CMSC 427/828M: Computer Graphics[1]
Spring 1997
Dave Mount

Lecture 1: Introduction

(Tuesday, Sep 2, 1997)

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 has been used often to describe algorithms solid-modeling. But its most widely recognized use is to describe the subfield of algorithm theory that involves the design and analysis of efficient algorithms for problems involving geometric inputs, primarily in 2-, 3-, or perhaps consant dimensional spaces. It primarily involves straight or flat objects (lines, line segments, polygons, planes, and polyhedra) as opposed to curves and surfaces. It is this latter sense of the term that we will be covering in this course.

The field developed rapidly in the late 70's and through the 80's and 90's, and it still continues to develop. Because of the area from which it grew (discrete algorithm design), the field of computational geometry has always emphasized problems of a discrete mathematic nature. For most problems in computational geometry the input is a finite set of points or other geometric objects, and the output is a typically some sort of structure consisting of a finite set of points or line segments.

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.

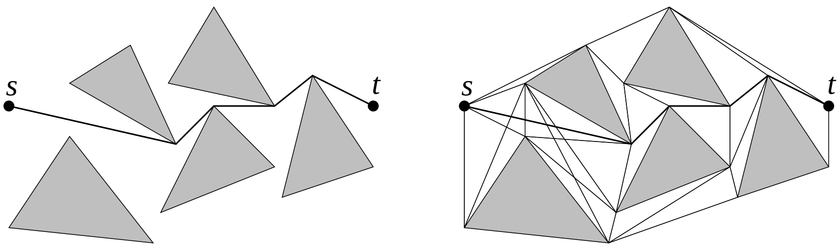


Figure 1: Shortest path problem.

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, and in $O(n \log n)$ by a very complex algorithm.) 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 naturea* 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. Furthmore, 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 theretically 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 practioners. 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.

A Grand Overview: 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.

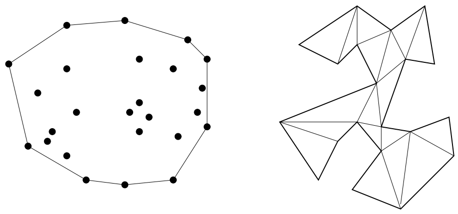


Figure 2: Convex hulls and polygon triangulation.

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 whic 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.

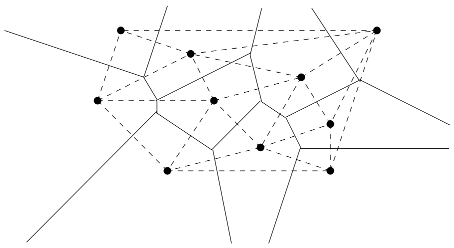


Figure 3: Voronoi diagram and Delaunay triangulation.

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.

Course Texts: The text for this course is the excellent new book *Computational Geometry: Algorithms and Applications* by de Berg, van Kreveld, Overmars and Schwarzkopf. Many of the algorithms, notations, definitions, and figures will be borrowed from their presentation. We'll refer to their book by their initials BKOS. We'll also refer to the excellent algorithms textbook *Introduction to Algorithms* by Cormen, Leiserson, and Rivest for basic algorithms and data structures.

Lecture 2: Geometric Background and Convex Hulls

(Thursday, Sep 4, 1997)
(Updated: 4:30pm, Sep 4, 1997)
Reading: Chapter 1 in BKOS (deBerg, vanKreveld, Overmars and Schwarzkopf).

Affine space: We will usually be rather informal in our presentation of geometry, but it is good to start things off on a somewhat formal footing. We begin with some background on Euclidean geometry, which is not presented in the text.

There are a number of different geometric systems that one can work under. This semester we will be working almost exclusively with Euclidean geometry, although we may introduce a few concepts from projective geometry later in the semester. We begin with a short review of geometry. Our approach will be different from what is done in most math texts and most computational geometry texts. The standard approach in math texts is to begin by assuming that everything is a "tuple of real numbers". But this approach does lend itself to object-oriented programming because it blurs the distinction between different geometric elements. The approach in computational geometry texts is to say nothing about geometric representations, but to rely upon an intuitive understanding of geometric concepts.

Rather than defining 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 differentiates Euclidean from 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. The following are the legal operations:

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

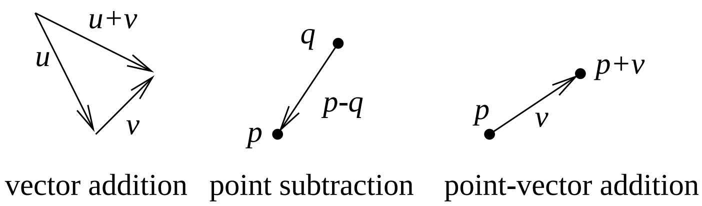


Figure 4: Affine operations.

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 any scalar α and two points p_0 and p_1, define the affine combination $\mathrm{Aff}(p_0, p_1, \alpha)$

to be:

$$(1-\alpha)\cdot p_0 + \alpha \cdot p_1 = p_0 + \alpha \cdot (p_1 - p_0).$$

Note that the left-hand side of this 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 the point $\mathrm{Aff}(p_0, p_1, \alpha)$ lies on the line connecting p_0 and p_1, and generally, as α varies from $-\infty$ to $+\infty$ it traces out all the points on this line.

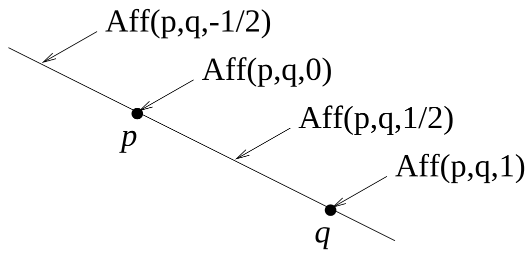


Figure 5: Affine combination.

In the special case where $0 \leq \alpha \leq 1$, $\mathrm{Aff}(p_0, p_1, \alpha)$ is a point that subdivides the line segment $\overline{p_0p_1}$ into two subsegments of relative sizes α to $1-\alpha$. When α is in this range, the operation is called a *convex combination*, and the set of all convex combinations traces out the line segment $\overline{p_0p_1}$.

Homogeneous Coordinates: In order to assign coordinates to points and vectors, we assume that there is a designated *standard coordinate system*, which is specified by an origin point and d orthogonal unit vectors. (In many applications, such as computer graphics and solid modeling, it is often convenient to define other local coordinate systems. We will not have much need of this though.)

To represent vectors and points in affine space, we use *homogeneous coordinates*. Suppose that we are working in d-dimensional affine space. It is common to represent both free vectors and points as $(d+1)$-tuples of real numbers. To represent a free vector we take its standard d-tuple of coordinates an prepend an additional 0 to the beginning. To represent a point we take its d-tuple and prepend an additional 1. (In many application areas, graphics and solid-modeling in particular, it is common to append the 0 or 1 to the end of the tuple. In principal there is no reason that one representation is better than the other, but beware that the concept of orientation defined below is reversed in odd dimensions in this case.)

This may sound rather ad-hoc, but it has some very nice algebraic properties. For example, if you take the difference of two points (which results in a free vector) observe that by simply subtracting their homogeneous coordinate tuples, component by component, you will get the proper representation of this free vector (since the first two coordinates will cancel each other giving 0). We will sometimes be sloppy. When it is clear that we are dealing with points, we may drop the homogenizing coordinate.

Orientation: So far all of the operations map numerical geometric entities into other numerical entities. In order to make discrete decisions, we need a geometric operation that is somewhat 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 relations in 1-space, called *orientation*.

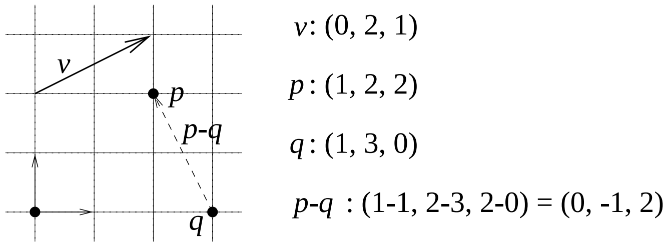


Figure 6: Homogeneous coordinates.

Given an ordered triple of three 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. Note that orientation depends on the order in which the points are given. 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.

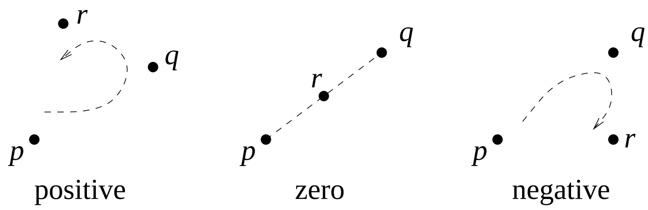


Figure 7: Orientations of the ordered triple (p, q, r).

Orientation is formally defined as the sign of the determinant of the points given in homogeneous coordinates. For example, in the plane, we have

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

Observe that in the 1-dimensional case, $\operatorname{Orient}(p, q)$ is just $q - p$. Hence it is positive if $p < q$, zero if $p = q$, and negative if $p > q$. Thus orientation generalized $<, =, >$ in 1-dimensional space.

Convexity: Now that we have discussed some of the basics, let us consider an initial problem. The computation of convex hulls is among the most basic problems in computational geometry. An $O(n \log n)$ algorithm for computing convex hulls was one of the earliest results in computational geometry (due to Ron Graham).

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. 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. Also many algorithms compute the convex hull as an initial stage in their execution, because convex polygons are often easier to deal with than point sets. 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.) A convex combination of three or more points is an linear combination of the points in which the coefficients sum to 1 and all the coefficients are in the interval $[0, 1]$.

In general, convex sets may have either straight or curved boundaries, may be bounded or unbounded (e.g. an infinite cone is convex), and may be topologically open or closed (that is, they may or may not contain their boundary). The convex hull of a finite set of points is necessarily a bounded, closed, convex polygon.

Convex hull problem: The (planar) *convex hull problem* is, given a set of n points P in the plane, output the vertices of the convex hull. Normally, polygons are presented in counterclockise order. For some reason our book outputs the hull in clockwise order, but obviously it is a trivial matter to convert from one to the other. The hull should consist only of *extreme points*, in the sense that if three points lie on an edge of the convex hull, then the middle point should not be output as part of the hull.

The book introduces 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. (It is not clear that this is necessarily a good thing to do. After all, if a point is inside the convex hull, then no update need be made. Perhaps a better algorithm should seek to add points in such a way that most of the points that are not on the hull are eliminated quickly from consideration. Nonetheless Graham's scan is easy to implement, and is optimal in the worst case, so it is not a bad algorithm by any means.)

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 are merged into a single cyclic list.

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 $\mathrm{Orient}(p, q, r) < 0$. When a new point p_i is added to the current hull, this may violate the right-hand turn invariant. So we check the last three points on the upper hull, including p_i. They fail to form a right-hand turn,

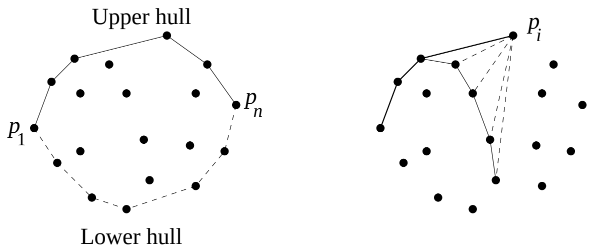


Figure 8: Convex hulls and Graham's scan.

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. For each point added, the amount of time spent is clearly $O(D_i + 1)$, where D_i is the number of points deleted in the process of adding point p_i. The reason is that each orientation test takes constant time, and we must perform one orientation test for each point deleted, perhaps along with one extra one (for the last point which is not deleted). Thus, the total running time is proportional to $\sum_{i=1}^{n}(D_i + 1) = n + \sum_{i=1}^{n} D_i$. How large is $\sum_i D_i$? Observe that 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 $O(2n) = O(n)$.

Lecture 3: More Convex Hulls

(Tuesday, Sep 9, 1997)
Reading: Chapter 1 in BKOS.

Degeneracies: Last time we presented the Graham's scan algorithm for computing convex hulls. Recall that the algorithm computes just the upper and lower hulls of a set of n points each in $O(n \log n)$ time. The method is an incremental algorithm that adds points to the hull in increasing order of x-coordinate.

One of the issues that we neglected to deal with was that of degenerate inputs. A degeneracy is somewhat hard to define formally, but a working definition is that, a *degeneracy* is any special configuration (of points, lines, etc) that will be broken (with very high probability) if the objects' defining coordinates are randomly perturbed. For example, three points form a degeneracy (in all dimensions strictly higher than 1) since any perturbation of any their coordinates will cause the orientation test to become nonzero.

Of course, we are only interested in degeneracies that affect the algorithm of interest. A functional definition of degeneracy is any configuration of points that causes a sign test based on real values (such as the orientation test) to return a zero value.

Let us begin by assuming that we perform all sign tests using exact arithmetic. There are two places in Graham's algorithm where degeneracies might arise. The first is that in sorting the points by their x-coordinates, there may be two points with the same x-coordinate. The second is when adding a point p_i and testing the last three consecutive points for the right-hand turn condition, the three points may be collinear.

For the case of equal x-coordinates, there is a very simple method for dealing with this, that is well worth remembering. We will imagine that an infinitesimal perturbation has been applied to the points. Such a perturbation should have the property of destroying degeneracies, but creating no new degeneracies. For example, to break ties in x-coordinates we could apply a small rotation of the plane.

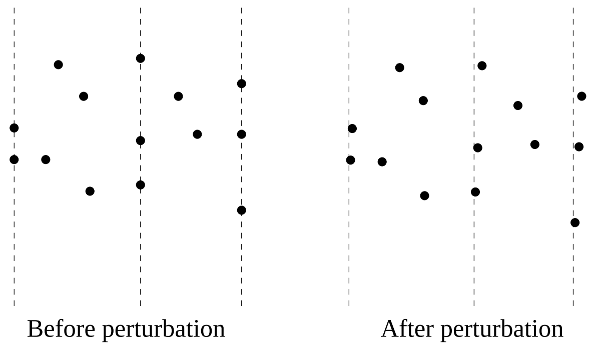


Figure 9: Perturbation to eliminate degeneracies.

If we rotate too much, we might create a new tie among x-coordinates. How small is small enough? How much numerical precision is needed? The trick is that we do not actually compute the perturbation numerically. Instead we perform the computation on the original exact inputs, but make all decisions *as if* this rotation had been applied. This technique, of simulating a perturbation is called *symbolic perturbation*.

For example, suppose that we rotate all the points infinitesimally clockwise about the origin of the coordinate system. If two points had the same x-coordinate prior to the rotation, then the one with the larger y-coordinate will have the larger x-coordinate after the rotation. Since the rotation is infinitesimal (infinitely small) then no new ties amongst x-coordinates will be created. To simulate the perturbation, we simply sort the points *lexicographically*, first by x-coordinate, and then amongst points with the same x-coordinate, by y-coordinate. They key is that the points have not actually been rotated numerically, but the algorithm behaves as if they had been.

We could have perturbed the points by a counterclockwise rotation as well, either would be fine. This perturbation also takes care of the question of where the upper hull ends and the lower hull begins if there are ties for the leftmost and rightmost points.

To deal with three collinear points, observe that since the points are sorted by x-coordinate as we visit them, the middle point of any such triple should not be considered as lying on the hull. So when implementing the right-hand turn condition, we should require that it is a "strict" right-hand turn otherwise the middle point is deleted.

If we were to use floating point arithmetic, the best we could hope for is an approximation to the convex hull, in the sense that if that the hull we output would be the correct hull for some small perturbation of the input. When dealing with floating point computation, the most important goal is that numerical errors should never cause the program to produce an output that is structurally unsound. For convex hulls this is not too complex. However, for other geometric problems it may be quite hard to devise an algorithm that insensitive to small numerical errors.

Quickhull: To show that there is not just one way to compute planar convex hulls, let us consider some other approaches. The first is called *quickhull*, and can be viewed as a 2-dimensional generalization of quicksort.

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 the expected number of points on the convex hull is $O(\log n)$. Here is a proof.

Theorem: Suppose that n points are uniformly distributed in a unit square. The expected number of points on the convex hull is $O(\log n)$.

Proof: First, we will break the hull into four parts, the upper-left, upper-right, lower-left, and lower-right hulls. This is done by breaking the hull at its leftmost, rightmost, topmost and bottommost points. We will show that each has $O(\log n)$ points. By symmetry, we may consider one, say the upper-right hull.

Rather than bounding the number of points on the hull, we will bound a larger quantity. A point $p_i = (x_i, y_i)$ is said to be *dominated* by another point p_j if $x_j \geq x_i$ and $y_j \geq y_i$. The *maxima* of a point set P are the points that are not dominated by any other point in P. Intuitively, the maxima form a staircase along the upper-right side of the point set. Observe that the set of maxima is a superset of the points on the upper-right hull, so an upper bound on the number of maxima is an upper bound on the number of points in the upper-right hull.

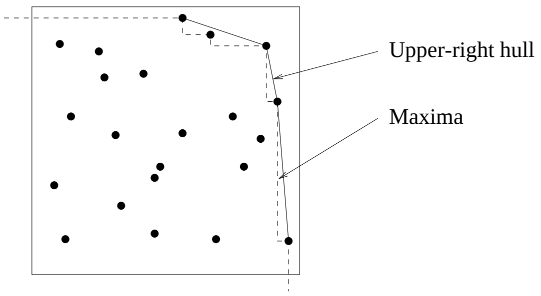


Figure 10: Upper-right hull and maxima.

Suppose that the points are sorted in decreasing order of x-coordinate. We will assume that all points have distinct coordinates to simplify things. Clearly p_i is a maximum if and only if p_i has the largest y-coordinate among the subset $\{p_1, p_2, \ldots, p_i\}$. What is the probability that this is true? The y-coordinates of these points are independent from each other. Therefore the probability that p_i has the largest y-coordinate is just $1/i$.

Thus, each point p_i is a maxima with probability $1/i$. The expected number of maxima is:

$$E_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n = O(\log n).$$

(The sum is the *harmonic series*, see Cormen, Leiserson, and Rivest, if you haven't seen this.) From the comments made earlier it follows that the expected number of points on all the hulls is at most 4 times this quantity, which 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.

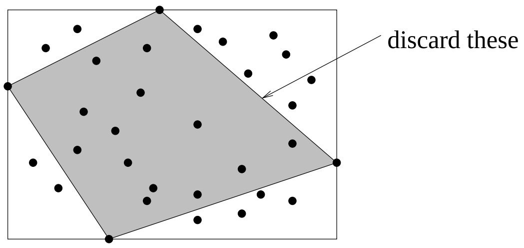


Figure 11: Quickhull's initial quadrilateral.

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.

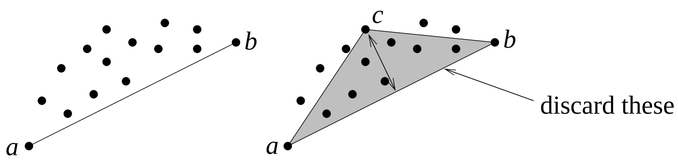


Figure 12: Quickhull elimination procedure.

How should c be selected? There are a number of possible selection criteria that one might think of. The suggested method is that c be the point that maximizes the perpendicular distance from the line ab. (Another possible choice might be the point that maximizes the angle cba or cab. It turns out that these are poor choices because they do not produce even splits 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. Then the running time is given by the recurrence:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= T(n_1) + T(n_2) + n, \end{aligned}$$

To solve the recurrence, it would be necessary to determine the expected values of (n_1, n_2), which would depend on the point distribution. 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).

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.

Other convex hull algorithms: There are still other convex hull algorithms. A natural question to ask is whether it is possible to improve on $O(n \log n)$? In fact, it is possible to show that any algorithm for convex hulls must at least sort the points along the hull. (We'll leave this as an exercise.) In the worst case, the hull has n points, and so this implies an $\Omega(n \log n)$ lower bound.

But if you know or expect fewer points on the hull, then you may be able to do much better. For example, if your inputs consist of points uniformly distributed in a square, then you expect the number of points on the hull to be only $O(\log n)$. Is it possible to find a better algorithm in this case? An algorithm that takes into consideration both the input and output size is called an *output sensitive algorithm*. Let n denote the number of input points and h denote the number of edges on the convex hull.

There is an algorithm called *Jarvis's march* which builds the hull in $O(nh)$ time by a process called "gift-wrapping". In $O(n)$ time it is possible to find the "next" edge on the hull, and this algorithm operates by walking around the hull one edge at a time. But this gives no improvement for the case of uniformly distributed points in the square. Nonetheless, many of the most practical higher dimensional convex hull algorithms are based on this approach.

There exists an $O(n \log h)$ time algorithm for the convex hull problem, and it can be shown that this algorithm is optimal with respect to output sensitive algorithms. The fist algorithm with this complexity was discovered back in the mid 1980's. But recently, in 1995, a much simpler algorithm was discovered.

Lecture 4: Line Segment Intersection

(Thursday, Sep 11, 1997)
Revised: Sept 17. (Fixed the analysis.)
Reading: Chapter 2 in BKOS.

Geometric intersections: One of the most basic problems in computational geometry is that of computing intersections. Intersection computation 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.
- Another example comes from computer graphics. The clipping problem involves determining what parts of a set of polygons are visible through a rectangular window. This involves determining the intersection of the polygons and the window, and then clipping the polygons at these points.

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.

We will let I denote the number of intersections. How should we count I in the degenerate configuration where three or more lines intersect in a single point? If k lines intersect in a single point, one might think of this as a single intersection point, or one might interpret this as $(\binom{k}{2}) = k(k-1)/2$ pairwise intersections. The former interpretation would lead to faster running times, and so would be the goal of a careful implementation. (And indeed the implementation in the book achieves this.) On the other hand, one might argue that such degeneracies are rare in practice, and so it is not worth worrying about. We will take the latter (sloppier) attitude, and refer to the text for a more careful implementation.

Complexity: 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.

The best 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.

Element uniqueness: Given a list of n numbers, does any number appear at least twice in the list.

(Note that element uniqueness can be solved in $O(n \log n)$ time by sorting and checking whether adjacent elements are equal.) Given this lower bound, even determining whether the same endpoint is used in two different line segments would require $\Omega(n \log n)$ time, and hence this is a lower bound for determining even the existence of a single intersection.

Note that this lower-bound result assumes the *algebraic decision tree model* of computation, in which all decisions are made by comparisons made based on exact algebraic operations $(+, -, *, /)$ applied to numeric inputs. Although this encompasses most of what one would consider to be "normal" geometric computations, there are alternative models of computation. For example, by taking mods, floors, or ceilings, you would be able to implement hashing. (This is a data access method taught in most data structures and algorithms courses. See Cormen, Leiserson, and Rivest, for example.) With hashing it is possible to solve the element uniqueness in expected $O(n)$ time. Unfortunately, even in this more powerful computational model, no one know how to determine whether any two line segments intersect in faster than $O(n \log n + I)$ time in the worst case.

Later in the semester we will discuss an optimal $O(n \log n + I)$ time algorithm for this problem.

Line segment intersection: In rather typical computational geometry fashion, our book does not discuss the issue of how to determine the intersection point of two line segments. Let $\overline{ab}$ and $\overline{cd}$ be two line segments, given by their endpoints. It is an easy exercise to determine *whether* these line segments intersect, simply by applying an appropriate combination of orientation tests.

To determine the coordinates of the intersect point involves solving a small system of equations. The most natural way to set up this computation is to introduce the notion of a *parametric representation* of the line segment. 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 introduct 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 such that $p(s) = p(t)$. 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 corresponds to the case where the line segments are parallel (and possibly 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 instances where the sweep line encounters an endpoints of a line segment (which are all known in advance) and when the sweep line encounters an intersection point of two line segments (which will be discovered as the algorithm executes).

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 intersect point, we must interchange the order of the intersecting line segments along the sweep line.

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 for now.

(1) No line segment is vertical. (Easily fixed through a symbolic perturbation.)

(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 points.

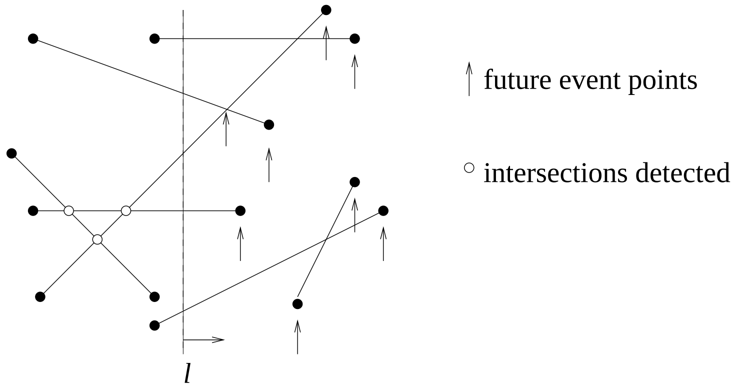


Figure 13: Plane sweep.

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.

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: Consider such a pair of segments, and assume that prior to this, all intersections have been correctly handled by the algorithm (thus the sweep line order prior to their intersection is correct). Consider the contents of the sweep line immediately prior to this event. We claim that at this time there could be no segment s_k between s_i and s_j along the sweep line. If there were, then either there would have to be endpoint of s_k prior to the intersection (implying this was not the event immediately prior to the intersection) or that s_k intersects either s_i or s_j (again implying the existence of an intermediate event).

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

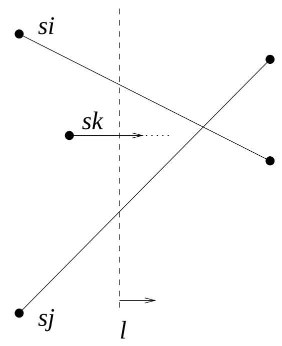


Figure 14: Correctness of plane sweep.

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 easily through a symbolic perturbation by sorting them lexicographically by (x, y). This has the same 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 decreasing 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 and plug it into the line equation for this line.

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 Algorithm: We can now present the complete plane-sweep algorithm.

(1) Initially, we insert all of the line segment endpoints 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.

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 5: DCEL's and Subdivision Intersection

(Tuesday, Sep 16, 1997)
Revised: Sep 18. (Augmented Figure 3 with alternative view.)
Reading: Chapter 2 in BKOS.

Topological Information: In most 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 subdivision* (or what might be called a *cell complex* in topology). A planar subdivision is defined by a graph with straight-line edges embedded in the plane 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.) This 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.

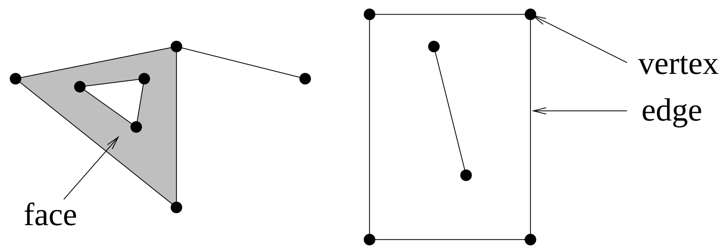


Figure 15: Planar straight-line subdivision.

Planar subdivisions will 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 edgs, F the number of faces in a connected planar graph. Euler's formula states that

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

If we allow the graph to be disconnected, and let C denote the number of connected components, then we have the 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 straight-line planar graph with n vertices has $O(n)$ edges and $O(n)$ faces.

Proof: Since the graph is a straight-line graph, there cannot be multiple edges between the same pair of vertices, and there cannot be self-loop edges. We will prove the theorem in the more general setting of planar graphs without these types of edges.

First, we will modify the graph in a way that it is still planar, but no more edges can be added. This modification will only increase the number of edges and faces. If the graph is disconnected, then add an edge between vertices in two different components, thus reducing the number of connected components by one and possibly increasing the number of edges and possibly the number faces each by one. Repeat until the graph is connected.

Next, if any face has more than three edges, then add an edge through this face, thus increasing the number of edges and faces each by one. (This cannot generally be done without adding curved lines, unless if there are more than three vertices on the convex hull of the vertex set. But since we are not requiring that this be a straight-line planar graph this is not an issue.)

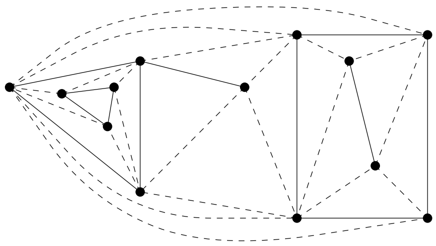


Figure 16: Modified planar graph.

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 involves 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, $3F' = 2E'$. Euler's formula states that $V + E' - F' = 2$. Using the fact that $E' = 3F'/2$ we have:

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

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

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

Thus, the number of faces is at most $2(V-2)$ and the number of edges is at most $3(V-2)$.

There are a number of reasonable representations that are used in practice. 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 discuss a simple and relatively elegant data structure called a *doubly-connected edge list* (or *DCEL*).

Doubly-connected Edge List: Like most representations for planar graphs, the DCEL is an edge-based representation, but vertex and face information is also included for whatever geometric application is using the data structure. There are three sets of records (which may be represented either as arrays or linked lists, as you prefer): a set of *vertex records*, a set of *edge records*, and a set of *face records*. Each undirected edge is represented by two directed edges (which our book calls *half edges*). Here is what each record contains for the data structure.

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 edge* 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. Our text does not make this assumption, and so presents a somewhat more general data structure. How to add these dummy edges is left as an exercise. 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 can access the destination as the origin of the twin edge). 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 *incident 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.)

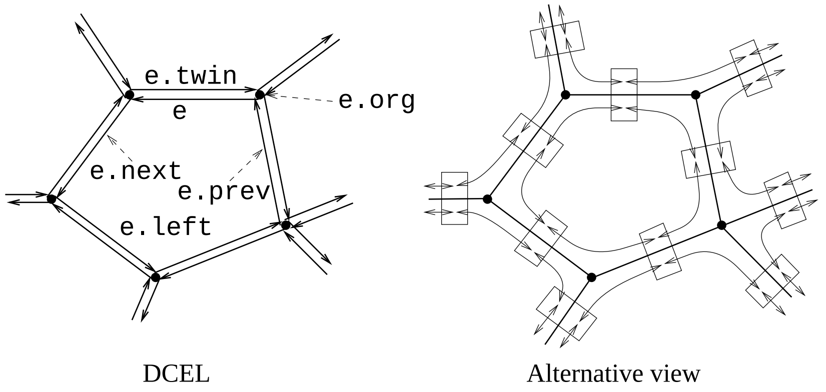


Figure 17: Doubly-connected edge list.

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. Furthermore, if we will not be needing the original subdivisions, it is possible to do this by modifying the existing subdivisions. (If not, then the process begins by making a copy of each.) The first part of the problem 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 segments come from the two different subdivisions. The process involves the following steps (and is illustrated in the figure).

(1) Create a new vertex v at the intersection point.

(2) Let a_1 and b_1 denote the edges that intersect, oriented from left to right across the sweep line. We split each of the two intersecting edges, by adding a vertex at the intersection point. Let a_2 and b_2 be the new edge pieces. They are created by the calls `Split(a1, a2)` and `Split(b1, b2)`, where the procedure is given below. This procedure creates the new edge, links it into place, and then returns the newly created edge. 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. (We will not give the details. Also note that the destination of a_1, that is the origin of a_1's twin must be updated, which we have omitted.)
This procedure creates both the new edge and its twin. The procedure `a1.dest` returns the destination of a_1, which is equivalent to `a1.twin.org`.

```
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;
}
```

(3) Link the four edges together.

```
Splice(edge &a1, edge &a2, edge &b1, edge &b2) {
    a1t = a1.twin;   a2t = a2.twin;      // find 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;
    }
```

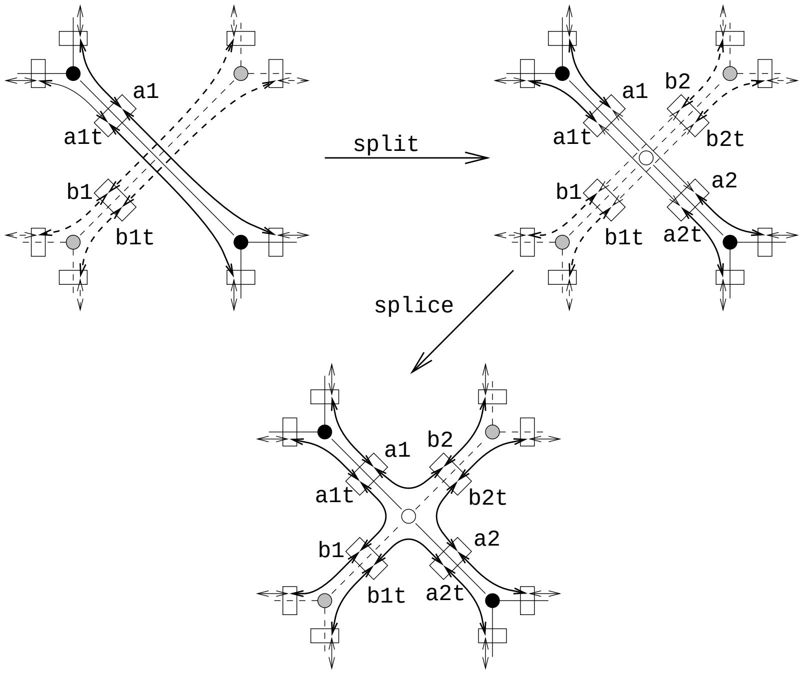


Figure 18: Updating the DCEL.

Lecture 6: Polygon Triangulation

(Thursday, Sep 18, 1997)
Revised: Sep 22. (Moved last section to Lecture 7.)
Reading: Chapter 3 in BKOS.

Simple Polygons: Today 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.

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.

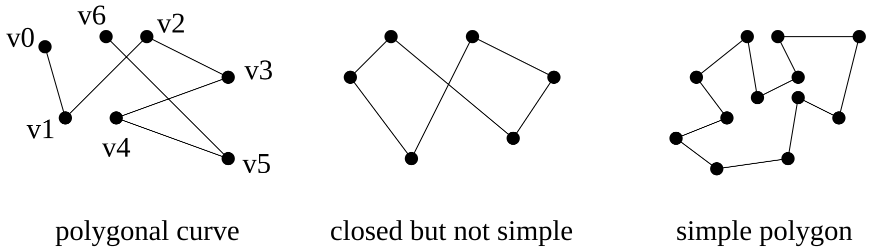


Figure 19: Polygonal curves

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.

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.

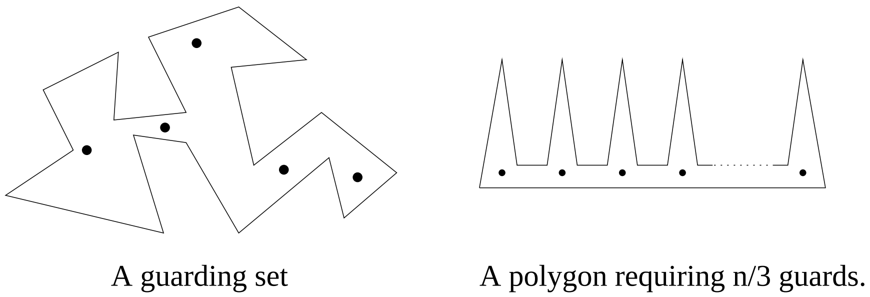


Figure 20: Guarding sets.

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 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'll refer you to the text for a proof. 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. For example, it fails to hold 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.

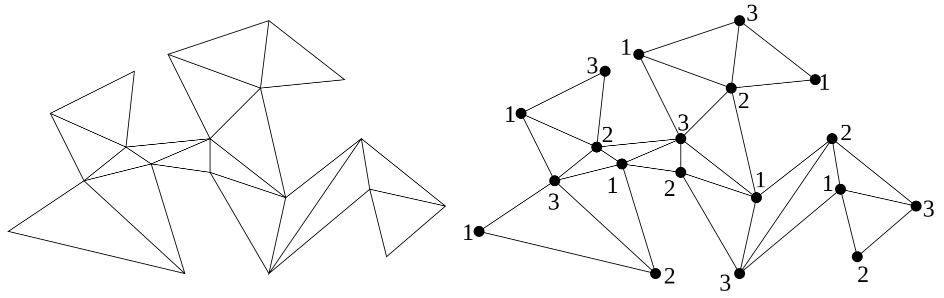


Figure 21: Polygon triangulation and a 3-coloring.

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 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.

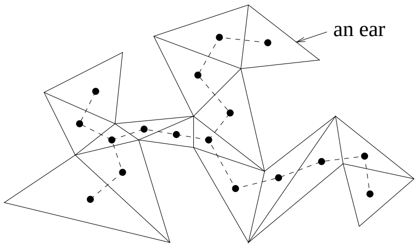


Figure 22: Dual graph of triangulation.

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: More Polygon Triangulation

(Tuesday, Sep 23, 1997)
Reading: Chapter 3 in BKOS.

The Polygon Triangulation Problem: The art-gallery exercise was intended as motivation for the problem of triangulating a simple polygon. This operation is 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.

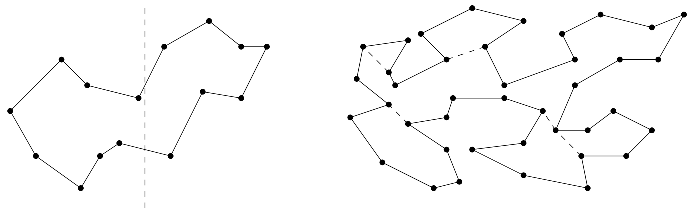


Figure 23: Monotonicity.

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 monotne. 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 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 the current vertex by adding diagonals, and then remove the triangulated region from further consideration.

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.

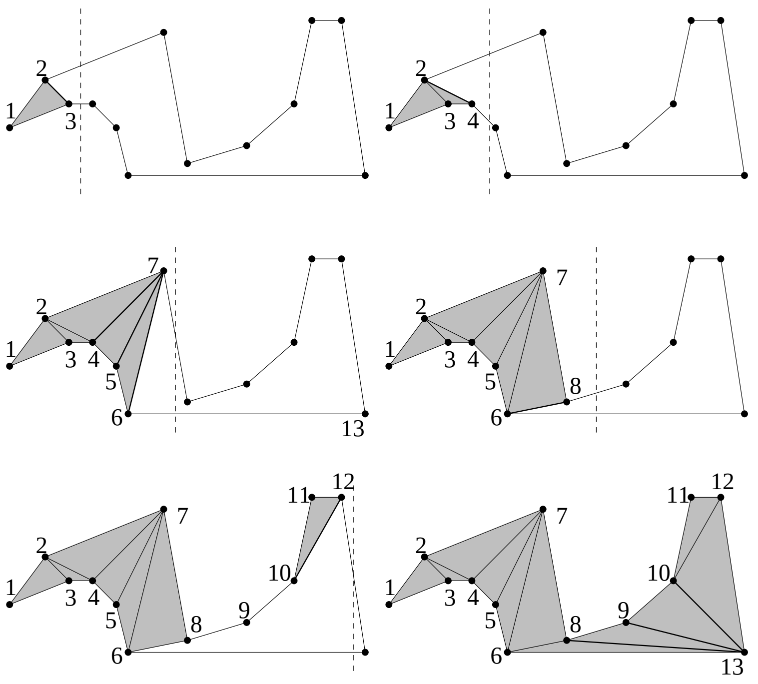


Figure 24: Triangulating a monotone polygon.

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). Now $u = v_{i-1}$, and the reflex chain consists of the single edge v_iv_{i-1}.

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 (which may mean that we add no diagonals).

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

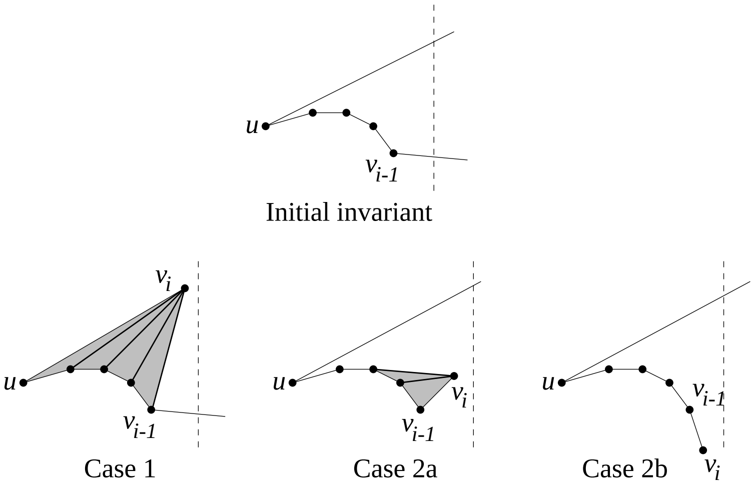


Figure 25: Triangulation cases.

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. Note that when we are done (analogous to Graham's scan) the remaining chain from v_i to u is a reflex chain.

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 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 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 the books 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 split vertices first (both edges lie to the right of the vertex). When a split vertex is encountered in the sweep, there will be an edge e_j of the polygon lying above and an edge e_k 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. But this would imply that there is a closer vertex lying between e_j and e_k. We will attach the split vertex to the closest vertex to the left of the sweep line which lies between e_j and e_k. Call this vertex the $helper(e_j)$. (We could have just as easily associated the helper with e_k, it doesn't really

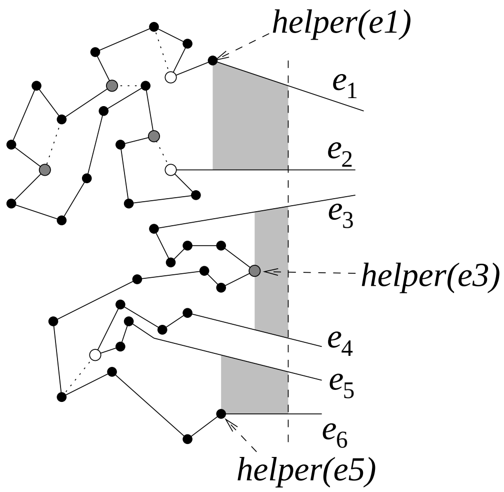


Figure 26: Split vertices, merge vertices, and helpers.

matter.) If there is no vertex between these edges, then $helper(e_j)$ is defined to be the left endpoint of e_j or e_k that lies closer to the sweep line. See the figure.

Note that $helper(e_j)$ 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 when the sweep line intersects e_j.

One way to visualize $helper(e_j)$ is to imagine a trapezoid with vertical sides and bounded above and below by e_j and e_k sweeping to the left of the current sweep line. The first vertex this sweeping trapezoid hits is the helper. These trapezoids are illustrated in the figure above.

Here are basic elements of the plane sweep algorithm to fix the split vertices. (We consider merge vertices later.)

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 6 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 and its 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.

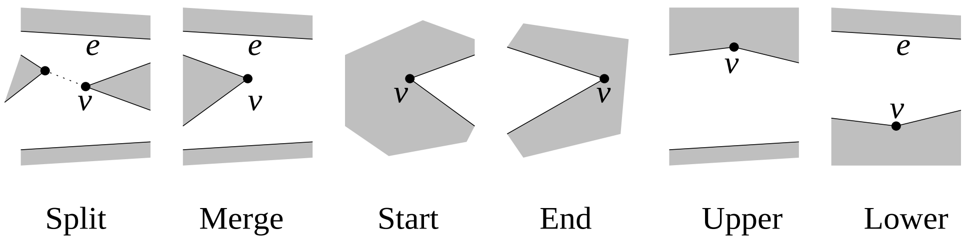


Figure 27: Plane sweep cases.

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: Intersection of Halfplanes

(Thursday, Sep 25, 1997)
Revised: Sept 30, fixed the "clockwise" bug in the last lemma.
Reading: Chapter 4 in BKOS, with some elements from Section 8.2 and Section 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? We may assume each halfplane is expressed by an inequality of the form

$$ax + by \leq c.$$

and the halfplane is represented by the three coefficients (a, b, c). The line bounding the halfplane is $ax + by = c$. Note that if we multiply a, b, and c by the same nonzero scalar value, then the line equation does not change. Thus, we can think of this triple as a form of homogeneous coordinates for lines. For example, if $c \neq 0$, then we can divide the equation through by c, yielding an equation of the form $(a/c)x + (b/c)y = 1$, which can be expressed by the homogeneous coordinates $(a/c, b/c, 1)$.

If the scalar multiple is negative, then the sense of the inequality will be reversed. Thus, we do not need a separate inequality of the form $ax + by \geq c$, since we can just negate all three coefficients to get the same effect.

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.

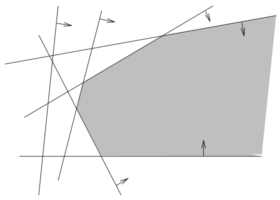


Figure 28: Halfplane intersection.

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 books 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 (ala DCEL's) 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 Cormen, Leiserson, and Rivest, for a proof.)

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.

However, there is a very simple plane-sweep approach to solving this problem. Suppose that we perform a left-to-right plane sweep to compute the intersection. Observe that by convexity the sweep line intersects each C_i in at most two points. Therefore, there are at most four points in the sweep line status at any time. Thus we do not need a fancy dictionary for storing the sweep line status. All the operations can be performed in constant time. Furthermore, you do not need a priority queue for the events. The only two events that are important are the leftmost points of C_1 and C_2 (note that they might stretch back to $-\infty$). To determine the next event point, you just need to check the four current edges for intersections, and check the four edges for their next endpoints. Since there are only a constant number of possibilities, each event can be handled in $O(1)$ time.

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 convex called *point-line duality*. In a nutshell there is a remarkable similarity

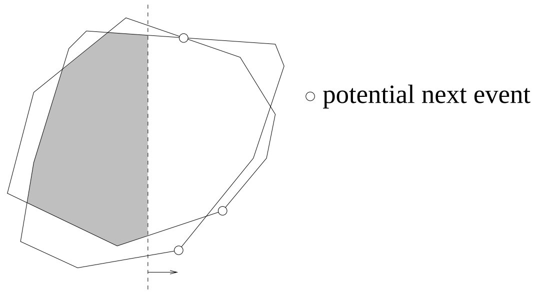


Figure 29: Convex polygon intersection.

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.)

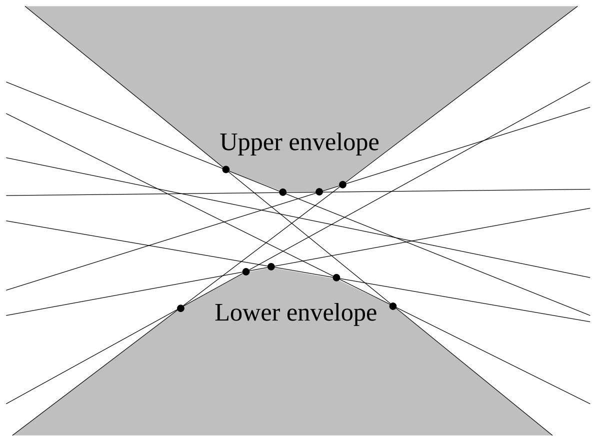


Figure 30: Lower and upper envelopes.

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:

$$\begin{aligned} \mathcal{L} &= \{(a, b) \mid p_y = ap_x - b\} \\ &= \{(a, b) \mid b = p_x a - p_y\}. \end{aligned}$$

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 $\ell : (y = ax - b)$ in the primal plane we define ℓ^* and p^* to be a point and line respectively in the dual plane defined by:

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

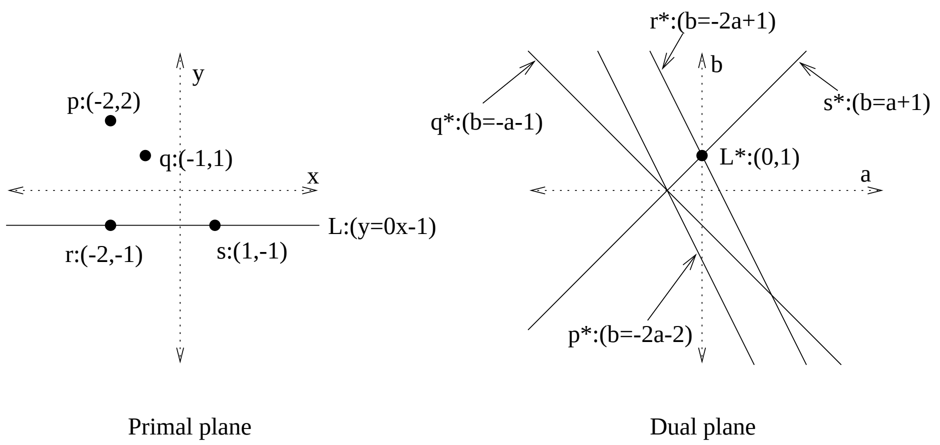


Figure 31: Dual transformation.

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.

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 $p_i p_j$ 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 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

(Tuesday, Sep 30, 1997)
Reading: Chapter 4 in BKOS.

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 the point that that is extreme in some given direction.

One particularly important application is that of linear programming. In linear programming (LP) we are given a set of linear inequalities, or *contraints*, 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:

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

where $a_{i,j}$, c_i, and b_i are given real numbers.

From a geometric perspective, the feasible region is the intersection of halfspaces, and hence is a convex polyhedron. We can think of the objective function as a vector c, and the problem is to find the point of the feasible region that is furthest in the direction c, called the *optimal vertex*. In many of our examples, we will imagine that the vector c is pointing down, and hence the problem is just that of finding the lowest point of the feasible region.

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

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.

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 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). This is exactly analogous to a applying a symbolic perturbation by rotating space.

2-dimensional LP: 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 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 open until a little over 10 years ago, when Karamarkar showed a polynomial time algorithm based on moving through the interior of the feasible region. (It should be mentioned that the method is polynomial in the number of number of constraints, the dimension, and the number of bits of precision in the numbers. Although this is good enough for practice, one of the major open problems left in the area is whether there is a *strongly polynomial time algorithm*, that is polynomial without the assumption that the numbers are of bounded precision.)

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. We will show that 2-dimensional LP problems can be solved in $O(n)$ time, and in fact more generally, any fixed dimensional linear programming can be solved in $O(n)$ time. The problem with using such an algorithm in practice is that the constant factors grow very rapidly with the dimension (faster than polynomial time). We will discuss one simple one later whose running time is $O(d!n)$. Thus these algorithms are only acceptable for relative small dimensional problems.

Incremental Construction: The 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:

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

and a nonzero objective function given by vector $c = (c_x, c_y)$. The problem is to find $p = (p_x, p_y)$ that is feasible and maximizes the dot product $c_x p_x + c_y p_y$. In our illustrations, we will assume that c is pointing downwards, e.g. $c = (0, -1)$. (This is our book's convention. It is a bit confusing, because note that 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. 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.

We will then add halfplanes one by one, $h_3, h_4, \ldots$, and with each addition we will update the current optimum vertex. After the addition of $\{h_1, h_2, \ldots, h_i\}$, let C_i denote the current feasible region (their common intersection) and let v_i denote the current feasible vertex. Our job is to update v_i with each new addition. (Note that we will not compute the C_i's explicitly, since doing so would require $O(n \log n)$ time, but it is handy to think about their existence for the purposes of proving things). Notice that with each new constraint, the feasible region 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. The first is that v_{i-1} is already satisfies constraint h_i (that is, it lies within this halfplane). If so, then it is easy to see that the optimum vertex does not change, that is $v_i = v_{i-1}$. On the other hand, if v_{i-1} violates constraint h_i, then we need to find a new optimum vertex. Where shall we look for it?

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. Intuitively, if the new optimum vertex did not lie on ℓ_i, then 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 a linearity are both very important for the proof.

So this leaves the question: 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.

Lecture 10: More Linear Programming

(Thursday, Oct 2, 1997)
Reading: Chapter 4 in BKOS.

Recap: Last time we presented a simple incremental algorithm for linear programming in the plane. We are given a set of n halfspaces, $h_1, h_2, \ldots, h_n$, and an objective function, described by a vector c. The goal is to find the point p that has the maximum dot product with c (that is, is furthest in direction c) subject to the constraint that it lies in the intersection of the halfspaces (the convex feasible region).

Let us recall the algorithm last time, from the perspective of linear programming in d-dimensional space. We are given a set of halfspaces and we want to find the point that lies in the intersection of the halfspaces and is most extreme in some direction. We start by finding an initial set of d halfspaces that define an initial bounded solution. We renumber the halfspaces so that these are the first d halfspaces. Let v_d be the initial optimum vertex. Then we add halfspaces one at a time. On adding the h_i, we test whether v_{i-1} is feasible with respect to h_i. If so, we set $v_i = v_{i-1}$. Otherwise, we claim that the new optimum must lie on the hyperplane that bounds h_i. We intersect all the previous $i-1$ halfspaces with this hyperplane (thus reducing the dimension of the problem by 1), project the objective function vector onto this hyperplane, and then recursively solve the resulting $d-1$ dimensional LP problem. The recursion bottoms out when we get down to a 1-dimensional LP problem (which is just the problem of intersecting a collection of intervals). Thus we solve a d-dimensional problem by repeatedly reducing it to a $d-1$ dimensional problem, until we get down to a 1-dimensional problem, which is trivial to solve.

Analysis: What is the worst-case running time of this algorithm in the planar case? 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:

$$\sum_{i=3}^{n}(i-1) \le \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2),$$

from standard results on summations (see Cormen, et al).

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. This is called a *randomized incremental algorithm*. For example, just prior to calling the algorithm given last time, we call a procedure for randomly permuting the initial input list. (Our text gives an example.)

We analyze the running time in the expected case where we average over all $n!$ possible permutations. It is important to observe that the analysis makes no assumptions about the nature of the input, only on the random permutation used. (In other words there are no bad inputs, only bad random permutations.) The fact that the first two halfplanes were chosen specially adds a confusing element to this. To be correct, we should think of them as being fixed, and we randomize over the $(n-2)!$ permutations of the remaining halfplanes. However, we'll try to ignore this complicating factor below.

To analyze the algorithm we use an interesting technique called *backward analysis*. Before discussing backward analysis, think of how a typical probabilistic analysis (e.g. the one used for quicksort) works. At each stage the algorithm has a set of random choices that might be made. (For example, in quicksort the choice is which element to select as the pivot.) Each choice has a certain probability of occuring. (Typically all choices are equally likely.) We analyze the effect of each choice on the running time, and then multiply each times it probability of occuring, and sum everything up to get the expected running time.

The difficulty of applying this to a problem like the linear programming problem is that at the next stage it is very difficult to predict what is going to happen, since it depends heavily on what has gone on up to this point. In a backwards analysis, rather than asking what is the effect of the *next* random choice on the running time, we ask what was the effect of the *previous* random choice. This may sound just like like a semantical distinction, but the remarkable fact is that it really makes things much easier to analyze. The reason is that we have full knowledge of the past.

Consider the example shown below. We have $i = 7$ random halfplanes that have been added, so far and $v_i = v$ is the current optimum vertex (see the figure). Rather than consider which halfplane will be added next, let's consider which one was added last. For example, if h_5 was the last to be added, then imagine the picture without h_5. Prior to this v would have been the optimum vertex. Therefore, if h_5 was added last, then it would have been added with $O(1)$ cost. This applies to all the other halfplanes as well except those that define v, namely h_4 and h_8. For example, if h_8 was added last, then prior to its addition, we would have had a different optimum vertex, namely v'. Thus if h_4 or h_8 were added last, then it would incur the higher running time of $O(i-1)$ to solve the 1-dimensional LP problem.

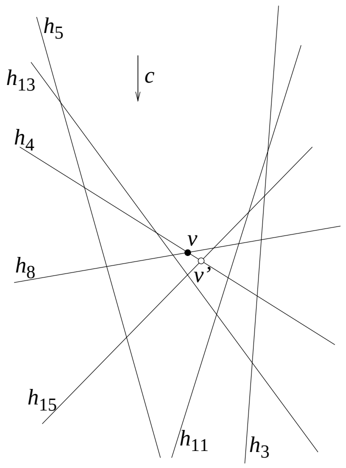


Figure 32: Backward analysis.

What is the probability of each of these events occuring? The important observation here is that the order in which halfplanes are inserted in completely independent of the geometric structure. Among the halfplanes in the diagram so far $\{h_3, h_4, h_5, h_8, h_{11}, h_{13}, h_{15}\}$, there are an equal number of permutations of these ending with h_8 as there are ending with h_3 as there are with any of them (6! in particular.) Here we are ignoring the two initial halfplanes, which are fixed. Imagine that they lie outside of the portion shown in the figure. Thus each halfplane has an equal probability of $1/i$ of being the last halfplane to be added. Among the i halfplanes present, two of them (the two that bound the current optimum vertex) would incur a cost of $O(i-1)$ on the running time, and the other $i-2$ would incur a cost of $O(1)$. Thus, there is a $2/i$ chance of incuring time $O(i-1)$ and a $(i-2)/i$ chance of incuring time $O(1)$. Thus the expected time for the i-th stage is:

$$\frac{2}{i}O(i-1) + \frac{i-2}{i}O(1) \leq O(1).$$

If we sum this up over all n stages, we just get $O(n)$ as the total expected running time (by the linearity of expectation).

This may all seem like a bit of probabilistic "magic". If you find it unbelievable, you are encouraged to draw a set of halfplanes and experiment with different orders to convince yourself (at least at an intuitive level) of the soundness of this technique. We will be applying it on other randomized algorithms this semester.

Unbounded LP's: We began by assuming that we could always find two (or generally d) initial halfplanes to provide us with an initial bounded optimal vertex. We never dealt with this issue, but we claim that it is always possible to determine either that the LP is unbounded, or to determine a pair of halfplanes that bound the feasible region in $O(n)$ time. Our book presents an algorithm for doing this in the plane. This algorithm has the nice feature that it generalizes to higher dimensions (although they leave the generalization as an exercise).

In the planar case there is a much simpler algorithm than this. Consider the outward pointing normal vectors for the various halfplanes (imagine they are scaled to unit length). They partition the unit circle into n angular sectors. Now consider the angular sector that contains the objective vector c. Consider the two halfplanes lying immediates clockwise and counterclockwise from c. If the angle between these two normals is less than 180 degrees, then the no matter how far apart they are, these two halfplanes will converge at a point p and the feasible set (if it is nonempty) will lie in the cone bounded by these halfplanes.

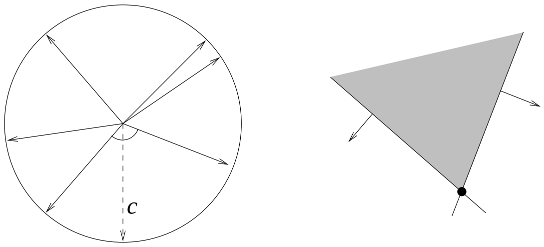


Figure 33: Boundedness test.

Intersecting with a hyperplane: Our book does not talk about how we go about intersecting given set of halfspaces with a hyperplane. (Suffice it to say that it is a geometric primitive that can be evaluated in $O(d)$ time per hyperplane.) For completeness let's consider how this might be done.

Suppose that the halfspace that we just added was given by the inequality:

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

The corresponding hyperplane is given by the equation:

$$\ell_i : a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,d}x_d = b_i.$$

We can express this more succinctly in matrix notation. Let A_i denote the $1 \times d$ vector consisting of the i-th row of the A matrix, $A_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,d})$. The inequality may be written $A_i x = b_i$, where x is a $d \times 1$ vector and b_i is a scalar.

We want to intersect the other halfspaces with this hyperplane. Furthermore, we would like to represent the result as a LP in $d-1$ dimensional problem. (Observe that after intersection the hyperplane still resides in d space.)

The idea is to apply one step of Gauss elimination using the equation of ℓ_i to eliminate a variable from all the other inequalities. Let us assume that $a_{i,1} \neq 0$ (if not, select a dimension k such that $a_{i,k} \neq 0$ and swap with the first). Consider an arbitrary constraint h_j that we wish to intersect with ℓ_i:

$$h_j : A_j x \leq b_j.$$

To eliminate the first dimension from h_j we multiply A_i by $(a_{j,1}/a_{i,1})$ and subtract from A_j, do the same for b_i and b_j:

$$\begin{aligned} A'_j &= A_j - \left(\frac{a_{j,1}}{a_{i,1}}\right) A_i \\ b'_j &= b_j - \left(\frac{a_{j,1}}{a_{i,1}}\right) b_i. \end{aligned}$$

To see that this works, suppose that x is a point on the hyperplane ℓ_i, implying that $A_i x = b_i$. Suppose that x satisfied constraint h_j. Then we have

$$\begin{aligned} A'_j x &= \left(A_j - \frac{a_{j,1}}{a_{i,1}} A_i\right) x \\ &= A_j x - \frac{a_{j,1}}{a_{i,1}} A_i x \\ &\leq b_j - \frac{a_{j,1}}{a_{i,1}} b_i \;=\; b'_j. \end{aligned}$$

Thus, every point on the hyperplane satisfies the modified constraint if and only if it satisfies the original constraint. A similar elimination can be performed to the objective vector c. It is also easy to show that (as is standard in Guass elimination) the first term of each equation vanishes, so we are left with a $d-1$ dimensional problem. Reversing the process allows us to project the $d-1$ dimensional solution back into d-space.

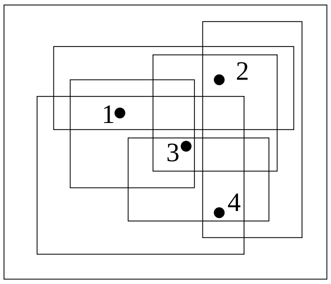
Lecture 11: Orthogonal Range Searching

(Tuesday, Oct 7, 1997)
Chapter 5 in BKOS.

Range Queries: We shift our focus from algorithm problems to data structures for the next few lectures. In general, we consider the question, 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.

Range queries are queries of the general form: given a set P of points, list (or count or compute some commutative function of) the subset of P lying within a given region. Regions may be rectangles, triangles, halfspaces, circles, etc. There are many data structures for processing range queries, depending on the type of region.

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. Given a particular class of ranges, a *range space* can be described abstractly as 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.



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 34: Rectangular range space.

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 thing about rectangular ranges is that they can be decomposed into a collection 1-dimensional searches.

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*. We are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ on the line, and given an interval $[x_{lo}, x_{hi}]$, report all the points lying within the interval. Our goal is to find a data structure that can answer these queries in $O(\log n + k)$ 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. However, this will not generalize to higher dimensions.

A basic approach to 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), such that the answer to any query can be expressed as a disjoint union of a small number of canonical subsets. Note that these subsets may overlap each other. The trick to solving a range searching problem is to identify a collection of canonical subsets having these properties, and then finding the proper subsets for a given range.

Suppose that the points of P are sorted in increasing order and then stored in the leaves of a balanced binary tree. We can associate each node of this tree with the (canonical) subset of point stored in the leaves that are descendents of this node. This gives rise to $O(n)$ canonical subsets. Further, we claim that the canonical subsets corresponding to any range can be computed in $O(\log n)$ time.

Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the leaf whose key is greater than or equal to x_{lo} and the leaf whose key is less than or equal to x_{hi}. Then we take all of the *maximal subtrees* lying between these two search paths. In particular, the two search paths will generally travel along some common subpath until reaching a node v_{split} where the two paths split. After this each search path goes its own way. When the left path to x_{lo} travels along a left edge of the tree, then the right subtree lies entirely within the interval. Similarly, when the right path to x_{hi} travels along a right edge, then the left subtree lies entirely within the interval. The leaves of these maximal subtrees are the desired canonical subsets whose disjoint union is the answer to the range query. To answer a range reporting query, we simply traverse these subtrees, reporting the points of their leaves. To answer a range counting query we store the total number of points in each subtree and sum all of these counts.

Since the search paths are of length at most $\log n$, it follows that there are $O(\log n)$ total canonical subsets. Thus the range counting query 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.

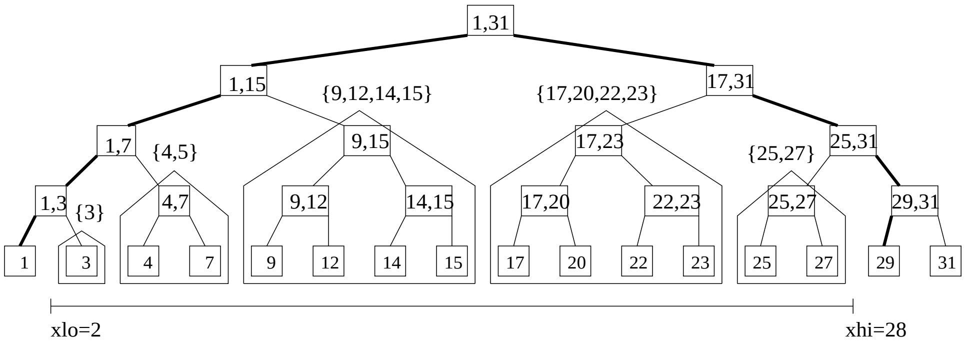


Figure 35: Canonical sets for interval queries.

Thus, 1-dimensional range queries can be answered in $O(\log n)$ time, using $O(n)$ storage. 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.

The idea behind a kd-tree is to extend the notion of a one dimension tree, but alternate is using the x- or y-coordinates to split on. In general dimension, the kd-tree cycles among the various possible splitting dimensions.

Each internal node of the kd-tree is associated with two quantities, a splitting dimension (either x or y), and a splitting value s. It has two subtrees. Since the splitting dimension alternates between x and y, some implementations do not store this explicitly, but keep track of it while traversing the tree. If the splitting dimension is x, then all points whose x-coordinates are less than or equal to s are stored in the left subtree and points whose x-coordinates are greater than or equal to s are stored in the right subtree. (If a point's coordinate is equal to s, then we reserve the right to store it on either side. This is done to allow us to balance the number of points in the left and right subtrees.) When a single point (or more generally a small constant number of points) remains, we store it in a leaf.

How is the splitting value chosen? To guarantee that the tree is balanced, the most common method is to let the splitting value be the median splitting coordinate. The resulting tree will have $O(\log n)$ height.

It is possible to build a kd-tree in $O(n \log n)$ time by a simple recursive procedure. The most costly step of the process is determining the median coordinate. However, if we presort the points into two cross-referenced lists, one sorted by x and the other sorted by y, then it is an easy matter to find the median at each step. The two lists can then be split in $O(n)$ time each, where n is the number of remaining points. This leads to a recurrence of the form $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$.

Searching the kd-tree: To answer an orthogonal range query we proceed as follows. Let R denote the query rectangle. Assume that the current splitting line is vertical. (The horizontal case is similar.) Let v denote the current node in the tree. Observe that each node of the tree is naturally associated with a rectangular region of space. Call this $reg(v)$. The search proceeds as follows. If v is a leaf, then we check the point(s) stored in v as to whether it lies in R. If so we report/count them. If v is an internal node, then we first consider the left subtree. If its region lies entirely within R then we call a different routine to enumerate all the points in this subtree (or for a counting query we return a precomputed count of the number of points in this subtree). If the left child's region partially overlaps R then we search it recursively. (If the left child's region does not overlap R at all, then we ignore it.) We do the same for the right child of v.

How many nodes does this method visit altogether? We claim that the total number is $O(\sqrt{n}+k)$, where k is the number of points reported.

Theorem: The times needed to answer an orthogonal range query using kd-tree with n points in $O(\sqrt{n}+k)$, where k is the number of reported points.

Proof: First observe that for each subtree with m points, we can enumerate the points of this subtree in $O(m)$ time, by a simple traversal. Thus, it suffices to bound the number of nodes visited in the search (not counting subtree enumeration).

To count the number of nodes visited, first observe that if we visit a node, then its associated region must intersect one of the four sides of the rectangle range. (Otherwise it lies entirely inside or outside, and so is not visited.) We will do this separately for each of the four sides, and multiply the result by four.

Rather than consider the line segment defining a side of the range rectangle, we consider the number of nodes visited if we had applied an orthogonal range query to the entire line on which this side lies. This will only overestimate the number of nodes visited.

Because the kd-tree processes even and odd levels differently, it will be important to analyze two levels at a time (or generally d levels at a time for dimension d). Let $Q(n)$ denote the query time for visiting a node v that has n points descended from it. Consider any orthogonal line that intersects the region associated with v. The key observation is that any horizontal or vertical line can intersect at most two of the four subregions associated with the grandchildren of v. (The other two will either be entirely contained within the halfspace or lie entirely outside the halfspace. In either case the algorithm will not make a recursive call on them.) Since each child has half as many points (because we split at the median), the number of points in each grandchild is roughly $n/4$. Thus we make at most two recursive calls on subtrees of size $n/4$. This gives the following recurrence:

$$Q(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2Q(n/4) + 1 & \text{otherwise.} \end{cases}$$

It is an easy process to expand this recursion, and derive the fact that it solves to $O(\sqrt{n})$. Including the factor of four, we still have an $O(\sqrt{n})$ bound on the total number of nodes visited, and adding the enumeration time the total time is $O(\sqrt{n}+k)$ as desired.

Lecture 12: More Orthogonal Range Searching

(Thursday, Oct 9, 1997)
Read: Chapter 5 in BKOS.

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.

The data structure is based on the concept of *leveling* a complex multi-dimensional search into a constant number of simpler range searches. 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 range query in the plane can be stated as two range queries: Find all the points whose x-coordinates are in the range $[x_{lo}, x_{hi}]$ and all the points whose y-coordinates are in the range $[y_{lo}, y_{hi}]$. Let us consider how to do this for 2-dimensional range queries, and then consider how to generalize the process. First, build a range tree for the first range query, which in this case is just a 1-dimensional range tree for the x-range. 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. These are the points lying in the subtree rooted at this node. 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 in the search. This constitutes the first level of the search tree. For the second level, for each node v in this x-range tree, we build an *auxiliary tree*, each of which is a y-coordinate range tree, which contains all the points in the canonical subset associated with v.

Thus the data structure consists of a x-range tree, such that each node 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.)

To answer a query, we determine the canonical sets that satisfy the first query (there will be $O(\log n)$ of them). Each of these sets is just represented as a node in the range tree. We know that that these sets are disjoint, and that every point in these sets lies within the range of x-coordinates. Thus to answer the query, we just need to find out which points from each canonical subset lies within the range of y-coordinates. To do this, for each canonical subset, we access the auxiliary tree for this node, and perform a 1-dimensional range search on the y-range. This process is illustrated in the following figure.

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 $O(\log n)$ 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.

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

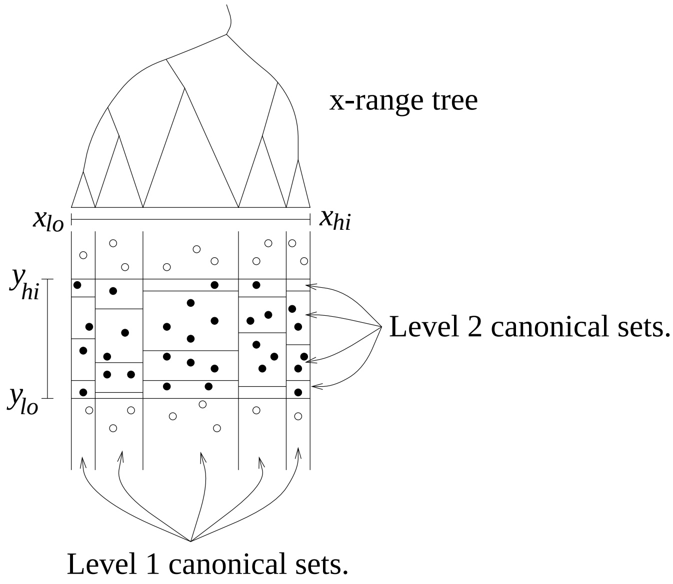


Figure 36: Range tree search.

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.

Summarizing, here is the basic idea to this (and many other query problems based on leveled searches). Let (S, R_1) denote the range space, consisting of points S and range sets R_1. Construct a data structure, which represents S by a collection of canonical sets, $\{S_1, S_2, \ldots, S_m\}$. For each canonical subset, construct a data structure for answering the second type of range query (and so on). The main property of the canonical subsets is that, for any range query, we can efficiently determine a small number of canonical sets whose disjoint union is equal to the answer to the query (in our case this was $O(\log n)$ subsets). Furthermore, this collection of canonical subsets can be determined efficiently (in our case in $O(\log n)$ time). To answer a range query, we solve the first range query, resulting in a collection of canonical subsets whose union is the answer to this query. We then invoke the second range query problem on each of these 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: $\langle p_5, p_2, p_3, p_4, p_1, p_6 \rangle$, and suppose that in v_L we store the points $\langle p_2, p_3, p_1 \rangle$ and in v_R we store $\langle p_5, p_4, p_6 \rangle$. This is shown 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.

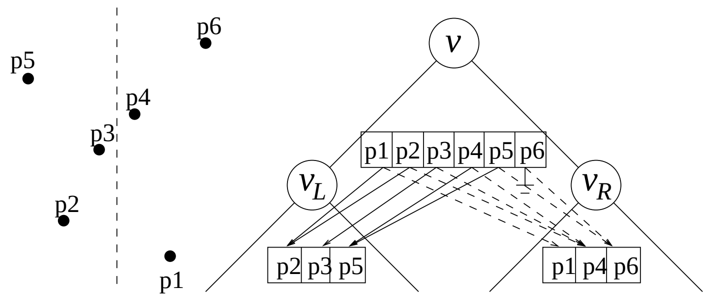


Figure 37: Cascaded search in range trees.

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: Planar Point Location

(Tuesday, Oct 14, 1997)
Today's material is not covered in our text.

Point Location: Today we consider is the *point location problem.* The problem (in 2-space) is: given a polygonal subdivision of the plane 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 book discusses how to handle degeneracies. As usual this is done by choosing a careful way to break ties.

It is remarkable that although this seems like such a simple and natural problem, it took quite a long time to discover an query time/space optimal solution. 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,

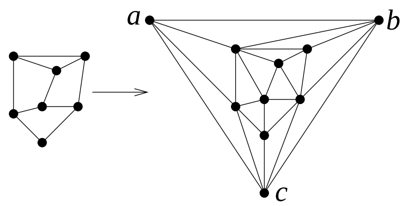


Figure 38: Triangulation of a planar subdivision.

many straight-line subdivisions, may already have convex faces and these can be triangulated easily in $O(n)$ time.

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 stupid triangulation algorithm, since the running time will be $O(1)$ anyway.)

The big question is whether such a subset exists. That is, given any triangulation can we always find a independent subset of vertices of bounded degree whose size is at least a constant fraction of the size of the whole vertex set? 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.

Theorem: 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.

Wow, 18? 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 "simple" solution is known to a problem it is often not long before a "practical" solution follows.

Before proving this theorem, we first complete the description of the point location algorithm. We start with T_0, and repeatedly select an independent set of vertices of degree at most 8. (However, never include the three vertices a, b, and c forming the outer face in such an independent set.) We delete these vertices from the graph, and retriangulate the resulting holes. Observe that each triangle in the new triangulation can overlap at most 8 triangles in the old 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 3 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 $4.5 \lg n$.

Point location algorithm: Now, to perform point location, we create a tree. The root of the tree is 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).

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.

The tree 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).

Construction and Analysis: 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.

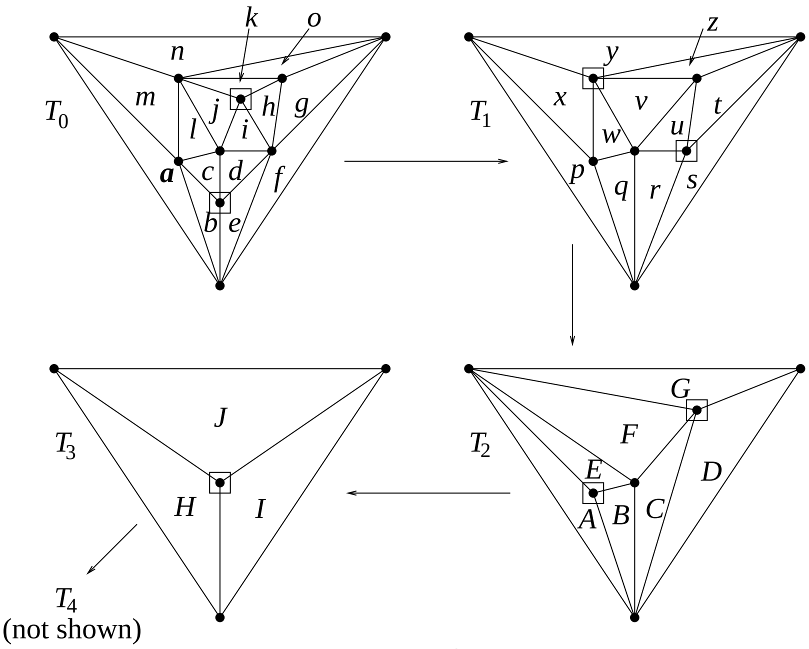
(1) Mark all nodes of degree ≥ 9.

(2) While there exists an unmarked node do the following:

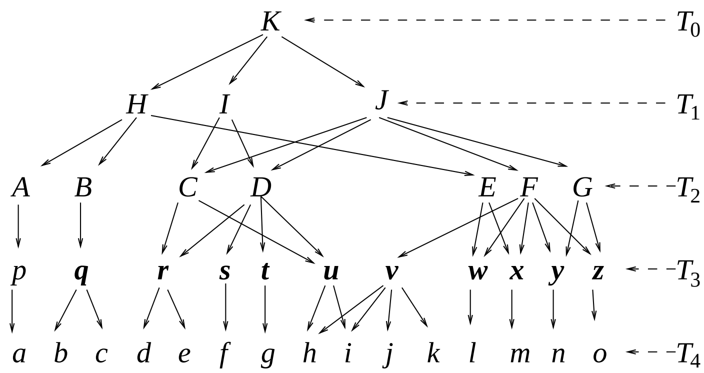
  (a) Choose an unmarked vertex v.

  (b) Add v to the independent set.

  (c) Mark v and all of its neighbors.



Triangulation Refinement.



Final Point Location Tree.

Figure 39: Kirkpatrick's point location algorithm.

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 14: More Planar Point Location

(Thursday, Oct 16, 1997)

Read: Chapt. 6 of BKOS.

More Point Location: Last time we presented Kirkpatrick's point location algorithm. Today we will consider another asymptotically optimal algorithm for point location, but the interesting element of this algorithm is that it is randomized, and in particular, it is based on a randomized incremental construction. We will show that the size, preprocessing time, and query time are $O(n)$, $O(n \log n)$ and $O(\log n)$, respectively, in the expected case. Here the expectation is independent of the choice of the polygonal subdivision or the query point. It depends only on the order in which the objects are inserted.

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 subdivision, 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. (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 bullet travels until it hits another line segment of S. The resulting "bullet paths", together with the initial line segments define the trapezoidal map. To avoid infinite bullet paths at the

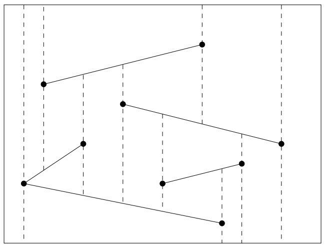


Figure 40: Trapezoidal map.

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.

First 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. The final trapezoidal map will be given as a subdivision, 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 $3n$. 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 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. (Hopefully, this is 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.

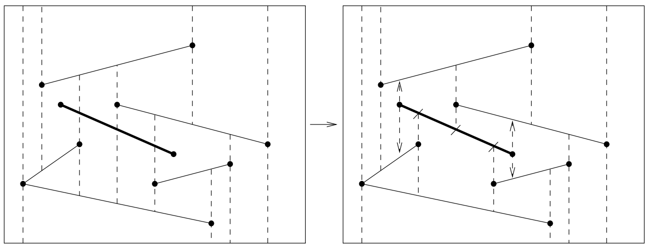


Figure 41: Incremental update.

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 detail out, which is how we locate the left endpoint of each new segment that we add. But ignoring the time for this (which we will see will be $O(\log n)$ time on average), we will show that the expected time to add each 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. 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:

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

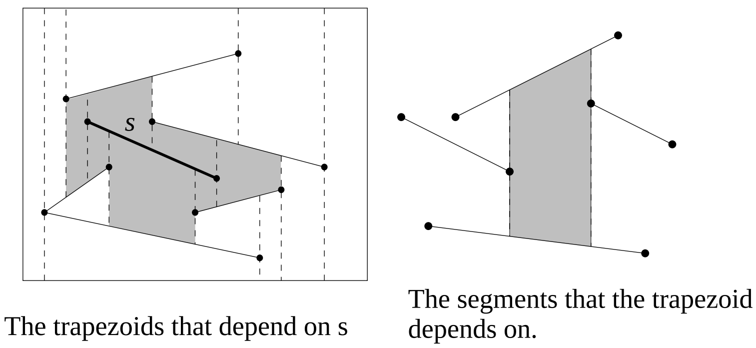


Figure 42: Trapezoid-segment dependencies.

This is much easier to determine. In particular, each trapezoid is bounded by four sides. 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

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

Lecture 15: Point Location in Trapezoidal Maps

(Tuesday, Oct 21, 1997)
Read: Chapt. 6 of BKOS.

Point Location: Last time we presented a randomized incremental algorithm for constructing a trapezoidal map in the plane. Today we consider how to modify this algorithm to answer point location queries. 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.

Recall from last time that we are treating 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: As in Kirkpatrick's algorithm, the point location data structure will be based on a rooted directed acyclic graph. In particular, to the query processor it will look like a binary tree, but there may be sharing of subtrees. There are two types of nodes, *x-nodes* and *y-nodes*. Each x-node contains the x-coordinate of an endpoint of one of the segments. Its two children correspond to the points lying to the left and to the right of this coordinate. Each y-node contains a pointer to a segment. The left and right children correspond to whether the query point is above or below line containing the segment, respectively. (We will visit a y-node only if we already know that the x-coordinate of the query point lies between the left and right endpoints of the 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 existing structure will be a point location structure for the existing trapezoidal map. In the figure below we show a simple example of what the data structure looks like for two line segments (from our textbook). 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 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).

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 have been deleted as a result of this new addition. This will be set of leaves in the current tree. Each such leaf will be replaced with a new subtree, which determines which one of the new trapezoids contains the query point. In Kirkpatrick's algorithm we just said "check each of the new triangles that overlapped one of the old triangles", and we will do essentially the same thing here. As in Kirkpatrick's algorithm, it will turn out that each old

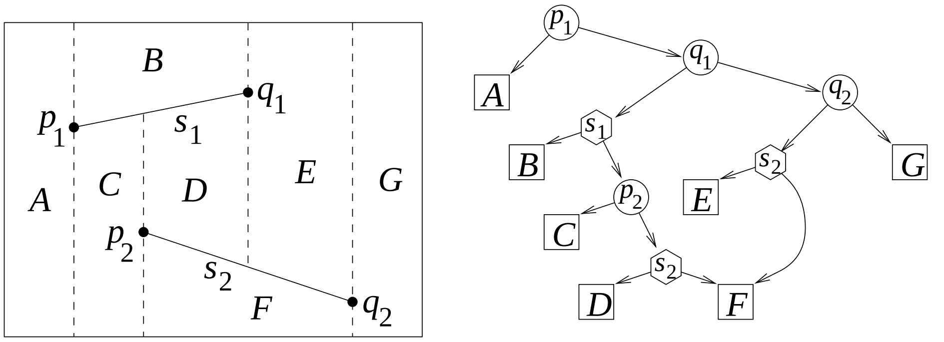


Figure 43: Trapezoidal map point location data structure.

trapezoid is overlapped by a constant number of new trapezoids. However, let us consider the process in a little more detail here.

Suppose that we add a line segment. This results in the replacement of an existing set of trapezoids with a set of new trapezoids. If the segment passes entirely through an existing trapezoid, then there will be two overlapping trapezoids in the new trapezoidal map, and thus we just need to compare against the newly added segment (one y-node). If the existing trapezoid contains one or both endpoints, then we need to test on which sides of these endpoints we lie (one or two x-nodes) and if we lie between them, then we need to test whether we lie above or below the line segment (one y-node). If we add a line segment to the example above, resulting in the replacement of C, D, E and G with new trapezoids, we will replace these leaves of the subtree as shown in the figure below. It is important to notice that (through sharing) each trapezoid appears exactly once as a leaf in the resulting structure.

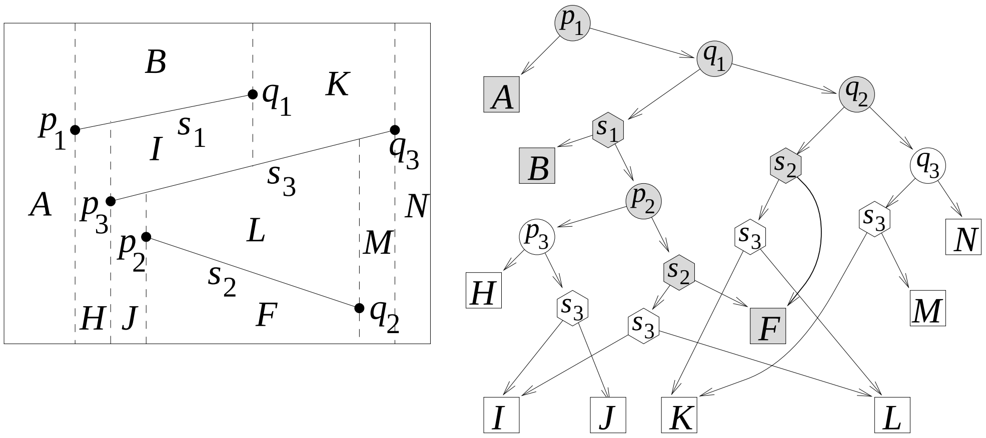


Figure 44: Line segment insertion.

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. This follows by observing that we create a constant number of new nodes for each existing trapezoid that was destroyed, and the number of new trapezoids is proportional to the number of old trapezoids that were destroyed. 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 to handle. (Imagine an adversary, that tries to select the worst-possible query point, but does not know what random choices the algorithm made in building the data structure.) We will show that no matter how the query point is selected, most random orderings of the line segments will lead to a search path of length $O(\log n)$ in the resulting tree. Since our analysis will be made without any assumptions on the final trapezoid in which q lies, it will apply equally well to all query points.

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. (For example, if q was in trapezoid C, before the addition of s_3 in the figure above, and q was in trapezoid J afterwards, then q would have to pass through two new levels of the structure as a result of this insertion.)

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. 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. Recall from last time that the trapezoid that contains q is dependent on at most four trapezoids, 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 be 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: The only problem with this result 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. The result 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 algorith 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: Review for the Midterm

(Thursday, Oct 23, 1997)

Midterm Exam: Oct 28. The exam will be closed-book, closed-notes, but you are allowed one cheat-sheet (front and back) of notes.

What we've covered so far:

Geometric Background: Affine geometry, homogeneous coordinates, orientation test, symbolici perturbation.

Convex hulls: Graham's scan, an $O(n \log n)$ algorithm.

Line Segment Intersection: Plane sweep algorithm, $O((n+k)\log n)$, where k is the number of intersections.

Representing planar subdivisions: Euler's formula, DCEL's, dual graphs.

Polygon Triangulation: The art-gallery problem, monotone decomposition by plane sweep in $O(n \log n)$ time, triangulating monotone polygons in $O(n)$ time. There exists an $O(n)$ time triangulation, but it is quite complicated.

Intersection of Halfplanes: Divide-and-conquer solution that runs in $O(n \log n)$ time.

Point-line duality: The lower and upper envelopes of a set of lines can be computed in $O(n \log n)$ time by applying point-line duality, and computing the convex hull of the dual points.

Linear Programming: Randomized incremental algorithm for low dimensional linear programming. Runs in $O(d!n)$ expected time in dimension d.

Orthogonal Range Searching: kd-trees and range trees. Both can be constructed in $O(n \log n)$ time. Showed that kd-trees could be used to answer orthogonal range queries in $O(\sqrt{n})$ time, and that range trees could answer them in $O(\log n)$ time. Both can be generalized to higher dimensions, with range trees taking $O(\log^{d-1} n)$ time.

Planar Point Location: Kirkpatrick's algorithm. $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time. Based on decimation of a triangulation.

Trapezoidal Maps: Randomized incremental construction and point location. $O(n \log n)$ expected preprocessing time, and $O(n)$ space and $O(\log n)$ query time.

Techniques:

- Plane sweep.
- Divide and conquer.
- (Randomized) incremental construction.
- Decomposing search problems into layers of trees.
- Point-line duality.
- Fractional cascading.
- Backwards analysis.

Lecture 17: Midterm

(Tuesday, Oct 28, 1997)
Midterm Exam today. No lecture.

Lecture 18: Voronoi diagrams

(Thursday, Oct 30, 1997)
Reading: BKOS, Chapt 7.

Euclidean Geometry: We now will make a subtle but important shift. Up to know, 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 which 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 $|v| = \sqrt{v_x^2 + v_y^2}$ (and in general dimension it is $|v| = \sqrt{v_1^2 + \ldots + v_d^2}$). The distance between two points p and q, denoted $\mathrm{dist}(p, q)$, 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 such that $\text{dist}(q, p_i) < \text{dist}(q, p_j)$. That is, the Voronoi cell for p_i consists of the set of points for which p_i is the unique *nearest neighbor* to q:

$$\mathcal{V}(p_i) = \{q \mid \text{dist}(p_i, q) \leq \text{dist}(p_j, q), \forall j \neq i\}.$$

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 $\text{Vor}(P)$ 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.

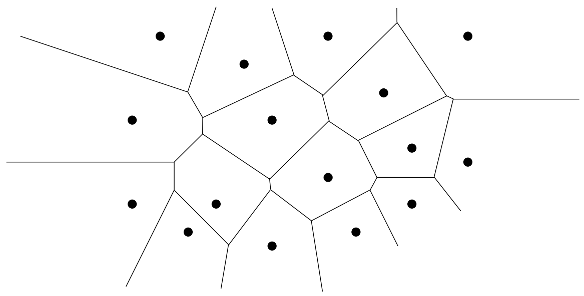


Figure 45: Voronoi diagram

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 locally at maximum distances from any other point in the set.

High clearance path planning: A robot wants to move around a set of obstacles. To minimize the possibility of collisions, it should stay as far away from the obstacles as possible. To do this, it should walk along the edges of the Voronoi diagram.

Properties of the Voronoi diagram: Some theoretical observations about the Voronoi diagram are warranted at this point.

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.

Voronoi vertices: It follows that vertex at which three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ intersect 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 assume that no four points 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 point is on the convex hull if and only if it is the closest point from some point at infinity.)

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.)

Delaunay Triangulation: 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. Among the many triangulations of a set of points, this one has a number of nice geometric properties, and is called the *Delaunay triangulation*.

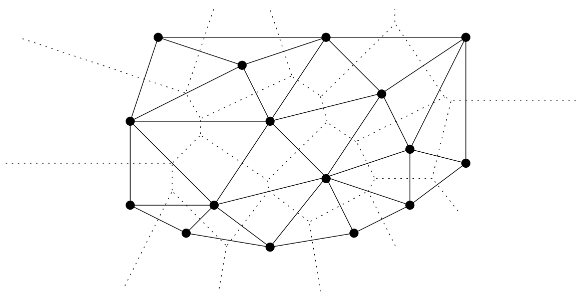


Figure 46: Delaunay triangulation.

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

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

Circumcircle property: The circumcircle of any triangle in the Delaunay triangulation is empty (contains no points 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 points and there is no closer point.)

Closest pair property: The closest pair of points in P are Delaunay neighbors.

There are a number of other interesting properties of Delaunay triangulations that are not so easy to prove.

Max-Min Angle criterion: Among all triangulations of the point set P, the Delaunay triangulation maximizes the minimum angle in the triangulation. (This only holds in dimension 2).

MST property: The minimum spanning tree of a set of points in the plane is a subgraph of the Delaunay triangulation.

This provides a good algorithm for computing MST's of points in the plane. First, compute the Delaunay triangulation of the point set (we will show this can be done in $O(n \log n)$ time), and then in $O(n \log n)$ time you can compute the MST of this sparse graph.

Lecture 19: Fortune's Voronoi Diagram Algorithm

(Tuesday, Nov 4, 1997)
Reading: BKOS, Chapt 7.

Computing Voronoi Diagrams: Last time we introduced the Voronoi diagram of a set of n points $P = \{p_1, \ldots, p_n\}$ in the plane, called *sites*. Recall that this is a planar straight-line subdivision with n faces, each a (possibly unbounded) convex polygon. The face, or *Voronoi cell* associated with site p_i, denoted $\mathcal{V}(p_i)$, consists of all the points in the plane that are strictly closer to p_i than to any other site. Recall that the edge joining cells for p_i and p_j is the perpendicular bisector between these points, and so a circle grown about any point on a Voronoi edge will touch these two sites and contain no other site in its interior. Also recall that the Voronoi vertices are of degree three (assuming general position) and a cicle grown about this point will touch all three sites an contain no other site in its center.

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.

Note from the comments made last time about the Delaunay triangulation, that the Voronoi diagram can either be computed directly, or else it can be built by first constructing the Delaunay triangulation of the sites, and then taking its dual graph (with some care as to where the Voronoi vertices are to be placed). Historically, the first algorithm for computing Voronoi diagrams is the simple incremental algorithm that was used for computing Delaunay triangulations. It was known for many years that this is not asymptotically optimal, and in fact ran in $O(n^2)$ time in the worst case, but this did not deter the practitioners who were quite happy with it. 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, Steve 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, but in the form of a Delaunay triangulation algorithm.

Before discussing Fortune's algorithm, it is interesting to consider why this algorithm wasn't 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 where the upcoming events will occur. The reason is that a site that lies ahead of the sweep line may generate a Voronoi vertex that lies behind the sweep line. It is these "unanticipated events" that make the design of a plane sweep algorithm challenging.

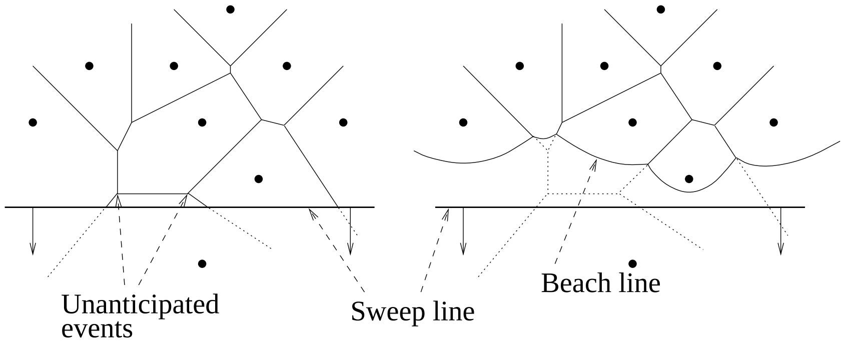


Figure 47: Plane sweep Voronoi diagrams.

Fortune's Algorithm: 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 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 a x-monotonic curve called a *beach line* (I guess because it looks like waves rolling up on a beach.) The beach line is formed from parabolic arcs. As the sweep line moves downward, the beach line moves downward as well, but the beach line's shape depends on the locations of the sites. We will see that the Voronoi diagram "grows out" of 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 "cut" 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. 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 point 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 abouts what points are still to appear below the sweep line).

What does the beach line look like? We know 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. With a little analytic geometry, it is easy to show that the parabola becomes "skinnier" as the site becomes closer to the line, the parabola degenerates into a vertical ray emanating from the site. (You should work out the equations to see why this is so.)

Thus, the beach line consists of the lower envelope of these parabolas. (By the way, if we were interested in computing the Voronoi diagram of line segments, the beach line is the boundary of the Voronoi cell of the sweep line itself.) 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 points 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 be interested in simulating the discrete event points where there is a "significant event", that is, any event that changes the topological structure of the beach line. It turns out these significant events will be of two varieties:

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. For the rest of the lecture we focus on the nature of the beach line, and how these events are discovered and processed.

Site events: A site event is generated whenever the 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.)

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

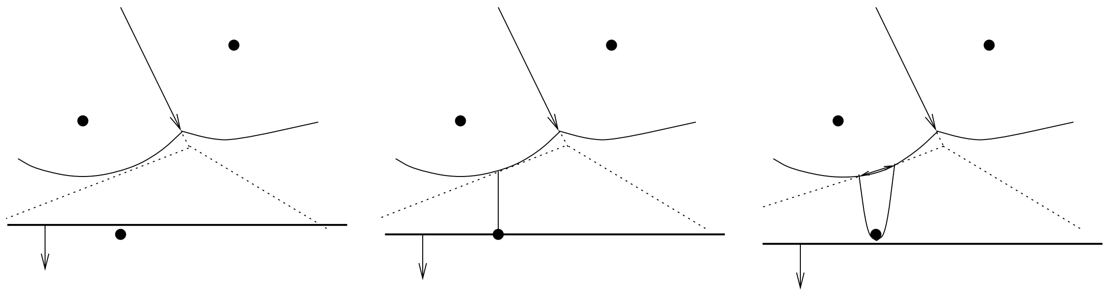


Figure 48: Site events.

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. However, the other type of event are somewhat harder to predict.

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 points 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.

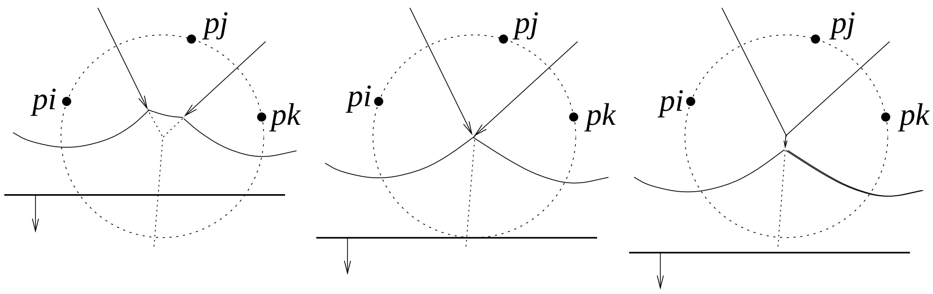


Figure 49: Vertex events.

Sweep-line algorithm: We can now present the algorithm is 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.)

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 their 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 $2n - 1$.

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. 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. Any old triple that involved p_j as its center arc is deleted from the priority queue, and we generate new events for each of the possible three new triples that result from this insertion.

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 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).

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 20: Delaunay Triangulations

(Thursday, Nov 6, 1997)
Special Announcement: Next Tuesday, Nov 11, class will meet in AVW 2460 to attend Prof. Defloriani's lecture on multiresolution surface representations. You will be responsible for material presented in Defloriani's talk.
Reading: BKOS, Chapt 9.

The Delaunay Triangulation: Last time we gave an algorithm for computing Voronoi diagrams. Today we consider the related structure, called a *Delaunay triangulation.* Recall that a triangulation of a point set P is a straight line planar subdivision whose vertices are the points of P, and which is maximal in the sense that no edge may be added without violating planarity. This implies that very internal face of this subdivision is a triangle, and the external face is bounded by the boundary of the convex hull of the point set.

There are many possible triangulations of a given point set, but one stands out as having the nicest geometric properties. This is the *Delaunay triangulation.* The Delaunay triangulation can be defined in a number of equivalent ways. The definition that we use is that it is the straight-line dual of the Voronoi diagram of the sites. In particular, the vertices are the sites themselves, and two sites are adjacent in the triangulation if and only if their Voronoi cells share a common edge.

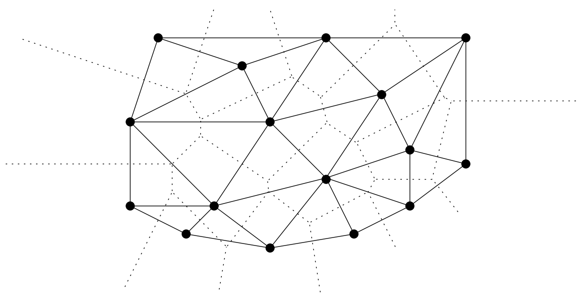


Figure 50: Delaunay Triangulation.

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 that the Delaunay triangulation has $2n - 2 - h$ triangles (as we saw on the exam) and $3n - 3 - h$ edges (by a similar proof). Recall that in all planar graphs the average degree of a vertex is a constant (at most 6).

Also it is not hard to prove that the graph is planar. (We know that the graph is topologically planar, since it is the dual of a planar graph, but this only implies that there is some way to draw the edges so they do not cross. It would still need to be proven that if the edges are drawn as straight line segments, then they still do not cross. See the text for details.)

An equivalent condition that we mentioned earlier is that every triangle in the Delaunay triangulation has the property, called the *empty circle property*, that the circumcircle of the three vertices of the triangle does not contain any other site of P in its interior.

Maximizing Angles and Edge Flipping: One of the interesting properties of Delaunay triangulations is that among all triangulations, the Delaunay triangulation maximizes the minimum angle. 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 maximum 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 $\theta_1 > \theta_2 > \theta_3$.

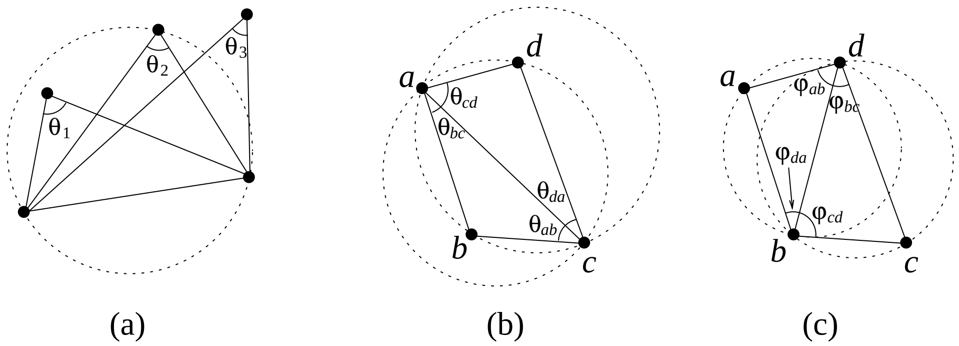


Figure 51: Angles and edge flips.

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 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 $\triangle abc$ (implying equivalently that b lies inside the circumcircle of $\triangle cda$), then if we flip the edge it will follows 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?), but even though their values might decrease it can be seen that neither of them is the minimum angle after the swap (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.

Randomized Algorithm: This observation provides a basis for a simple but potentially very slow algorithm. Simply build any initial triangulation (e.g the plane sweep triangulation presented in the exam) and then start performing edge flips until every pair of triangles satisfies the empty circle property. However, there are no good bounds on the number of edge flips required, or what a good strategy would be for selecting them. Instead, 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 trapezoidal map algorithm in that builds its own point-location data structure as a side effect. 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 the number of structural changes in the diagram is not very large (it will be $O(1)$ in the expected case for each insertion). As with other incremental algorithm, we need some way of keeping track of where newly inserted sites are to be placed in the diagram. Unlike the trapezoidal map algorithm, we will not create a data structure, but will instead use a simpler method that puts each of the uninserted points into buckets according to the triangle that it overlaps in the current triangulation. In this case, we will need to argue that the number of times that a site is rebucketed on average is not too large (it will be $O(\log n)$ in the expected case).

Incremental update: 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 to 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. It can be shown that a site d lies in the circumcircle determined by the counterclockwise triangle $\triangle abc$ 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.

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

When we add the next site, p_i, the problem is to convert the current Delaunay triangulation into a new Delaunay triangulation. This will be done by creating an illegal (nonDelaunay) triangulation containing the new point, 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.

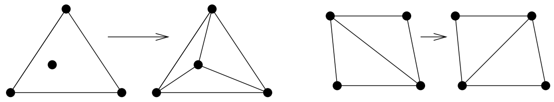


Figure 52: Basic triangulation changes.

Here is how the algorithm works. We start with an initial triangulation. Guibas, Knuth and Sharir suggest starting with the triangle of three sites "at infinity". (Not just any enclosing triangle will work, since we need to be sure that the newly added vertices will not affect the structure of the circumcircles, and hence the interior structure of the triangulation). This guarantees that all sites to be added, will lie within some triangle of the existing triangulation.

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, 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 other side of the edge that does not include p. If this vertex fails the incircle test, then we swap the edge (creating two new triangles that are adjacent to p) and repeat the same test with these triangles. An example is shown below.

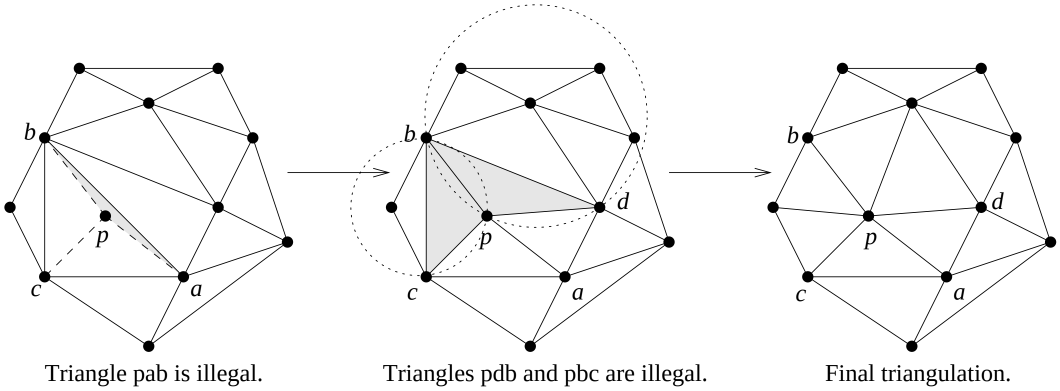


Figure 53: Point insertion.

The following is a description of the algorithm (Guibas, Knuth, and Sharir give a nonrecursive version). The current triangulation is kept in a global data structure. The edges in the following algorithm are actually pointers to the quad-edge data structure.

Insert(p) {
    Find the triangle $\triangle abc$ containing p;
    Insert edges pa, pb, and pc into triangulation;
    SwapTest(ab);

```
        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 (in(p, a, b, d) {
            Replace edge ab with pd;
            SwaptTest(ad);
            SwaptTest(db);
        }
    }
```

As you can see, the algorithm is very simple. The only things that have to be implemented are (1) the DCEL (or other data structure) to store the triangulation, (2) the incircle test, and (3) locating the triangle that contains p. Tasks (1) and (2) are straightforward.

Task (3) can be accomplished by one of two means. Our text discusses the idea of building a point-location data structure for dealing with this. The other, somewhat simpler, approach is based on a simple bucketing idea which involves the same total amount of work, but it a bit easier to implement. Think of each triangle of the current triangulation as a *bucket* that holds the future sites that lie within this triangle and are yet to be inserted. Whenever an edge is flipped, or when a triangle is split into three triangles through point insertion, then some triangles are destroyed and are replaced by a constant number of new triangles. When this happens, all the sites in the buckets associated with the old triangles are rebucketed according to the new triangles that they lie in.

There is only one major issue in establishing the correctness of the algorithm. When we performed empty-circle tests, we only tested (1) triangles containing the site p, and (2) only sites that lay on the opposite side of an edge of such a triangle. To establish (1), observe that it suffices to consider only triangles containing p because since p is the only newly added site, it is the only site that can cause a violation of the empty-circle property.

To establish (2) we argue that if for every site d, which is opposite from p along some edge ab, lies outside the circumcircle of pab, then all these circumcircles are empty. A complete proof 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 (e.g. 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 high school geometry.) However, this violates the assumption that the initial triangulation (before the insertion of p) was a Delaunay triangulation.

Analysis: To analyze the algorithm we need to bound two things: (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.

We argue (1) that the expected number of edge changes with each insertion is $O(1)$ by a simple application of backwards analysis. Observe that whenever the triangulation performs an edge swap, it always adds a new edge to p. Therefore the total number of changes made in the triangulation for the insertion of p is equal to the degree of p *after* the insertion is completed. By backwards analysis, we know that each of the sites that are currently in the triangulation are equally likely to be deleted. (With the exception of the three initial sites at infinity, but they only slightly bias this argument, so we ignore them.)

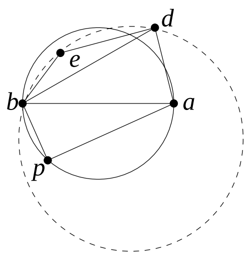


Figure 54: Proof of sufficiency of testing neighboring sites.

By the argument just given, we know that the total number of edge changes needed if p were the last site added is just the degree of p in the resulting triangulation. However, we also know that the average degree of a vertex in a planar graph is constant. Therefore, the expected number of changes to the structure is $O(1)$ per insertion.

Next we argue (2) 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 i-th stage of the algorithm (after i sites have been inserted into the triangulation). Among the remaining $n-i$ sites, we claim that the probability that this site changes triangles is at most $3/i$. Observe that when we insert a new site p, all of the newly created triangles include the site p. Each triangle in the current triangulation is determined by exactly 3 sites. If none of these 3 was the last site to be inserted, then no (uninserted) site in this triangle needs to be rebucketed. Thus, for each of the remaining $n-i$ sites, the probability that this site needs to be rebucketed is at most $3/i$. (The probability is independent of the site distribution, and depends only on the sequence with which the sites are inserted.) Thus, the total expected number of rebucketings is given by the sum

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

Lecture 21: DeFloriani's Lecture

(Tuesday, Nov 11, 1997)
Special Announcement: Today's class will meet in AVW 2460 to attend Prof. Defloriani's lecture on multiresolution surface representations. You will be responsible for material presented in Defloriani's talk.

Lecture 22: Line Arrangements

(Thursday, Nov 13, 1997)
Reading: BKOS, Chapt 8.

Arrangements: So far we have studied a few of the most important (in my opinion) geometric structures: convex hulls and Voronoi diagrams and Delaunay triangulations. The next most important structure (again in my opinion) in computational geometry is that of a *line arrangement*. As with hulls and Voronoi diagrams, it is possible to define arrangements 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.

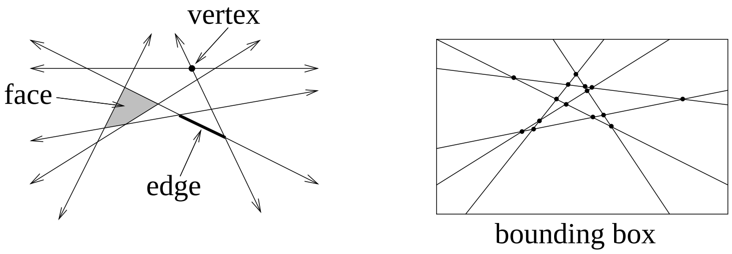


Figure 55: Arrangement of lines.

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 will not require randomization. In fact, its asymptotic running time will be the same, $O(n^2)$, no matter what order we insert the lines. This is asymptotically optimal, since there 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.

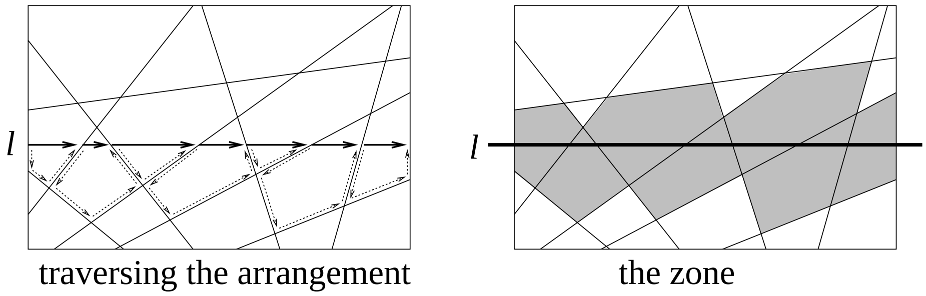


Figure 56: Traversing an arrangement and zones.

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. (Details are left as an exercise.)

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 $O(n^2)$. 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 basis 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_n 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 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.

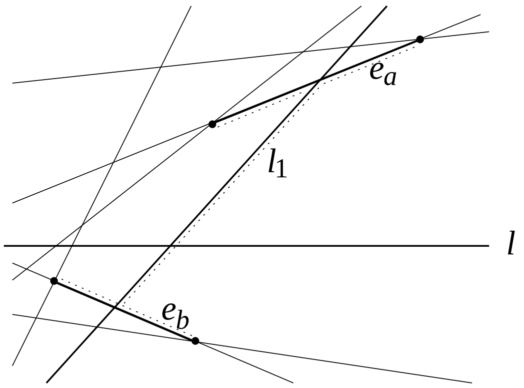


Figure 57: Proof of the Zone Theorem.

Lecture 23: Applications of Arrangements

(Tuesday, Nov 18, 1997)
Reading: This material is not covered in the text.
Revised: Nov 19. Added material about angular sorting.

Sweeping Arrangements: Last time we showed how to construct an arrangement of lines in the plane in $O(n^2)$ time. Since an arrangement is of size $O(n^2)$, this is optimal. Usually it is not sufficient just to build the arrangement, but necessary to traverse the arrangement as well. In some instances, any graph traversal will suffice. For other problems it is nice to perform the traversal in some order, for example as a plane sweep from left to right.

If an arrangement is to be built just so it can be swept, then maybe you didn't need to construct the entire arrangement after all. You can just perform the plane sweep on the lines, exactly as we did for the line segment intersection algorithm. Assuming that we are sweeping from left to right, the initial position of the sweep line is at $x = -\infty$ (computed symbolically). The sweep line status maintains the lines in top to bottom order according to their intersection with the sweep line. The initial order is according to increasing order of slope. Then the sweep line proceeds exactly as it did in the algorithm for determining line segment intersections, but the only events are intersection events (since there are no endpoints). The sweep ends when the last intersection event is processed. Sweeping an arrangement in this way takes $O(n^2 \log n)$ time, but only $O(n)$ space (since we need only maintain the current sweep line). For many applications involving arrangements, this offers a reasonable tradeoff between time and space.

But is the $O(\log n)$ factor necessary? In many applications involving plane sweep it is not necessary to sweep in strictly left-to-right order, as long as the vertices lying on each line of the arrangement are swept from left-to-right. In this case we can save the $O(\log n)$ factor by constructing the arrangement, then (thinking of the edges as being directed from left to right) perform a *topological sort* of the vertices. A topological sort of an acyclic directed graph is an ordering of the vertices of a directed graph such that for each edge (u, v) in the graph, u appears before v in the order. It is well known that the vertices of a graph can be topologically sorted in time that is proportional to the size of the graph, or $O(n^2)$ in this case. (See CLR for details.) Given the topological ordering of the vertices of the arrangement, we can sweep the arrangement by simply extracting the vertices in this order. This gives an $O(n^2)$ time and $O(n^2)$ space algorithm for sweeping arrangements.

Is it possible to sweep arrangements in $O(n^2)$ time but with only $O(n)$ space? It turns out that the answer is yes. There is an algorithm called *topological plane sweep* which incorporates

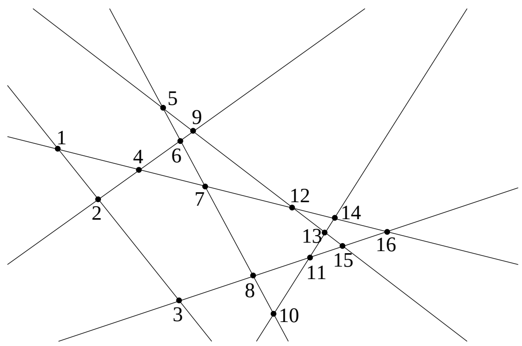


Figure 58: One possible topological ordering of the vertices of an arrangement.

the topological nature of the above sweeping algorithm with the incremental nature of plane sweep.

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, determine whether 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 duality transformation described in Lecture 8. A point $p = (p_x, p_y)$ and line $\ell : (y = ax - b)$ in the primal plane are mapped through duality to dual point ℓ^* and dual line p^* by the following relationship.

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

Observe 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.

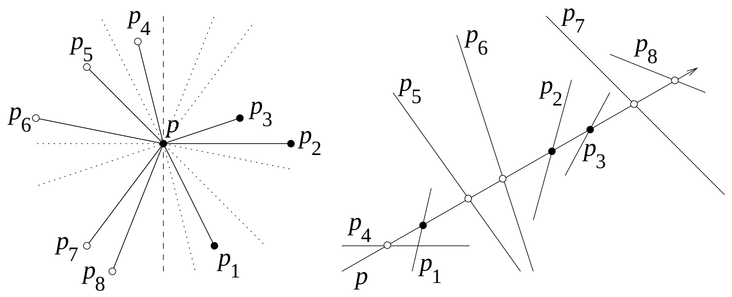


Figure 59: Arrangements and angular sequences.

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 an 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.

Lecture 24: More Applications of Arrangements

(Thursday, Nov 20, 1997)
Reading: The material on discrepancies is in Chapt 8 of BKOS. The other material is not covered in the text.

Applications of Arrangements: Today we will discuss a few of applications of line arrangements in the plane. This is to give a feel for how one converts problems into dual form and then uses arrangements to solve these problems. Recall the duality operator from Lecture 8. We defined 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 denoted it using a asterisk ($*$) as a superscript. Thus, given point $p = (p_x, p_y)$ and line $\ell : (y = ax - b)$ in the primal plane we define ℓ^* and p^* to be a point and line respectively in the dual plane defined by:

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

The main property of this transformation was the following *order reversing* property: a 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.

Narrowest 3-corridor: As mentioned above, in this problem we are given a set P of n points in the plane, and an integer k, $1 \leq k \leq n$, and we wish to determine the narrowest pair of parallel lines that enclose at least k points of the set. In this case we will define the vertical distance between the lines as the distance to minimize. (It is easy, but a bit tricky to adapt the algorithm for perpendicular distance.)

To simplify the presentation, we assume that $k = 3$. The generalization to general k is an exercise. We will assume that no three points of P are collinear. This will allow us to assume that the narrowest corridor contains exactly three points and has width strictly greater than zero. (Observe that if three points were collinear, then they would correspond to three lines that intersect at a common vertex in the arrangement, which could be tested as we build the arrangement.) We will also assume that no two points have the same x-coordinate. The dual transformation that we consider cannot represent vertical lines, but this assumption will imply that the solution is not a vertical line. We could fix this by using homogeneous coordinates, but we will not worry about it now.

If we dualize the points of P, then in dual space we have a set L of n lines, $\{\ell_1, \ell_2, \ldots, \ell_n\}$. The slope of each dual-line is the x-coordinate of the corresponding point of P, and its y-intercept is the negation of the point's y-coordinate.

A narrowest 3-corridor in the primal plane consists of two parallel lines ℓ_a and ℓ_b in primal space. Their duals ℓ_a^* and ℓ_b^* are dual points, which have the same x-coordinates (since the lines are parallel), and the vertical distance between these points, is the difference in the y-intercepts of the two primal lines. Thus the vertical width of the corridor, is the vertical length of the line segment.

In the primal plane, there are exactly three points lying in the corridor, that is, there are three points that are both above ℓ_b and below ℓ_a. Thus, by the order reversing property, in the dual plane, there are three dual lines that pass both below point ℓ_b^* and above ℓ_a^*. Combining all these observations it follows that the dual formulation of the narrowest 3-corridor problem is the following:

Shortest vertical 3-stabber: Given an arrangement of n lines, determine the shortest vertical segment that stabs three lines of the arrangement.

It is easy to show (by a simple perturbation argument) that the shortest vertical 3-stabber may be assumed to have one of its endpoints on a vertex of the arrangement, implying that the other endpoint lies on the line of the arrangement lying immediately above or below this vertex. (In the primal plane the significance is that we can assume that the minimum 3-corridor one

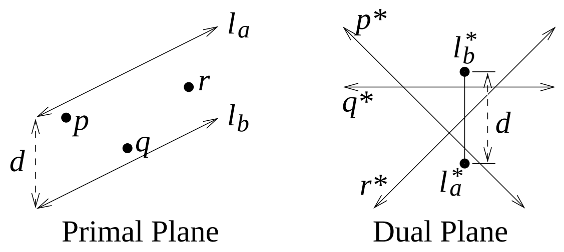


Figure 60: Narrowest 3-corridor: primal and dual form.

of the lines passes through 2 of the points, and the other passes through a third point, and there are no points within the interior of the corridor. This is shown in the figure below.

We can compute the minimum 3-stabber in an arrangement, by a simple plane sweep of the arrangement (using a vertical sweep line). Whenever we encounter a vertex of the arrangement, we consider the distance to the edge of the arrangement lying immediately above this vertex and the edge lying immediately below. We can solve this problem by topological plane sweep in $O(n^2)$ time and $O(n)$ space.

We can also solve this by constructing the arrangement and the computing the (vertical) trapezoidal map. Each trapezoidal edge will correspond to a corridor. The shortest such edge is the final answer. This leads to an $O(n^2)$ time and space solution.

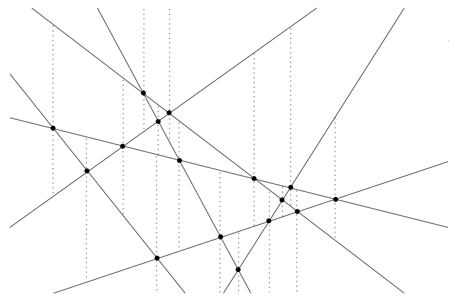


Figure 61: Narrowest corridor and trapezoidal maps.

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, 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

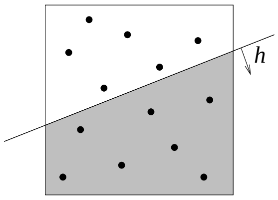


Figure 62: Discrepancy of a point set.

upper bound) of this quantity over all halfplanes:

$$\Delta(S) = \sup_h \Delta_S(h).$$

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 alterning $\mu(h)$.

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 can rotate this line about p and hence increase or decrease $\mu(h)$ without altering $\mu_S(h)$, to increase their difference.

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 dualized 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, $\mathcal{L}_k$, 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.)

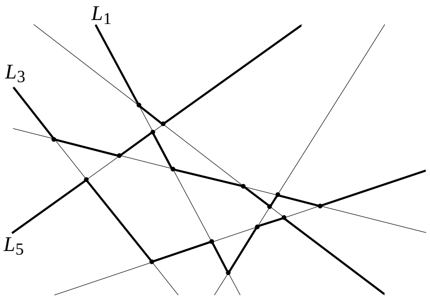


Figure 63: Levels in an arrangement.

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 25: Shortest Paths and Visibility Graphs

(Tuesday, Nov 25, 1997)

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 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 mutually visible.

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 the line segment (v, w) lies entirely in free space.

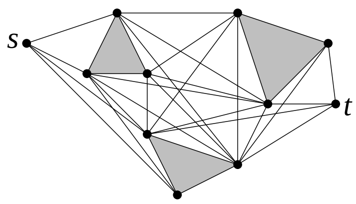


Figure 64: Visibility graph.

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 you are a stickler for exactness, 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: Next 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 purpose assumption that no three vertices are colinear, but this is not hard to handle with some care. The algorithm is *not* output sensitive. An $O(n \log n + k)$ algorithm does exist, but it 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. Using the fact that angular sorting can be implemented through topologically sweeping the a line arrangement in $O(n^2)$ time, we manage to shave off the extra $O(\log n)$ factor in running time.

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.

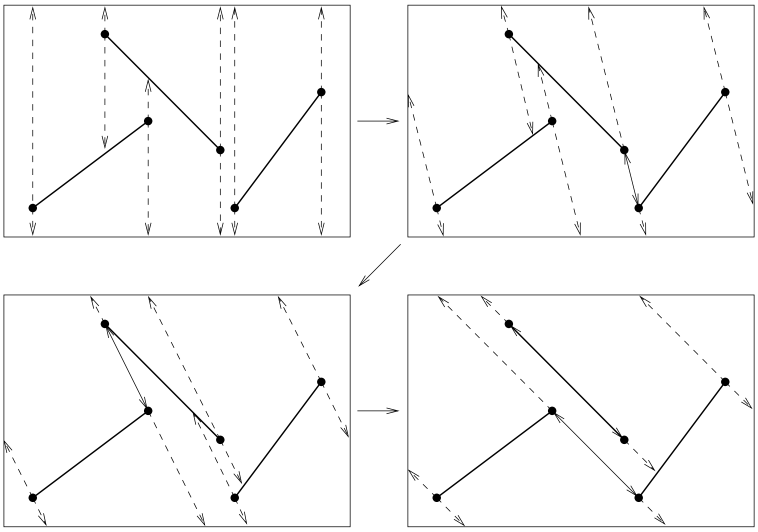


Figure 65: Visibility graph by multiple angular sweep.

Let us consider this "multiple angular sweep" in greater detail. Observe that a significant event occurs whenever a bullet path jumps from one line segment to another. This occurs when θ reaches the slope of the line joining two visible vertices v and w. Unfortunately, it is somewhat complicated to keep track of which vertices 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 vertices, whether they are visible or not. By duality, the slope of such an event will correspond to the a-value of the intersection of lines v^* and w^* in the dual arrangement. (Convince yourself of this before going on.) Thus, by sweeping the arrangement we will generate all these events.

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.

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).

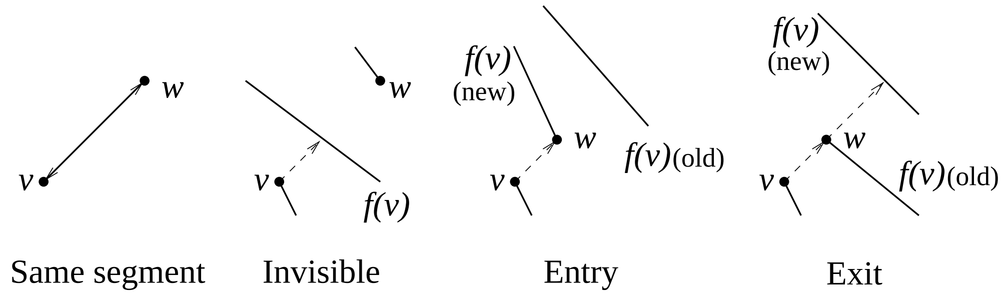


Figure 66: Possible events.

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 "nonevent".

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 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.

There is one subtlety about topological sweep that we should be careful about though. Remember that topological sweep and regular plane sweep are not identical, in that a topological sweep does process events in strictly left to right order. However, we are assured that along any dual line, the sweep will process events from left to right along the line.

Here is our fear. In the *Segment Exit* event, we accessed $f(w)$. But since the sweeps along the dual lines for v and w are proceeding independently in general, might it be that $f(w)$ is lagging behind or has moved ahead of the sweep along v? If so, this could be disastrous, because $f(w)$ would be set for some other angle θ. To see why this is not a danger, we need to look at what is happening in the arrangement. At the moment of this event, we are processing a line that passes through both v and w. In the dual plane, this line corresponds to the intersection point of lines v^* and w^* in the dual plane. The topological plane sweep sweeps each vertex of the arrangement exactly once, implying that these events will be processed at the same time in the topological plane sweep, and so there is no danger of one being ahead or lagging behind the other.

There are examples of algorithms where the natural events to be processed in topological plane sweep do not correspond to single vertices, but rather to groups of events in different parts of the arrangement that are supposed to occur at the same time, but might be out of phase because of the topological nature of the sweep. Does this mean that topological sweep cannot be used? Sometimes yes and sometimes no. Recall that the topological sweep arises from a topological sort of a DAG (the directed edges of the arrangement). Sometimes it is possible to add additional edges (i.e., additional ordering constraints) thus forcing the sweep to behave itself.

Lecture 26: Motion Planning

(Tuesday, Dec 2, 1997)
Reading: Chapt 13 in BKOS.

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 very 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. There are interesting variants of motion planning in environments where the obstacles either unknown or are known but move. In unknown environments, the robot must have some way of sensing its environment, either by sight or touch. In environments with moving objects, the robot must be capable of detecting the presense of moving objects early enough that it can avoid colliding with them. Henceforth, let S denote the set of obstacles defining the environment.

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 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 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*.

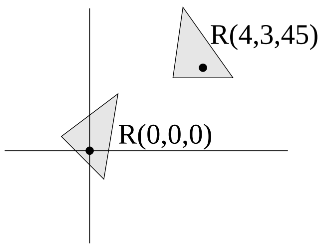


Figure 67: Configurations of a translating and rotating robot.

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.

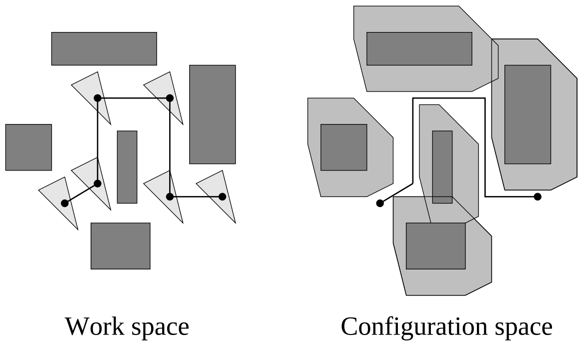


Figure 68: Work space and configuration space.

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 $C_{\text{free}}(\mathcal{R}, S)$.

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 are typically 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.

Planning the Motion of a Point Robot: As mentioned above, we can reduce complex motion planning problems to the problem of planning the motion of a point in free configuration space. First we will consider the question of how to plan the motion of a point amidst a set of polygonal obstacles in the plane, and then we will consider the question of how to construct configuration spaces.

To determine whether there is a path from one point to another of free configuration space, we will subdivide free space into simple convex regions. In the plane, we already know how to do this by computing a trapezoidal map. We can construct a trapezoidal map for all of the line segments bounding the obstacles, then throw away any faces that lie in the forbidden space. We also assume that we have a point location data structure for the trapezoidal map.

Next, we create a planar graph, called a *road map*, based on the trapezoidal map. To do this we create a vertex in the center of each trapezoid and a vertex at the midpoint of each vertical edge. We create edges joining each center vertex to the vertices on its (at most four) edges.

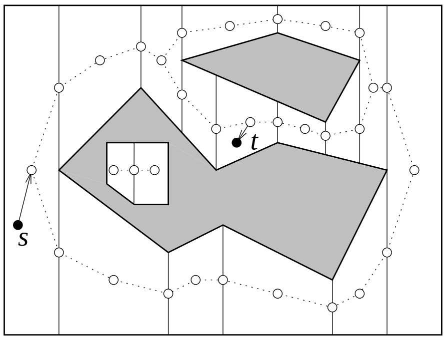


Figure 69: Motion planning using road maps.

Now to answer the motion planning problem, we assume we are given the start point s and destination point t. We locate the trapezoids containing these two points, and connect them to the corresponding center vertices. We can join them by a straight line segment, because the cells of the subdivision are convex. Then we determine whether there is a path in the road map graph between these two vertices, say by breadth-first search. Note that this will not necessarily produce the shortest path, but if there is a path from one position to the other, it will find it.

This description ignores many practical issues that arise in motion planning, but it is the basis for many practical motion planning problems. More realistic configuration spaces will contain more information (for example, encodings of the current joint rotation velocities) and will usually refine the road map to a much finer extent, so that short paths can be approximated well, as well as handling other elements such as guaranteeing minimal clearances around obstacles.

Configuration Obstacles and Minkowski Sums: Now that we know how to find paths in configuration space, let us consider how to build configuration space for a set of polygonal obstacles. We again 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 (x, y), so that $\mathcal{R}(x, y)$ denotes the placement of the robot. Given an obstacle P, the configuration obstacle is defined as all the placements of $\mathcal{R}$ that intersect P, that is

$$\mathcal{CP} = \{(x, y) \mid \mathcal{R}(x, y) \cap P \neq \emptyset\}.$$

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:

$$S_1 \oplus S_2 = \{p + q \mid p \in S_1, \quad q \in S_2\}.$$

Also, define $-S = \{-p \mid p \in S\}$. Observe that for the case of a translating robot, we could define $\mathcal{R}(x, y)$ as $\mathcal{R}(0, 0) + \{(x, y)\}$.

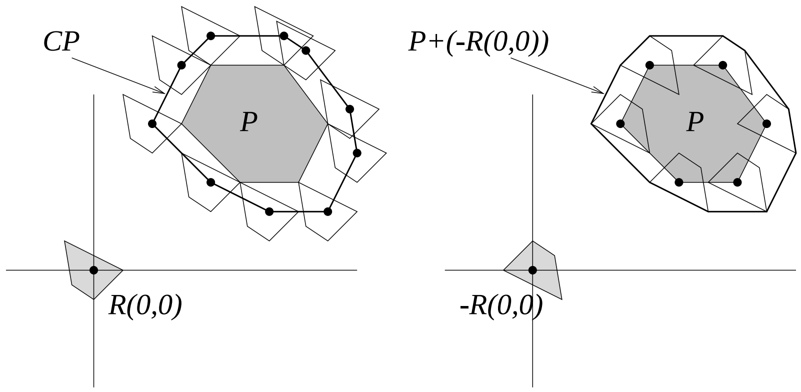


Figure 70: Configuration obstacles and Minkowski sums.

Claim: Given a planar translating robot $\mathcal{R}$ and an obstacle P, then the C-obstacle of P is $P \oplus (-\mathcal{R}(0, 0))$.

Proof: We show that $\mathcal{R}(x, y)$ intersects P if and only if $(x, y) \in P \oplus (-\mathcal{R}(0, 0))$. First, if $\mathcal{R}(x, y)$ intersects P, then there must be a point $q = (q_x, q_y)$ such that $q \in P \cap \mathcal{R}(x, y)$. This implies that $(q_x - x, q_y - y) \in \mathcal{R}(0, 0)$, and hence $(x - q_x, y - q_y) \in -\mathcal{R}(0, 0)$. Since $q \in P$, by adding (q_x, q_y) to this we have $(x, y) \in P \oplus (-\mathcal{R}(0, 0))$.

Conversely, if $(x, y) \in P \oplus (-\mathcal{R}(0, 0))$, then there must be points $(p_x, p_y) \in P$ and $(r_x, r_y) \in \mathcal{R}(0, 0)$ such that $(p_x - r_x, p_y - r_y) = (x, y)$. Thus we have $(p_x, p_y) = (r_x + x, r_y + y)$, which implies that P intersects $\mathcal{R}(x, y)$.

Since it is an easy matter to compute $-\mathcal{R}(0, 0)$ 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. The algorithm is based on the following observation. Given a vector $\vec{d}$, We say that a point p is *extreme* in direction $\vec{d}$ if it maximizes the dot product $p \cdot \vec{d}$.

Observation: Given two polygons P and R, then the set of extreme points of $P \oplus R$ in direction $\vec{d}$ is the set of sums of points p and r that are extreme in direction $\vec{d}$ for P and R, respectively.

The book leaves the proof as an exercise. It follows easily by the linearity of the dot product.

From this observation, it follows that there is a simple rotating calipers algorithm for computing $P \oplus R$, when both are convex polygons. In particular, we perform an angular sweep by considering a unit vector $\vec{d}$ rotating counterclockwise around a circle. As $\vec{d}$ rotates, it is an easy matter to keep track of the vertex or edge of P and R that is extreme in this direction. Whenever $\vec{d}$ is perpendicular to an edge of either P or R, we add this edge to the vertex of the other polygon. The algorithm is given in the text, and is illustrated in the figure below.

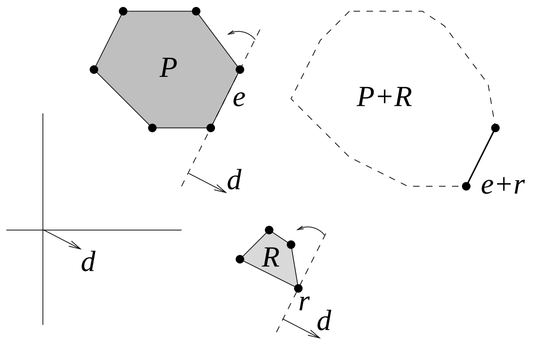


Figure 71: Computing Minkowski sums.

Assuming P and R are convex, observe that each edge of P and each edge of R contributes exactly one edge to $P + R$. (If two edges are parallel and on the same side of the polygons, then these edges will be combined into one edge, which is as long as their sum.) Thus we have the following.

Claim: Given two convex polygons, P and R, with n and m edges respectively, their Minkowski sum $P \oplus R$ can be computed in $O(n + m)$ time, and consist of at most $n + m$ edges.

Lecture 27: More Motion Planning

(Thursday, Dec 4, 1997)
Reading: Chapt 13 in BKOS.

Motion planning: Recall that the motion planning problem is that of determining whether there is a path from one placement (or configuration) of a robot to another, subject to the constraint that the robot does not intersect any of a set of polygonal obstacles. We showed that this problem could be solved by transforming the obstacles into *configuration space*, where each point in this space corresponds to a particular configuration or position of the robot. Then the problem reduces to determining whether it is possible to get from the starting configuration point to the ending configuration point. The transformed obstacles are called *configuration obstacles* or *C-obstacles*.

We also showed that in the special case where the robot $\mathcal{R}$ can be translated but not rotated (i.e., the configuration space is two-dimensional) then, given a work-space obstacle P, the corresponding configuration obstacle $\mathcal{CP}$ is the Minkowski sum $P \oplus (-\mathcal{R}(0, 0))$. We also showed that if P and $\mathcal{R}$ are convex, then it is possible to compute this Minkowski sum by a rotating calipers algorithm in $O(n + m)$ time, where n and m are the number of sides in P and $\mathcal{R}$, respectively.

Complexity of Minkowski Sums: Today we continue to consider the case of translation only (because it is the simplest nontrivial case), and consider the questions of how we can compute a complete description of the configuration space, and what the combinatorial complexity of this description might be.

To begin with, let's see just how bad things might be. Suppose you are given a robot R with m sides and a set of work-space obstacle P with n sides. How many sides might the Minkowski sum $P \oplus R$ have in the worst case? $O(n+m)$? $O(nm)$, even more? The complexity generally depends on what special properties if any P and R have.

Nonconvex Robot and Nonconvex Obstacles: Suppose that both R and P are nonconvex simple polygons. Let m be the number of sides of R and n be the number of sides of P. How many sides might there be in the Minkowski sum $P \oplus R$ in the worst case? We can derive a quick upper bound as follows. First observe that if we triangulate P, we can break it into the union of at most $n-2$ triangles. That is:

$$\begin{aligned} P &= \cup_{i=1}^{n-2} T_i, \\ R &= \cup_{j=1}^{m-2} S_j. \end{aligned}$$

It follows that

$$P \oplus R = \cup_{i=1}^{n-2} \cup_{j=1}^{m-2} (T_i \oplus S_j).$$

Thus, the Minkowski sum is the union of $O(nm)$ polygons, each of constant complexity. Thus, there are $O(nm)$ sides in all of these polygons. The arrangement of all of these line segments can have at most $O(n^2m^2)$ intersection points (if each side intersects with each other), and hence this is an upper bound on the number of vertices in the final result.

Could things really be this bad? Yes they could. Consider the two polygons in the figure below left. There are $O(n^2m^2)$ ways that these two polygons can be "docked", as shown on the right. The Minkowski sum $P \oplus -R$ is shown in the text. Notice that the large size is caused by the number of holes. (It might be argued that this is not fair, since we are not really interested in the entire Minkowski sum, just a single face of the Minkowski sum. Proving bounds on the complexity of a single face is an interesting problem, and the analysis is quite complex.)

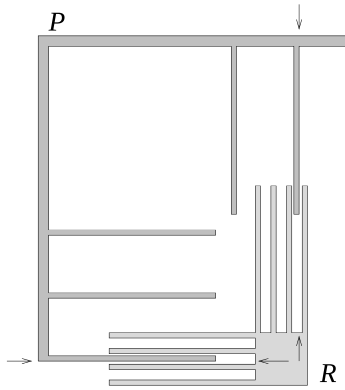


Figure 72: Minkowski sum of $O(n^2m^2)$ complexity.

As a final observation, notice that the upper bound holds even if P (and R for that matter) is not a single simple polygon, but any union of n triangles.

Convex Robot and Nonconvex Obstacles: We have seen that the worst-case complexity of the Minkowski sum might range from $O(n+m)$ to as high as $O(n^2m^2)$, which is quite a gap. Let us consider an intermediate but realistic situation. Suppose that we assume that P is an arbitrary n-sided simple polygon, and R is a convex m-sided polygon. Typically m is much smaller than n. What is the combinatorial complexity of $P \oplus R$ in the worst case? As before we can observe that P can be decomposed into the union of $n-2$ triangles T_i, implying that

$$P \oplus R = \cup_{i=1}^{n-2}(T_i \oplus R).$$

Each Minkowski sum in the union is of complexity $m+3$. So the question is how many sides might there be in the union of $O(n)$ convex polygons each with $O(m)$ sides? We could derive a bound on this quantity, but it will give a rather poor bound on the worst-case complexity. To see why, consider the limiting case of $m = 3$. We have the union of n convex objects, each of complexity $O(1)$. This could have complexity as high as $\Omega(n^2)$, as seen by generating a criss-crossing pattern of very skinny triangles. But, if you try to construct such a counter example, you won't be able to do it.

To see why such a counterexample is impossible, suppose that you start with nonintersecting triangles, and then take the Minkowski sum with some convex polygon. The claim is that it is impossible to generate this sort of criss-cross arrangement. So how complex an arrangement can you construct? We will show the following.

Theorem: Let R be a convex m-gon and P and simple n-gon, then the Minkowski sum $P \oplus R$ has total complexity $O(nm)$.

Is $O(nm)$ an attainable bound? The idea is to go back to our analogy of "scraping" R around the boundary of P. Can we arrange P such that most of the edges of R scrape over most of the n vertices of P? Suppose that R is a regular convex polygon with m sides, and that P has the comb structure shown in the figure below, where the teeth of the comb are separated by a distance at least as large as the diameter of R. In this case R will have many sides scrape across each of the pointy ends of the teeth, implying that the final Minkowski sum will have total complexity $\Omega(nm)$.

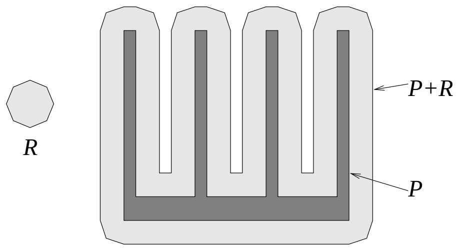


Figure 73: Minkowski sum of $O(nm)$ complexity.

The Union of Pseudodisks: 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 $T_i \oplus R$, where the T_i's have disjoint interiors. The configuration that we want to avoid is the criss-cross pattern shown above. How do we prove that such a pattern cannot be created? The key is in the way the Minkowski sums of disjoint objects can intersect.

We call a pair of convex objects o_1 and o_2 are a *pair of pseudodisks* if the both of the differences $o_1 \backslash o_2$ and $o_2 \backslash o_1$ are connected. In particular, if the objects intersect, then they do not cross through one another.

Note that the being a *pseudodisk* is not a property of a single object, but is the property of pairs of objects. A collection of objects is said to be a *collection of pseudodisks* if each pair is a pair of pseudodisks. The theorem above will follow from the next two lemmas

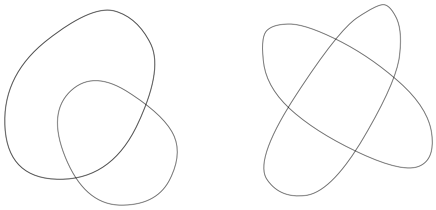


Figure 74: Pseudodisks.

Lemma 1: Given a set convex objects $T_1, T_2, \ldots, T_n$ with disjoint interiors, and convex R, the set

$$\{T_i \oplus R \mid 1 \leq i \leq n\}$$

is a collection of pseudodisks.

Lemma 2: Given a collection of pseudodisks, with a total of n vertices, the complexity of their union is $O(n)$. (In our case, the total number of vertices is $O(nm)$.)

First we prove Lemma 1. Consider two polygons T_1 and T_2 with disjoint interiors. We want to show that $T_1 \oplus R$ and $T_2 \oplus R$ do not cross over one another. There is a very easy way to "visualize" why this is true, but it is not a rigorous proof. Recall that the Minkowski sum $T \oplus R$ is related to scraping the negation $-R$ around the boundary of T. Suppose we scrape $-R$ around the union of T_1 and T_2. If the interiors of these two polygons do not intersect, then either these two scrapings do not intersect each other at all, or else they intersect in exactly two points. But, from convexity, it is "plausible" that there should not be more than two intersections.

Here is a careful proof. Recall from yesterday's lecture that given any directional unit vector $\vec{d}$, the *most extreme* point of R in direction $\vec{d}$ is the point $r \in R$ 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 $T_1 \oplus R$ that is most extreme in direction d is the sum of the points t and r that are most extreme for T_1 and 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.

Now, if to the contrary $T_1 \oplus R$ and $T_2 \oplus R$ 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 $(T_1 \oplus R) \cup (T_2 \oplus R)$ such that $p_1, p_3 \in T_1 \oplus R$ and $p_2, p_4 \in T_2 \oplus R$. 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 R, T_1 and T_2 in direction $\vec{d}$, 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

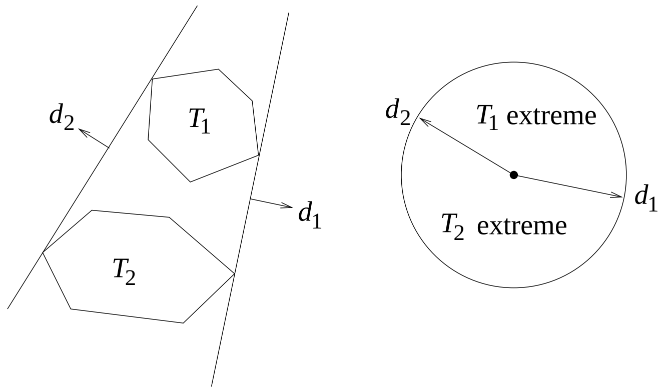


Figure 75: Alternation of extremes.

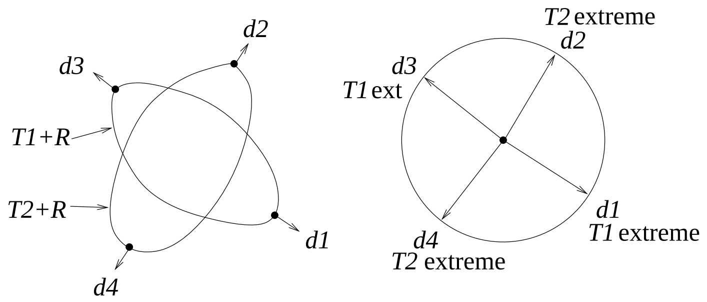


Figure 76: Proof of Lemma 1.

at most one alternation in extreme points for nonintersecting convex polygons. See the figure below.

Next we prove Lemma 2. 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 psuedodisks, 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.

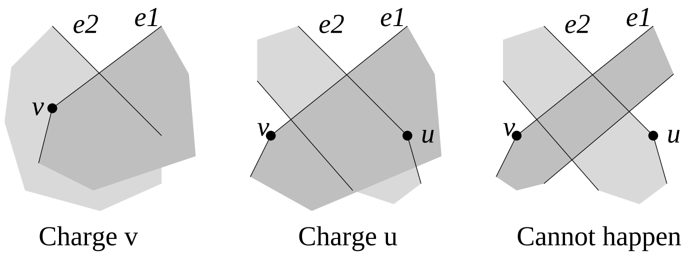


Figure 77: Proof of Lemma 2.

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.

Lecture 28: Final Review

(Tuesday, Dec 9, 1997)

Announcements: Class is cancelled this Thursday. Office hours on Wednesday will be moved up to 2:00–3:00pm. I'll be holding usual office hours on Monday, from 3:00–4:00pm on the day before the final.

Final Exam: Tues, Dec 16, 8:00–10:00am. The exam will be closed-book, closed-notes, but you are allowed two cheat-sheets (front and back).

Before the Midterm: Convex hulls, line segment intersection, planar graphs and DCEL's, polygon triangulation, intersection of halfspaces, linear programming in low dimensions, orthogonal range searching, planar point location, trapezoidal maps. You should be aware of the results, definitions and general techniques, but I will not ask you for detailed information (e.g. simulating Kirkpatrick's algorithm on a data set).

Voronoi diagrams: Recall the definition of Voronoi diagrams. We presented Fortune's algorithm, which operated by sweeping a "distorted" sweep-line. Voronoi diagrams have many applications in problems dealing with distances. More generally, Voronoi diagrams are an example of the notion of subdividing the plane into regions according to some criterion. In this case, we subdivide the plane into regions according to who the nearest neighbor is. It is possible to define other sorts of diagrams. For example, the *2nd-order Voronoi diagram* is a subdivision of the plane according to who your first two nearest neighbors are (and the kth-order diagram is defined similarly for the k nearest neighbors). Also, the furthest point Voronoi diagram is defined according to which point is the furthest from this point. Voronoi diagrams can be computed for line segments as well, and are often used in motion planning applications.

Delaunay Triangulations: This is the dual of the Voronoi diagram. We discussed properties, such as the fact that the Delaunay triangulation maximized the minimum angle. We presented a randomized incremental algorithm for the Delaunay triangulation. Generally, triangulations play an important role in computational geometry since they allow us to reduce complex domains to a collection of simpler domains (triangles). DeFloriani described how any algorithm for incrementally updating a triangulation could be used for generating a hierarchical surface representation (similar to Kirkpatrick's algorithm). You are not responsible for the details of her presentation, but for understanding how update rules for triangulations can be used to generate hierarchical surface representations.

Line Arrangements: We showed that line arrangements together with duality could be used for answering many problems having to do with lines and points in the plane. We presented a simple incremental construction algorithm for arrangements in the plane, proved its $O(n^2)$ running time with the zone theorem. We discussed topological plane sweep as a method for traversing an planar arrangement. Duality and arrangements can also be generalized to higher dimensions.

We presented a number of applications of arrangements. These included sorting angular sequences, computing the narrowest 3-corridor, computing the maximum discrepancy (and discussed the notion of a level in an arrangement). On Homework 4, Problem 2 we also saw that there are problems that can be solved by using the arrangement to "conceptualize" a solution, even though the actual solution may not involve constructing the arrangement. You should probably expect at least one problem on the exam whose solution will require use of arrangements.

Shortest Paths and Visibility Graphs: We discussed the problem of computing shortest paths in the plane, and how visibility graphs could be used to reduce the shortest path problem to a problem on graphs. Notice that using visibility graphs does not extend to three dimensional shortest path problems (since a shortest path may bend anywhere along the interior of an edge). Nonetheless, because graph problems are often much easier to solve, the technique of extracting a graph from a geometric setting, is an important concept. We saw in Homework 4, Problem 4 that some special shortest path problems can be solved without explicitly constructing a visibility graph at all. For example, shortest paths in simple polygons can be solved in $O(n)$ time, once the polygon has been triangulated. (The solution is somewhat more complicated, but still similar to the homework solution.)

Motion Planning: We introduced configuration spaces, and discussed the idea of mapping workspace obstacles into obstacles in configuration space. We discussed Minkowski sums as a method for computing configuration obstacles where translation is involved. We showed how to compute the Minkowski sum of two convex polygons in linear time. We also discussed the combinatorial complexity of the union of a geometric objects, when these objects satisfied special conditions. Although the general complexity could be as high as $O(n^2)$, we showed that if the objects are convex pseudodisks, then the complexity only has complexity $O(n)$.

Lecture X01: Randomized Trapezoidal Decomposition

(Supplemental)

Randomized Incremental Algorithms: We consider the following problem. Given a set of (possibly intersecting) line segments in the plane, subdivide the plane into a collection of trapezoids, formed by shooting a bullet to the left and right of each vertex and each intersection point until it hits the first object. (Following Mulmuley's presentatio

The running time of this algorithm will be $O(k + n \log n)$, with *high probability*, where n is the number of segments and k is the number of intersection points between the segments. More succinctly, the running time is $\tilde{O}(k + n \log n)$. Mulmuley uses the notation $f(n) \in \tilde{O}(g(n))$ if, for all sufficiently large n, $f(n) < cg(n)$ for some constant c, with probability $1 - 1/p(n)$, where $p(n)$ is a polynomial whose degree increases with c. Thus, we can reduce the probability that the running time fails to be $O(g(n))$ as low as we like, up to the reciprocal of high degree polynomial function of n.) Observe that this running time is asymptotically better than the previous, in the sense that we have removed the $\log n$ multiplicative factor from in front of the k. The price we have paid, is that the running time is now randomized, and so not guaranteed.

Randomized Incremental Algorithm: The basic idea of the algorithm is described below.

(1) Randomly *permute* the set of segments. Let N^i denote the first i segments. In general, $H(N^i)$ will denote the trapezoidal decomposition of the first i segments.

(2) Initialize the structure to a trivial "empty" trapezoidal decomposition (e.g. a large empty enclosing box), $H(N^0)$.

(3) One by one add the segments in random order. For each segment, do the following.

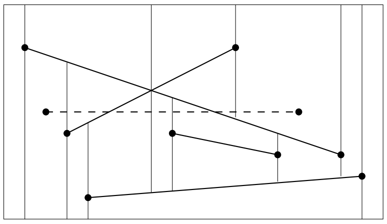
   (a) *Locate* the trapezoid of $H(n^{i-1})$ containing the left endpoint of the segment. (There are a couple of ways to do this. More on this later.)

   (b) *Trace* the segment from one trapezoid to the next. At the endpoints of the segment, *split* the current trapezoid by adding a vertical wall. For each trapezoid determine the point of entry and point of exit of the segment. If the segment crosses the top or bottom then create new intersection points, and *split* the trapezoid by adding a vertical wall.

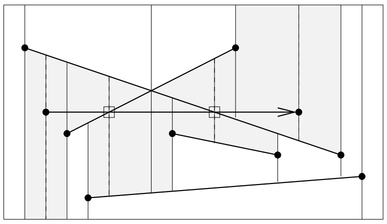
   (c) After tracing the segment, determine the vertical walls of $H(N^{i-1})$ that were stabbed by the segment. For each such wall, *trim* it back to the portion containing its supporting vertex. This results in the *merger* of two adjacent trapezoids.

The implementation of the algorithm relies on the ability to perform the necessary local manipulations of the planar graph representing the trapezoidal decomposition. In particular, we assume that we can (1) trace a segment through a face of the decomposition, (2) split a face (by the addition of a vertical wall) and (3) merge two adjacent faces (when a vertical wall is trimmed) in time proportional to the complexity of the face, that is, the number of vertices (equivalently edges) on the face. We let f denote a trapezoidal face of the decomposition, and let $c(f)$ denote the complexity of this face. Observe that the running time of the above algorithm is $O(c(f))$ for each face that is traced by this procedure, because there can only be a constant number (at most 2) split performed for each trapezoid, and a constant number (at most 2) merges for each trapezoid.

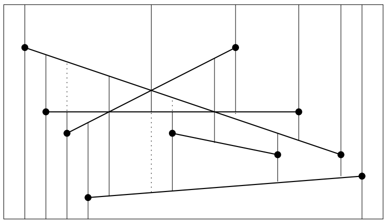
The other question we left open was how to locate the left-endpoint from which to start the tracing. We do this using a simple *bucketing* strategy. For each of the n left-endpoints, we store which trapezoid of the current subdivision contains this point, and with each trapezoid f we associate a list $L(f)$, which contains the segments whose left-endpoint lies within this face. When we wish to trace a segment, we can determine the trapezoid containing this segment



Initial decomposition.



Trace and split.



Trim and merge.

Figure 78: Randomized incremental trapezoidal decomposition.

in constant time. When a segment intersects a trapezoid f, it may split a trapezoid f into a constant number of new trapezoids. We can walk through the list $L(f)$ and determine which of the new trapezoids contains a given point in constant time each. We form new left-endpoint lists for each of the new faces, $L(f_1)$, $L(f_2)$, etc. When we merge two trapezoids, we simply concatenate their lists. Observe that both of these operations can be performed in time $O(l(f))$, where $l(f) = |L(f)|$.

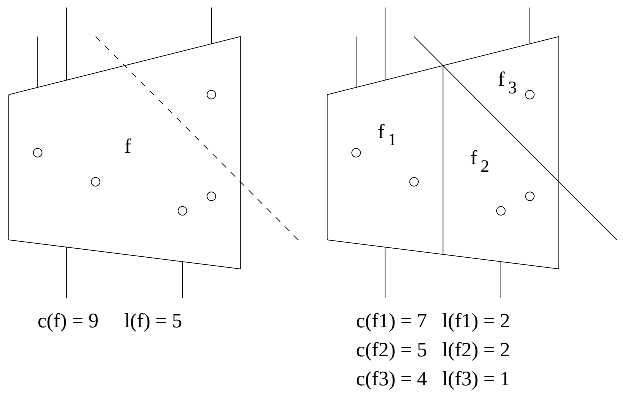


Figure 79: Splitting a face.

Lemma: Consider the insertion of segment into $H(N^{i-1})$ which intersects faces $F = \{f_1, f_2, \ldots, f_k\}$ in the decomposition. The time to insert this segment is

$$O\left(\sum_{f \in F} (c(f) + l(f))\right).$$

Analysis: Analyzing the expected case running time of this algorithm is quite a tricky task. We want to show that, no matter what n segments are given initially, if we randomize of all possible insertion orders, the total expected time to build the decomposition is $O(k + n \log n)$. (We will only prove that this is an expected bound. The proof that this holds with high probability takes some additional work.) In particular, we seem to need to be able to determine the expected value of $\sum_f (c(f) + l(f))$ over all segments that might be inserted next.

This task would be quite daunting, if it were not for a very clever analysis trick, called *backward analysis*. Here is the idea: we imagine that we are running time algorithm backwards, deleting segments one at a time. (Whether we count backwards or forwards makes no difference. The running time of a given stage is still given by the lemma above.) Observe that if all insertion orders are equally likely, then the last segment to be deleted in the reversed algorithm is equally likely to be any one of the segments that already exists in the decomposition $H(N^i)$. Since every segment of N^i is equally likely to be deleted, to determine the expected time for the i-th stage, it suffices to average over all possible segments i to be deleted, and for each determine the complexity if this were the segment chosen.

When we delete some segment from the $H(N^i)$, observe that the only trapezoids that would be affected from its deletion consist of the trapezoids of $H(N^i)$ that are adjacent to this segment. An example is shown in the figure below.

Lemma: For each j, $1 \leq j \leq i$, let F_j denote the faces that the j-th segment intersects in $H(N^i)$. The expected time for the last stage in the construction of $H(N^i)$ is on the order

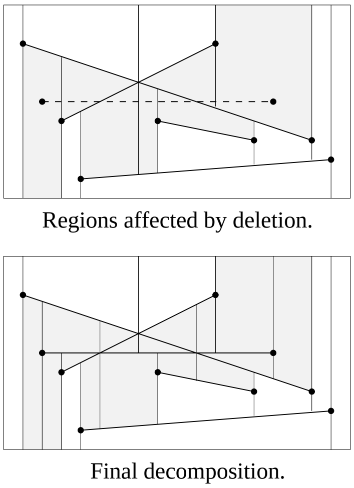


Figure 80: Affected trapezoids.

of

$$E(i) = \frac{1}{i}\sum_{j=1}^{i}\sum_{f\in F_j}(c(f)+l(f)).$$

We seem to be no closer to our goal, since there does not appear to be an easy way to analyze the sums of complexities of all the trapezoids that a given segment intersects. However, the crucial trick is not to count segment by segment, but to count trapezoid by trapezoid. Assuming that the segments are in general position, observe the following important fact.

Lemma: (Bounded degree property) Each trapezoid in a trapezoidal decomposition is adjacent to at most 4 segments. (Namely the segments immediately above and below, and the segments which touch the left and right walls of the trapezoid (either because of an endpoint or an intersection point).

Thus if we take the complexity of each trapezoid, and multiply by 4, we will have an upper bound on the sum of complexities of all the trapezoids that are intersected by all segments.

$$\begin{aligned} E(i) &\le \frac{1}{i}\sum_{f\in H(N^i)} 4(c(f)+l(f)) \\ &\le \frac{4}{i}\left(\sum_{f\in H(N^i)} c(f) + \sum_{f\in H(N^i)} l(f)\right). \end{aligned}$$

Now, this is something we can analyze. We know the value of $\sum_f c(f)$ is the sum of all the edges summed over all the faces in the decomposition. However, this counts every edge in the decomposition twice. Since the decomposition is a planar graph, this is proportional to the number of vertices in the decomposition currently. The value of $\sum_f l(f)$ is just equal to $n-i$, since it includes all the left endpoints of segments that have yet to be added. If we let k_i denote the number of intersection points at the current stage of the decomposition, the number of vertices in $H(N^i)$ is just $2i+k_i$, so the above formula simplifies to

$$E(i) \le \frac{4}{i}(2i+k_i+n-i)$$

$$\leq \frac{c(n+k_i)}{i}.$$

for some constant c. The interesting thing at this point is that, although our analysis was conditional on the structure of $H(N^i)$, the resulting bound is almost entirely independent of this structure. (Only the value k_i is dependent.)

At this point we can see where the analysis is going. If we ignore the k_i term in the above, we see that $E(i) \leq cn/i$. To get the total time, we sum these up, getting at total expected time of

$$\begin{aligned} TE(n) &= \sum_{i=1}^{n} E(i) \\ &\leq \sum_{i=1}^{n} \frac{cn}{i} \\ &= cn \sum_{i=1}^{n} \frac{1}{i} \\ &\approx cn \ln n \in O(n \log n). \end{aligned}$$

The last line uses the well known fact that the Harmonic series, $\sum_{1 \leq i \leq n} 1/i$ tends to $\ln n$.

We need to get a handle on what k_i is expected to be. Obviously the value of k_i should eventually approach k, the total number of intersections as i approaches n. The interesting fact is that this quantity approaches k quadratically, not linearly.

Lemma: For fixed $i \geq 0$, the expected value of k_i, assuming that N^i is a random sample of N of size i, is $O(ki^2/n^2)$.

Proof: For each intersection point v between two segments, s_1 and s_2, let I_v denote the random variable that this 1 if this intersection is part of the decomposition $H(N^i)$ and 0 otherwise. Observe that the expected value of k_i is $\sum_v I_v$. However, v occurs within I_v if and only if both s_1 and s_2 occur within the first i randomly selected segments. The probability that each one has been selected alone is i/n. The probability that both have been selected is roughly i^2/n^2 (for i and n large). Thus the expected value of k_i is k times this quantity, or $O(ki^2/n^2)$.

To complete the analysis, we make use of another basic fact from summations. $\sum_{1 \leq i \leq n} i \approx n^2/2$.

$$\begin{aligned} TE(n) &= \sum_{i=1}^{n} E(i) \\ &\leq \sum_{i=1}^{n} \frac{c(n+k_i)}{i} \\ &= \sum_{i=1}^{n} \frac{cn}{i} + \sum_{i=1}^{n} \frac{ck_i}{i} \\ &= cn \sum_{i=1}^{n} \frac{1}{i} + c \sum_{i=1}^{n} \frac{ki^2}{n^2 i} \\ &\approx cn \ln n + \frac{ck}{n^2} \sum_{i=1}^{n} i \\ &\approx cn \ln n + ck/2 \\ &\in O(n \log n + k). \end{aligned}$$

Lecture X02: Voronoi Diagrams

(Supplemental)
Read: 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.

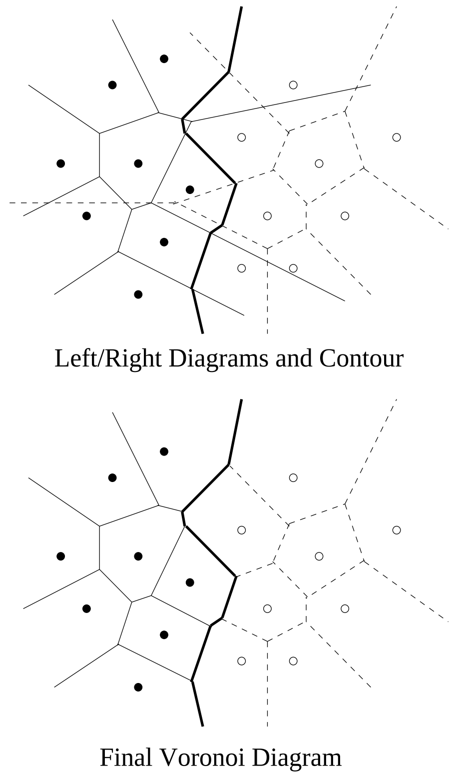


Figure 81: Merging Voronoi diagrams.

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.

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

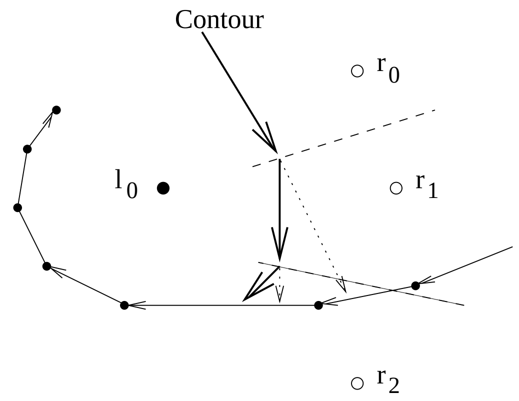


Figure 82: Tracing the contour.

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 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 X03: Delaunay Triangulations and Convex Hulls

(Supplemental)
Read: 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.

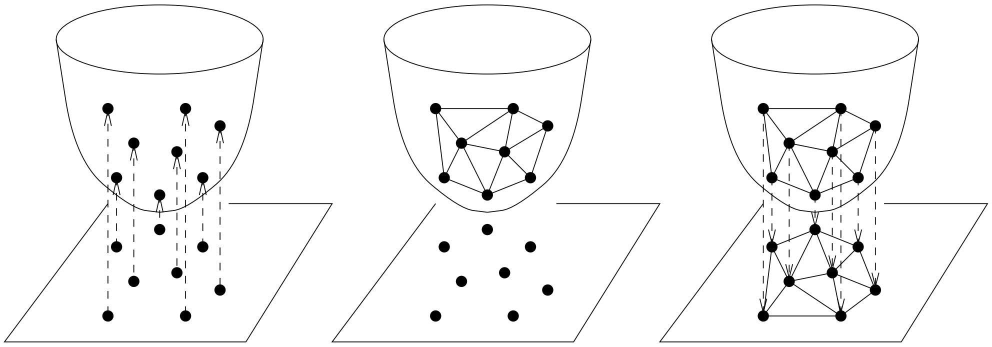


Figure 83: Delaunay triangulations and convex hull.

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.

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.

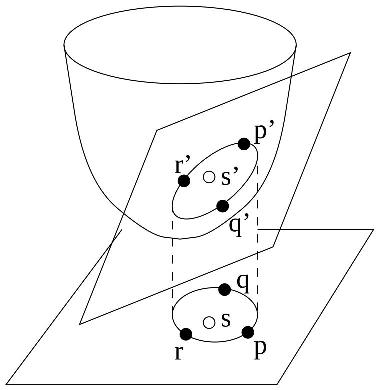


Figure 84: Planes and circles.

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.

$$\mathrm{in}(p,q,r,s) \;=\; \det\begin{pmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{pmatrix} > 0.$$

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'.

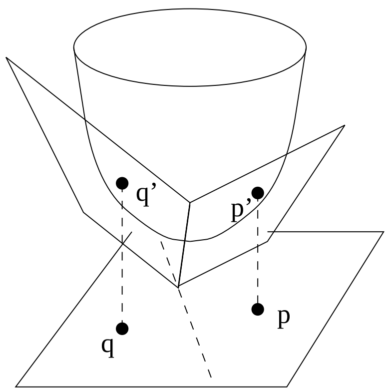


Figure 85: Intersection of halfspaces.

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 X04: Topological Plane Sweep

(Supplemental)

Read: 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 $1 \le i \le n-1$, 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.

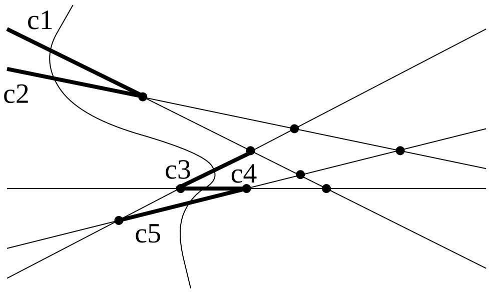


Figure 86: Topological line and associated cut.

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 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.

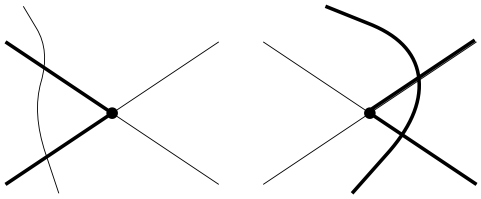


Figure 87: Elementary step.

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.

The important things about the UHT and LHT is that they give us an easy way to determine

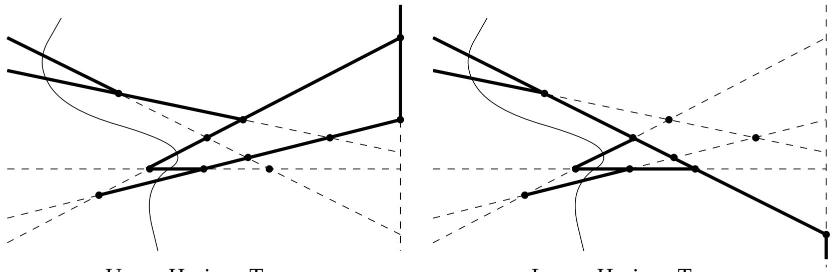


Figure 88: Upper and lower horizon trees.

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.

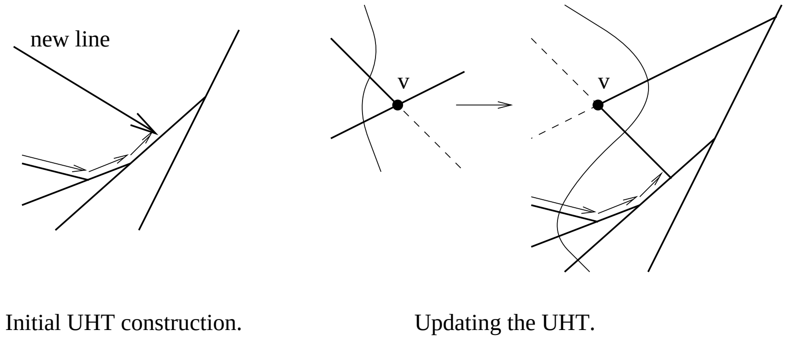


Figure 89: Constructing and updating the UHT.

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 X05: Ham-Sandwich Cuts

(Supplemental)

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.

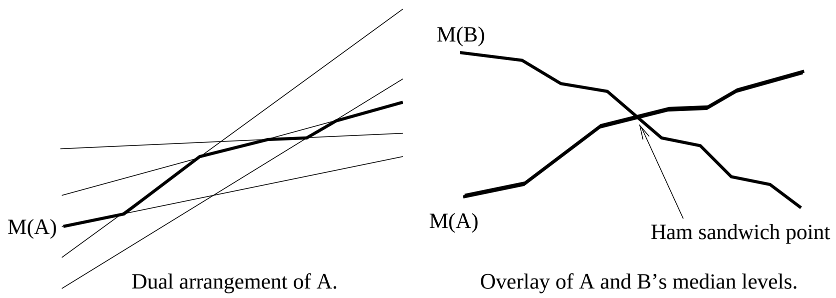


Figure 90: Ham sandwich: Dual formulation.

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.

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.