blue, and long blue and short red. Let us consider the first one, since the other one is symmetrical.

As before, sort the long segments of R_u in ascending order according to y-coordinate, letting $\langle r_1, r_2, \ldots, r_{n_u} \rangle$ denote this ordered list. These segments naturally subdivide the slab into $n_u + 1$ trapezoids. For each short segment $b \in B_u^*$, perform two binary searches among the segments of R_u to find the lowest segment r_i and the highest segment r_j that b intersects. (See the figure above, right.) Then b intersects all the red segments $r_i, r_{i+1}, \ldots, r_j$.

Thus, after $O(\log n_u)$ time for the binary searches, the segments of R_u intersecting b can be counted in $O(1)$ time, for a total time of $O(m_u^* \log n_u)$. Reporting can be done in time proportional to the number of

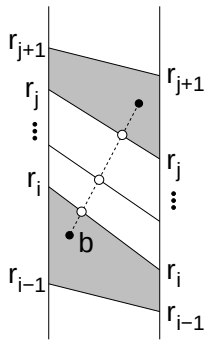


Figure 51: Red-blue intersection counting/reporting: Long-short intersections.

intersections reported. Adding this to the time for the long blue and short red case, we have a total time complexity of $O(m_u^* \log n_u + n_u^* \log m_u)$.

If we let $N_u = m_u + n_u + m_u^* + n_u^*$, then observe that the total time to process vertex u is $O(N_u \log N_u)$ time. Summing this over all nodes of the tree, and recalling that $\sum_u N_u = O(N \log N)$ we have a total time complexity of

$$T(N) \;=\; \sum_u N_u \log N_u \;\le\; \left(\sum_u N_u\right) \log N \;=\; O(N \log^2 N).$$

Correctness: To show that the algorithm is correct, we assert that each bichromatic intersection is counted exactly once. For any bichromatic intersection between b_i and r_j consider the leaf associated with the atomic interval containing this intersection point. As we move up to the ancestors of this leaf, we will encounter b_i in the standard list of one of these ancestors, denoted u_i, and will encounter r_j at some node, denoted u_j. If $u_i = u_j$ then this intersection will be detected as a long-long intersection at this node. Otherwise, one is a proper ancestor of the other, and this will be detected as a long-short intersection (with the ancestor long and descendent short).

Lecture 14: Planar Point Location and Trapezoidal Maps

Reading: Chapter 6 of the 4M's.

Point Location: The *point location problem* (in 2-space) is: given a polygonal subdivision of the plane (that is, a PSLG) with n vertices, preprocess this subdivision so that given a query point q, we can efficiently determine which face of the subdivision contains q. We may assume that each face has some identifying label, which is to be returned. We also assume that the subdivision is represented in any "reasonable" form (e.g. as a DCEL). In general q may coincide with an edge or vertex. To simplify matters, we will assume that q does not lie on an edge or vertex, but these special cases are not hard to handle.

Our goal is to develop a data structure with $O(n)$ that can answer queries in $O(\log n)$ time. For many years the best methods known had an extra log factor, either in the space or in the query time. Kirkpatrick achieved a breakthrough by presenting a time/space optimal algorithm. Kirkpatrick's algorithm has fairly high constant factors. Somewhat simpler and more practical optimal algorithms were discovered since then. We will present perhaps the simplest and most practical of the known optimal algorithms. The method is based on a randomized incremental construction, the same technique used in our linear programming algorithm.

Trapezoidal Map: The algorithm is based on a construction called a *trapezoidal map* (which also goes under many other names in the computational geometry literature). Although we normally think of the input to a point location algorithm as being a planar polygonal subdivision (or PSLG), we will define the algorithm under the

assumption that the input is just a collection of line segments $S = \{s_1, s_2, \ldots, s_n\}$, such that these line segments do not intersect except possibly at their endpoints. To construct a trapezoidal map, imagine shooting a bullet vertically upwards and downwards from each vertex in the polygonal subdivision until it hits another segment of S. (For simplicity, we will assume that there are no vertical segments in the initial subdivision and no two segments have the same x-coordinate. Both of these are easy to handle with an appropriate symbolic perturbation.) The resulting "bullet paths", together with the initial line segments define the trapezoidal map. To avoid infinite bullet paths at the top and bottom of the subdivision, we may assume that the initial subdivision is contained entirely within a large bounding rectangle. An example is shown in the figure below.

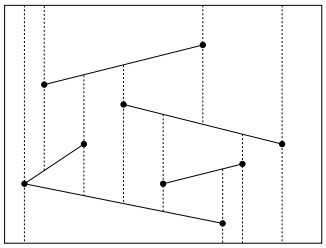


Figure 52: Trapezoidal map.

Observe that all the faces of the resulting subdivision are trapezoids with vertical sides. The left or right side might degenerate to a line segment of length zero, implying that the resulting trapezoid degenerates to a triangle. We claim that the process of converting an arbitrary polygonal subdivision into a trapezoidal decomposition increases its size by at most a constant factor. Actually this follows from the facts that we only increase the number of vertices by a constant factor and the graph is planar. But since constant factor expansions in space are significant, it is a good idea to work this through carefully. We assume that the final trapezoidal map will be given as a polygonal subdivision of the plane (represented, say, using a DCEL).

Claim: Given a polygonal subdivision with n segments, the resulting trapezoidal map has at most $6n + 4$ vertices and $3n + 1$ trapezoids.

Proof: To prove the bound on the number of vertices, observe that each vertex shoots two bullet paths, each of which will result in the creation of a new vertex. Thus each original vertex gives rise to three vertices in the final map. Since each segment has two vertices, this implies at most $6n$ vertices.

To bound the number of trapezoids, observe that for each trapezoid in the final map, its left side (and its right as well) is bounded by a vertex of the original polygonal subdivision. The left endpoint of each line segment can serve as the left bounding vertex for two trapezoids (one above the line segment and the other below) and the right endpoint of a line segment can serve as the left bounding vertex for one trapezoid. Thus each segment of the original subdivision gives rise to at most three trapezoids, for a total of $3n$ trapezoids. The last trapezoid is the one bounded by the left side of the bounding box.

An important fact to observe about each trapezoid is that it is *defined* (that is, its existence is determined) by exactly four entities from the original subdivision: a segment on top, a segment on the bottom, a bounding vertex on the left, and a bounding vertex on the right. This simple observation will play an important role in the analysis.

Trapezoidal decompositions, like triangulations, are interesting data structures in their own right. It is another example of the idea of converting a complex shape into a disjoint collection of simpler objects. The fact that the sides are vertical makes trapezoids simpler than arbitrary quadrilaterals. Finally observe that the trapezoidal decomposition is a refinement of the original polygonal subdivision, and so once we know which face of the trapezoidal map a query point lies in, we will know which face of the original subdivision it lies in (either implicitly, or because we label each face of the trapezoidal map in this way).

Construction: We could construct the trapezoidal map easily by plane sweep. (This should be an easy exercise by this point, but think about how you would do it.) We will build the trapezoidal map by a randomized incremental algorithm, because the point location algorithm is based on this construction. (In fact, historically, this algorithm arose as a method for computing the trapezoidal decomposition of a collection of intersecting line segments, and the point location algorithm arose as an artifact that was needed in the construction.)

The incremental algorithm starts with the initial bounding rectangle (that is, one trapezoid) and then we add the segments of the polygonal subdivision one by one in random order. As each segment is added, we update the trapezoidal map. Let S_i denote the subset consisting of the first i (random) segments, and let $\mathcal{T}_i$ denote the resulting trapezoidal map.

To perform the update this we need to know which trapezoid the left endpoint of the segment lies in. We will let this question go until later, since it will be answered by the point location algorithm itself. Then we trace the line segment from left to right, determining which trapezoids it intersects. Finally, we go back to these trapezoids and "fix them up". There are two things that are involved in fixing. First, the left and right endpoints of the new segment need to have bullets fired from them. Second, one of the earlier bullet paths might hit this line segment. When that happens the bullet path must be trimmed back. (We know which vertices are from the original subdivision vertices, so we know which side of the bullet path to trim.) The process is illustrated in the figure below.

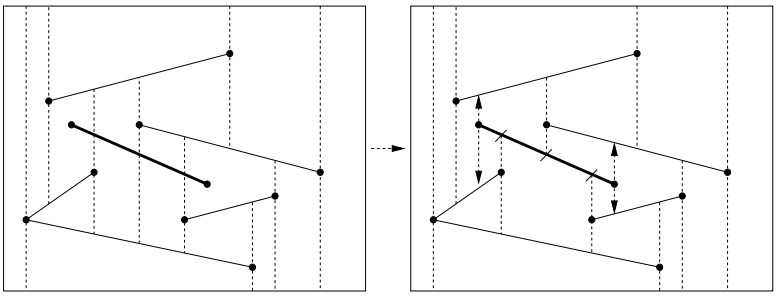


Figure 53: Incremental update.

Observe that the structure of the trapezoidal decomposition does not depend on the order in which the segments are added. This observation will be important for the probabilistic analysis. The following is also important to the analysis.

Claim: Ignoring the time spent to locate the left endpoint of an segment, the time that it takes to insert the ith segment and update the trapezoidal map is $O(k_i)$, where k_i is the number of newly created trapezoids.

Proof: Consider the insertion of the ith segment, and let K denote the number of bullet paths that this segment intersects. We need to shoot four bullets (two from each endpoint) and then trim each of the K bullet paths, for a total of $K + 4$ operations that need to be performed. If the new segment did not cross any of the bullet paths, then we would get exactly four new trapezoids. For each of the K bullet paths we cross, we add one more to the number of newly created trapezoids, for a total of $K + 4$. Thus, letting $k_i = K + 4$ be the number of trapezoids created, the number of update operations is exactly k_i. Each of these operations can be performed in $O(1)$ time given any reasonable representation of the trapezoidal map (e.g. a DCEL).

Analysis: We left one important detail out, namely, how we locate the trapezoid containing left endpoint of each new segment that we add. Let's ignore this for now. (We will see later that this is $O(\log n)$ time on average). We will show that the expected time to add each new segment is $O(1)$. Since there are n insertions, this will lead to a total expected time complexity of $O(n(1 + \log n)) = O(n \log n)$.

We know that the size of the final trapezoidal map is $O(n)$. It turns out that the total size of the point location data structure will actually be proportional to the number of new trapezoids that are created with each insertion. In the

worst case, when we add the ith segment, it might cut through a large fraction of the existing $O(i)$ trapezoids, and this would lead to a total size proportional to $\sum_{i=1}^{n} i = n^2$. However, the magic of the incremental construction is that this does not happen. We will show that on average, each insertion results in only a constant number of trapezoids being created.

(You might stop to think about this for a moment, because it is rather surprising at first. Clearly if the segments are short, then each segment might not intersect very many trapezoids. But what if all the segments are long? It seems as though it might be possible to construct a counterexample. Give it a try before you read this.)

Lemma: Consider the randomized incremental construction of a trapezoidal map, and let k_i denote the number of new trapezoids created when the ith segment is added. Then $E[k_i] = O(1)$, where the expectation is taken over all permutations of the segments.

Proof: The analysis will be based on a backwards analysis. Recall that such an analysis is based on analyzing the expected value assuming that the last insertion was random.

Let $\mathcal{T}_i$ denote the trapezoidal map after the insertion of the ith segment. Because we are averaging over all permutations, among the i segments that are present in $\mathcal{T}_i$, each one has an equal probability $1/i$ of being the last one to have been added. For each of the segments s we want to count the number of trapezoids that would have been created, had s been the last segment to be added. Let's say that a trapezoid Δ *depends* on an segment s, if s would have caused Δ to be created, had s been added last. We want to count the number of trapezoids that depend on each segment, and then compute the average over all segments. If we let $\delta(\Delta, s) = 1$ if segment s depends on Δ, and 0 otherwise, then the expected complexity is

$$E[k_i] = \frac{1}{i} \sum_{s \in S_i} \sum_{\Delta \in \mathcal{T}_i} \delta(\Delta, s).$$

Some segments might have resulted in the creation of lots of trapezoids and other very few. How do we get a handle on this quantity? The trick is, rather than count the number of trapezoids that depend on each segment, we count the number segments that each trapezoid depends on. (The old combinatorial trick of reversing the order of summation.) In other words we want to compute:

$$E[k_i] = \frac{1}{i} \sum_{\Delta \in \mathcal{T}_i} \sum_{s \in S_i} \delta(\Delta, s).$$

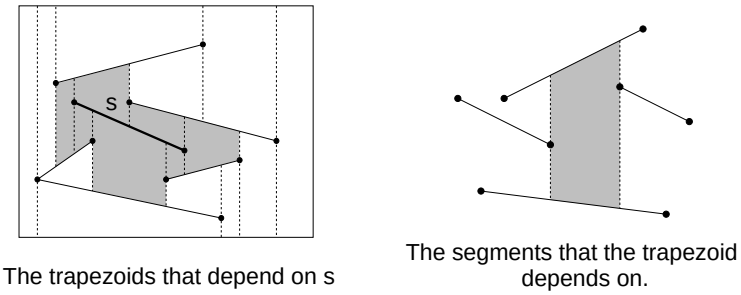


Figure 54: Trapezoid-segment dependencies.

This is much easier to determine. In particular, each trapezoid is bounded by at most four sides (recall that degenerate trapezoids are possible). The top and bottom sides are each determined by a segment of S_i, and clearly if either of these was the last to be added, then this trapezoid would have come into existence as a result. The left and right sides are each determined by a endpoint of a segment in S_i, and clearly if either of these was the last to be added, then this trapezoid would have come into existence. Thus, each

trapezoid is dependent on at most four segments, implying that $\sum_{s \in S_i} \delta(\Delta, s) \leq 4$. Since $\mathcal{T}_i$ consists of $O(i)$ trapezoids we have

$$E[k_i] \leq \frac{1}{i} \sum_{\Delta \in \mathcal{T}_i} 4 = \frac{1}{i} 4|\mathcal{T}_i| = \frac{1}{i} 4O(i) = O(1).$$

Since the expected number of new trapezoids created with each insertion is $O(1)$, it follows that the total number of trapezoids that are created in the entire process is $O(n)$. This fact is important in bounding the total time needed for the randomized incremental algorithm. The only question that we have not considered in the construction is how to locate the trapezoid that contains left endpoint of each newly added segment. We will consider this question, and the more general question of how to do point location next time.

Lecture 15: More on Planar Point Location

Reading: Chapter 6 of the 4M's.

Point Location: Last time we presented a randomized incremental algorithm for building a trapezoidal map. Today we consider how to modify this algorithm to answer point location queries for the resulting trapezoidal decomposition. The preprocessing time will be $O(n \log n)$ in the expected case (as was the time to construct the trapezoidal map), and the space and query time will be $O(n)$ and $O(\log n)$, respectively, in the expected case. Note that this may be applied to any spatial subdivision, by treating it as a set of line segments, and then building the resulting trapezoidal decomposition and using this data structure.

Recall that we treat the input as a set of segments $S = \{s_1, \ldots, s_n\}$ (permuted randomly), that S_i denotes the subset consisting of the first i segments of S, and $\mathcal{T}_i$ denotes the trapezoidal map of S_i. One important element of the analysis to remember from last time is that each time we add a new line segment, it may result in the creation of the collection of new trapezoids, which were said to *depend* on this line segment. We presented a backwards analysis that the number of new trapezoids that are created with each stage is expected to be $O(1)$. This will play an important role in today's analysis.

Point Location Data Structure: The point location data structure is based on a rooted directed acyclic graph. Each node will either two or zero outgoing edges. Nodes with zero outgoing edges are called *leaves*. There will be one leaf for each trapezoid in the map. The other nodes are called *internal nodes*, and they are used to guide the search to the leaves. This is not a binary tree, however, because subtrees may be shared.

There are two types of internal nodes, *x-nodes* and *y-nodes*. Each x-node contains the x-coordinate x_0 of an endpoint of one of the segments. Its two children correspond to the points lying to the left and to the right of the vertical line $x = x_0$. Each y-node contains a pointer to a line segment of the subdivision. The left and right children correspond to whether the query point is above or below the line containing this segment, respectively. Note that the search will reach a y-node only if we have already verified that the x-coordinate of the query point lies within the vertical slab that contains this segment.

Our construction of the point location data structure mirrors the incremental construction of the trapezoidal map. In particular, if we freeze the construction just after the insertion of any segment, the current structure will be a point location structure for the current trapezoidal map. In the figure below we show a simple example of what the data structure looks like for two line segments. The circular nodes are the x-nodes and the hexagonal nodes are the y-nodes. There is one leaf for each trapezoid. The y-nodes are shown as hexagons. For example, if the query point is in trapezoid D, we would first detect that it is to the right of enpoint p_1, then left of q_1, then below s_1 (the right child), then right of p_2, then above s_2 (the left child).

Incremental Construction: The question is how do we build this data structure incrementally? First observe that when a new line segment is added, we only need to adjust the portion of the tree that involves the trapezoids that

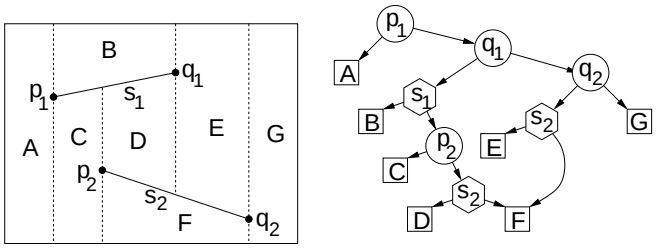


Figure 55: Trapezoidal map point location data structure.

have been deleted as a result of this new addition. Each trapezoid that is deleted will be replaced with a search structure that determines the newly created trapezoid that contains it.

Suppose that we add a line segment s. This results in the replacement of an existing set of trapezoids with a set of new trapezoids. As a consequence, we will replace the leaves associated with each such deleted trapezoid with a small search structure, which locates the new trapezoid that contains the query point. There are three cases that arise, depending on how many endpoints of the segment lie within the current trapezoid.

Single (left or right) endpoint: A single trapezoid A is replaced by three trapezoids, denoted X, Y, and Z. Letting p denote the endpoint, we create an x-node for p, and one child is a leaf node for the trapezoid X that lies outside vertical projection of the segment. For the other child, we create a y-node whose children are the trapezoids Y and Z lying above and below the segment, respectively. (See the figure below left.)

No segment endpoints: This happens when the segment cuts completely through a trapezoid. A single trapezoid is replaced by two trapezoids, one above and one below the segment, denoted Y and Z. We replace the leaf node for the original trapezoid with a y-node whose children are leaf nodes associated with Y and Z. (This case is not shown in the figure.)

Two segment endpoints: This happens when the segment lies entirely inside the trapezoid. In this case one trapezoid A is replaced by four trapezoids, U, X, Y, and Z. Letting p and q denote the left and right endpoints of the segment, we create two x-nodes, one for the p and the other for q. We create a y-node for the line segment, and join everything together as shown in the figure.

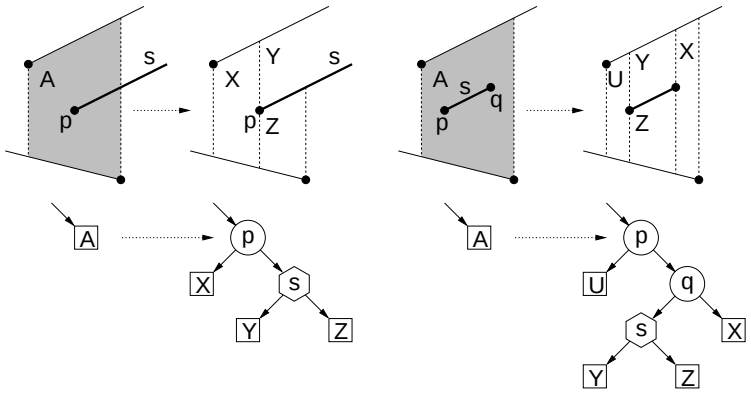


Figure 56: Line segment insertion and updates to the point location structure. The single-endpoint case (left) and the two-endpoint case (right). The no-endpoint case is not shown.

It is important to notice that (through sharing) each trapezoid appears exactly once as a leaf in the resulting structure. An example showing the complete transformation to the data structure after adding a single segment is shown in the figure below.

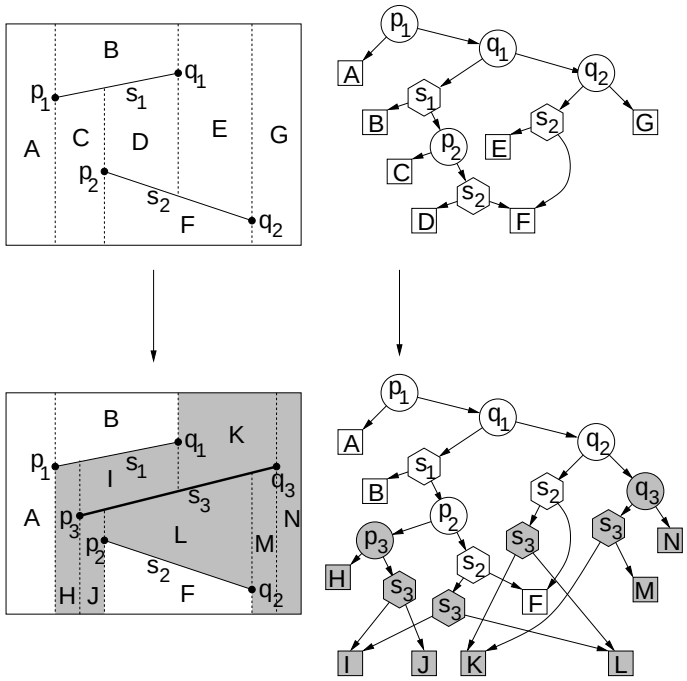


Figure 57: Line segment insertion.

Analysis: We claim that the size of the point location data structure is $O(n)$ and the query time is $O(\log n)$, both in the expected case. As usual, the expectation depends only on the order of insertion, not on the line segments or the location of the query point.

To prove the space bound of $O(n)$, observe that the number of new nodes added to the structure with each new segment is proportional to the number of newly created trapezoids. Last time we showed that with each new insertion, the expected number of trapezoids that were created was $O(1)$. Therefore, we add $O(1)$ new nodes with each insertion in the expected case, implying that the total size of the data structure is $O(n)$.

Analyzing the query time is a little subtler. In a normal probabilistic analysis of data structures we think of the data structure as being fixed, and then compute expectations over random queries. Here the approach will be to imagine that we have exactly one query point to handle. The query point can be chosen arbitrarily (imagine an adversary that tries to select the worst-possible query point) but this choice is made without knowledge of the random choices the algorithm makes. We will show that for any query point, most random orderings of the line segments will lead to a search path of length $O(\log n)$ in the resulting tree.

Let q denote the query point. Rather than consider the search path for q in the final search structure, we will consider how q moves incrementally through the structure with the addition of each new line segment. Let Δ_i denote the trapezoid of the map that q lies in after the insertion of the first i segments. Observe that if $\Delta_{i-1} = \Delta_i$, then insertion of the ith segment did not affect the trapezoid that q was in, and therefore q will stay where it is relative to the current search structure. (For example, if q was in trapezoid B prior to adding s_3 in the figure above, then the addition of s_3 does not incur any additional cost to locating q.) However, if $\Delta_{i-1} \neq \Delta_i$, then the insertion of the ith segment caused q's trapezoid to be deleted. As a result, q must locate itself with respect to the newly created trapezoids that overlap Δ_{i-1}. Since there are a constant number of such trapezoids (at most four), there will be $O(1)$ work needed to locate q with respect to these. In particular, q may fall as much as three levels in the search tree. (The worst case occurs in the two-endpoint case, if the query point falls into one of the trapezoids X or Y lying above or below the segment.)

To compute the expected length of the search path, it suffices to compute the probability that the trapezoid that

contains q changes as a result of the ith insertion. Let P_i denote this probability (where the probability is taken over random insertion orders, irrespective of the choice of q). Since q could fall through up to three levels in the search tree as a result of each the insertion, the expected length of q's search path in the final structure is at most

$$\sum_{i=1}^{n} 3P_i.$$

We will show that $P_i \leq 4/i$. From this it will follow that the expected path length is at most

$$\sum_{i=1}^{n} 3\frac{4}{i} = 12\sum_{i=1}^{n} \frac{1}{i},$$

which is roughly $12 \ln n = O(\log n)$ by the Harmonic series.

To show that $P_i \leq 4/i$, we apply a backwards analysis. In particular, consider the trapezoid that contains q *after* the ith insertion. Recall from last time that this trapezoid is dependent on at most four segments, which define the top and bottom edges, and the left and right sides of the trapezoid. Since each segment is equally likely to be the last segment to have been added, the probability that the last insertion caused q to belong to a new trapezoid is at most $4/i$. This completes the proof.

Guarantees on Search Time: One shortcoming with this analysis is that even though the search time is provably small in the expected case for a given query point, it might still be the case that once the data structure has been constructed there is a single very long path in the search structure, and the user repeatedly performs queries along this path. Hence, the analysis provides no guarantees on the running time of all queries.

Although we will not prove it, the book presents a stronger result, namely that the length of the maximum search path is also $O(\log n)$ with high probability. In particular, they prove the following.

Lemma: Given a set of n non-crossing line segments in the plane, and a parameter $\lambda > 0$, the probability that the total depth of the randomized search structure exceeds $3\lambda \ln(n+1)$, is at most $2/(n+1)^{\lambda \ln 1.25 - 3}$.

For example, for $\lambda = 20$, the probability that the search path exceeds $60 \ln(n+1)$ is at most $2/(n+1)^{1.5}$. (The constant factors here are rather weak, but a more careful analysis leads to a better bound.)

Nonetheless, this itself is enough to lead to variant of the algorithm for which $O(\log n)$ time is guaranteed. Rather than just running the algorithm once and taking what it gives, instead keep running it and checking the structure's depth. As soon as the depth is at most $c \log n$ for some suitably chosen c, then stop here. Depending on c and n, the above lemma indicates how long you may need to expect to repeat this process until the final structure has the desired depth. For sufficiently large c, the probability of finding a tree of the desired depth will be bounded away from 0 by some constant factor, and therefore after a constant number of trials (depending on this probability) you will eventually succeed in finding a point location structure of the desired depth. A similar argument can be applied to the space bounds.

Theorem: Given a set of n non-crossing line segments in the plane, in expected $O(n \log n)$ time, it is possible to construct a point location data structure of (worst case) size $O(n)$ that can answer point location queries in (worst case) time $O(\log n)$.

Lecture 16: Voronoi Diagrams and Fortune's Algorithm

Reading: Chapter 7 in the 4M's.

Euclidean Geometry: We now will make a subtle but important shift. Up to now, virtually everything that we have done has not needed the notion of angles, lengths, or distances (except for our work on circles). All geometric

tests were made on the basis of orientation tests, a purely affine construct. But there are important geometric algorithms that depend on nonaffine quantities such as distances and angles. Let us begin by defining the *Euclidean length* of a vector $v = (v_x, v_y)$ in the plane to be $|v| = \sqrt{v_x^2 + v_y^2}$. In general, in dimension d it is

$$|v| = \sqrt{v_1^2 + \ldots + v_d^2}.$$

The distance between two points p and q, denoted $\mathrm{dist}(p, q)$ or $|pq|$, is defined to be $|p - q|$.

Voronoi Diagrams: Voronoi diagrams (like convex hulls) are among the most important structures in computational geometry. A Voronoi diagram records information about what is close to what. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane (or in any dimensional space), which we call *sites*. Define $\mathcal{V}(p_i)$, the *Voronoi cell* for p_i, to be the set of points q in the plane that are closer to p_i than to any other site. That is, the Voronoi cell for p_i is defined to be:

$$\mathcal{V}(p_i) = \{q \mid |p_i q| < |p_j q|, \forall j \neq i\}.$$

Another way to define $\mathcal{V}(p_i)$ is in terms of the intersection of halfplanes. Given two sites p_i and p_j, the set of points that are strictly closer to p_i than to p_j is just the *open* halfplane whose bounding line is the perpendicular bisector between p_i and p_j. Denote this halfplane $h(p_i, p_j)$. It is easy to see that a point q lies in $\mathcal{V}(p_i)$ if and only if q lies within the intersection of $h(p_i, p_j)$ for all $j \neq i$. In other words,

$$\mathcal{V}(p_i) = \cap_{j \neq i} h(p_i, p_j).$$

Since the intersection of halfplanes is a (possibly unbounded) convex polygon, it is easy to see that $\mathcal{V}(p_i)$ is a (possibly unbounded) convex polygon. Finally, define the *Voronoi diagram* of P, denoted $\mathrm{Vor}(P)$ to be what is left of the plane after we remove all the (open) Voronoi cells. It is not hard to prove (see the text) that the Voronoi diagram consists of a collection of line segments, which may be unbounded, either at one end or both. An example is shown in the figure below.

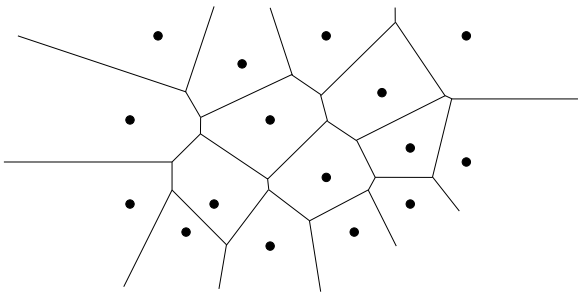


Figure 58: Voronoi diagram

Voronoi diagrams have a number of important applications. These include:

Nearest neighbor queries: One of the most important data structures problems in computational geometry is solving nearest neighbor queries. Given a point set P, and given a query point q, determine the closest point in P to q. This can be answered by first computing a Voronoi diagram and then locating the cell of the diagram that contains q. (We have already discussed point location algorithms.)

Computational morphology: Some of the most important operations in morphology (used very much in computer vision) is that of "growing" and "shrinking" (or "thinning") objects. If we grow a collection of points, by imagining a grass fire starting simultaneously from each point, then the places where the grass fires meet will be along the Voronoi diagram. The *medial axis* of a shape (used in computer vision) is just a Voronoi diagram of its boundary.

Facility location: We want to open a new Blockbuster video. It should be placed as far as possible from any existing video stores. Where should it be placed? It turns out that the vertices of the Voronoi diagram are the points that are locally at maximum distances from any other point in the set.

Neighbors and Interpolation: Given a set of measured height values over some geometric terrain. Each point has (x, y) coordinates and a height value. We would like to interpolate the height value of some query point that is not one of our measured points. To do so, we would like to interpolate its value from neighboring measured points. One way to do this, called *natural neighbor interpolation*, is based on computing the Voronoi neighbors of the query point, assuming that it has one of the original set of measured points.

Properties of the Voronoi diagram: Here are some observations about the structure of Voronoi diagrams in the plane.

Voronoi edges: Each point on an edge of the Voronoi diagram is equidistant from its two nearest neighbors p_i and p_j. Thus, there is a circle centered at such a point such that p_i and p_j lie on this circle, and no other site is interior to the circle.

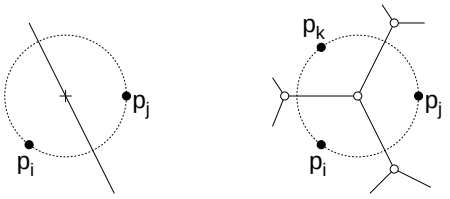


Figure 59: Properties of the Voronoi diagram.

Voronoi vertices: It follows that the vertex at which three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ intersect, called a *Voronoi vertex* is equidistant from all sites. Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior.

Degree: If we make the general position assumption that no four sites are cocircular, then the vertices of the Voronoi diagram all have degree three.

Convex hull: A cell of the Voronoi diagram is unbounded if and only if the corresponding site lies on the convex hull. (Observe that a site is on the convex hull if and only if it is the closest point from some point at infinity.) Thus, given a Voronoi diagram, it is easy to extract the convex hull in linear time.

Size: If n denotes the number of sites, then the Voronoi diagram is a planar graph (if we imagine all the unbounded edges as going to a common vertex infinity) with exactly n faces. It follows from Euler's formula that the number of Voronoi vertices is at most $2n - 5$ and the number of edges is at most $3n - 6$. (See the text for details.)

Computing Voronoi Diagrams: There are a number of algorithms for computing Voronoi diagrams. Of course, there is a naive $O(n^2 \log n)$ time algorithm, which operates by computing $\mathcal{V}(p_i)$ by intersecting the $n - 1$ bisector halfplanes $h(p_i, p_j)$, for $j \neq i$. However, there are much more efficient ways, which run in $O(n \log n)$ time. Since the convex hull can be extracted from the Voronoi diagram in $O(n)$ time, it follows that this is asymptotically optimal in the worst-case.

Historically, $O(n^2)$ algorithms for computing Voronoi diagrams were known for many years (based on incremental constructions). When computational geometry came along, a more complex, but asymptotically superior $O(n \log n)$ algorithm was discovered. This algorithm was based on divide-and-conquer. But it was rather complex, and somewhat difficult to understand. Later, Steven Fortune invented a plane sweep algorithm for the problem, which provided a simpler $O(n \log n)$ solution to the problem. It is his algorithm that we will discuss. Somewhat later still, it was discovered that the incremental algorithm is actually quite efficient, if it is run as a randomized incremental algorithm. We will discuss this algorithm later when we talk about the dual structure, called a Delaunay triangulation.

Fortune's Algorithm: Before discussing Fortune's algorithm, it is interesting to consider why this algorithm was not invented much earlier. In fact, it is quite a bit trickier than any plane sweep algorithm we have seen so far. The key to any plane sweep algorithm is the ability to discover all "upcoming" events in an efficient manner. For example, in the line segment intersection algorithm we considered all pairs of line segments that were adjacent in the sweep-line status, and inserted their intersection point in the queue of upcoming events. The problem with the Voronoi diagram is that of predicting when and where the upcoming events will occur. Imagine that you are designing a plane sweep algorithm. Behind the sweep line you have constructed the Voronoi diagram based on the points that have been encountered so far in the sweep. The difficulty is that a site that lies ahead of the sweep line may generate a Voronoi vertex that lies behind the sweep line. How could the sweep algorithm know of the existence of this vertex until it sees the site. But by the time it sees the site, it is too late. It is these *unanticipated events* that make the design of a plane sweep algorithm challenging. (See the figure below.)

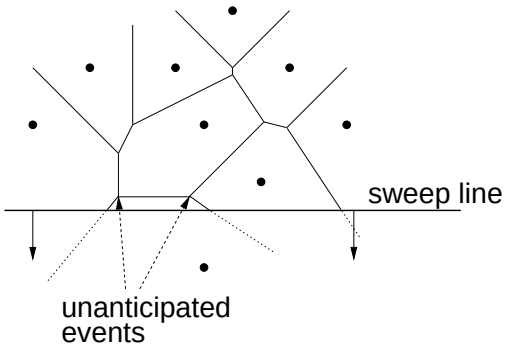


Figure 60: Plane sweep for Voronoi diagrams. Note that the position of the indicated vertices depends on sites that have not yet been encountered by the sweep line, and hence are unknown to the algorithm. (Note that the sweep line moves from top to bottom.)

Fortune made the clever observation of rather than computing the Voronoi diagram through plane sweep in its final form, instead to compute a "distorted" but topologically equivalent version of the diagram. This distorted version of the diagram was based on a transformation that alters the way that distances are measured in the plane. The resulting diagram had the same topological structure as the Voronoi diagram, but its edges were parabolic arcs, rather than straight line segments. Once this distorted diagram was generated, it was an easy matter to "undistort" it to produce the correct Voronoi diagram.

Our presentation will be different from Fortune's. Rather than distort the diagram, we can think of this algorithm as distorting the sweep line. Actually, we will think of two objects that control the sweeping process. First, there will be a horizontal sweep line, moving from top to bottom. We will also maintain an x-monotonic curve called a *beach line*. (It is so named because it looks like waves rolling up on a beach.) The beach line is a monotone curve formed from pieces of parabolic arcs. As the sweep line moves downward, the beach line follows just behind. The job of the beach line is to prevent us from seeing unanticipated events until the sweep line encounters the corresponding site.

The Beach Line: In order to make these ideas more concrete, recall that the problem with ordinary plane sweep is that sites that lie below the sweep line may affect the diagram that lies above the sweep line. To avoid this problem, we will maintain only the portion of the diagram that cannot be affected by anything that lies below the sweep line. To do this, we will subdivide the halfplane lying above the sweep line into two regions: those points that are closer to some site p above the sweep line than they are to the sweep line itself, and those points that are closer to the sweep line than any site above the sweep line.

What are the geometric properties of the boundary between these two regions? The set of points q that are equidistant from the sweep line to their nearest site above the sweep line is called the *beach line*. Observe that for any point q above the beach line, we know that its closest site cannot be affected by any site that lies below

the sweep line. Hence, the portion of the Voronoi diagram that lies above the beach line is "safe" in the sense that we have all the information that we need in order to compute it (without knowing about which sites are still to appear below the sweep line).

What does the beach line look like? Recall from high school geometry that the set of points that are equidistant from a site lying above a horizontal line and the line itself forms a parabola that is open on top (see the figure below, left). With a little analytic geometry, it is easy to show that the parabola becomes "skinnier" as the site becomes closer to the line. In the degenerate case when the line contains the site the parabola degenerates into a vertical ray shooting up from the site. (You should work through the distance equations to see why this is so.)

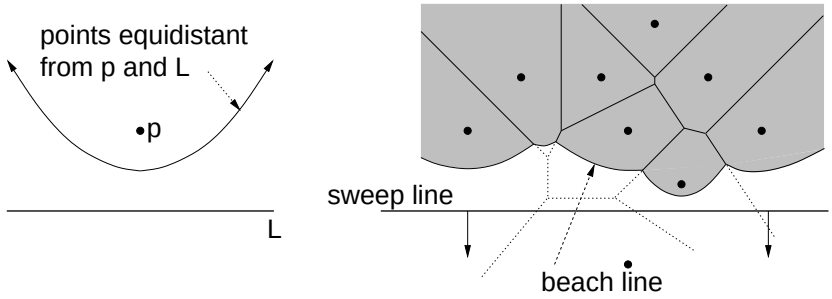


Figure 61: The beach line. Notice that only the portion of the Voronoi diagram that lies above the beach line is computed. The sweep line status maintains the intersection of the Voronoi diagram with the beach line.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site. Note that the parabola of some sites above the beach line will not touch the lower envelope and hence will not contribute to the beach line. Because the parabolas are x-monotone, so is the beach line. Also observe that the vertex where two arcs of the beach line intersect, which we call a *breakpoint*, is a point that is equidistant from two sites and the sweep line, and hence must lie on some Voronoi edge. In particular, if the beach line arcs corresponding to sites p_i and p_j share a common breakpoint on the beach line, then this breakpoint lies on the Voronoi edge between p_i and p_j. From this we have the following important characterization.

Lemma: The beach line is an x-monotone curve made up of parabolic arcs. The breakpoints of the beach line lie on Voronoi edges of the final diagram.

Fortune's algorithm consists of simulating the growth of the beach line as the sweep line moves downward, and in particular tracing the paths of the breakpoints as they travel along the edges of the Voronoi diagram. Of course, as the sweep line moves the parabolas forming the beach line change their shapes continuously. As with all plane-sweep algorithms, we will maintain a sweep-line status and we are interested in simulating the discrete event points where there is a "significant event", that is, any event that changes the topological structure of the Voronoi diagram and the beach line.

Sweep Line Status: The algorithm maintain the current location (y-coordinate) of the sweep line. It stores, in left-to-right order the set of sites that define the beach line. **Important:** The algorithm never needs to store the parabolic arcs of the beach line. It exists solely for conceptual purposes.

Events: There are two types of events.

Site events: When the sweep line passes over a new site a new arc will be inserted into the beach line.

Vertex events: (What our text calls *circle events*.) When the length of a parabolic arc shrinks to zero, the arc disappears and a new Voronoi vertex will be created at this point.

The algorithm consists of processing these two types of events. As the Voronoi vertices are being discovered by vertex events, it will be an easy matter to update a DCEL for the diagram as we go, and so to link the entire diagram together. Let us consider the two types of events that are encountered.

Site events: A site event is generated whenever the horizontal sweep line passes over a site. As we mentioned before, at the instant that the sweep line touches the point, its associated parabolic arc will degenerate to a vertical ray shooting up from the point to the current beach line. As the sweep line proceeds downwards, this ray will widen into an arc along the beach line. To process a site event we will determine the arc of the sweep line that lies directly above the new site. (Let us make the general position assumption that it does not fall immediately below a vertex of the beach line.) We then split this arc of the beach line in two by inserting a new infinitesimally small arc at this point. As the sweep proceeds, this arc will start to widen, and eventually will join up with other edges in the diagram. (See the figure below.)

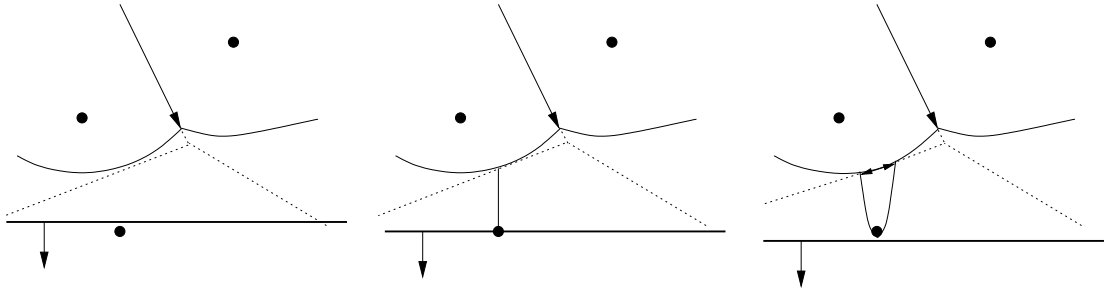


Figure 62: Site events.

It is important to consider whether this is the only way that new arcs can be introduced into the sweep line. In fact it is. We will not prove it, but a careful proof is given in the text. As a consequence of this proof, it follows that the maximum number of arcs on the beach line can be at most $2n - 1$, since each new point can result in creating one new arc, and splitting an existing arc, for a net increase of two arcs per point (except the first).

The nice thing about site events is that they are all known in advance. Thus, after sorting the points by y-coordinate, all these events are known.

Vertex events: In contrast to site events, vertex events are generated dynamically as the algorithm runs. As with the line segment plane sweep algorithm, the important idea is that each such event is generated by objects that are *neighbors* on the beach line. However, unlike the segment intersection where pairs of consecutive segments generated events, here triples of points generate the events.

In particular, consider any three consecutive sites p_i, p_j, and p_k whose arcs appear consecutively on the beach line from left to right. (See the figure below.) Further, suppose that the circumcircle for these three sites lies at least partially below the current sweep line (meaning that the Voronoi vertex has not yet been generated), and that this circumcircle contains no points lying below the sweep line (meaning that no future point will block the creation of the vertex).

Consider the moment at which the sweep line falls to a point where it is tangent to the lowest point of this circle. At this instant the circumcenter of the circle is equidistant from all three sites and from the sweep line. Thus all three parabolic arcs pass through this center point, implying that the contribution of the arc from p_j has disappeared from the beach line. In terms of the Voronoi diagram, the bisectors (p_i, p_j) and (p_j, p_k) have met each other at the Voronoi vertex, and a single bisector (p_i, p_k) remains. (See the figure below.)

Sweep-line algorithm: We can now present the algorithm in greater detail. The main structures that we will maintain are the following:

(Partial) Voronoi diagram: The partial Voronoi diagram that has been constructed so far will be stored in a DCEL. There is one technical difficulty caused by the fact that the diagram contains unbounded edges. To handle this we will assume that the entire diagram is to be stored within a large bounding box. (This box should be chosen large enough that all of the Voronoi vertices fit within the box.)

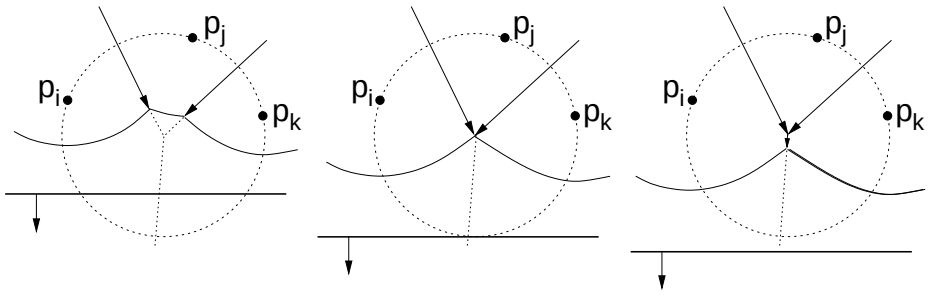


Figure 63: Vertex events.

Beach line: The beach line is represented using a dictionary (e.g. a balanced binary tree or skip list). An important fact of the construction is that *we do not explicitly store the parabolic arcs*. They are just there for the purposes of deriving the algorithm. Instead for each parabolic arc on the current beach line, we store the site that gives rise to this arc. Notice that a site may appear multiple times on the beach line (in fact linearly many times in n). But the total length of the beach line will never exceed $2n - 1$. (You should try to construct an example where a single site contributes multiple arcs to the beach line.)

Between each consecutive pair of sites p_i and p_j, there is a breakpoint. Although the breakpoint moves as a function of the sweep line, observe that it is possible to compute the exact location of the breakpoint as a function of p_i, p_j, and the current y-coordinate of the sweep line. In particular, the breakpoint is the center of a circle that passes through p_i, p_j and is tangent to the sweep line. Thus, as with beach lines, *we do not explicitly store breakpoints*. Rather, we compute them only when we need them.

The important operations that we will have to support on the beach line are

(1) Given a fixed location of the sweep line, determine the arc of the beach line that intersects a given vertical line. This can be done by a binary search on the breakpoints, which are computed "on the fly". (Think about this.)

(2) Compute predecessors and successors on the beach line.

(3) Insert an new arc p_i within a given arc p_j, thus splitting the arc for p_j into two. This creates three arcs, p_j, p_i, and p_j.

(4) Delete an arc from the beach line.

It is not difficult to modify a standard dictionary data structure to perform these operations in $O(\log n)$ time each.

Event queue: The event queue is a priority queue with the ability both to insert and delete new events. Also the event with the largest y-coordinate can be extracted. For each site we store its y-coordinate in the queue.

For each consecutive triple p_i, p_j, p_k on the beach line, we compute the circumcircle of these points. (We'll leave the messy algebraic details as an exercise, but this can be done in $O(1)$ time.) If the lower endpoint of the circle (the minimum y-coordinate on the circle) lies below the sweep line, then we create a vertex event whose y-coordinate is the y-coordinate of the bottom endpoint of the circumcircle. We store this in the priority queue. Each such event in the priority queue has a cross link back to the triple of sites that generated it, and each consecutive triple of sites has a cross link to the event that it generated in the priority queue.

The algorithm proceeds like any plane sweep algorithm. We extract an event, process it, and go on to the next event. Each event may result in a modification of the Voronoi diagram and the beach line, and may result in the creation or deletion of existing events.

Here is how the two types of events are handled:

Site event: Let p_i be the current site. We shoot a vertical ray up to determine the arc that lies immediately above this point in the beach line. Let p_j be the corresponding site. We split this arc, replacing it with the triple of arcs p_j, p_i, p_j which we insert into the beach line. Also we create new (dangling) edge for the Voronoi diagram which lies on the bisector between p_i and p_j. Some old triples that involved p_j may be deleted and some new triples involving p_i will be inserted.

For example, suppose that prior to insertion we had the beach-line sequence

$$\langle p_1, p_2, p_j, p_3, p_4 \rangle.$$

The insertion of p_i splits the arc p_j into two arcs, denoted p'_j and p''_j. Although these are separate arcs, they involve the same site, p_j. The new sequence is

$$\langle p_1, p_2, p'_j, p_i, p''_j, p_3, p_4 \rangle.$$

Any event associated with the old triple p_2, p_j, p_3 will be deleted. We also consider the creation of new events for the triples p_2, p'_j, p_i and p_i, p''_j, p_3. Note that the new triple p'_j, p_i, p''_j cannot generate an event because it only involves two distinct sites.

Vertex event: Let p_i, p_j, and p_k be the three sites that generate this event (from left to right). We delete the arc for p_j from the beach line. We create a new vertex in the Voronoi diagram, and tie the edges for the bisectors (p_i, p_j), (p_j, p_k) to it, and start a new edge for the bisector (p_i, p_k) that starts growing down below. Finally, we delete any events that arose from triples involving this arc of p_j, and generate new events corresponding to consecutive triples involving p_i and p_k (there are two of them).

For example, suppose that prior to insertion we had the beach-line sequence

$$\langle p_1, p_i, p_j, p_k, p_2 \rangle.$$

After the event we have the sequence

$$\langle p_1, p_i, p_k, p_2 \rangle.$$

We remove any events associated with the triples p_1, p_i, p_j and p_j, p_k, p_2. (The event p_i, p_j, p_k has already been removed since we are processing it now.) We also consider the creation of new events for the triples p_1, p_i, p_k and p_i, p_k, p_2.

The analysis follows a typical analysis for plane sweep. Each event involves $O(1)$ processing time plus a constant number accesses to the various data structures. Each of these accesses takes $O(\log n)$ time, and the data structures are all of size $O(n)$. Thus the total time is $O(n \log n)$, and the total space is $O(n)$.

Lecture 17: Delaunay Triangulations

Reading: Chapter 9 in the 4M's.

Delaunay Triangulations: Last time we gave an algorithm for computing Voronoi diagrams. Today we consider the related structure, called a *Delaunay triangulation* (DT). Since the Voronoi diagram is a planar graph, we may naturally ask what is the corresponding dual graph. The vertices for this dual graph can be taken to be the sites themselves. Since (assuming general position) the vertices of the Voronoi diagram are of degree three, it follows that the faces of the dual graph (excluding the exterior face) will be triangles. The resulting dual graph is a triangulation of the sites, the Delaunay triangulation.

Delaunay triangulations have a number of interesting properties, that are consequences of the structure of the Voronoi diagram.

Convex hull: The boundary of the exterior face of the Delaunay triangulation is the boundary of the convex hull of the point set.

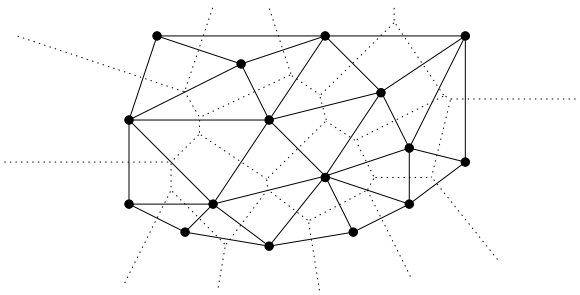


Figure 64: Delaunay triangulation.

Circumcircle property: The circumcircle of any triangle in the Delaunay triangulation is empty (contains no sites of P).

Empty circle property: Two sites p_i and p_j are connected by an edge in the Delaunay triangulation, if and only if there is an empty circle passing through p_i and p_j. (One direction of the proof is trivial from the circumcircle property. In general, if there is an empty circumcircle passing through p_i and p_j, then the center c of this circle is a point on the edge of the Voronoi diagram between p_i and p_j, because c is equidistant from each of these sites and there is no closer site.)

Closest pair property: The closest pair of sites in P are neighbors in the Delaunay triangulation. (The circle having these two sites as its diameter cannot contain any other sites, and so is an empty circle.)

If the sites are not in general position, in the sense that four or more are cocircular, then the Delaunay triangulation may not be a triangulation at all, but just a planar graph (since the Voronoi vertex that is incident to four or more Voronoi cells will induce a face whose degree is equal to the number of such cells). In this case the more appropriate term would be *Delaunay graph*. However, it is common to either assume the sites are in general position (or to enforce it through some sort of symbolic perturbation) or else to simply triangulate the faces of degree four or more in any arbitrary way. Henceforth we will assume that sites are in general position, so we do not have to deal with these messy situations.

Given a point set P with n sites where there are h sites on the convex hull, it is not hard to prove by Euler's formula that the Delaunay triangulation has $2n-2-h$ triangles, and $3n-3-h$ edges. The ability to determine the number of triangles from n and h only works in the plane. In 3-space, the number of tetrahedra in the Delaunay triangulation can range from $O(n)$ up to $O(n^2)$. In dimension n, the number of simplices (the d-dimensional generalization of a triangle) can range as high as $O(n^{\lceil d/2 \rceil})$.

Minimum Spanning Tree: The Delaunay triangulation possesses some interesting properties that are not directly related to the Voronoi diagram structure. One of these is its relation to the minimum spanning tree. Given a set of n points in the plane, we can think of the points as defining a *Euclidean graph* whose edges are all $\binom{n}{2}$ (undirected) pairs of distinct points, and edge (p_i, p_j) has weight equal to the Euclidean distance from p_i to p_j. A minimum spanning tree is a set of $n-1$ edges that connect the points (into a free tree) such that the total weight of edges is minimized. We could compute the MST using Kruskal's algorithm. Recall that Kruskal's algorithm works by first sorting the edges and inserting them one by one. We could first compute the Euclidean graph, and then pass the result on to Kruskal's algorithm, for a total running time of $O(n^2 \log n)$.

However there is a much faster method based on Delaunay triangulations. First compute the Delaunay triangulation of the point set. We will see later that it can be done in $O(n \log n)$ time. Then compute the MST of the Delaunay triangulation by Kruskal's algorithm and return the result. This leads to a total running time of $O(n \log n)$. The reason that this works is given in the following theorem.

Theorem: The minimum spanning tree of a set of points P (in any dimension) is a subgraph of the Delaunay triangulation.

Proof: Let T be the MST for P, let $w(T)$ denote the total weight of T. Let a and b be any two sites such that ab is an edge of T. Suppose to the contrary that ab is not an edge in the Delaunay triangulation. This implies that there is no empty circle passing through a and b, and in particular, the circle whose diameter is the segment ab contains a site, call it c. (See the figure below.)

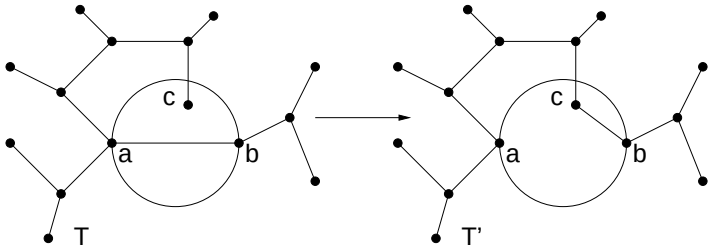


Figure 65: The Delaunay triangulation and MST.

The removal of ab from the MST splits the tree into two subtrees. Assume without loss of generality that c lies in the same subtree as a. Now, remove the edge ab from the MST and add the edge bc in its place. The result will be a spanning tree T' whose weight is

$$w(T') = w(T) + |bc| - |ab| < w(T).$$

The last inequality follows because ab is the diameter of the circle, implying that $|bc| < |ab|$. This contradicts the hypothesis that T is the MST, completing the proof.

By the way, this suggests another interesting question. Among all triangulations, we might ask, does the Delaunay triangulation minimize the total edge length? The answer is no (and there is a simple four-point counterexample). However, this claim was made in a famous paper on Delaunay triangulations, and you may still hear it quoted from time to time. The triangulation that minimizes total edge weight is called the *minimum weight triangulation*. To date, no polynomial time algorithm is known for computing it, and the problem is not known to be NP-complete.

Maximizing Angles and Edge Flipping: Another interesting property of Delaunay triangulations is that among all triangulations, the Delaunay triangulation maximizes the minimum angle. This property is important, because it implies that Delaunay triangulations tend to avoid skinny triangles. This is useful for many applications where triangles are used for the purposes of interpolation.

In fact a much stronger statement holds as well. Among all triangulations with the same smallest angle, the Delaunay triangulation maximizes the second smallest angle, and so on. In particular, any triangulation can be associated with a sorted *angle sequence*, that is, the increasing sequence of angles $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ appearing in the triangles of the triangulation. (Note that the length of the sequence will be the same for all triangulations of the same point set, since the number depends only on n and h.)

Theorem: Among all triangulations of a given point set, the Delaunay triangulation has the lexicographically largest angle sequence.

Before getting into the proof, we should recall a few basic facts about angles from basic geometry. First, recall that if we consider the circumcircle of three points, then each angle of the resulting triangle is exactly half the angle of the minor arc subtended by the opposite two points along the circumcircle. It follows as well that if a point is inside this circle then it will subtend a larger angle and a point that is outside will subtend a smaller angle. This in the figure part (a) below, we have $\theta_1 > \theta_2 > \theta_3$.

We will not give a formal proof of the theorem. (One appears in the text.) The main idea is to show that for any triangulation that fails to satisfy the empty circle property, it is possible to perform a local operation, called

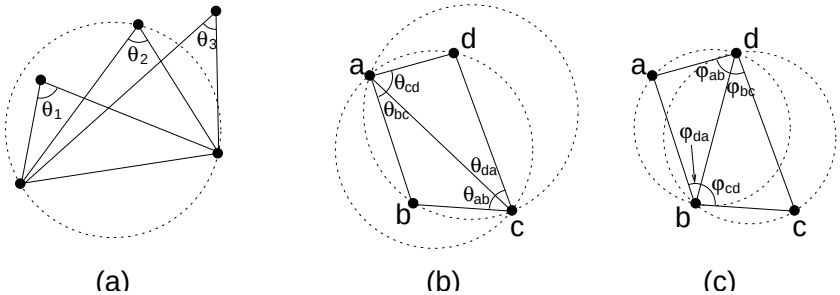


Figure 66: Angles and edge flips.

an *edge flip*, which increases the lexicographical sequence of angles. An edge flip is an important fundamental operation on triangulations in the plane. Given two adjacent triangles $\triangle abc$ and $\triangle cda$, such that their union forms a convex quadrilateral $abcd$, the edge flip operation replaces the diagonal ac with bd. Note that it is only possible when the quadrilateral is convex. Suppose that the initial triangle pair violates the empty circle condition, in that point d lies inside the circumcircle of $\triangle abc$. (Note that this implies that b lies inside the circumcircle of $\triangle cda$.) If we flip the edge it will follow that the two circumcircles of the two resulting triangles, $\triangle abd$ and $\triangle bcd$ are now empty (relative to these four points), and the observation above about circles and angles proves that the minimum angle increases at the same time. In particular, in the figure above, we have

$$\phi_{ab} > \theta_{ab} \qquad \phi_{bc} > \theta_{bc} \qquad \phi_{cd} > \theta_{cd} \qquad \phi_{da} > \theta_{da}.$$

There are two other angles that need to be compared as well (can you spot them?). It is not hard to show that, after swapping, these other two angles cannot be smaller than the minimum of θ_{ab}, θ_{bc}, θ_{cd}, and θ_{da}. (Can you see why?)

Since there are only a finite number of triangulations, this process must eventually terminate with the lexicographically maximum triangulation, and this triangulation must satisfy the empty circle condition, and hence is the Delaunay triangulation.

Lecture 18: Delaunay Triangulations: Incremental Construction

Reading: Chapter 9 in the 4M's.

Constructing the Delaunay Triangulation: We will present a simple randomized $O(n \log n)$ expected time algorithm for constructing Delaunay triangulations for n sites in the plane. The algorithm is remarkably similar in spirit to the randomized algorithm for trapezoidal map algorithm in that not only builds the triangulation but also provides a point-location data structure as well. We will not discuss the point-location data structure in detail, but the details are easy to fill in.

As with any randomized incremental algorithm, the idea is to insert sites in random order, one at a time, and update the triangulation with each new addition. The issues involved with the analysis will be showing that after each insertion the expected number of structural changes in the diagram is $O(1)$. As with other incremental algorithm, we need some way of keeping track of where newly inserted sites are to be placed in the diagram. We will describe a somewhat simpler method than the one we used in the trapezoidal map. Rather than building a data structure, this one simply puts each of the uninserted points into a bucket according to the triangle that contains it in the current triangulation. In this case, we will need to argue that the expected number of times that a site is rebucketed is $O(\log n)$.

Incircle Test: The basic issue in the design of the algorithm is how to update the triangulation when a new site is added. In order to do this, we first investigate the basic properties of a Delaunay triangulation. Recall that a

triangle $\triangle abc$ is in the Delaunay triangulation, if and only if the circumcircle of this triangle contains no other site in its interior. (Recall that we make the general position assumption that no four sites are cocircular.) How do we test whether a site d lies within the interior of the circumcircle of $\triangle abc$? It turns out that this can be reduced to a determinant computation. First off, let us assume that the sequence $\langle abcd \rangle$ defines a counterclockwise convex polygon. (If it does not because d lies inside the triangle $\triangle abc$ then clearly d lies in the circumcircle for this triangle. Otherwise, we can always relabel abc so this is true.) Under this assumption, d lies in the circumcircle determined by the $\triangle abc$ if and only if the following determinant is positive. This is called the *incircle test*. We will assume that this primitive is available to us.

$$\text{inCircle}(a, b, c, d) \;=\; \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} > 0.$$

We will not prove the correctness of this test, but a simpler assertion, namely that if the above determinant is equal to zero, then the four points are cocircular. The four points are cocircular if there exists a center point $q = (q_x, q_y)$ and a radius r such that

$$(a_x - q_x)^2 + (a_y - q_y)^2 = r^2,$$

and similarly for the other three points. Expanding this and collecting common terms we have

$$(a_x^2 + a_y^2) - 2q_x(a_x) - 2q_y(a_y) + (q_x^2 + q_y^2 - r^2)(1) = 0,$$

and similarly for the other three points, b, c, and d. If we let X_1, X_2, X_3 and X_4 denote the columns of the above matrix we have

$$X_3 - 2q_x X_1 - 2q_y X_2 + (q_x^2 + q_y^2 - r^2)X_4 = 0.$$

Thus, the columns of the above matrix are linearly dependent, implying that their determinant is zero. We will leave the completion of the proof as an exercise. Next time we will show how to use the incircle test to update the triangulation, and present the complete algorithm.

Incremental update: When we add the next site, p_i, the problem is to convert the current Delaunay triangulation into a new Delaunay triangulation containing this site. This will be done by creating a non-Delaunay triangulation containing the new site, and then incrementally "fixing" this triangulation to restore the Delaunay properties. The fundamental changes will be: (1) adding a site to the middle of a triangle, and creating three new edges, and (2) performing an *edge flip*. Both of these operations can be performed in $O(1)$ time, assuming that the triangulation is maintained, say, as a DCEL.

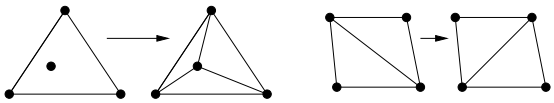


Figure 67: Basic triangulation changes.

The algorithm that we will describe has been known for many years, but was first analyzed by Guibas, Knuth, and Sharir. The algorithm starts within an initial triangulation such that all the points lie in the convex hull. This can be done by enclosing the points in a large triangle. Care must be taken in the construction of this enclosing triangle. It is not sufficient that it simply contain all the points. It should be the case that these points do not lie in the circumcircles of any of the triangles of the final triangulation. Our book suggests computing a triangle that contains all the points, but then fudging with the incircle test so that these points act as if they are invisible.

The sites are added in random order. When a new site p is added, we find the triangle $\triangle abc$ of the current triangulation that contains this site (we will see how later), insert the site in this triangle, and join this site to

the three surrounding vertices. This creates three new triangles, $\triangle pab$, $\triangle pbc$, and $\triangle pca$, each of which may or may not satisfy the empty-circle condition. How do we test this? For each of the triangles that have been added, we check the vertex of the triangle that lies on the opposite side of the edge that does not include p. (If there is no such vertex, because this edge is on the convex hull, then we are done.) If this vertex fails the incircle test, then we swap the edge (creating two new triangles that are adjacent to p). This replaces one triangle that was incident to p with two new triangles. We repeat the same test with these triangles. An example is shown in the figure below.

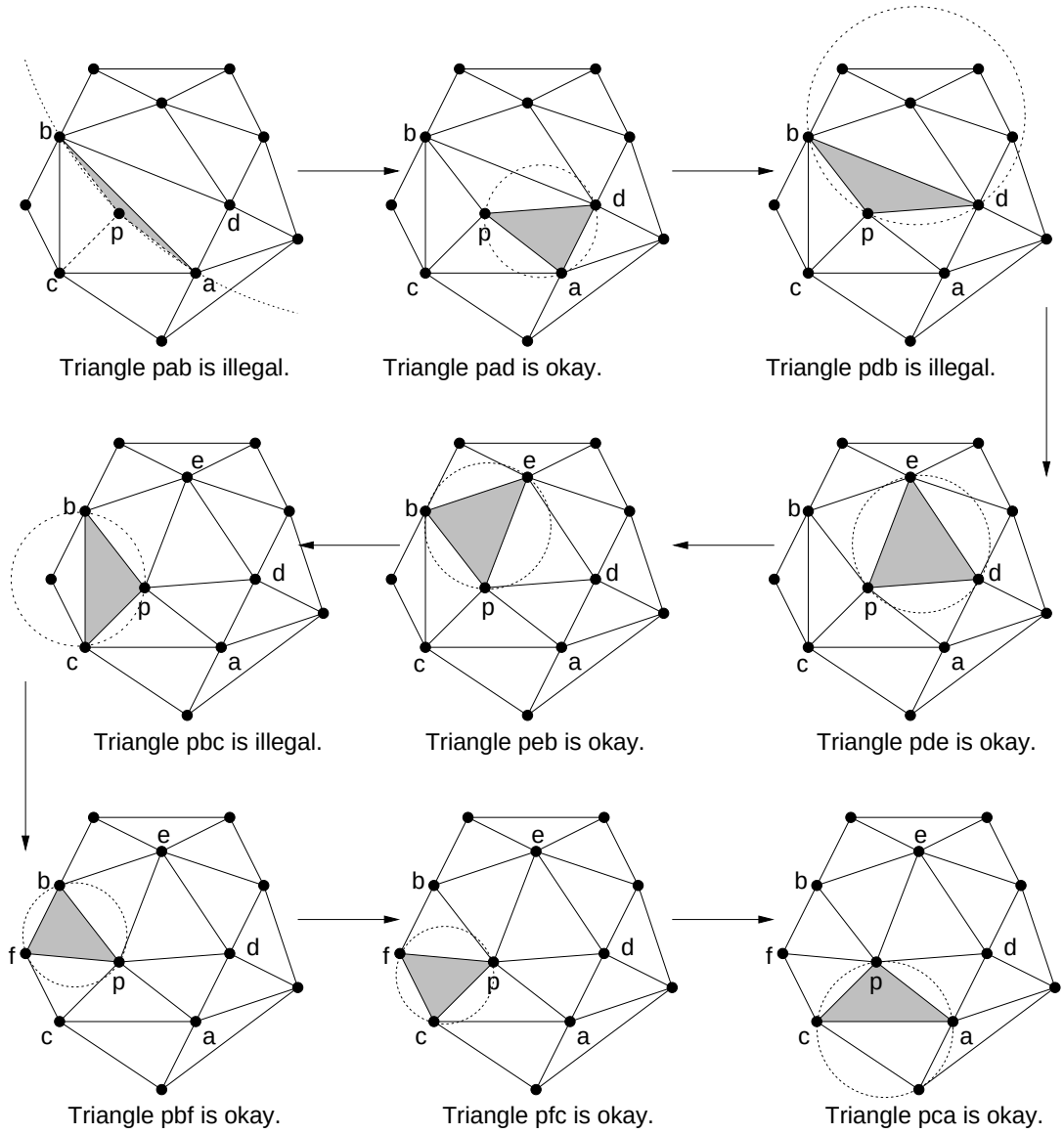


Figure 68: Point insertion.

The code for the incremental algorithm is shown in the figure below. The current triangulation is kept in a global data structure. The edges in the following algorithm are actually pointers to the DCEL.

There is only one major issue in establishing the correctness of the algorithm. When we performed empty-circle tests, we only tested triangles containing the site p, and only sites that lay on the opposite side of an edge of such

```
Insert(p) {
    Find the triangle △abc containing p;
    Insert edges pa, pb, and pc into triangulation;
    SwapTest(ab);              // Fix the surrounding edges
    SwapTest(bc);
    SwapTest(ca);
}

SwapTest(ab) {
    if (ab is an edge on the exterior face) return;
    Let d be the vertex to the right of edge ab;
    if (inCircle(p, a, b, d) {     // d violates the incircle test
        Flip edge ab for pd;
        SwaptTest(ad);             // Fix the new suspect edges
        SwaptTest(db);
    }
}
```

a triangle. We need to establish that these tests are sufficient to guarantee that the final triangulation is indeed Delaunay.

First, we observe that it suffices to consider only triangles that contain p in their circumcircle. The reason is that p is the only newly added site, it is the only site that can cause a violation of the empty-circle property. Clearly the triangle that contained p must be removed, since its circumcircle definitely contains p. Next, we need to argue that it suffices to check only the neighboring triangles after each edge flip. Consider a triangle $\triangle pab$ that contains p and consider the vertex d belonging to the triangle that lies on the opposite side of edge ab. We argue that if d lies outside the circumcircle of pab, then no other point of the point set can lie within this circumcircle.

A complete proof of this takes some effort, but here is a simple justification. What could go wrong? It might be that d lies outside the circumcircle, but there is some other site, say, a vertex e of a triangle adjacent to d, that lies inside the circumcircle. This is illustrated in the following figure. We claim that this cannot happen. It can be shown that if e lies within the circumcircle of $\triangle pab$, then a must lie within the circumcircle of $\triangle bde$. (The argument is a exercise in geometry.) However, this violates the assumption that the initial triangulation (before the insertion of p) was a Delaunay triangulation.

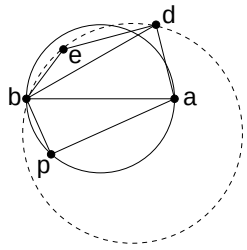


Figure 69: Proof of sufficiency of testing neighboring sites.

As you can see, the algorithm is very simple. The only things that need to be implemented are the DCEL (or other data structure) to store the triangulation, the incircle test, and locating the triangle that contains p. The first two tasks are straightforward. The point location involves a little thought.

Point Location: The point location can be accomplished by one of two means. Our text discusses the idea of building

a history graph point-location data structure, just as we did in the trapezoid map case. A simpler approach is based on the idea of maintaining the uninserted sites in a set of *buckets*. Think of each triangle of the current triangulation as a *bucket* that holds the sites that lie within this triangle and have yet to be inserted. Whenever an edge is flipped, or when a triangle is split into three triangles through point insertion, some old triangles are destroyed and are replaced by a constant number of new triangles. When this happens, we lump together all the sites in the buckets corresponding to the deleted triangles, create new buckets for the newly created triangles, and reassign each site into its new bucket. Since there are a constant number of triangles created, this process requires $O(1)$ time per site that is rebucketed.

Analysis: To analyze the expected running time of algorithm we need to bound two quantities: (1) how many changes are made in the triangulation on average with the addition of each new site, and (2) how much effort is spent in rebucketing sites. As usual, our analysis will be in the worst-case (for any point set) but averaged over all possible insertion orders.

We argue first that the expected number of edge changes with each insertion is $O(1)$ by a simple application of backwards analysis. First observe that (assuming general position) the structure of the Delaunay triangulation is independent of the insertion order of the sites so far. Thus, any of the existing sites is equally likely to have been the last site to be added to the structure. Suppose that some site p was the last to have been added. How much work was needed to insert p? Observe that the initial insertion of p involved the creation of three new edges, all incident to p. Also, whenever an edge swap is performed, a new edge is added to p. These are the only changes that the insertion algorithm can make. Therefore the total number of changes made in the triangulation for the insertion of p is proportional to the degree of p *after* the insertion is complete. Thus the work needed to insert p is proportional to p's degree after the insertion.

Thus, by a backwards analysis, the expected time to insert the last point is equal to the average degree of a vertex in the triangulation. (The only exception are the three initial vertices at infinity, which must be the first sites to be inserted.) However, from Euler's formula, we know that the average degree of a vertex in any planar graph is at most 6. (To see this, recall that a planar graph can have at most $3n$ edges, and the sum of vertex degrees is equal to twice the number of edges, which is at most $6n$.) Thus, (irrespective of which insertion this was) the expected number of edge changes for each insertion is $O(1)$.

Next we argue that the expected number of times that a site is rebucketed (as to which triangle it lies in) is $O(\log n)$. Again this is a standard application of backwards analysis. Consider the ith stage of the algorithm (after i sites have been inserted into the triangulation). Consider any one of the remaining $n - i$ sites. We claim that the probability that this site changes triangles is at most $3/i$ (under the assumption that any of the i points could have been the last to be added).

To see this, let q be an uninserted site and let Δ be the triangle containing q after the ith insertion. As observed above, after we insert the ith site all of the newly created triangles are incident to this site. Since Δ is incident to exactly three sites, if any of these three was added last, then Δ would have come into existence after this insertion, implying that q required (at least one) rebucketing. On the other hand, if none of these three was the last to have been added, then the last insertion could not have caused q to be rebucketed. Thus, (ignoring the three initial sites at infinity) the probability that q required rebucketing after the last insertion is exactly $3/i$. Thus, the total number of points that required rebucketings as part of the last insertion is $(n - i)3/i$. To get the total expected number of rebucketings, we sum over all stages, giving

$$\sum_{i=1}^{n} \frac{3}{i}(n-i) \;\leq\; \sum_{i=1}^{n} \frac{3}{i}n \;=\; 3n \sum_{i=1}^{n} \frac{1}{i} \;=\; 3n \ln n + O(1).$$

Thus, the total expected time spent in rebucketing is $O(n \log n)$, as desired.

There is one place in the proof that we were sloppy. (Can you spot it?) We showed that the number of points that required rebucketing is $O(n \log n)$, but notice that when a point is inserted, many rebucketing operations may be needed (one for the initial insertion and one for each additional edge flip). We will not give a careful analysis of the total number of individual rebucketing operations per point, but it is not hard to show that the expected total

number of individual rebucketing operations will not be larger by more than a constant factor. The reason is that (as argued above) each new insertion only results in a constant number of edge flips, and hence the number of individual rebucketings per insertion is also a constant. But a careful proof should consider this. Such a proof is given in our text book.

Lecture 19: Line Arrangements

Reading: Chapter 8 in the 4M's.

Arrangements: So far we have studied a few of the most important structures in computational geometry: convex hulls, Voronoi diagrams and Delaunay triangulations. Perhaps, the next most important structure is that of a *line arrangement*. As with hulls and Voronoi diagrams, it is possible to define arrangements (of $d-1$ dimensional hyperplanes) in any dimension, but we will concentrate on the plane. As with Voronoi diagrams, a line arrangement is a polygonal subdivision of the plane. Unlike most of the structures we have seen up to now, a line arrangement is not defined in terms of a set of points, but rather in terms of a set L of lines. However, line arrangements are used mostly for solving problems on point sets. The connection is that the arrangements are typically constructed in the dual plane. We will begin by defining arrangements, discussing their combinatorial properties and how to construct them, and finally discuss applications of arrangements to other problems in computational geometry.

Before discussing algorithms for computing arrangements and applications, we first provide definitions and some important theorems that will be used in the construction. A finite set L of lines in the plane subdivides the plane. The resulting subdivision is called an *arrangement*, denoted $\mathcal{A}(L)$. Arrangements can be defined for curves as well as lines, and can also be defined for $(d-1)$-dimensional hyperplanes in dimension d. But we will only consider the case of lines in the plane here. In the plane, the arrangement defines a planar graph whose vertices are the points where two or more lines intersect, edges are the intersection free segments (or rays) of the lines, and faces are (possibly unbounded) convex regions containing no line. An example is shown below.

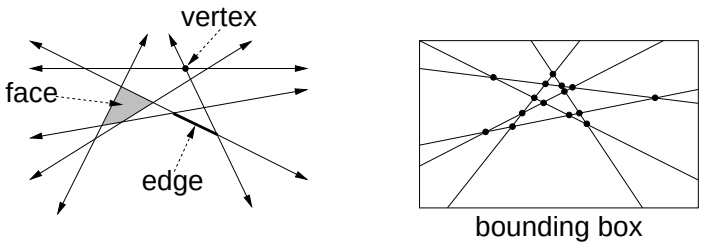


Figure 70: Arrangement of lines.

An arrangement is said to be *simple* if no three lines intersect at a common point. We will make the usual general position assumptions that no three lines intersect in a single point. This assumption is easy to overcome by some sort of symbolic perturbation.

An arrangement is not formally a planar graph, because it has unbounded edges. We can fix this (topologically) by imagining that a vertex is added at infinity, and all the unbounded edges are attached to this vertex. A somewhat more geometric way to fix this is to imagine that there is a bounding box which is large enough to contain all the vertices, and we tie all the unbounded edges off at this box. Rather than computing the coordinates of this huge box (which is possible in $O(n^2)$ time), it is possible to treat the sides of the box as existing at infinity, and handle all comparisons symbolically. For example, the lines that intersect the right side of the "box at infinity" have slopes between $+1$ and -1, and the order in which they intersect this side (from top to bottom) is in decreasing order of slope. (If you don't see this right away, think about it.)

The *combinatorial complexity* of an arrangement is the total number of vertices, edges, and faces in the arrangement. The following shows that all of these quantities are $O(n^2)$.

Theorem: Give a set of n lines L in the plane in general position,

(i) the number of vertices in $\mathcal{A}(L)$ is $\binom{n}{2}$,

(ii) the number of edges in $\mathcal{A}(L)$ is n^2, and

(iii) the number of faces in $\mathcal{A}(L)$ is $\binom{n}{2} + n + 1$.

Proof: The fact that the number of vertices is $\binom{n}{2}$ is clear from the fact that each pair of lines intersects in a single point. To prove that the number of edges is n^2, we use induction. The basis case is trivial (1 line and 1 edge). When we add a new line to an arrangement of $n-1$ lines (having $(n-1)^2$ edges by the induction hypothesis) we split $n-1$ existing edges, thus creating $n-1$ new edges, and we add n new edges from the $n-1$ intersections with the new line. This gives a total of $(n-1)^2 + (n-1) + n = n^2$. The number of faces comes from Euler's formula, $v - e + f = 2$ (with the little trick that we need to create one extra vertex to attach all the unbounded edges to).

Incremental Construction: Arrangements are used for solving many problems in computational geometry. But in order to use arrangements, we first must be able to construct arrangements. We will present a simple incremental algorithm, which builds an arrangement by adding lines one at a time. Unlike most of the other incremental algorithms we have seen so far, this one is *not randomized*. Its asymptotic running time will be the same, $O(n^2)$, no matter what order we insert the lines. This is asymptotically optimal, since this is the size of the arrangement. The algorithm will also require $O(n^2)$, since this is the amount of storage needed to store the final result. (Later we will consider the question of whether it is possible to traverse an arrangement without actually building it.)

As usual, we will assume that the arrangement is represented in any reasonable data structure for planar graphs, a DCEL for example. Let $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ denote the lines. We will simply add lines one by one to the arrangement. (The order does not matter.) We will show that the i-th line can be inserted in $O(i)$ time. If we sum over i, this will give $O(n^2)$ total time.

Suppose that the first $i-1$ lines have already been added, and consider the effort involved in adding ℓ_i. Recall our assumption that the arrangement is assumed to lie within a large bounding box. Since each line intersects this box twice, the first $i-1$ lines have subdivided the boundary of the box into $2(i-1)$ edges. We determine where ℓ_i intersects the box, and which of these edge it crosses intersects. This will tell us which face of the arrangement ℓ_i first enters.

Next we trace the line through the arrangement, from one face to the next. Whenever we enter a face, the main question is where does the line exit the face? We answer the question by a very simple strategy. We walk along the edges face, say in a counterclockwise direction (recall that a DCEL allows us to do this) and as we visit each edge we ask whether ℓ_i intersects this edge. When we find the edge through which ℓ_i exits the face, we jump to the face on the other side of this edge (again the DCEL supports this) and continue the trace. This is illustrated in the figure below.

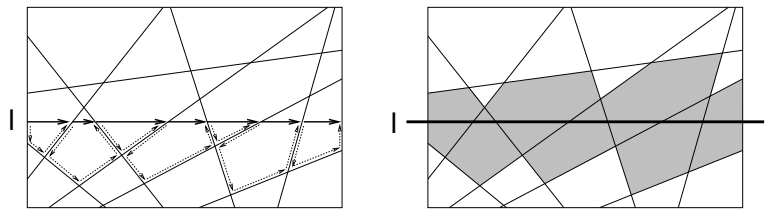


Figure 71: Traversing an arrangement and zones.

Once we know the points of entry and exit into each face, the last matter is to update the DCEL by inserting edges for each entry-exit pair. This is easy to handle in constant time, by adjusting the appropriate pointers in the DCEL.

Clearly the time that it takes to perform the insertion is proportional to the total number of edges that have been traversed in this tracing process. A naive argument says that we encounter $i-1$ lines, and hence pass through i faces (assuming general position). Since each face is bounded by at most i lines, each facial traversal will take $O(i)$ time, and this gives a total $O(i^2)$. Hey, what went wrong? Above we said that we would do this in $O(i)$ time. The claim is that the traversal does indeed traverse only $O(i)$ edges, but to understand why, we need to delve more deeply into a concept of a *zone* of an arrangement.

Zone Theorem: The most important combinatorial property of arrangements (which is critical to their efficient construction) is a rather surprising result called the *zone theorem*. Given an arrangement $\mathcal{A}$ of a set L of n lines, and given a line ℓ that is not in L, the *zone* of ℓ in $\mathcal{A}(\ell)$, denoted $Z_{\mathcal{A}}(\ell)$, is the set of faces whose closure intersects ℓ. The figure above illustrates a zone for the line ℓ. For the purposes of the above construction, we are only interested in the edges of the zone that lie below ℓ_i, but if we bound the total complexity of the zone, then this will be an upper bound on the number of edges traversed in the above algorithm. The combinatorial complexity of a zone (as argued above) is at most $O(n^2)$. The Zone theorem states that the complexity is actually much smaller, only $O(n)$.

Theorem: (The Zone Theorem) Given an arrangement $\mathcal{A}(L)$ of n lines in the plane, and given any line ℓ in the plane, the total number of edges in all the cells of the zone $Z_A(\ell)$ is at most $6n$.

Proof: As with most combinatorial proofs, the key is to organize everything so that the counting can be done in an easy way. Note that this is not trivial, because it is easy to see that any one line of L might contribute many segments to the zone of ℓ. The key in the proof is finding a way to add up the edges so that each line appears to induce only a constant number of edges into the zone.

The proof is based on a simple inductive argument. We will first imagine, through a suitable rotation, that ℓ is horizontal, and further that none of the lines of L is horizontal (through an infinitesimal rotation). We split the edges of the zone into two groups, those that bound some face from the left side and those that bound some face from the right side. More formally, since each face is convex, if we split it at its topmost and bottommost vertices, we get two convex chains of edges. The *left-bounding edges* are on the left chain and the *right-bounding edges* are on the right chain. We will show that there are at most $3n$ lines that bounded faces from the left. (Note that an edge of the zone that crosses ℓ itself contributes only once to the complexity of the zone. In the book's proof they seem to count this edge twice, and hence their bound they get a bound of $4n$ instead. We will also ignore the edges of the bounding box.)

For the base case, when $n = 1$, then there is exactly one left bounding edge in ℓ's zone, and $1 \leq 3n$. Assume that the hypothesis is true for any set of $n-1$ lines. Consider the rightmost line of the arrangement to intersect ℓ. Call this ℓ_1. (Selecting this particular line is very important for the proof.) Suppose that we consider the arrangement of the other $n-1$ lines. By the induction hypothesis there will be at most $3(n-1)$ left-bounding edges in the zone for ℓ.

Now let us add back ℓ_1 and see how many more left-bounding edges result. Consider the rightmost face of the arrangement of $n-1$ lines. Note that all of its edges are left-bounding edges. Line ℓ_1 will intersect ℓ within this face. Let e_a and e_b denote the two edges of this that ℓ_1 intersects, one above ℓ and the other below ℓ. The insertion of ℓ_1 creates a new left bounding edge along ℓ_1 itself, and splits the left bounding edges e_a and e_b into two new left bounding edges for a net increase of three edges. Observe that ℓ_1 cannot contribute any other left-bounding edges to the zone, because (depending on slope) either the line supporting e_a or the line supporting e_b blocks ℓ_1's visibility from ℓ. (Note that it might provide right-bounding edges, but we are not counting them here.) Thus, the total number of left-bounding edges on the zone is at most $3(n-1)+3 \leq 3n$, and hence the total number of edges is at most $6n$, as desired.

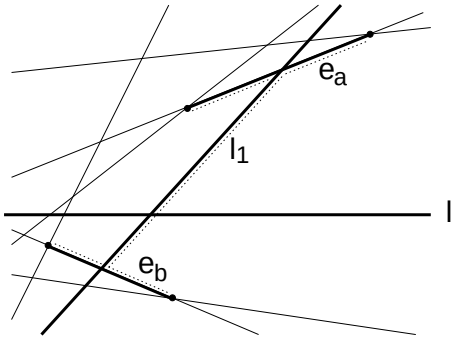


Figure 72: Proof of the Zone Theorem.

Lecture 20: Applications of Arrangements

Reading: Chapter 8 in the 4M's. Some of this material is not covered in the book.

Applications of Arrangements and Duality: The computational and mathematical tools that we have developed with geometric duality and arrangements allow a large number of problems to be solved. Here are some examples. Unless otherwise stated, all these problems can be solved in $O(n^2)$ time and $O(n^2)$ space by constructing a line arrangement, or in $O(n^2)$ time and $O(n)$ space through topological plane sweep.

General position test: Given a set of n points in the plane, determine whether any three are collinear.

Minimum area triangle: Given a set of n points in the plane, determine the minimum area triangle whose vertices are selected from these points.

Minimum k-corridor: Given a set of n points, and an integer k, determine the narrowest pair of parallel lines that enclose at least k points of the set. The distance between the lines can be defined either as the vertical distance between the lines or the perpendicular distance between the lines.

Visibility graph: Given line segments in the plane, we say that two points are *visible* if the interior of the line segment joining them intersects none of the segments. Given a set of n non-intersecting line segments, compute the *visibility graph*, whose vertices are the endpoints of the segments, and whose edges a pairs of visible endpoints.

Maximum stabbing line: Given a set of n line segments in the plane, compute the line that stabs (intersects) the maximum number of these line segments.

Hidden surface removal: Given a set of n non-intersecting polygons in 3-space, imagine projecting these polygon onto a plane (either orthogonally or using perspective). Determine which portions of the polygons are visible from the viewpoint under this projection.

Note that in the worst case, the complexity of the final visible scene may be as high as $O(n^2)$, so this is asymptotically optimal. However, since such complex scenes rarely occur in practice, this algorithm is really only of theoretical interest.

Ham Sandwich Cut: Given n red points and m blue points, find a single line that simultaneously bisects these point sets. It is a famous fact from mathematics (called the *Ham-Sandwich Theorem*) that such a line always exists. If the point sets are separated by a line, then this can be done in time: $O(n + m)$, space: $O(n + m)$.

Sorting all angular sequences: Here is a natural application of duality and arrangements that turns out to be important for the problem of computing visibility graphs. Consider a set of n points in the plane. For each point p in this set we want to perform an angular sweep, say in counterclockwise order, visiting the other $n - 1$ points of

the set. For each point, it is possible to compute the angles between this point and the remaining $n-1$ points and then sort these angles. This would take $O(n \log n)$ time per point, and $O(n^2 \log n)$ time overall.

With arrangements we can speed this up to $O(n^2)$ total time, getting rid of the extra $O(\log n)$ factor. Here is how. Recall the point-line dual transformation. A point $p = (p_x, p_y)$ and line $\ell : (y = ax - b)$ in the primal plane are mapped through to a dual point ℓ^* and dual line p^* as follows:

$$\begin{aligned} \ell^* &= (a, b) \\ p^* &: (b = p_x a - p_y). \end{aligned}$$

Recall that the a-coordinate in the dual plane corresponds to the slope of a line in the primal plane. Suppose that p is the point that we want to sort around, and let $p_1, p_2, \ldots, p_n$ be the points in final angular order about p.

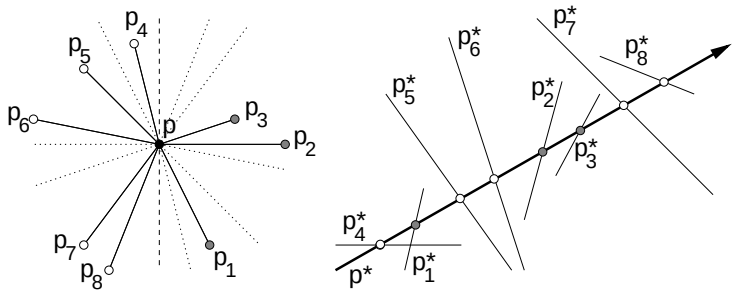


Figure 73: Arrangements and angular sequences.

Consider the arrangement defined by the dual lines p_i^*. How is this order revealed in the arrangement? Consider the dual line p^*, and its intersection points with each of the dual lines p_i^*. These form a sequence of vertices in the arrangement along p^*. Consider this sequence ordered from left to right. It would be nice if this order were the desired circular order, but this is not quite correct. The a-coordinate of each of these vertices in the dual arrangement is the slope of some line of the form $\overline{pp_i}$. Thus, the sequence in which the vertices appear on the line is a *slope ordering* of the points about p_i, not an *angular ordering*.

However, given this slope ordering, we can simply test which primal points lie to the left of p (that is, have a smaller x-coordinate in the primal plane), and separate them from the points that lie to the right of p (having a larger x-coordinate). We partition the vertices into two sorted sequences, and then concatenate these two sequences, with the points on the right side first, and the points on the left side later. The resulting is an angular sequence starting with the angle -90 degrees and proceeding up to $+270$ degrees.

Thus, once the arrangement has been constructed, we can reconstruct each of the angular orderings in $O(n)$ time, for a total of $O(n^2)$ time.

Maximum Discrepancy: Next we consider a problem derived from computer graphics and sampling. Suppose that we are given a collection of n points S lying in a unit square $U = [0, 1]^2$. We want to use these points for random sampling purposes. In particular, the property that we would like these points to have is that for any halfplane h, we would like the size of the fraction of points of P that lie within h should be roughly equal to the area of intersection of h with U. That is, if we define $\mu(h)$ to be the area of $h \cap U$, and $\mu_S(h) = |S \cap h|/|S|$, then we would like $\mu(h) \approx \mu_S(h)$ for all h. This property is important when point sets are used for things like sampling and Monte-Carlo integration.

To this end, we define the *discrepancy* of S with respect to a halfplane h to be

$$\Delta_S(h) = |\mu(h) - \mu_S(h)|.$$

For example, in the figure below (a), the area of $h \cap U$ is $\mu(h) = 0.625$, and there are 7 out of 13 points in h, thus $\mu_S(h) = 7/13 = 0.538$. Thus the discrepancy of h is $|0.625 - 0.538| = 0.087$. Define the *halfplane*

discrepancy of S to be the maximum (or more properly the supremum, or least upper bound) of this quantity over all halfplanes:

$$\Delta(S) = \sup_h \Delta_S(h).$$

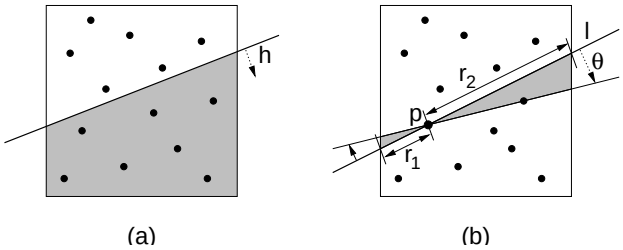


Figure 74: Discrepancy of a point set.

Since there are an uncountably infinite number of halfplanes, it is important to derive some sort of *finiteness criterion* on the set of halfplanes that might produce the greatest discrepancy.

Lemma: Let h denote the halfplane that generates the maximum discrepancy with respect to S, and let ℓ denote the line that bounds h. Then either (i) ℓ passes through at least two points of S, or (ii) ℓ passes through one point of S, and this point is the midpoint of the line segment $\ell \cap U$.

Remark: If a line passes through one or more points of S, then should this point be included in $\mu_S(h)$? For the purposes of computing the maximum discrepancy, the answer is to either include or omit the point, whichever will generate the larger discrepancy. The justification is that it is possible to perturb h infinitesimally so that it includes none or all of these points without altering $\mu(h)$.

Proof: If ℓ does not pass through any point of S, then (depending on which is larger $\mu(h)$ or $\mu_S(h)$) we can move the line up or down without changing $\mu_S(h)$ and increasing or decreasing $\mu(h)$ to increase their difference. If ℓ passes through a point $p \in S$, but is not the midpoint of the line segment $\ell \cap U$, then we claim that we can rotate this line about p and hence increase or decrease $\mu(h)$ without altering $\mu_S(h)$, to increase their difference.

To establish the claim, consider the figure above (b). Suppose that the line ℓ passes through point p and let $r_1 < r_2$ denote the two lengths along ℓ from p to the sides of the square. Observe that if we rotate ℓ through a small angle θ, then to a first order approximation, the loss due to area of the triangle on the left is $r_1^2\theta/2$, since this triangle can be approximated by an angular sector of a circle of radius r_1 and angle θ. The gain due to the area of the triangle on the right is $r_2^2\theta/2$. Thus, since $r_1 < r_2$ this rotation will increase the area of region lying below h infinitessimally. A rotation in the opposite decreases the area infinitessimally. Since the number of points bounded by h does not change as a function of θ, the discrepancy cannot be achieved as long as such a rotation is possible.

(Note that this proof reveals something slightly stronger. If ℓ contacts two points, the line segment between these points must contain the midpoint of the $\ell \cap U$. Do you see why?)

Since for each point $p \in S$ there are only a constant number of lines ℓ (at most two, I think) through this point such that p is the midpoint of $\ell \cap U$, it follows that there are at most $O(n)$ lines of type (i) above, and hence the discrepancy of all of these lines can be tested in $O(n^2)$ time.

To compute the discrepancies of the other lines, we can dualize the problem. In the primal plane, a line ℓ that passes through two points $p_i, p_j \in S$, is mapped in the dual plane to a point ℓ^* at which the lines p_i^* and p_j^* intersect. This is just a vertex in the arrangement of the dual lines for S. So, if we have computed the arrangement, then all we need to do is to visit each vertex and compute the discrepancy for the corresponding primal line.

It is easy to see that the area $\ell \cap U$ of each corresponding line in the primal plane can be computed in $O(1)$ time. So, all that is needed is to compute the number of points of S lying below each such line. In the dual plane, the corresponds to determining the number of dual lines that lie below or above each vertex in the arrangement. If we know the number of dual lines that lie strictly above each vertex in the arrangement, then it is trivial to compute the number of lines that lie below by subtraction.

We define a point to be at *level* k, denoted $\mathcal{L}_k$, in an arrangement if there are at most $k-1$ lines above this point and at most $n-k$ lines below this point. The k-th level of an arrangement is an x-monotone polygonal curve, as shown below. For example, the upper envelope of the lines is level 1 of the arrangement, and the lower envelope is level n of the arrangement. Note that a vertex of the arrangement can be on multiple levels. (Also note that our definition of level is exactly one greater than our text's definition.)

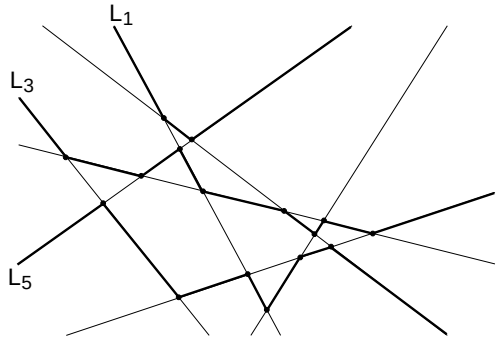


Figure 75: Levels in an arrangement.

We claim that it is an easy matter to compute the level of each vertex of the arrangement (e.g. by plane sweep). The initial levels at $x = -\infty$ are determined by the slope order of the lines. As the plane sweep proceeds, the index of a line in the sweep line status is its level. Thus, by using topological plane sweep, in $O(n^2)$ time we can compute the minimum and maximum level number of each vertex in the arrangement. From the order reversing property, for each vertex of the dual arrangement, the minimum level number minus 1 indicates the number of primal points that lie strictly below the corresponding primal line and the maximum level number is the number of dual points that lie on or below this line. Thus, given the level numbers and the fact that areas can be computed in $O(1)$ time, we can compute the discrepancy in $O(n^2)$ time and $O(n)$ space, through topological plane sweep.

Lecture 21: Shortest Paths and Visibility Graphs

Reading: The material on visibility graphs is taken roughly from Chapter 15, but we will present slightly more efficient variant of the one that appears in this chapter.

Shortest paths: We are given a set of n disjoint polygonal *obstacles* in the plane, and two points s and t that lie outside of the obstacles. The problem is to determine the shortest path from s to t that avoids the interiors of the obstacles. (It may travel along the edges or pass through the vertices of the obstacles.) The complement of the interior of the obstacles is called *free space*. We want to find the shortest path that is constrained to lie entirely in free space.

Today we consider a simple (but perhaps not the most efficient) way to solve this problem. We assume that we measure lengths in terms of Euclidean distances. How do we measure paths lengths for curved paths? Luckily, we do not have to, because we claim that the shortest path will always be a polygonal curve.

Claim: The shortest path between any two points that avoids a set of polygonal obstacles is a polygonal curve, whose vertices are either vertices of the obstacles or the points s and t.

Proof: We show that any path π that violates these conditions can be replaced by a slightly shorter path from s to t. Since the obstacles are polygonal, if the path were not a polygonal curve, then there must be some point p in the interior of free space, such that the path passing through p is not locally a line segment. If we consider any small neighborhood about p (small enough to not contain s or t or any part of any obstacle), then since the shortest path is not locally straight, we can shorten it slightly by replacing this curved segment by a straight line segment jointing one end to the other. Thus, π is not shortest, a contradiction.

Thus π is a polygonal path. Suppose that it contained a vertex v that was not an obstacle vertex. Again we consider a small neighbor hood about v that contains no part of any obstacle. We can shorten the path, as above, implying that π is not a shortest path.

From this it follows that the edges that constitute the shortest path must travel between s and t and vertices of the obstacles. Each of these edges must have the property that it does not intersect the interior of any obstacle, implying that the endpoints must be visible to each other. More formally, we say that two points p and q are *mutually visible* if the open line segment joining them does not intersect the interior of any obstacle. By this definition, the two endpoints of an obstacle edge are not mutually visible, so we will explicitly allow for this case in the definition below.

Definition: The *visibility graph* of s and t and the obstacle set is a graph whose vertices are s and t the obstacle vertices, and vertices v and w are joined by an edge if v and w are either mutually visible or if (v, w) is an edge of some obstacle.

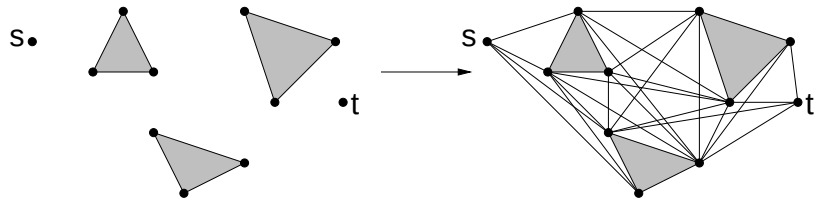


Figure 76: Visibility graph.

It follows from the above claim that the shortest path can be computed by first computing the visibility graph and labeling each edge with its Euclidean length, and then computing the shortest path by, say, Dijkstra's algorithm (see CLR). Note that the visibility graph is not planar, and hence may consist of $\Omega(n^2)$ edges. Also note that, even if the input points have integer coordinates, in order to compute distances we need to compute square roots, and then sums of square roots. This can be approximated by floating point computations. (If exactness is important, this can really be a problem, because there is no known polynomial time procedure for performing arithmetic with arbitrary square roots of integers.)

Computing the Visibility Graph: We give an $O(n^2)$ procedure for constructing the visibility graph of n line segments in the plane. The more general task of computing the visibility graph of an arbitrary set of polygonal obstacles is a very easy generalization. In this context, two vertices are visible if the line segment joining them does not intersect any of the obstacle line segments. However, we allow each line segment to contribute itself as an edge in the visibility graph. We will make the general position assumption that no three vertices are collinear, but this is not hard to handle with some care. The algorithm is *not* output sensitive. If k denotes the number of edges in the visibility graph, then an $O(n \log n + k)$ algorithm does exist, but it is quite complicated.

The text gives an $O(n^2 \log n)$ time algorithm. We will give an $O(n^2)$ time algorithm. Both algorithms are based on the same concept, namely that of performing an angular sweep around each vertex. The text's algorithm operates by doing this sweep one vertex at a time. Our algorithm does the sweep for all vertices simultaneously. We use the fact (given in the lecture on arrangements) that this angular sort can be performed for all vertices in $O(n^2)$ time. If we build the entire arrangement, this sorting algorithm will involve $O(n^2)$ space. However it can be implemented in $O(n)$ space using an algorithm called *topological plane sweep*. Topological plane sweep

provides a way to sweep an arrangement of lines using a "flexible" sweeping line. Because events do not need to sorted, we can avoid the $O(\log n)$ factor, which would otherwise be needed to maintain the priority queue.

Here is a high-level intuitive view of the algorithm. First, recall the algorithm for computing trapezoidal maps. We shoot a bullet up and down from every vertex until it hits its first line segment. This implicitly gives us the vertical visibility relationships between vertices and segments. Now, we imagine that angle θ continuously sweeps out all slopes from $-\infty$ to $+\infty$. Imagine that all the bullet lines attached to all the vertices begin to turn slowly counterclockwise. If we play the mind experiment of visualizing the rotation of these bullet paths, the question is what are the significant event points, and what happens with each event? As the sweep proceeds, we will eventually determine everything that is visible from every vertex in every direction. Thus, it should be an easy matter to piece together the edges of the visibility graph as we go.

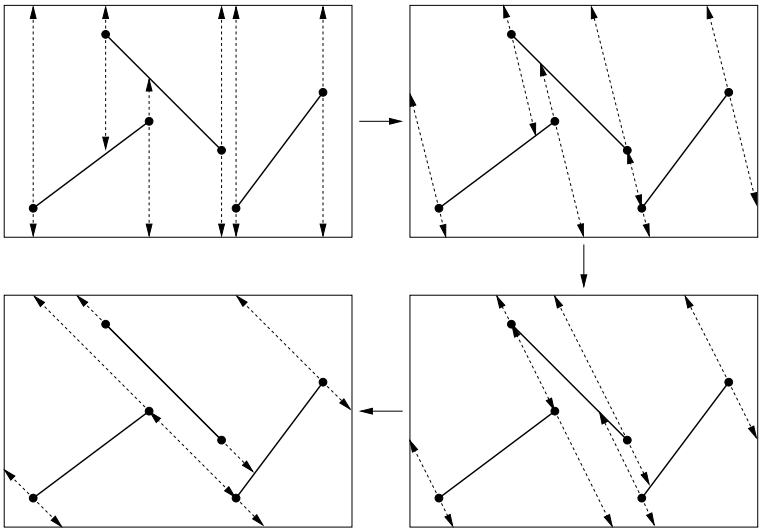


Figure 77: Visibility graph by multiple angular sweep.

Let us consider this "multiple angular sweep" in greater detail.

It is useful to view the problem both in its primal and dual form. For each of the $2n$ segment endpoints $v = (v_a, v_b)$, we consider its dual line $v^* : y = v_a x - v_b$. Observe that a significant event occurs whenever a bullet path in the primal plane jumps from one line segment to another. This occurs when θ reaches the slope of the line joining two visible endpoints v and w. Unfortunately, it is somewhat complicated to keep track of which endpoints are visible and which are not (although if we could do so it would lead to a more efficient algorithm). Instead we will take events to be *all* angles θ between two endpoints, whether they are visible or not. By duality, the slope of such an event will correspond to the a-coordinate of the intersection of dual lines v^* and w^* in the dual arrangement. (Convince yourself of this.) Thus, by sweeping the arrangement of the $2n$ dual lines from left-to-right, we will enumerate all the slope events in angular order.

Next let's consider what happens at each event point. Consider the state of the angular sweep algorithm for some slope θ. For each vertex v, there are two bullet paths emanating from v along the line with slope θ. Call one the *forward bullet path* and the other the *backward bullet path*. Let $f(v)$ and $b(v)$ denote the line segments that these bullet paths hit, respectively. If either path does not hit any segment then we store a special null value. As θ varies the following events can occur. Assuming (through symbolic perturbation) that each slope is determined by exactly two lines, whenever we arrive at an events slope θ there are exactly two vertices v and w that are involved. Here are the possible scenarios:

Same segment: If v and w are endpoints of the same segment, then they are visible, and we add the edge (v, w) to the visibility graph.

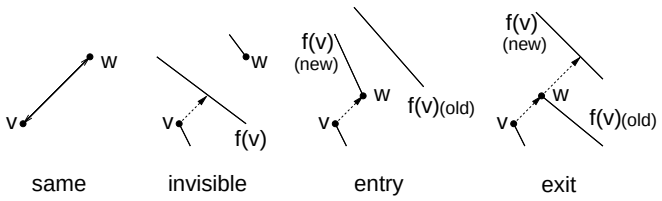


Figure 78: Possible events.

Invisible: Consider the distance from v to w. First, determine whether w lies on the same side as $f(v)$ or $b(v)$. For the remainder, assume that it is $f(v)$. (The case of $b(v)$ is symmetrical).

Compute the contact point of the bullet path shot from v in direction θ with segment $f(v)$. If this path hits $f(v)$ strictly before w, then we know that w is not visible to v, and so this is a "non-event".

Segment entry: Consider the segment that is incident to w. Either the sweep is just about to enter this segment or is just leaving it. If we are entering the segment, then we set $f(v)$ to this segment.

Segment exit: If we are just leaving this segment, then the bullet path will need to shoot out and find the next segment that it hits. Normally this would require some searching. (In particular, this is one of the reasons that the text's algorithm has the extra $O(\log n)$ factor—to perform this search.) However, we claim that the answer is available to us in $O(1)$ time.

In particular, since we are sweeping over w at the same time that we are sweeping over v. Thus we know that the bullet extension from w hits $f(w)$. All we need to do is to set $f(v) = f(w)$.

This is a pretty simple algorithm (although there are a number of cases). The only information that we need to keep track of is (1) a priority queue for the events, and (2) the $f(v)$ and $b(v)$ pointers for each vertex v. The priority queue is not stored explicitly. Instead it is available from the line arrangement of the duals of the line segment vertices. By performing a topological sweep of the arrangement, we can process all of these events in $O(n^2)$ time.

Lecture 22: Motion Planning

Reading: Chapt 13 in the 4M's.

Motion planning: Last time we considered the problem of computing the shortest path of a point in space around a set of obstacles. Today we will study a much more general approach to the more general problem of how to plan the motion of one or more robots, each with potentially many degrees of freedom in terms of its movement and perhaps having articulated joints.

Work Space and Configuration Space: The environment in which the robot operates is called its *work space*, which consists of a set of obstacles that the robot is not allowed to intersect. We assume that the work space is static, that is, the obstacles do not move. We also assume that a complete geometric description of the work space is available to us.

For our purposes, a *robot* will be modeled by two main elements. The first is a *configuration*, which is a finite sequence of values that fully specifies the position of the robot. The second element is the robot's geometric shape description. Combined these two element fully define the robot's exact position and shape in space. For example, suppose that the robot is a 2-dimensional polygon that can translate and rotate in the plane. Its configuration may be described by the (x, y) coordinates of some reference point for the robot, and an angle θ that describes its orientation. Its geometric information would include its shape (say at some canonical position), given, say, as a simple polygon. Given its geometric description and a configuration (x, y, θ), this uniquely

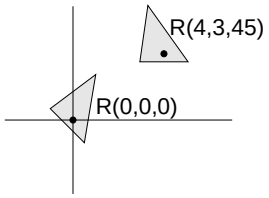


Figure 79: Configurations of a translating and rotating robot.

determines the exact position $\mathcal{R}(x, y, \theta)$ of this robot in the plane. Thus, the position of the robot can be identified with a point in the robot's *configuration space*.

A more complex example would be an *articulated arm* consisting of a set of links, connected to one another by a set of *revolute joints*. The configuration of such a robot would consist of a vector of joint angles. The geometric description would probably consist of a geometric representation of the links. Given a sequence of joint angles, the exact shape of the robot could be derived by combining this configuration information with its geometric description. For example, a typical 3-dimensional industrial robot has six joints, and hence its configuration can be thought of as a point in a 6-dimensional space. Why six? Generally, there are three degrees of freedom needed to specify a location in 3-space, and 3 more degrees of freedom needed to specify the direction and orientation of the robot's end manipulator.

Given a point p in the robot's configuration space, let $\mathcal{R}(p)$ denote the *placement* of the robot at this configuration. The figure below illustrates this in the case of the planar robot defined above.

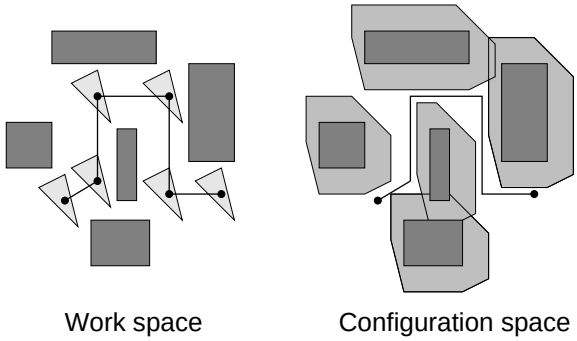


Figure 80: Work space and configuration space.

Because of limitations on the robot's physical structure and the obstacles, not every point in configuration space corresponds to a legal placement of the robot. Any configuration which is illegal in that it causes the robot to intersect one of the obstacles is called a *forbidden configuration*. The set of all forbidden configurations is denoted $C_{\text{forb}}(\mathcal{R}, S)$, and all other placements are called *free configurations*, and the set of these configurations is denoted $C_{\text{free}}(\mathcal{R}, S)$, or *free space*.

Now consider the *motion planning* problem in robotics. Given a robot $\mathcal{R}$, an work space S, and initial configuration s and final configuration t (both points in the robot's free configuration space), determine (if possible) a way to move the robot from one configuration to the other without intersecting any of the obstacles. This reduces to the problem of determining whether there is a path from s to t that lies entirely within the robot's free configuration space. Thus, we map the task of computing a robot's motion to the problem of finding a path for a single point through a collection of obstacles.

Configuration spaces are typically higher dimensional spaces, and can be bounded by curved surfaces (especially when rotational elements are involved). Perhaps the simplest case to visualize is that of translating a convex polygonal robot in the plane amidst a collection of polygonal obstacles. In this cased both the work space and

configuration space are two dimensional. Consider a reference point placed in the center of the robot. As shown in the figure above, the process of mapping to configuration space involves replacing the robot with a single point (its reference point) and "growing" the obstacles by a compensating amount. These grown obstacles are called *configuration obstacles* or C-obstacles.

This approach while very general, ignores many important practical issues. It assumes that we have complete knowledge of the robot's environment and have perfect knowledge and control of its placement. As stated we place no requirements on the nature of the path, but in reality physical objects can not be brought to move and stop instantaneously. Nonetheless, this abstract view is very powerful, since it allows us to abstract the motion planning problem into a very general framework.

For the rest of the lecture we will consider a very simple case of a convex polygonal robot that is translating among a convex set of obstacles. Even this very simple problem has a number of interesting algorithmic issues.

Configuration Obstacles and Minkowski Sums: Let us consider how to build a configuration space for a set of polygonal obstacles. We consider the simplest case of translating a convex polygonal robot amidst a collection of convex obstacles. If the obstacles are not convex, then we may subdivide them into convex pieces.

Consider a robot $\mathcal{R}$, whose placement is defined by a translation $\vec{p} = (x, y)$. Let $\mathcal{R}(x, y)$ (also denoted $\mathcal{R}(\vec{p})$) be the placement of the robot with its reference point at $\vec{p}$. Given an obstacle P, the *configuration obstacle* is defined as all the placements of $\mathcal{R}$ that intersect P, that is

$$\mathcal{CP} = \{\vec{p} \,|\, \mathcal{R}(\vec{p}) \cap P \neq \emptyset\}.$$

One way to visualize $\mathcal{CP}$ is to imagine "scraping" $\mathcal{R}$ along the boundary of P and seeing the region traced out by $\mathcal{R}$'s reference point.

The problem we consider next is, given $\mathcal{R}$ and P, compute the configuration obstacle $\mathcal{CP}$. To do this, we first introduce the notion of a *Minkowski sum*. Let us violate our notions of affine geometry for a while, and think of points (x, y) in the plane as vectors. Given any two sets S_1 and S_2 in the plane, define their *Minkowski sum* to be the set of all pairwise sums of points taken from each set:

$$S_1 \oplus S_2 \;=\; \{\vec{p} + \vec{q} \,|\, \vec{p} \in S_1,\;\; \vec{q} \in S_2\}.$$

Also, define $-S = \{-\vec{p} \,|\, \vec{p} \in S\}$. Observe that for the case of a translating robot, we can define $\mathcal{R}(\vec{p})$ as $\mathcal{R} \oplus \vec{p}$.

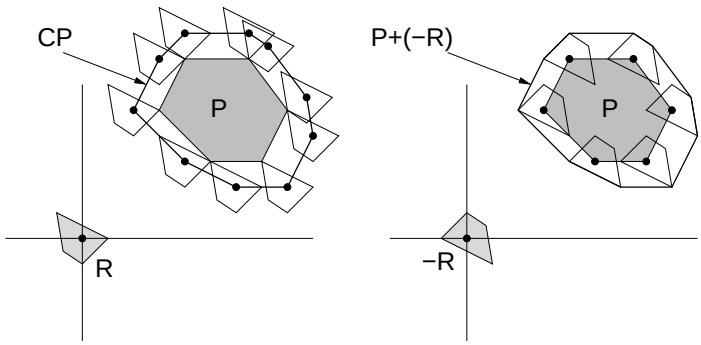


Figure 81: Configuration obstacles and Minkowski sums.

Claim: Given a translating robot $\mathcal{R}$ and an obstacle P, $\mathcal{CP} = P \oplus (-\mathcal{R})$.

Proof: We show that $\mathcal{R}(\vec{q})$ intersects P if and only if $\vec{q} \in P \oplus (-\mathcal{R})$. Note that $\vec{q} \in P \oplus (-\mathcal{R})$ if and only if there exist $\vec{p} \in P$ and $\vec{r} \in \mathcal{R}$ such that $\vec{q} = \vec{p} - \vec{r}$. Similarly, $\mathcal{R}(\vec{q})(= \mathcal{R} \oplus \vec{q})$ intersects P if and only if there exists points $\vec{r} \in \mathcal{R}$ and $\vec{p} \in P$ such that $\vec{r} + \vec{q} = \vec{p}$. These two conditions are clearly equivalent.

Note that the proof made no use of the convexity of $\mathcal{R}$ or P. It works for any shapes and in any dimension. However, computation of the Minkowski sums is most efficient for convex polygons.

It is an easy matter to compute $-\mathcal{R}$ in linear time (by simply negating all of its vertices) the problem of computing the C-obstacle $\mathcal{CP}$ reduces to the problem of computing a Minkowski sum of two convex polygons. We claim that this can be done in $O(m+n)$ time, where m is the number of vertices in $\mathcal{R}$ and n is the number of vertices in P. We will leave the construction as an exercise.

Complexity of Minkowski Sums: We have shown that free space for a translating robot is the complement of a union of C-obstacles $\mathcal{CP}_i$, each of which is a Minkowski sum of the form $P_i \oplus \mathcal{R}$, where P_i ranges over all the obstacles in the environment. If P_i and $\mathcal{R}$ are polygons, then the resulting region will be a union of polygons. How complex might this union be, that is, how many edges and vertices might it have?

To begin with, let's see just how bad things might be. Suppose you are given a robot $\mathcal{R}$ with m sides and a set of work-space obstacle P with n sides. How many sides might the Minkowski sum $P \oplus \mathcal{R}$ have in the worst case? $O(n+m)$? $O(nm)$, even more? The complexity generally depends on what special properties if any P and $\mathcal{R}$ have.

The Union of Pseudodisks: Consider a translating robot given as an m-sided convex polygon and a collection of polygonal obstacles having a total of n vertices. We may assume that the polygonal obstacles have been triangulated into at most n triangles, and so, without any loss of generality, let us consider an instance of an m-sided robot translating among a set of n triangles. As argued earlier, each C-obstacle has $O(3+m) = O(m)$ sides, for a total of $O(nm)$ line segments. A naive analysis suggests that this many line segments might generate as many as $O(n^2m^2)$ intersections, and so the complexity of the free space can be no larger. However, we assert that the complexity of the space will be much smaller, in fact its complexity will be $O(nm)$.

To show that $O(nm)$ is an upper bound, we need some way of extracting the special geometric structure of the union of Minkowski sums. Recall that we are computing the union of $T_i \oplus \mathcal{R}$, where the T_i's have disjoint interiors. A set of convex objects $\{o_1, o_2, \ldots, o_n\}$ is called a *collection of pseudodisks* if for any two distinct objects o_i and o_j both of the set-theoretic differences $o_i \backslash o_j$ and $o_j \backslash o_i$ are connected. That is, if the objects intersect then they do not "cross through" one another. Note that the pseudodisk property is not a property of a single object, but a property that holds among a set of objects.

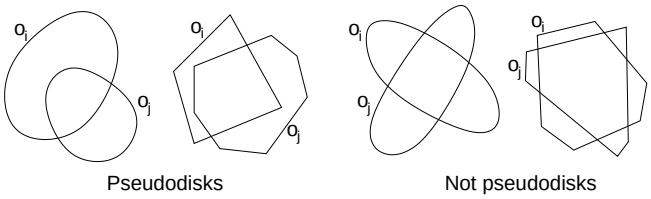


Figure 82: Pseudodisks.

Lemma 1: Given a set convex objects $T_1, T_2, \ldots, T_n$ with disjoint interiors, and convex $\mathcal{R}$, the set

$$\{T_i \oplus \mathcal{R} \mid 1 \leq i \leq n\}$$

is a collection of pseudodisks.

Proof: Consider two polygons T_1 and T_2 with disjoint interiors. We want to show that $T_1 \oplus \mathcal{R}$ and $T_2 \oplus \mathcal{R}$ do not cross over one another.

Given any directional unit vector $\vec{d}$, the *most extreme* point of $\mathcal{R}$ in direction $\vec{d}$ is the point $r \in \mathcal{R}$ that maximizes the dot product $(\vec{d} \cdot r)$. (Recall that we treat the "points" of the polygons as if they were vectors.) The point of $T_1 \oplus \mathcal{R}$ that is most extreme in direction d is the sum of the points t and r that are most extreme for T_1 and $\mathcal{R}$, respectively.

Given two convex polygons T_1 and T_2 with disjoint interiors, they define two outer tangents, as shown in the figure below. Let $\vec{d_1}$ and $\vec{d_2}$ be the outward pointing perpendicular vectors for these tangents. Because these polygons do not intersect, it follows easily that as the directional vector rotates from $\vec{d_1}$ to $\vec{d_2}$, T_1 will be the more extreme polygon, and from $\vec{d_2}$ to $\vec{d_1}$ T_2 will be the more extreme. See the figure below.

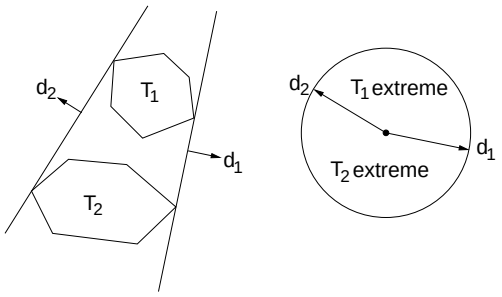


Figure 83: Alternation of extremes.

Now, if to the contrary $T_1 \oplus \mathcal{R}$ and $T_2 \oplus \mathcal{R}$ had a crossing intersection, then observe that we can find points p_1 p_2, p_3, and p_4, in cyclic order around the boundary of the convex hull of $(T_1 \oplus \mathcal{R}) \cup (T_2 \oplus \mathcal{R})$ such that $p_1, p_3 \in T_1 \oplus \mathcal{R}$ and $p_2, p_4 \in T_2 \oplus \mathcal{R}$. First consider p_1. Because it is on the convex hull, consider the direction $\vec{d_1}$ perpendicular to the supporting line here. Let r, t_1, and t_2 be the extreme points of $\mathcal{R}$, T_1 and T_2 in direction $\vec{d_1}$, respectively. From our basic fact about Minkowski sums we have

$$p_1 = r + t_1 \qquad p_2 = r + t_2.$$

Since p_1 is on the convex hull, it follows that t_1 is more extreme than t_2 in direction $\vec{d_1}$, that is, T_1 is more extreme than T_2 in direction $\vec{d_1}$. By applying this same argument, we find that T_1 is more extreme than T_2 in directions $\vec{d_1}$ and $\vec{d_3}$, but that T_2 is more extreme than T_1 in directions $\vec{d_2}$ and $\vec{d_4}$. But this is impossible, since from the observation above, there can be at most one alternation in extreme points for nonintersecting convex polygons. See the figure below.

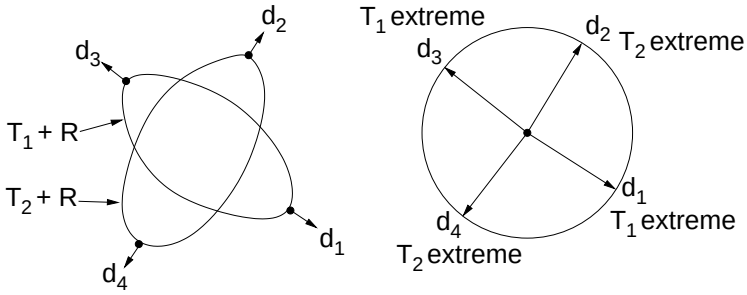


Figure 84: Proof of Lemma 1.

Lemma 2: Given a collection of pseudodisks, with a total of n vertices, the complexity of their union is $O(n)$.

Proof: This is a rather cute combinatorial lemma. We are given some collection of pseudodisks, and told that altogether they have n vertices. We claim that their entire union has complexity $O(n)$. (Recall that in general the union of n convex polygons can have complexity $O(n^2)$, by criss-crossing.) The proof is based on a clever charging scheme. Each vertex in the union will be charged to a vertex among the original pseudodisks, such that no vertex is charged more than twice. This will imply that the total complexity is at most $2n$.

There are two types of vertices that may appear on the boundary. The first are vertices from the original polygons that appear on the union. There can be at most n such vertices, and each is charged to itself. The more troublesome vertices are those that arise when two edges of two pseudodisks intersect each other. Suppose that two edges e_1 and e_2 of pseudodisks P_1 and P_2 intersect along the union. Follow edge e_1 into the interior of the pseudodisk e_2. Two things might happen. First, we might hit the endpoint v of this e_1 before leaving the interior P_2. In this case, charge the intersection to v. Note that v can get at most two such charges, one from either incident edge. If e_1 passes all the way through P_2 before coming to the endpoint, then try to do the same with edge e_2. Again, if it hits its endpoint before coming out of P_1, then charge to this endpoint. See the figure below.

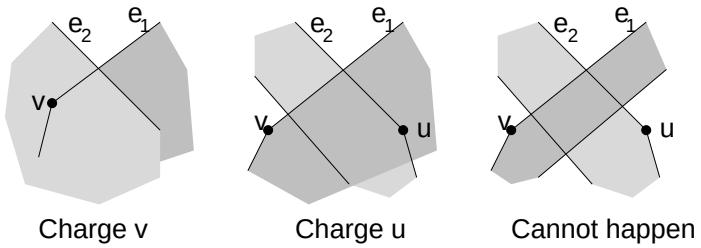


Figure 85: Proof of Lemma 2.

But what do we do if both e_1 shoots straight through P_2 and e_2 shoots straight through P_1? Now we have no vertex to charge. This is okay, because the pseudodisk property implies that this cannot happen. If both edges shoot completely through, then the two polygons must cross over each other.

Recall that in our application of this lemma, we have n C-obstacles, each of which has at most $m + 3$ vertices, for a total input complexity of $O(nm)$. Since they are all pseudodisks, it follows from Lemma 2 that the total complexity of the free space is $O(nm)$.

Lecture 23: DCEL's and Subdivision Intersection

Reading: Chapter 2 in the 4M's.

Doubly-connected Edge List: We consider the question of how to represent plane straight-line graphs (or PSLG). The DCEL is a common *edge-based representation*. Vertex and face information is also included for whatever geometric application is using the data structure. There are three sets of records one for each element in the PSLG: *vertex records*, a *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*.

We will make a simplifying assumption that faces do not have holes inside of them. This assumption can be satisfied by introducing some number of *dummy edges* joining each hole either to the outer boundary of the face, or to some other hole that has been connected to the outer boundary in this way. With this assumption, it may be assumed that the edges bounding each face form a single cyclic list.

Vertex: Each vertex stores its coordinates, along with a pointer to any incident directed edge that has this vertex as its origin, `v.inc_edge`.

Edge: Each undirected edge is represented as two directed edges. Each edge has a pointer to the oppositely directed edge, called its *twin*. Each directed edge has an *origin* and *destination* vertex. Each directed edge is associate with two faces, one to its left and one to its right.

We store a pointer to the origin vertex `e.org`. (We do not need to define the destination, `e.dest`, since it may be defined to be `e.twin.org`.)

We store a pointer to the face to the left of the edge `e.left` (we can access the face to the right from the twin edge). This is called the dent face. We also store the next and previous directed edges in counterclockwise order about the incident face, `e.next` and `e.prev`, respectively.

Face: Each face f stores a pointer to a single edge for which this face is the incident face, `f.inc_edge`. (See the text for the more general case of dealing with holes.)

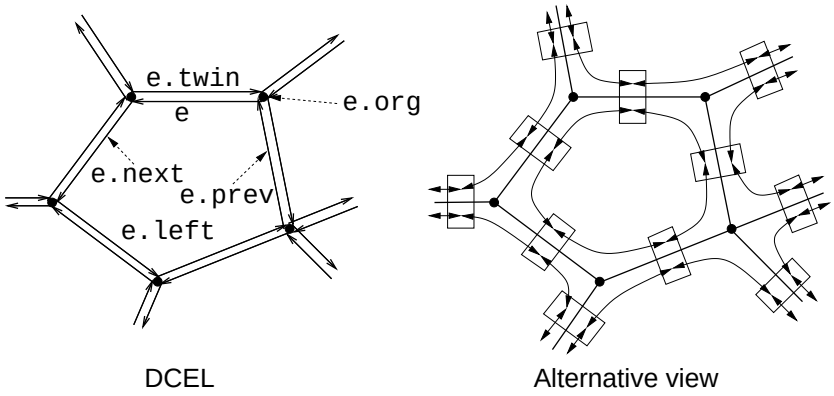


Figure 86: Doubly-connected edge list.

The figure shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternative way of viewing the data structure that gives a better sense of the connectivity structure is based on covering each edge with a two element block, one for e and the other for its twin. The next and prev pointers provide links around each face of the polygon. The next pointers are directed counterclockwise around each face and the prev pointers are directed clockwise.

Of course, in addition the data structure may be enhanced with whatever application data is relevant. In some applications, it is not necessary to know either the face or vertex information (or both) at all, and if so these records may be deleted. See the book for a complete example.

For example, suppose that we wanted to enumerate the vertices that lie on some face f. Here is the code:

Vertex enumeration using DCEL

```
enumerate_vertices(Face f) {
    Edge start = f.inc_edge;
    Edge e = start;
    do {
        output e.org;
        e = e.next;
    } while (e != start);
}
```

Merging subdivisions: Let us return to the applications problem that lead to the segment intersection problem. Suppose that we have two planar subdivisions, S_1 and S_2, and we want to compute their overlay. In particular, this is a subdivision whose vertices are the union of the vertices of each subdivision and the points of intersection of the line segments in the subdivision. (Because we assume that each subdivision is a planar graph, the only new vertices that could arise will arise from the intersection of two edges, one from S_1 and the other from S_2.) Suppose that each subdivision is represented using a DCEL. Can we adapt the plane-sweep algorithm to generate the DCEL of the overlaid subdivision?

The answer is yes. The algorithm will destroy the original subdivisions, so it may be desirable to copy them before beginning this process. The first part of the process is straightforward, but perhaps a little tedious. This part consists of building the edge and vertex records for the new subdivision. The second part involves building the face records. It is more complicated because it is generally not possible to know the face structure at the moment that the sweep is advancing, without looking "into the future" of the sweep to see whether regions will merge. (You might try to convince yourself of this.) The entire subdivision is built first, and then the face information is constructed and added later. We will skip the part of updating the face information (see the text).

For the first part, the most illustrative case arises when the sweep is processing an intersection event. In this case the two segments arise as two edges a_1 and b_1 from the two subdivisions. We will assume that we select the half-edges that are directed from left to right across the sweep-line. The process is described below (and is illustrated in the figure below). It makes use of two auxiliary procedures. $\text{Split}(a_1, a_2)$ splits an edge a_1 at its midpoint into two consecutive edges a_1 followed by a_2, and links a_2 into the structure. $\text{Splice}(a_1, a_2, b_1, b_2)$ takes two such split edges and links them all together.

Merge two edges into a common subdivision

Merge(a_1, b_1) :

(1) Create a new vertex v at the intersection point.

(2) Split each of the two intersecting edges, by adding a vertex at the common intersection point. Let a_2 and b_2 be the new edge pieces. They are created by the calls $a_2 = \text{Split}(a_1)$ and $b_2 = \text{Split}(b_1)$ given below.

(3) Link the four edges together by invoking $\text{Splice}(a_1, a_2, b_1, b_2)$, given below.

The splitting procedure creates the new edge, links it into place. After this the edges have been split, but they are not linked to each other. The edge constructor is given the origin and destination of the new edge and creates a new edge and its twin. The procedure below initializes all the other fields. Also note that the destination of a_1, that is the origin of a_1's twin must be updated, which we have omitted. The splice procedure interlinks four edges around a common vertex in the counterclockwise order a_1 (entering), b_1 (entering), a_2 (leaving), b_2 (leaving).

Split an edge into two edges

```
Split(edge &a1, edge &a2) {              // a2 is returned
    a2 = new edge(v, a1.dest());         // create edge (v,a1.dest)
    a2.next  = a1.next;   a1.next.prev  = a2;
    a1.next  = a2;        a2.prev  = a1;
    a1t = a1.twin;        a2t = a2.twin; // the twins
    a2t.prev = a1t.prev;  a1t.prev.next = a2t;
    a1t.prev = a2t;       a2t.next  = a1t;
}
```

Splice four edges together

```
Splice(edge &a1, edge &a2, edge &b1, edge &b2) {
    a1t = a1.twin;   a2t = a2.twin;      // get the twins
    b1t = b1.twin;   b2t = b2.twin;
    a1.next  = b2;   b2.prev  = a1;      // link the edges together
    b2t.next = a2;   a2.prev  = b2t;
    a2t.next = b1t;  b1t.prev = a2t;
    b1.next  = a1t;  a1t.prev = b1;
}
```

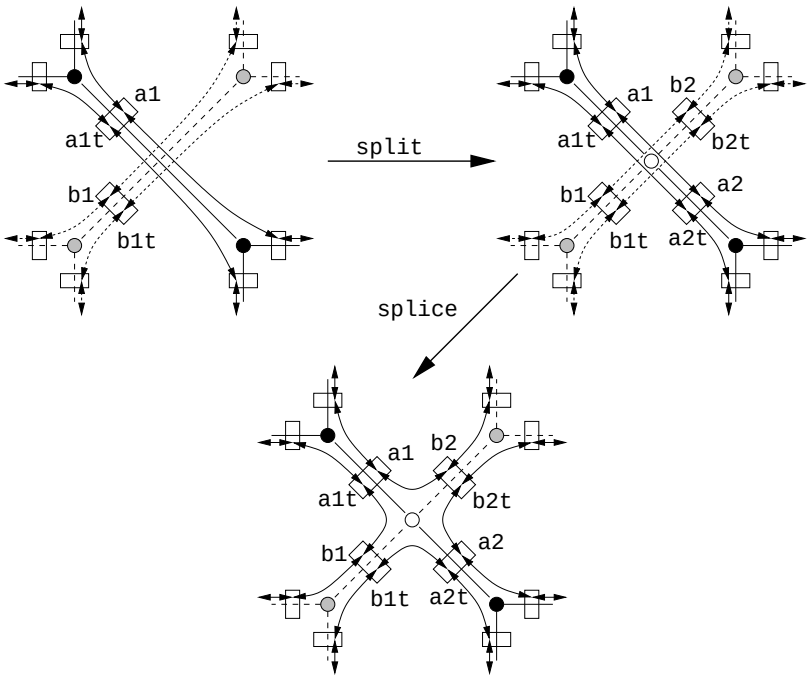


Figure 87: Updating the DCEL.

Lecture 24: Interval Trees

Reading: Chapter 10 in the 4M's.

Segment Data: So far we have considered geometric data structures for storing points. However, there are many others types of geometric data that we may want to store in a data structure. Today we consider how to store orthogonal (horizontal and vertical) line segments in the plane. We assume that a line segment is represented by giving its pair of *endpoints*. The segments are allowed to intersect one another.

As a basic motivating query, we consider the following *window query*. Given a set of orthogonal line segments S, which have been preprocessed, and given an orthogonal query rectangle W, count or report all the line segments of S that intersect W. We will assume that W is closed and solid rectangle, so that even if a line segment lies entirely inside of W or intersects only the boundary of W, it is still reported. For example, given the window below, the query would report the segments that are shown with solid lines, and segments with broken lines would not be reported.

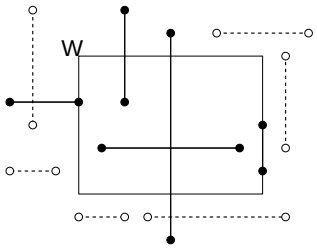


Figure 88: Window Query.

Window Queries for Orthogonal Segments: We will present a data structure, called the *interval tree*, which (combined with a range tree) can answer window counting queries for orthogonal line segments in $O(\log^2 n)$ time, where n is the number line segments. It can report these segments in $O(k+\log^2 n)$ time, where and k is the total number of segments reported. The interval tree uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.

We will consider the case of range reporting queries. (There are some subtleties in making this work for counting queries.) We will derive our solution in steps, starting with easier subproblems and working up to the final solution. To begin with, observe that the set of segments that intersect the window can be partitioned into three types: those that have no endpoint in W, those that have one endpoint in W, and those that have two endpoints in W.

We already have a way to report segments of the second and third types. In particular, we may build a range tree just for the $2n$ endpoints of the segments. We assume that each endpoint has a cross-link indicating the line segment with which it is associated. Now, by applying a range reporting query to W we can report all these endpoints, and follow the cross-links to report the associated segments. Note that segments that have both endpoints in the window will be reported twice, which is somewhat unpleasant. We could fix this either by sorting the segments in some manner and removing duplicates, or by marking each segment as it is reported and ignoring segments that have already been marked. (If we use marking, after the query is finished we will need to go back an "unmark" all the reported segments in preparation for the next query.)

All that remains is how to report the segments that have no endpoint inside the rectangular window. We will do this by building two separate data structures, one for horizontal and one for vertical segments. A horizontal segment that intersects the window but neither of its endpoints intersects the window must pass entirely through the window. Observe that such a segment intersects any vertical line passing from the top of the window to the bottom. In particular, we could simply ask to report all horizontal segments that intersect the left side of W. This is called a *vertical segment stabbing query*. In summary, it suffices to solve the following subproblems (and remove duplicates):

Endpoint inside: Report all the segments of S that have at least one endpoint inside W. (This can be done using a range query.)

Horizontal through segments: Report all the horizontal segments of S that intersect the left side of W. (This reduces to a vertical segment stabbing query.)

Vertical through segments: Report all the vertical segments of S that intersect the bottom side of W. (This reduces to a horizontal segment stabbing query.)

We will present a solution to the problem of vertical segment stabbing queries. Before dealing with this, we will first consider a somewhat simpler problem, and then modify this simple solution to deal with the general problem.

Vertical Line Stabbing Queries: Let us consider how to answer the following query, which is interesting in its own right. Suppose that we are given a collection of horizontal line segments S in the plane and are given an (infinite) vertical query line $\ell_q : x = x_q$. We want to report all the line segments of S that intersect ℓ_q. Notice that for the purposes of this query, the y-coordinates are really irrelevant, and may be ignored. We can think of each horizontal line segment as being a closed *interval* along the x-axis. We show an example in the figure below on the left.

As is true for all our data structures, we want some balanced way to decompose the set of intervals into subsets. Since it is difficult to define some notion of order on intervals, we instead will order the endpoints. Sort the interval endpoints along the x-axis. Let $\langle x_1, x_2, \ldots, x_{2n} \rangle$ be the resulting sorted sequence. Let x_{med} be the median of these $2n$ endpoints. Split the intervals into three groups, L, those that lie strictly to the left of x_{med}, R those that lie strictly to the right of x_{med}, and M those that contain the point x_{med}. We can then define a binary tree by putting the intervals of L in the left subtree and recursing, putting the intervals of R in the right subtree and recursing. Note that if $x_q < x_{\text{med}}$ we can eliminate the right subtree and if $x_q > x_{\text{med}}$ we can eliminate the left subtree. See the figure right.

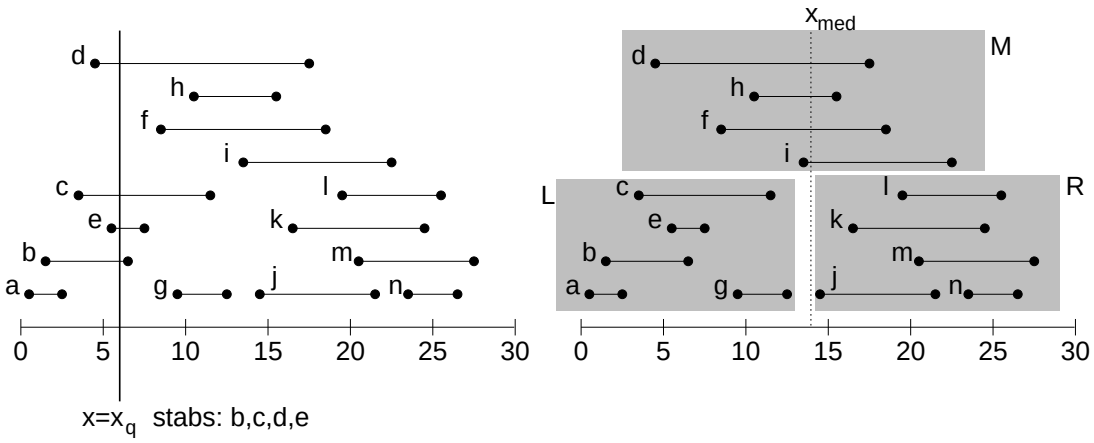


Figure 89: Line Stabbing Query.

But how do we handle the intervals of M that contain x_{med}? We want to know which of these intervals intersects the vertical line ℓ_q. At first it may seem that we have made no progress, since it appears that we are back to the same problem that we started with. However, we have gained the information that all these intervals intersect the vertical line $x = x_{\text{med}}$. How can we use this to our advantage?

Let us suppose for now that $x_q \leq x_{\text{med}}$. How can we store the intervals of M to make it easier to report those that intersect ℓ_q. The simple trick is to sort these lines in increasing order of their left endpoint. Let M_L denote the resulting sorted list. Observe that if some interval in M_L does not intersect ℓ_q, then its left endpoint must be to the right of x_q, and hence none of the subsequent intervals intersects ℓ_q. Thus, to report all the segments of M_L that intersect ℓ_q, we simply traverse the sorted list and list elements until we find one that does not intersect ℓ_q, that is, whose left endpoint lies to the right of x_q. As soon as this happens we terminate. If k' denotes the total number of segments of M that intersect ℓ_q, then clearly this can be done in $O(k' + 1)$ time.

On the other hand, what do we do if $x_q > x_{\text{med}}$? This case is symmetrical. We simply sort all the segments of M in a sequence, M_R, which is sorted from right to left based on the right endpoint of each segment. Thus each element of M is stored twice, but this will not affect the size of the final data structure by more than a constant factor. The resulting data structure is called an *interval tree*.

Interval Trees: The general structure of the interval tree was derived above. Each node of the interval tree has a left child, right child, and itself contains the median x-value used to split the set, x_{med}, and the two sorted sets M_L and M_R (represented either as arrays or as linked lists) of intervals that overlap x_{med}. We assume that there is a constructor that builds a node given these three entities. The following high-level pseudocode describes the basic recursive step in the construction of the interval tree. The initial call is `root = IntTree(S)`, where S is the initial set of intervals. Unlike most of the data structures we have seen so far, this one is not built by the successive insertion of intervals (although it would be possible to do so). Rather we assume that a set of intervals S is given as part of the constructor, and the entire structure is built all at once. We assume that each interval in S is represented as a pair $(x_{\text{lo}}, x_{\text{hi}})$. An example is shown in the following figure.

We assert that the height of the tree is $O(\log n)$. To see this observe that there are $2n$ endpoints. Each time through the recursion we split this into two subsets L and R of sizes at most half the original size (minus the elements of M). Thus after at most $\lg(2n)$ levels we will reduce the set sizes to 1, after which the recursion bottoms out. Thus the height of the tree is $O(\log n)$.

Implementing this constructor efficiently is a bit subtle. We need to compute the median of the set of all endpoints, and we also need to sort intervals by left endpoint and right endpoint. The fastest way to do this is to presort all these values and store them in three separate lists. Then as the sets L, R, and M are computed, we

```
IntTreeNode IntTree(IntervalSet S) {
    if (|S| == 0) return null                              // no more

    xMed = median endpoint of intervals in S               // median endpoint

    L = {[xlo, xhi] in S | xhi < xMed}                     // left of median
    R = {[xlo, xhi] in S | xlo > xMed}                     // right of median
    M = {[xlo, xhi] in S | xlo <= xMed <= xhi}             // contains median
    ML = sort M in increasing order of xlo                 // sort M
    MR = sort M in decreasing order of xhi

    t = new IntTreeNode(xMed, ML, MR)                      // this node
    t.left  = IntTree(L)                                   // left subtree
    t.right = IntTree(R)                                   // right subtree
    return t
}
```

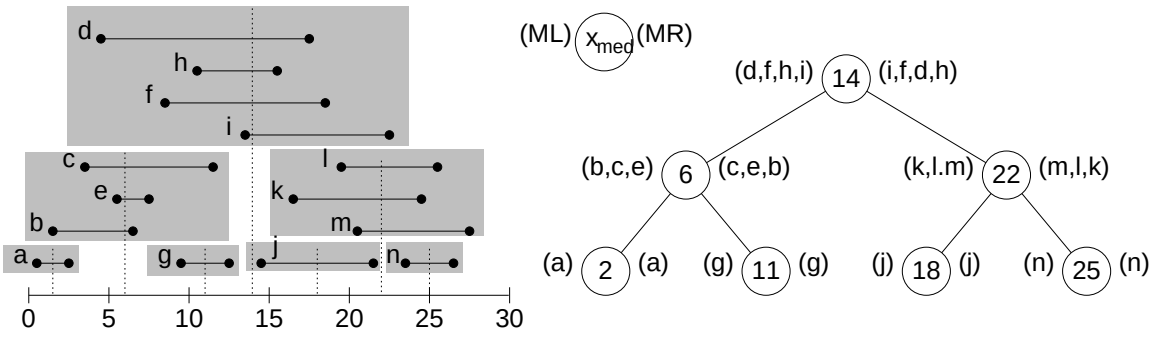


Figure 90: Interval Tree.

simply copy items from these sorted lists to the appropriate sorted lists, maintaining their order as we go. If we do so, it can be shown that this procedure builds the entire tree in $O(n \log n)$ time.

The algorithm for answering a stabbing query was derived above. We summarize this algorithm below. Let x_q denote the x-coordinate of the query line.

Line Stabbing Queries for an Interval Tree

```
stab(IntTreeNode t, Scalar xq) {
    if (t == null) return                          // fell out of tree
    if (xq < t.xMed) {                             // left of median?
        for (i = 0; i < t.ML.length; i++) {        // traverse ML
            if (t.ML[i].lo <= xq) print(t.ML[i])// ..report if in range
            else break                             // ..else done
        }
        stab(t.left, xq)                           // recurse on left
    }
    else {                                         // right of median
        for (i = 0; i < t.MR.length; i++) {        // traverse MR
            if (t.MR[i].hi >= xq) print(t.MR[i])// ..report if in range
            else break                             // ..else done
        }
        stab(t.right, xq)                          // recurse on right
    }
}
```

This procedure actually has one small source of inefficiency, which was intentionally included to make code look more symmetric. Can you spot it? Suppose that $x_q = t.x_{\text{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that are strictly to the right of x_{med} and so is a waste of effort. However it does not affect the asymptotic running time.

As mentioned earlier, the time spent processing each node is $O(1 + k')$ where k' is the total number of points that were recorded at this node. Summing over all nodes, the total reporting time is $O(k + v)$, where k is the total number of intervals reported, and v is the total number of nodes visited. Since at each node we recurse on only one child or the other, the total number of nodes visited v is $O(\log n)$, the height of the tree. Thus the total reporting time is $O(k + \log n)$.

Vertical Segment Stabbing Queries: Now let us return to the question that brought us here. Given a set of horizontal line segments in the plane, we want to know how many of these segments intersect a vertical line segment. Our approach will be exactly the same as in the interval tree, except for how the elements of M (those that intersect the splitting line $x = x_{\text{med}}$) are handled.

Going back to our interval tree solution, let us consider the set M of horizontal line segments that intersect the splitting line $x = x_{\text{med}}$ and as before let us consider the case where the query segment q with endpoints (x_q, y_{lo}) and (x_q, y_{hi}) lies to the left of the splitting line. The simple trick of sorting the segments of M by their left endpoints is not sufficient here, because we need to consider the y-coordinates as well. Observe that a segment of M stabs the query segment q if and only if the left endpoint of a segment lies in the following semi-infinite rectangular region.

$$\{(x, y) \mid x \leq x_q \text{ and } y_{\text{lo}} \leq y \leq y_{\text{hi}}\}.$$

This is illustrated in the figure below. Observe that this is just an orthogonal range query. (It is easy to generalize the procedure given last time to handle semi-infinite rectangles.) The case where q lies to the right of x_{med} is symmetrical.

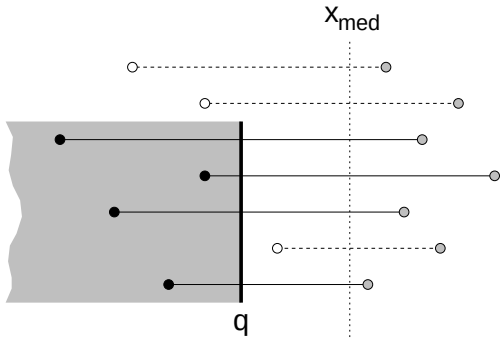


Figure 91: The segments that stab q lie within the shaded semi-infinite rectangle.

So the solution is that rather than storing M_L as a list sorted by the left endpoint, instead we store the left endpoints in a 2-dimensional range tree (with cross-links to the associated segments). Similarly, we create a range tree for the right endpoints and represent M_R using this structure.

The segment stabbing queries are answered exactly as above for line stabbing queries, except that part that searches M_L and M_R (the for-loops) are replaced by searches to the appropriate range tree, using the semi-infinite range given above.

We will not discuss construction time for the tree. (It can be done in $O(n \log n)$ time, but this involves some thought as to how to build all the range trees efficiently). The space needed is $O(n \log n)$, dominated primarily from the $O(n \log n)$ space needed for the range trees. The query time is $O(k + \log^3 n)$, since we need to answer $O(\log n)$ range queries and each takes $O(\log^2 n)$ time plus the time for reporting. If we use the spiffy version of range trees (which we mentioned but never discussed) that can answer queries in $O(k + \log n)$ time, then we can reduce the total time to $O(k + \log^2 n)$.

Lecture 25: Kirkpatrick's Planar Point Location

Today's material is not covered in our text. See the book by Preparata and Shamos, *Computational Geometry*, Section 2.2.2.3.

Point Location: The *point location problem* (in 2-space) is: given a polygonal subdivision of the plane (that is, a PSLG) with n vertices, preprocess this subdivision so that given a query point q, we can efficiently determine which face of the subdivision contains q. We may assume that each face has some identifying label, which is to be returned. We also assume that the subdivision is represented in any "reasonable" form (e.g. as a DCEL). In general q may coincide with an edge or vertex. To simplify matters, we will assume that q does not lie on an edge or vertex, but these special cases are not hard to handle.

It is remarkable that although this seems like such a simple and natural problem, it took quite a long time to discover a method that is optimal with respect to both query time and space. It has long been known that there are data structures that can perform these searches reasonably well (e.g. quad-trees and kd-trees), but for which no good theoretical bounds could be proved. There were data structures of with $O(\log n)$ query time but $O(n \log n)$ space, and $O(n)$ space but $O(\log^2 n)$ query time.

The first construction to achieve both $O(n)$ space and $O(\log n)$ query time was a remarkably clever construction due to Kirkpatrick. It turns out that Kirkpatrick's idea has some large embedded constant factors that make it less attractive practically, but the idea is so clever that it is worth discussing, nonetheless. Later we will discuss a more practical randomized method that is presented in our text.

Kirkpatrick's Algorithm: Kirkpatrick's idea starts with the assumption that the planar subdivision is a triangulation, and further that the outer face is a triangle. If this assumption is not met, then we begin by triangulating all the faces of the subdivision. The label associated with each triangular face is the same as a label for the original face that contained it. For the outer face is not a triangle, first compute the convex hull of the polygonal subdivision, triangulate everything inside the convex hull. Then surround this convex polygon with a large triangle (call the vertices a, b, and c), and then add edges from the convex hull to the vertices of the convex hull. It may sound like we are adding a lot of new edges to the subdivision, but recall from earlier in the semester that the number of edges and faces in any straight-line planar subdivision is proportional to n, the number of vertices. Thus the addition only increases the size of the structure by a constant factor.

Note that once we find the triangle containing the query point in the augmented graph, then we will know the original face that contains the query point. The triangulation process can be performed in $O(n \log n)$ time by a plane sweep of the graph, or in $O(n)$ time if you want to use sophisticated methods like the linear time polygon triangulation algorithm. In practice, many straight-line subdivisions, may already have convex faces and these can be triangulated easily in $O(n)$ time.

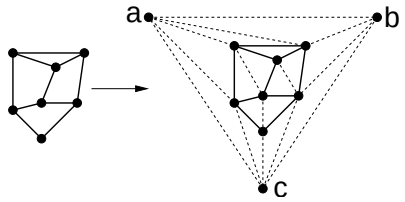


Figure 92: Triangulation of a planar subdivision.

Let T_0 denote the initial triangulation. What Kirkpatrick's method does is to produce a sequence of triangulations, $T_0, T_1, T_2, \ldots, T_k$, where $k = O(\log n)$, such that T_k consists only of a single triangle (the exterior face of T_0), and each triangle in T_{i+1} overlaps a constant number of triangles in T_i.

We will see how to use such a structure for point location queries later, but for now let us concentrate on how to build such a sequence of triangulations. Assuming that we have T_i, we wish to compute T_{i+1}. In order to guarantee that this process will terminate after $O(\log n)$ stages, we will want to make sure that the number of vertices in T_{i+1} decreases by some constant factor from the number of vertices in T_i. In particular, this will be done by carefully selecting a subset of vertices of T_i and deleting them (and along with them, all the edges attached to them). After these vertices have been deleted, we need retriangulate the resulting graph to form T_{i+1}. The question is: How do we select the vertices of T_i to delete, so that each triangle of T_{i+1} overlaps only a constant number of triangles in T_i?

There are two things that Kirkpatrick observed at this point, that make the whole scheme work.

Constant degree: We will make sure that each of the vertices that we delete have constant ($\leq d$) degree (that is, each is adjacent to at most d edges). Note that the when we delete such a vertex, the resulting *hole* will consist of at most $d - 2$ triangles. When we retriangulate, each of the new triangles, can overlap at most d triangles in the previous triangulation.

Independent set: We will make sure that no two of the vertices that are deleted are adjacent to each other, that is, the vertices to be deleted form an *independent set* in the current planar graph T_i. This will make retriangulation easier, because when we remove m independent vertices (and their incident edges), we create m independent *holes* (non triangular faces) in the subdivision, which we will have to retriangulate. However, each of these holes can be triangulated independently of one another. (Since each hole contains a constant number of vertices, we can use any triangulation algorithm, even brute force, since the running time will be $O(1)$ in any case.)

An important question to the success of this idea is whether we can always find a sufficiently large independent set of vertices with bounded degree. We want the size of this set to be at least a constant fraction of the current

number of vertices. Fortunately, the answer is "yes," and in fact it is quite easy to find such a subset. Part of the trick is to pick the value of d to be large enough (too small and there may not be enough of them). It turns out that $d = 8$ is good enough.

Lemma: Given a planar graph with n vertices, there is an independent set consisting of vertices of degree at most 8, with at least $n/18$ vertices. This independent set can be constructed in $O(n)$ time.

We will present the proof of this lemma later. The number 18 seems rather large. The number is probably smaller in practice, but this is the best bound that this proof generates. However, the size of this constant is one of the reasons that Kirkpatrick's algorithm is not used in practice. But the construction is quite clever, nonetheless, and once a optimal solution is known to a problem it is often not long before a practical optimal solution follows.

Kirkpatrick Structure: Assuming the above lemma, let us give the description of how the point location data structure, the *Kirkpatrick structure*, is constructed. We start with T_0, and repeatedly select an independent set of vertices of degree at most 8. We never include the three vertices a, b, and c (forming the outer face) in such an independent set. We delete the vertices from the independent set from the graph, and retriangulate the resulting *holes*. Observe that each triangle in the new triangulation can overlap at most 8 triangles in the previous triangulation. Since we can eliminate a constant fraction of vertices with each stage, after $O(\log n)$ stages, we will be down to the last 3 vertices.

The constant factors here are not so great. With each stage, the number of vertices falls by a factor of $17/18$. To reduce to the final three vertices, implies that $(18/17)^k = n$ or that

$$k = \log_{18/17} n \approx 12 \lg n.$$

It can be shown that by always selecting the vertex of smallest degree, this can be reduced to a more palatable $4.5 \lg n$.

The data structure is based on this decomposition. The root of the structure corresponds to the single triangle of T_k. The nodes at the next lower level are the triangles of T_{k-1}, followed by T_{k-2}, until we reach the leaves, which are the triangles of our initial triangulation, T_0. Each node for a triangle in triangulation T_{i+1}, stores pointers to all the triangles it overlaps in T_i (there are at most 8 of these). Note that this structure is a directed acyclic graph (DAG) and not a tree, because one triangle may have many parents in the data structure. This is shown in the following figure.

To locate a point, we start with the root, T_k. If the query point does not lie within this single triangle, then we are done (it lies in the exterior face). Otherwise, we search each of the (at most 8) triangles in T_{k-1} that overlap this triangle. When we find the correct one, we search each of the triangles in T_{k-2} that overlap this triangles, and so forth. Eventually we will find the triangle containing the query point in the last triangulation, T_0, and this is the desired output. See the figure below for an example.

Construction and Analysis: The structure has $O(\log n)$ levels (one for each triangulation), it takes a constant amount of time to move from one level to the next (at most 8 point-in-triangle tests), thus the total query time is $O(\log n)$. The size of the data structure is the sum of sizes of the triangulations. Since the number of triangles in a triangulation is proportional to the number of vertices, it follows that the size is proportional to

$$n(1 + 17/18 + (17/18)^2 + (17/18)^3 + \ldots) \leq 18n,$$

(using standard formulas for geometric series). Thus the data structure size is $O(n)$ (again, with a pretty hefty constant).

The last thing that remains is to show how to construct the independent set of the appropriate size. We first present the algorithm for finding the independent set, and then prove the bound on its size.

(1) Mark all nodes of degree ≥ 9.

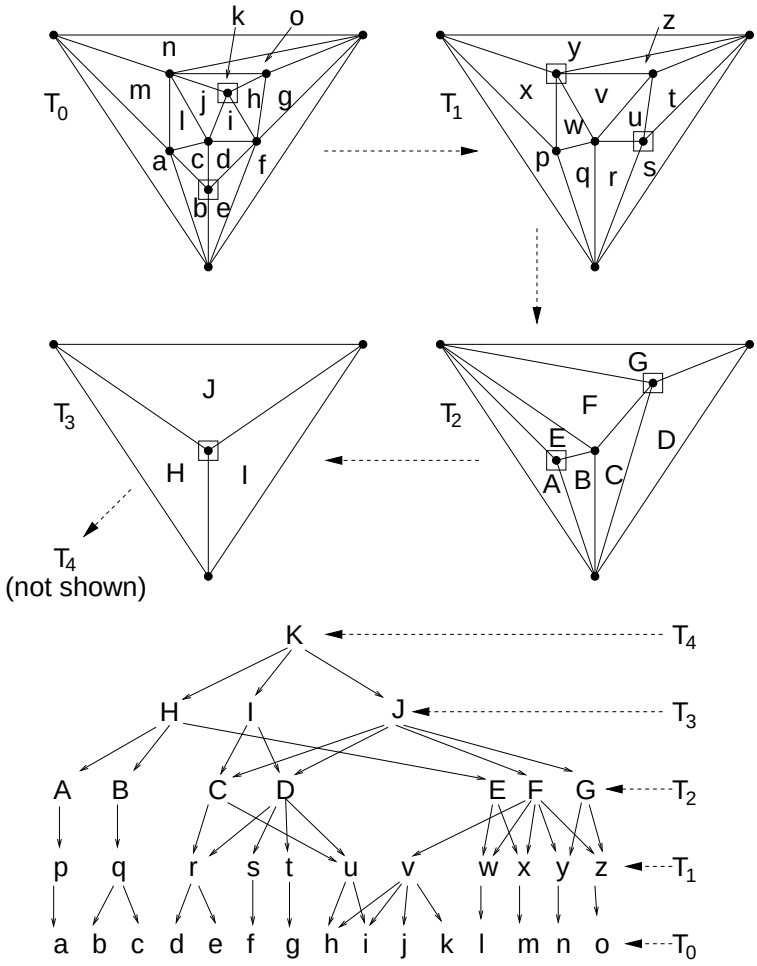


Figure 93: Kirkpatrick's point location structure.

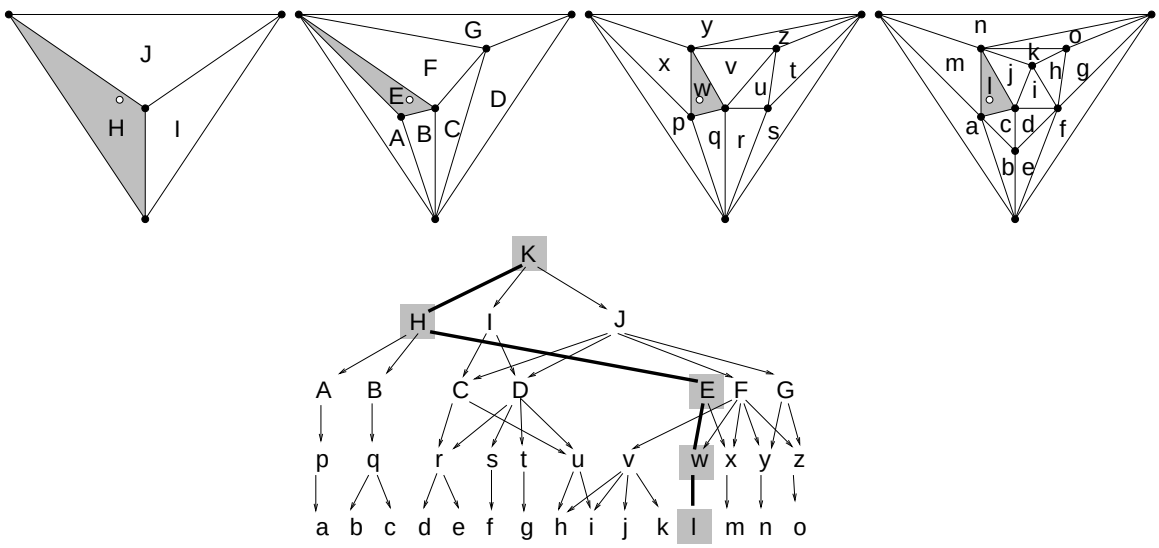


Figure 94: Point location search.

(2) While there exists an unmarked node do the following:

(a) Choose an unmarked vertex v.

(b) Add v to the independent set.

(c) Mark v and all of its neighbors.

It is easy to see that the algorithm runs in $O(n)$ time (e.g., by keeping unmarked vertices in a stack and representing the triangulation so that neighbors can be found quickly.)

Intuitively, the argument that there exists a large independent set of low degree is based on the following simple observations. First, because the average degree in a planar graph is less than 6, there must be a lot of vertices of degree at most 8 (otherwise the average would be unattainable). Second, whenever we add one of these vertices to our independent set, only 8 other vertices become ineligible for inclusion in the independent set.

Here is the rigorous argument. Recall from Euler's formula, that if a planar graph is fully triangulated, then the number of edges e satisfies $e = 3n - 6$. If we sum the degrees of all the vertices, then each edge is counted twice. Thus the average degree of the graph is

$$\sum_v \deg(v) = 2e = 6n - 12 < 6n.$$

Next, we claim that there must be at least $n/2$ vertices of degree 8 or less. To see why, suppose to the contrary that there were more than $n/2$ vertices of degree 9 or greater. The remaining vertices must have degree at least 3 (with the possible exception of the 3 vertices on the outer face), and thus the sum of all degrees in the graph would have to be at least as large as

$$9\frac{n}{2} + 3\frac{n}{2} = 6n,$$

which contradicts the equation above.

Now, when the above algorithm starts execution, at least $n/2$ vertices are initially unmarked. Whenever we select such a vertex, because its degree is 8 or fewer, we mark at most 9 new vertices (this node and at most 8 of its neighbors). Thus, this step can be repeated at least $(n/2)/9 = n/18$ times before we run out of unmarked vertices. This completes the proof.

Lecture 26: Lower Bound for Convex Hulls

Lower Bound: We have presented Chan's $O(n \log h)$ time algorithm for convex hulls. We will show that this result is asymptotically optimal in the sense that any algorithm for computing the convex hull of n points with h points on the hull requires $\Omega(n \log h)$ time. The proof is a generalization of the proof that sorting a set of n numbers requires $\Omega(n \log n)$ comparisons.

If you recall the proof that sorting takes at least $\Omega(n \log n)$ comparisons, it is based on the idea that any sorting algorithm can be described in terms of a *decision tree*. Each comparison has at most 3 outcomes ($<$, $=$, or $>$). Each such comparison corresponds to an internal node in the tree. The execution of an algorithm can be viewed as a traversal along a path in the resulting 3-ary tree. The height of the tree is a lower bound on the worst-case running time of the algorithm. There are at least $n!$ different possible inputs, each of which must be reordered differently, and so you have a 3-ary tree with at least n leaves. Any such tree must have $\Omega(\log_3 n!)$ height. Using Stirling's approximation for $n!$, this solves to $\Omega(n \log n)$ height.

We will give an $\Omega(n \log h)$ lower bound for the convex hull problem. In fact, we will give an $\Omega(n \log h)$ lower bound on the following simpler decision problem, whose output is either yes or no.

Convex Hull Size Verification Problem (CHSV): Given a point set P and integer h, does the convex hull of P have h distinct vertices?

Clearly if this takes $\Omega(n \log h)$ time, then computing the hull must take at least as long. As with sorting, we will assume that the computation is described in the form of a decision tree. Assuming that the algorithm uses only comparisons is too restrictive for computing convex hulls, so we will generalize the model of computation to allow for more complex functions. We assume that we are allowed to compute any algebraic function of the point coordinates, and test the sign of the resulting function. The result is called a *algebraic decision tree*.

The input to the CHSV problem is a sequence of $2n = N$ real numbers. We can think of these numbers as forming a vector $(z_1, z_2, \ldots, z_N) = \vec{z} \in R^N$, which we will call a *configuration*. Each node of the decision tree is associated with a multivariate algebraic formula of degree at most d. e.g.

$$f(\vec{z}) = z_1 z_4 - 2 z_3 z_6 + 5 z_6^2,$$

would be an algebraic function of degree 2. The node branches in one of three ways, depending on whether the result is negative, zero, or positive. (Computing orientations and dot-products both fall into this category.) Each leaf of the resulting tree corresponds to a possible answer that the algorithm might give.

For each input vector $\vec{z}$ to the CHSV problem, the answer is either "yes" or "no". The set of all "yes" points is just a subset of points $Y \subset R^N$, that is a region in this space. Given an arbitrary input $\vec{z}$ the purpose of the decision tree is to tell us whether this point is in Y or not. This is done by walking down the tree, evaluating the functions on $\vec{z}$ and following the appropriate branches until arriving at a leaf, which is either labeled "yes" (meaning $\vec{z} \in Y$) or "no". An abstract example (not for the convex hull problem) of a region of configuration space and a possible algebraic decision tree (of degree 1) is shown in the following figure. (We have simplified it by making it a binary tree.) In this case the input is just a pair of real numbers.

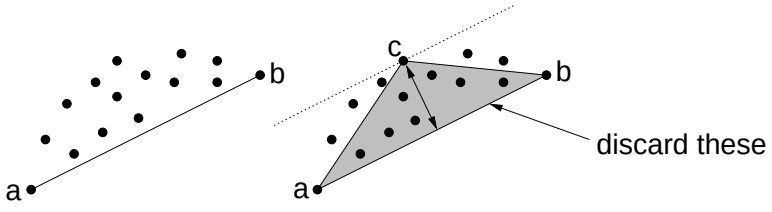


Figure 95: The geometric interpretation of an algebraic decision tree.

We say that two points $\vec{u}, \vec{v} \in Y$ are in the same *connected component* of Y if there is a path in R^N from $\vec{u}$ to $\vec{v}$ such that all the points along the path are in the set Y. (There are two connected components in the figure.) We will make use of the following fundamental result on algebraic decision trees, due to Ben-Or. Intuitively, it states that if your set has M connected components, then there must be at least M leaves in any decision tree for the set, and the tree must have height at least the logarithm of the number of leaves.

Theorem: Let $Y \in R^N$ be any set and let T be any d-th order algebraic decision tree that determines membership in W. If W has M disjoint connected components, then T must have height at least $\Omega((\log M) - N)$.

We will begin our proof with a simpler problem.

Multiset Size Verification Problem (MSV): Given a multiset of n real numbers and an integer k, confirm that the multiset has exactly k distinct elements.

Lemma: The MSV problem requires $\Omega(n \log k)$ steps in the worst case in the d-th order algebraic decision tree

Proof: In terms of points in R^n, the set of points for which the answer is "yes" is

$$Y = \{(z_1, z_2, \ldots, z_n) \in R^n \mid |\{z_1, z_2, \ldots, z_n\}| = k\}.$$

It suffices to show that there are at least $k!k^{n-k}$ different connected components in this set, because by Ben-Or's result it would follow that the time to test membership in Y would be

$$\Omega(\log(k!k^{n-k}) - n) \;=\; \Omega(k \log k + (n-k)\log k - n) \;=\; \Omega(n \log k).$$

Consider the all the tuples $(z_1, \ldots, z_n)$ with $z_1, \ldots z_k$ set to the distinct integers from 1 to k, and $z_{k+1} \ldots z_n$ each set to an arbitrary integer in the same range. Clearly there are $k!$ ways to select the first k elements and k^{n-k} ways to select the remaining elements. Each such tuple has exactly k distinct items, but it is not hard to see that if we attempt to continuously modify one of these tuples to equal another one, we must change the number of distinct elements, implying that each of these tuples is in a different connected component of Y.

To finish the lower bound proof, we argue that any instance of MSV can be reduced to the convex hull size verification problem (CHSV). Thus any lower bound for MSV problem applies to CHSV as well.

Theorem: The CHSV problem requires $\Omega(n \log h)$ time to solve.

Proof: Let $Z = (z_1, \ldots, z_n)$ and k be an instance of the MSV problem. We create a point set $\{p_1, \ldots, p_n\}$ in the plane where $p_i = (z_i, z_i^2)$, and set $h = k$. (Observe that the points lie on a parabola, so that all the points are on the convex hull.) Now, if the multiset Z has exactly k distinct elements, then there are exactly $h = k$ points in the point set (since the others are all duplicates of these) and so there are exactly h points on the hull. Conversely, if there are h points on the convex hull, then there were exactly $h = k$ distinct numbers in the multiset to begin with in Z.

Thus, we cannot solve CHSV any faster than $\Omega(n \log h)$ time, for otherwise we could solve MSV in the same time.

The proof is rather unsatisfying, because it relies on the fact that there are many duplicate points. You might wonder, does the lower bound still hold if there are no duplicates? Kirkpatric and Seidel actually prove a stronger (but harder) result that the $\Omega(n \log h)$ lower bound holds even you assume that the points are distinct.

Lecture 27: Voronoi Diagrams

Reading: O'Rourke, Chapt 5, Mulmuley, Sect. 2.8.4.

Planar Voronoi Diagrams: Recall that, given n points $P = \{p_1, p_2, \ldots, p_n\}$ in the plane, the Voronoi polygon of a point p_i, $V(p_i)$, is defined to be the set of all points q in the plane for which p_i is among the closest points to q in P. That is,

$$V(p_i) = \{q \ : \ |p_i - q| \leq |p_j - q|, \forall j \neq i\}.$$

The union of the boundaries of the Voronoi polygons is called the *Voronoi diagram* of P, denoted $VD(P)$. The dual of the Voronoi diagram is a triangulation of the point set, called the *Delaunay triangulation*. Recall from our discussion of quad-edge data structure, that given a good representation of any planar graph, the dual is easy to construct. Hence, it suffices to show how to compute either one of these structures, from which the other can be derived easily in $O(n)$ time.

There are four fairly well-known algorithms for computing Voronoi diagrams and Delaunay triangulations in the plane. They are

Divide-and-Conquer: (For both VD and DT.) The first $O(n \log n)$ algorithm for this problem. Not widely used because it is somewhat hard to implement. Can be generalized to higher dimensions with some difficulty. Can be generalized to computing Vornoi diagrams of line segments with some difficulty.

Randomized Incremental: (For DT and VD.) The simplest. $O(n \log n)$ time with high probability. Can be generalized to higher dimensions as with the randomized algorithm for convex hulls. Can be generalized to computing Voronoi diagrams of line segments fairly easily.

Fortune's Plane Sweep: (For VD.) A very clever and fairly simple algorithm. It computes a "deformed" Voronoi diagram by plane sweep in $O(n \log n)$ time, from which the true diagram can be extracted easily. Can be generalized to computing Voronoi diagrams of line segments fairly easily.

Reduction to convex hulls: (For DT.) Computing a Delaunay triangulation of n points in dimension d can be reduced to computing a convex hull of n points in dimension $d + 1$. Use your favorite convex hull algorithm. Unclear how to generalize to compute Voronoi diagrams of line segments.

We will cover all of these approaches, except Fortune's algorithm. O'Rourke does not give detailed explanations of any of these algorithms, but he does discuss the idea behind Fortune's algorithm. Today we will discuss the divide-and-conquer algorithm. This algorithm is presented in Mulmuley, Section 2.8.4.

Divide-and-conquer algorithm: The divide-and-conquer approach works like most standard geometric divide-and-conquer algorithms. We split the points according to x-coordinates into 2 roughly equal sized groups (e.g. by presorting the points by x-coordinate and selecting medians). We compute the Voronoi diagram of the left side, and the Voronoi diagram of the right side. Note that since each diagram alone covers the entire plane, these two diagrams overlap. We then merge the resulting diagrams into a single diagram.

The merging step is where all the work is done. Observe that every point in the the plane lies within two Voronoi polygons, one in $VD(L)$ and one in $VD(R)$. We need to resolve this overlap, by separating overlapping polygons. Let $V(l_0)$ be the Voronoi polygon for a point from the left side, and let $V(r_0)$ be the Voronoi polygon for a point on the right side, and suppose these polygons overlap one another. Observe that if we insert the bisector between l_0 and r_0, and through away the portions of the polygons that lie on the "wrong" side of the bisector, we resolve the overlap. If we do this for every pair of overlapping Voronoi polygons, we get the final Voronoi diagram. This is illustrated in the figure below.

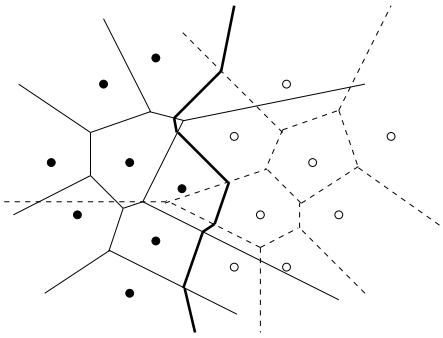
The union of these bisectors that separate the left Voronoi diagram from the right Voronoi diagram is called the *contour*. A point is on the contour if and only if it is equidistant from 2 points in S, one in L and one in R.

(0) Presort the points by x-coordinate (this is done once).

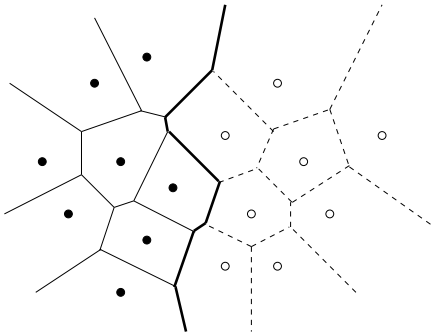
(1) Split the point set S by a vertical line into two subsets L and R of roughly equal size.

(2) Compute $VD(L)$ and $VD(R)$ recursively. (These diagrams overlap one another.)

(3) Merge the two diagrams into a single diagram, by computing the *contour* and discarding the portion of the $VD(L)$ that is to the right of the contour, and the portion of $VD(R)$ that is to the left of the contour.



Left/Right Diagrams and Contour



Final Voronoi Diagram

Figure 96: Merging Voronoi diagrams.

Assuming we can implement step (3) in $O(n)$ time (where n is the size of the remaining point set) then the running time will be defined by the familiar recurrence

$$T(n) = 2T(n/2) + n,$$

which we know solves to $O(n \log n)$.

Computing the contour: What makes the divide-and-conquer algorithm somewhat tricky is the task of computing the contour. Before giving an algorithm to compute the contour, let us make some observations about its geomtetric structure. Let us make the usual simplifying assumptions that no 4 points are cocircular.

Lemma: The contour consists of a single polygonal curve (whose first and last edges are semiinfinite) which is monotone with respect to the y-axis.

Proof: A detailed proof is a real hassle. Here are the main ideas, though. The contour separates the plane into two regions, those points whose nearest neighbor lies in L from those points whose nearest neighbor lies in R. Because the contour locally consists of points that are equidistant from 2 points, it is formed from pieces that are perpendicular bisectors, with one point from L and the other point from R. Thus, it is a piecewise polygonal curve. Because no 4 points are cocircular, it follows that all vertices in the Voronoi diagram can have degree at most 3. However, because the contour separates the plane into only 2 types of regions, it can contain only vertices of degree 2. Thus it can consist only of the disjoint union of closed curves (actually this never happens, as we will see) and unbounded curves. Observe that if we orient the contour counterclockwise with respect to each point in R (clockwise with respect to each point in L), then each segment must be directed in the $-y$ directions, because L and R are separated by a vertical line. Thus, the contour contains no horizontal cusps. This implies that the contour cannot contain any closed curves, and hence contains only vertically monotone unbounded curves. Also, this orientability also implies that there is only one such curve.

Lemma: The topmost (bottommost) edge of the contour is the perpendicular bisector for the two points forming the upper (lower) tangent of the left hull and the right hull.

Proof: This follows from the fact that the vertices of the hull correspond to unbounded Voronoi polygons, and hence upper and lower tangents correspond to unbounded edges of the contour.

These last two theorem suggest the general approach. We start by computing the upper tangent, which we know can be done in linear time (once we know the left and right hulls, or by prune and search). Then, we start tracing the contour from top to bottom. When we are in Voronoi polygons $V(l_0)$ and $V(r_0)$ we trace the bisector between l_0 and r_0 in the negative y-direction until its first contact with the boundaries of one of these polygons. Suppose that we hit the boundary of $V(l_0)$ first. Assuming that we use a good data structure for the Voronoi diagram (e.g. quad-edge data structure) we can determine the point l_1 lying on the other side of this edge in the left Voronoi diagram. We continue following the contour by tracing the bisector of l_1 and r_0.

However, in order to insure efficiency, we must be careful in how we determine where the bisector hits the edge of the polygon. Consider the figure shown below. We start tracing the contour between l_0 and r_0. By walking along the boundary of $V(l_0)$ we can determine the edge that the contour would hit first. This can be done in time proportional to the number of edges in $V(l_0)$ (which can be as large as $O(n)$). However, we discover that before the contour hits the boundary of $V(l_0)$ it hits the boundary of $V(r_0)$. We find the new point r_1 and now trace the bisector between l_0 and r_1. Again we can compute the intersection with the boundary of $V(l_0)$ in time proportional to its size. However the contour hits the boundary of $V(r_1)$ first, and so we go on to r_2. As can be seen, if we are not smart, we can rescan the boundary of $V(l_0)$ over and over again, until the contour finally hits the boundary. If we do this $O(n)$ times, and the boundary of $V(l_0)$ is $O(n)$, then we are stuck with $O(n^2)$ time to trace the contour.

We have to avoid this repeated rescanning. However, there is a way to scan the boundary of each Voronoi polygon at most once. Observe that as we walk along the contour, each time we stay in the same polygon $V(l_0)$, we are adding another edge onto its Voronoi polygon. Because the Voronoi polygon is convex, we know

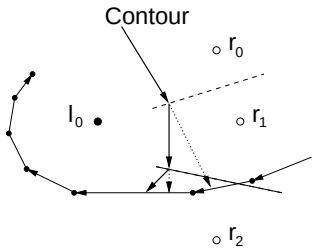


Figure 97: Tracing the contour.

that the edges we are creating turn consistently in the same direction (clockwise for points on the left, and counterclockwise for points on the right). To test for intersections between the contour and the current Voronoi polygon, we trace the boundary of the polygon clockwise for polygons on the left side, and counterclockwise for polygons on the right side. Whenever the contour changes direction, we continue the scan from the point that we left off. In this way, we know that we will never need to rescan the same edge of any Voronoi polygon more than once.

Lecture 28: Delaunay Triangulations and Convex Hulls

Reading: O'Rourke 5.7 and 5.8.

Delaunay Triangulations and Convex Hulls: At first, Delaunay triangulations and convex hulls appear to be quite different structures, one is based on metric properties (distances) and the other on affine properties (collinearity, coplanarity). Today we show that it is possible to convert the problem of computing a Delaunay triangulation in dimension d to that of computing a convex hull in dimension $d + 1$. Thus, there is a remarkable relationship between these two structures.

We will demonstrate the connection in dimension 2 (by computing a convex hull in dimension 3). Some of this may be hard to visualize, but see O'Rourke for illustrations. (You can also reason by analogy in one lower dimension of Delaunay triangulations in 1-d and convex hulls in 2-d, but the real complexities of the structures are not really apparent in this case.)

The connection between the two structures is the *paraboloid* $z = x^2 + y^2$. Observe that this equation defines a surface whose vertical cross sections (constant x or constant y) are parabolas, and whose horizontal cross sections (constant z) are circles. For each point in the plane, (x, y), the *vertical projection* of this point onto this paraboloid is $(x, y, x^2 + y^2)$ in 3-space. Given a set of points S in the plane, let S' denote the projection of every point in S onto this paraboloid. Consider the *lower convex hull* of S'. This is the portion of the convex hull of S' which is visible to a viewer standing at $z = -\infty$. We claim that if we take the lower convex hull of S', and project it back onto the plane, then we get the Delaunay triangulation of S. In particular, let $p, q, r \in S$, and let p', q', r' denote the projections of these points onto the paraboloid. Then $p'q'r'$ define a *face* of the lower convex hull of S' if and only if $\triangle pqr$ is a triangle of the Delaunay triangulation of S. The process is illustrated in the following figure.

The question is, why does this work? To see why, we need to establish the connection between the triangles of the Delaunay triangulation and the faces of the convex hull of transformed points. In particular, recall that

Delaunay condition: Three points $p, q, r \in S$ form a Delaunay triangle if and only if the circumcircle of these points contains no other point of S.

Convex hull condition: Three points $p', q', r' \in S'$ form a face of the convex hull of S' if and only if the plane passing through p', q', and r' has all the points of S' lying to one side.

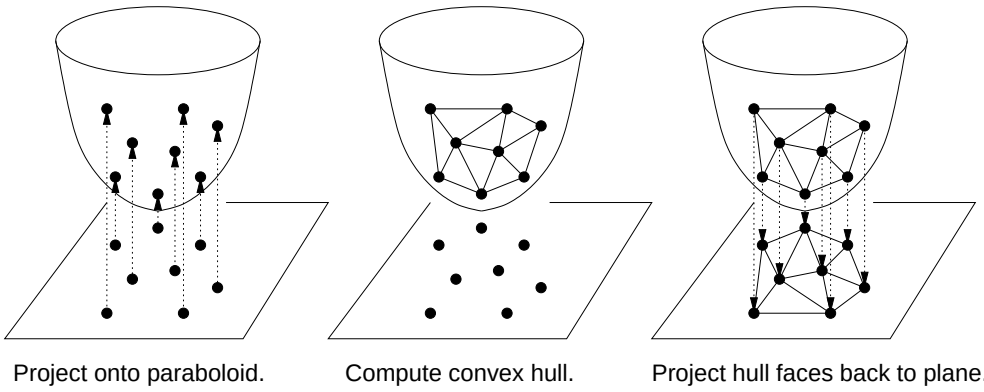


Figure 98: Delaunay triangulations and convex hull.

Clearly, the connection we need to establish is between the emptiness of circumcircles in the plane and the emptiness of halfspaces in 3-space. We will prove the following claim.

Lemma: Consider 4 distinct points p, q, r, s in the plane, and let p', q', r', s' be their respective projections onto the paraboloid, $z = x^2 + y^2$. The point s lies within the circumcircle of p, q, r if and only if s' lies on the lower side of the plane passing through p', q', r'.

To prove the lemma, first consider an arbitrary (nonvertical) plane in 3-space, which we assume is tangent to the paraboloid above some point (a, b) in the plane. To determine the equation of this tangent plane, we take derivatives of the equation $z = x^2 + y^2$ with respect to x and y giving

$$\frac{\partial z}{\partial x} = 2x, \qquad \frac{\partial z}{\partial y} = 2y.$$

At the point $(a, b, a^2 + b^2)$ these evaluate to $2a$ and $2b$. It follows that the plane passing through these point has the form

$$z = 2ax + 2by + \gamma.$$

To solve for γ we know that the plane passes through $(a, b, a^2 + b^2)$ so we solve giving

$$a^2 + b^2 = 2a \cdot a + 2b \cdot b + \gamma,$$

Implying that $\gamma = -(a^2 + b^2)$. Thus the plane equation is

$$z = 2ax + 2by - (a^2 + b^2).$$

If we shift the plane upwards by some positive amount r^2 we get the plane

$$z = 2ax + 2by - (a^2 + b^2) + r^2.$$

How does this plane intersect the paraboloid? Since the paraboloid is defined by $z = x^2 + y^2$ we can eliminate z giving

$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + r^2,$$

which after some simple rearrangements is equal to

$$(x - a)^2 + (y - b)^2 = r^2.$$

This is just a circle. Thus, we have shown that the intersection of a plane with the paraboloid produces a space curve (which turns out to be an ellipse), which when projected back onto the (x, y)-coordinate plane is a circle centered at (a, b).

Thus, we conclude that the intersection of an arbitrary lower halfspace with the paraboloid, when projected onto the (x, y)-plane is the interior of a circle. Going back to the lemma, when we project the points p, q, r onto the paraboloid, the projected points p', q' and r' define a plane. Since p', q', and r', lie at the intersection of the plane and paraboloid, the original points p, q, r lie on the projected circle. Thus this circle is the (unique) circumcircle passing through these p, q, and r. Thus, the point s lies within this circumcircle, if and only if its projection s' onto the paraboloid lies within the lower halfspace of the plane passing through p, q, r.

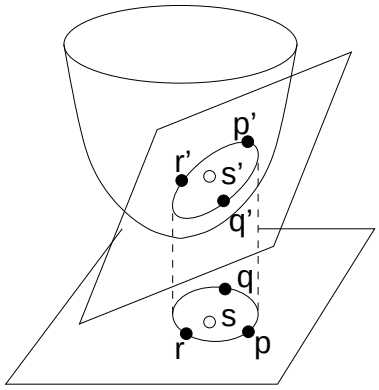


Figure 99: Planes and circles.

Now we can prove the main result.

Theorem: Given a set of points S in the plane (assume no 4 are cocircular), and given 3 points $p, q, r \in S$, the triangle $\triangle pqr$ is a triangle of the Delaunay triangulation of S if and only if triangle $\triangle p'q'r'$ is a face of the lower convex hull of the projected set S'.

From the definition of Delaunay triangulations we know that $\triangle pqr$ is in the Delaunay triangulation if and only if there is no point $s \in S$ that lies within the circumcircle of pqr. From the previous lemma this is equivalent to saying that there is no point s' that lies in the lower convex hull of S', which is equivalent to saying that $p'q'r'$ is a face of the lower convex hull. This completes the proof.

In order to test whether a point s lies within the circumcircle defined by p, q, r, it suffices to test whether s' lies within the lower halfspace of the plane passing through p', q', r'. If we assume that p, q, r are oriented counterclockwise in the plane this this reduces to determining whether the quadruple p', q', r', s' is positively oriented, or equivalently whether s lies to the left of the oriented circle passing through p, q, r.

This leads to the incircle test we presented last time.

$$\text{in}(p, q, r, s) \;=\; \det \begin{pmatrix} p_x & p_y & p_x^2 + p_y^2 & 1 \\ q_x & q_y & q_x^2 + q_y^2 & 1 \\ r_x & r_y & r_x^2 + r_y^2 & 1 \\ s_x & s_y & s_x^2 + s_y^2 & 1 \end{pmatrix} > 0.$$

Voronoi Diagrams and Upper Envelopes: We know that Voronoi diagrams and Delaunay triangulations are dual geometric structures. We have also seen (informally) that there is a dual relationship between points and lines in the plane, and in general, points and planes in 3-space. From this latter connection we argued that the problems of computing convex hulls of point sets and computing the intersection of halfspaces are somehow "dual" to one another. It turns out that these two notions of duality, are (not surprisingly) interrelated. In particular, in the

same way that the Delaunay triangulation of points in the plane can be transformed to computing a convex hull in 3-space, it turns out that the Voronoi diagram of points in the plane can be transformed into computing the intersection of halfspaces in 3-space.

Here is how we do this. For each point $p = (a, b)$ in the plane, recall the tangent plane to the paraboloid above this point, given by the equation

$$z = 2ax + 2by - (a^2 + b^2).$$

Define $H^+(p)$ to be the set of points that are above this halfplane, that is, $H^+(p) = \{(x, y, z) \mid z \geq 2ax + 2by - (a^2 + b^2)\}$. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of points. Consider the intersection of the halfspaces $H^+(p_i)$. This is also called the *upper envelope* of these halfspaces. The upper envelope is an (unbounded) convex polyhedron. If you project the edges of this upper envelope down into the plane, it turns out that you get the Voronoi diagram of the points.

Theorem: Given a set of points S in the plane (assume no 4 are cocircular), let H denote the set of upper halfspaces defined by the previous transformation. Then the Voronoi diagram of H is equal to the projection onto the (x, y)-plane of the 1-skeleton of the convex polyhedron which is formed from the intersection of halfspaces of S'.

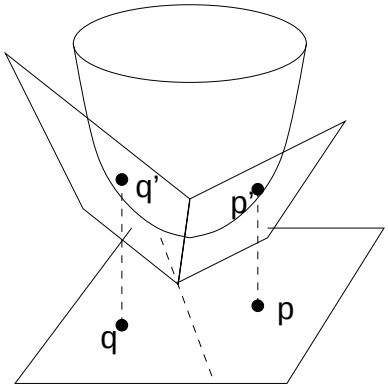


Figure 100: Intersection of halfspaces.

It is hard to visualized this surface, but it is not hard to show why this is so. Suppose we have 2 points in the plane, $p = (a, b)$ and $q = (c, d)$. The corresponding planes are:

$$z = 2ax + 2by - (a^2 + b^2) \qquad \text{and} \qquad z = 2cx + 2dy - (c^2 + d^2).$$

If we determine the intersection of the corresponding planes and project onto the (x, y)-coordinate plane (by eliminating z from these equations) we get

$$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2).$$

We claim that this is the perpendicular bisector between (a, b) and (c, d). To see this, observe that it passes through the midpoint $((a + c)/2, (b + d)/2)$ between the two points since

$$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2).$$

and, its slope is $-(a-c)/(b-d)$, which is the negative reciprocal of the line segment from (a, b) to (c, d). From this it can be shown that the intersection of the upper halfspaces defines a polyhedron whose edges project onto the Voronoi diagram edges.

Lecture 29: Topological Plane Sweep

Reading: The material on topological plane sweep is not discussed in any of our readings. The algorithm for topological plane sweep can be found in the paper, "Topologically sweeping an arrangement" by H. Edelsbrunner and L. J. Guibas, *J. Comput. Syst. Sci.*, 38 (1989), 165–194, with Corrigendum in the same journal, volume 42 (1991), 249–251.

Topological Plane Sweep: In the last two lectures we have introduced arrangements of lines and geometric duality as important tools in solving geometric problems on lines and points. Today give an efficient algorithm for sweeping an arrangement of lines.

As we will see, many problems in computational geometry can be solved by applying line-sweep to an arrangement of lines. Since the arrangement has size $O(n^2)$, and since there are $O(n^2)$ events to be processed, each involving an $O(\log n)$ heap deletion, this typically leads to algorithms running in $O(n^2 \log n)$ time, using $O(n^2)$ space. It is natural to ask whether we can dispense with the additional $O(\log n)$ factor in running time, and whether we need all of $O(n^2)$ space (since in theory we only need access to the current $O(n)$ contents of the sweep line).

We discuss a variation of plane sweep called *topological plane sweep*. This method runs in $O(n^2)$ time, and uses only $O(n)$ space (by essentially constructing only the portion of the arrangement that we need at any point). Although it may appear to be somewhat sophisticated, it can be implemented quite efficiently, and is claimed to outperform conventional plane sweep on arrangements of any significant size (e.g. over 20 lines).

Cuts and topological lines: The algorithm is called *topological* plane sweep because we do not sweep a straight vertical line through the arrangement, but rather we sweep a curved *topological line* that has the essential properties of a vertical sweep line in the sense that this line intersects each line of the arrangement exactly once. The notion of a topological line is an intuitive one, but it can be made formal in the form of something called a *cut*. Recall that the faces of the arrangement are convex polygons (possibly unbounded). (Assuming no vertical lines) the edges incident to each face can naturally be partitioned into the edges that are *above* the face, and those that are *below* the face. Define a *cut* in an arrangement to be a sequence of edges $c_1, c_2, \ldots, c_n$, in the arrangement, one taken from each line of the arrangement, such that for $1 \leq i \leq n-1$, c_i and c_{i+1} are incident to the same face of the arrangement, and c_i is above the face and c_{i+1} is below the face. An example of a topological line and the associated cut is shown below.

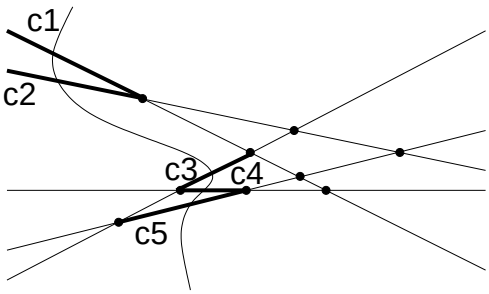


Figure 101: Topological line and associated cut.

The topological plane sweep starts at the *leftmost cut* of the arrangement. This consists of all the left-unbounded edges of the arrangement. Observe that this cut can be computed in $O(n \log n)$ time, because the lines intersect the cut in inverse order of slope. The topological sweep line will sweep to the right until we come to the rightmost cut, which consists all of the right-unbounded edges of the arrangement. The sweep line advances by a series of what are called *elementary steps*. In an elementary steps, we find two consecutive edges on the cut that meet at a vertex of the arrangement (we will discuss later how to determine this), and push the topological

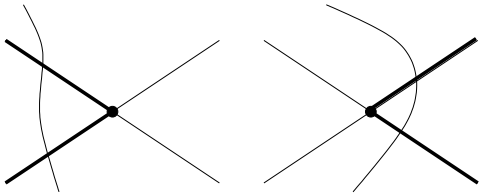


Figure 102: Elementary step.

sweep line through this vertex. Observe that on doing so these two lines swap in their order along the sweep line. This is shown below.

It is not hard to show that an elementary step is always possible, since for any cut (other than the rightmost cut) there must be two consecutive edges with a common right endpoint. In particular, consider the edge of the cut whose right endpoint has the smallest x-coordinate. It is not hard to show that this endpoint will always allow an elementary step. Unfortunately, determining this vertex would require at least $O(\log n)$ time (if we stored these endpoints in a heap, sorted by x-coordinate), and we want to perform each elementary step in $O(1)$ time. Hence, we will need to find some other method for finding elementary steps.

Upper and Lower Horizon Trees: To find elementary steps, we introduce two simple data structures, the *upper horizon tree* (UHT) and the *lower horizon tree* (LHT). To construct the upper horizon tree, trace each edge of the cut to the right. When two edges meet, keep only the one with the higher slope, and continue tracing it to the right. The lower horizon tree is defined symmetrically. There is one little problem in these definitions in the sense that these trees need not be connected (forming a forest of trees) but this can be fixed conceptually at least by the addition of a vertical line at $x = +\infty$. For the upper horizon we think of its slope as being $+\infty$ and for the lower horizon we think of its slope as being $-\infty$. Note that we consider the left endpoints of the edges of the cut as not belonging to the trees, since otherwise they would not be trees. It is not hard to show that with these modifications, these are indeed trees. Each edge of the cut defines exactly one line segment in each tree. An example is shown below.

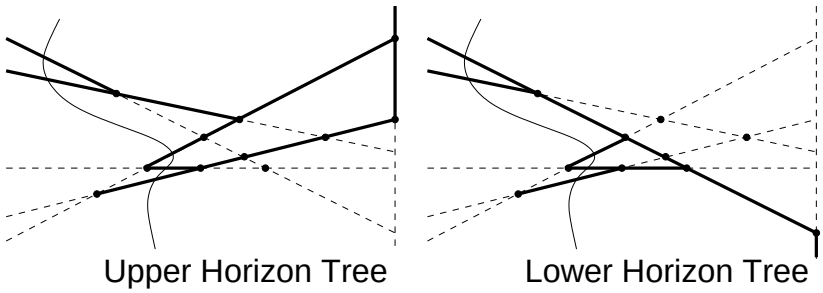


Figure 103: Upper and lower horizon trees.

The important things about the UHT and LHT is that they give us an easy way to determine the right endpoints of the edges on the cut. Observe that for each edge in the cut, its right endpoint results from a line of smaller slope intersecting it from above (as we trace it from left to right) or from a line of larger slope intersecting it from below. It is easy to verify that the UHT and LHT determine the first such intersecting line of each type, respectively. It follows that if we intersect the two trees, then the segments they share in common correspond exactly to the edges of the cut. Thus, by knowing the UHT and LHT, we know where are the right endpoints are, and from this we can determine easily which pairs of consecutive edges share a common right endpoint, and from this we can determine all the elementary steps that are legal. We store all the legal steps in a stack (or queue, or any list is fine), and extract them one by one.

The sweep algorithm: Here is an overview of the topological plane sweep.

(1) Input the lines and sort by slope. Let C be the initial (leftmost) cut, a list of lines in decreasing order of slope.

(2) Create the initial UHT incrementally by inserting lines in decreasing order of slope. Create the initial LHT incrementally by inserting line in increasing order of slope. (More on this later.)

(3) By consulting the LHT and UHT, determine the right endpoints of all the edges of the initial cut, and for all pairs of consecutive lines (l_i, l_{i+1}) sharing a common right endpoint, store this pair in stack S.

(4) Repeat the following elementary step until the stack is empty (implying that we have arrived at the rightmost cut).

  (a) Pop the pair (l_i, l_{i+1}) from the top of the stack S.

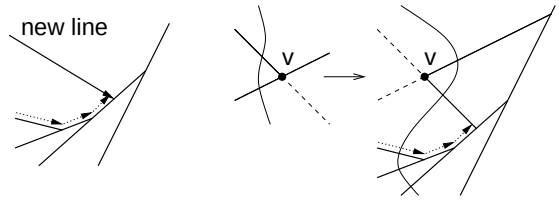
  (b) Swap these lines within C, the cut (we assume that each line keeps track of its position in the cut).

  (c) Update the horizon trees. (More on this later.)

  (d) Consulting the changed entries in the horizon tree, determine whether there are any new cut edges sharing right endpoints, and if so push them on the stack S.

The important unfinished business is to show that we can build the initial UHT and LHT in $O(n)$ time, and to show that, for each elementary step, we can update these trees and all other relevant information in $O(1)$ *amortized time*. By *amortized time* we mean that, even though a single elementary step can take more than $O(1)$ time, the total time needed to perform all $O(n^2)$ elementary steps is $O(n^2)$, and hence the average time for each step is $O(1)$.

This is done by an adaptation of the same incremental "face walking" technique we used in the incremental construction of line arrangements. Let's consider just the UHT, since the LHT is symmetric. To create the initial (leftmost) UHT we insert the lines one by one in decreasing order of slope. Observe that as each new line is inserted it will start above all of the current lines. The uppermost face of the current UHT consists of a convex polygonal chain, see the figure below left. As we trace the newly inserted line from left to right, there will be some point at which it first hits this upper chain of the current UHT. By walking along the chain from left to right, we can determine this intersection point. Each segment that is walked over is never visited again by this initialization process (because it is no longer part of the upper chain), and since the initial UHT has a total of $O(n)$ segments, this implies that the total time spent in walking is $O(n)$. Thus, after the $O(n \log n)$ time for sorting the segments, the initial UHT tree can be built in $O(n)$ additional time.



Initial UHT construction. Updating the UHT.

Figure 104: Constructing and updating the UHT.

Next we show how to update the UHT after an elementary step. The process is quite similar, as shown in the figure right. Let v be the vertex of the arrangement which is passed over in the sweep step. As we pass over v, the two edges swap positions along the sweep line. The new lower edge, call it l, which had been cut off of the UHT by the previous lower edge, now must be reentered into the tree. We extend l to the left until it contacts an edge of the UHT. At its first contact, it will terminate (and this is the only change to be made to the UHT). In order to find this contact, we start with the edge immediately below l the current cut. We traverse the face of the UHT in counterclockwise order, until finding the edge that this line intersects. Observe that we must eventually

find such an edge because l has a lower slope than the other edge intersecting at v, and this edge lies in the same face.

Analysis: A careful analysis of the running time can be performed using the same amortization proof (based on pebble counting) that was used in the analysis of the incremental algorithm. We will not give the proof in full detail. Observe that because we maintain the set of legal elementary steps in a stack (as opposed to a heap as would be needed for standard plane sweep), we can advance to the next elementary step in $O(1)$ time. The only part of the elementary step that requires more than constant time is the update operations for the UHT and LHT. However, we claim that the total time spent updating these trees is $O(n^2)$. The argument is that when we are tracing the edges (as shown in the previous figure) we are "essentially" traversing the edges in the *zone* for L in the arrangement. (This is not quite true, because there are edges above l in the arrangement, which have been cut out of the upper tree, but the claim is that their absence cannot increase the complexity of this operation, only decrease it. However, a careful proof needs to take this into account.) Since the zone of each line in the arrangement has complexity $O(n)$, all n zones have total complexity $O(n^2)$. Thus, the total time spent in updating the UHT and LHT trees is $O(n^2)$.

Lecture 30: Ham-Sandwich Cuts

Reading: This material is not covered in our book.

Ham Sandwich Cuts of Linearly Separated Point Sets: We are given n red points A, and m blue points B, and we want to compute a single line that simultaneously bisects both sets. (If the cardinality of either set is odd, then the line passes through one of the points of the set.) We make the simplifying assumption that the sets are separated by a line. (This assumption makes the problem much simpler to solve, but the general case can still be solved in $O(n^2)$ time using arrangements.)

To make matters even simpler we assume that the points have been translated and rotated so this line is the y-axis. Thus all the red points (set A) have positive x-coordinates, and hence their dual lines have positive slopes, whereas all the blue points (set B) have negative x-coordinates, and hence their dual lines have negative slopes. As long as we are simplifying things, let's make one last simplification, that both sets have an odd number of points. This is not difficult to get around, but makes the pictures a little easier to understand.

Consider one of the sets, say A. Observe that for each slope there exists one way to bisect the points. In particular, if we start a line with this slope at positive infinity, so that all the points lie beneath it, and drop in downwards, eventually we will arrive at a unique placement where there are exactly $(n-1)/2$ points above the line, one point lying on the line, and $(n-1)/2$ points below the line (assuming no two points share this slope). This line is called the *median line* for this slope.

What is the dual of this median line? If we dualize the points using the standard dual transformation: $\mathcal{D}(a,b)$: $y = ax - b$, then we get n lines in the plane. By starting a line with a given slope above the points and translating it downwards, in the dual plane we moving a point from $-\infty$ upwards in a vertical line. Each time the line passes a point in the primal plane, the vertically moving point crosses a line in the dual plane. When the translating line hits the median point, in the dual plane the moving point will hit a dual line such that there are exactly $(n-1)/2$ dual lines above this point and $(n-1)/2$ dual lines below this point. We define a point to be at *level* k, $\mathcal{L}_k$, in an arrangement if there are at most $k-1$ lines above this point and at most $n-k$ lines below this point. The median level in an arrangement of n lines is defined to be the $\lceil (n-1)/2 \rceil$-th level in the arrangement. This is shown as $M(A)$ in the following figure on the left.

Thus, the set of bisecting lines for set A in dual form consists of a polygonal curve. Because this curve is formed from edges of the dual lines in A, and because all lines in A have positive slope, this curve is monotonically increasing. Similarly, the median for B, $M(B)$, is a polygonal curve which is monotonically decreasing. It follows that A and B must intersect at a unique point. The dual of this point is a line that bisects both sets.

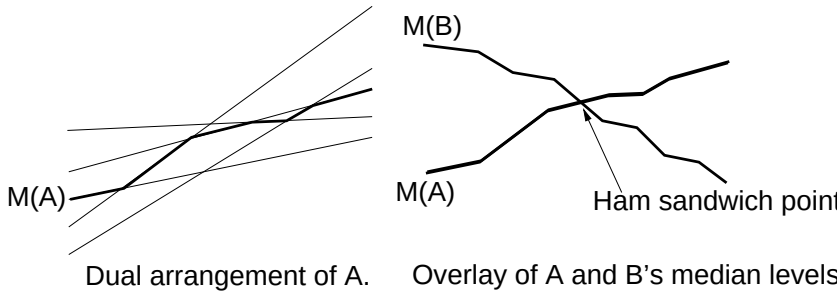


Figure 105: Ham sandwich: Dual formulation.

We could compute the intersection of these two curves by a simultaneous topological plane sweep of both arrangements. However it turns out that it is possible to do much better, and in fact the problem can be solved in $O(n + m)$ time. Since the algorithm is rather complicated, I will not describe the details, but here are the essential ideas. The algorithm operates by prune and search. In $O(n + m)$ time we will generate a hypothesis for where the ham sandwich point is in the dual plane, and if we are wrong, we will succeed in throwing away a constant fraction of the lines from future consideration.

First observe that for any vertical line in the dual plane, it is possible to determine in $O(n + m)$ time whether this line lies to the left or the right of the intersection point of the median levels, $M(A)$ and $M(B)$. This can be done by computing the intersection of the dual lines of A with this line, and computing their median in $O(n)$ time, and computing the intersection of the dual lines of B with this line and computing their median in $O(m)$ time. If A's median lies below B's median, then we are to the left of the ham sandwich dual point, and otherwise we are to the right of the ham sandwich dual point. It turns out that with a little more work, it is possible to determine in $O(n+m)$ time whether the ham sandwich point lies to the right or left of a line of *arbitrary* slope. The trick is to use prune and search. We find two lines L_1 and L_2 in the dual plane (by a careful procedure that I will not describe). These two lines define four quadrants in the plane. By determining which side of each line the ham sandwich point lies, we know that we can throw away any line that does not intersect this quadrant from further consideration. It turns out that by a judicious choice of L_1 and L_2, we can guarantee that a fraction of at least $(n + m)/8$ lines can be thrown away by this process. We recurse on the remaining lines. By the same sort of analysis we made in the Kirkpatrick and Seidel prune and search algorithm for upper tangents, it follows that in $O(n + m)$ time we will find the ham sandwich point.