# CMSC 754
# Computational Geometry[1]

David M. Mount
Department of Computer Science
University of Maryland
Spring 2012

# Lecture 1: Introduction to Computational Geometry

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

The field of computational geometry developed rapidly in the late 70's and through the 80's and 90's, and it still continues to develop. Historically, computational geometry developed as a generalization of the study of algorithms for sorting and searching in 1-dimensional space to problems involving multi-dimensional inputs. Because of its history, the field of computational geometry has focused mostly on problems in 2-dimensional space and to a lesser extent in 3-dimensional space. When problems are considered in multi-dimensional spaces, it is usually assumed that the dimension of the space is a small constant (say, 10 or lower). Nonetheless, recent work in this area has considered a limited set of problems in very high dimensional spaces, particularly with respect to approximation algorithms. In this course, our focus will be largely on problems in 2-dimensional space, with occasional forays into spaces of higher dimensions.

Because the field was developed by researchers whose training was in discrete algorithms (as opposed to numerical analysis) the field has also focused more on the discrete nature of geometric problems (combinatorics and topology, in particular), as opposed to continuous issues. The field primarily deals with straight or flat objects (lines, line segments, polygons, planes, and polyhedra) or simple curved objects such as circles. This is in contrast, say, to fields such as solid modeling, which focus on issues involving curves and surfaces and their representations.

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.

**A Typical Problem in Computational Geometry:** Here is an example of a typical problem, called the *shortest path problem*. Given a set polygonal obstacles in the plane, find the shortest obstacle-avoiding path from some given start point to a given goal point (see Fig. 1). 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.

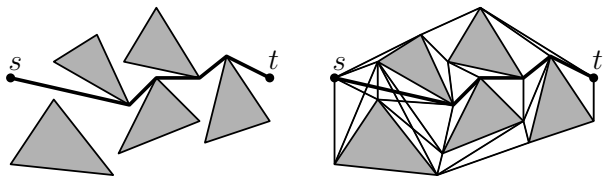



Fig. 1: Shortest path problem.

The measure of the quality of an algorithm in computational geometry has traditionally been its *asymptotic worst-case running time*. Thus, an algorithm running in $O(n)$ time is better than one running in $O(n \log n)$ time which is better than one running in $O(n^2)$ time. (This particular problem can be solved in $O(n^2 \log n)$ time by a fairly simple algorithm, in $O(n \log n)$ by a relatively complex algorithm, and it can be approximated quite well by an algorithm whose running time is $O(n \log n)$.) In some cases *average case* running time is

considered instead. However, for many types of geometric inputs (this one for example) it is difficult to define input distributions that are both easy to analyze and representative of typical inputs.

**Strengths Computational Geometry:**

**Development of Geometric Tools:** Prior to computational geometry, there were many *ad hoc* solutions to geometric computational problems, some efficient, some inefficient, and some simply incorrect. Because of its emphasis of mathematical rigor, computational geometry has made great strides in establishing correct, provably efficient algorithmic solutions to many of these problems.

**Emphasis on Provable Efficiency:** Prior to the development of computational geometry little was understood about the computational complexity of many geometric computations. For example, given an encoding of all the zip code regions in the USA, and given a latitude and longitude from a GPS device, how long should it take to compute the zip code associated with the location? How should the computation time depend on the amount of preprocessing time and space available? Computational geometry put such questions on the firm grounding of asymptotic complexity, and in some cases it has been possible to prove that algorithms discovered in this area are optimal solutions.

**Emphasis on Correctness/Robustness:** Prior to the development of computational geometry, many of the software systems that were developed were troubled by bugs arising from the confluence of the continuous nature of geometry and the discrete nature of computation. For example, given two line segments in the plane, do they intersect? This problem is remarkably tricky to solve since two line segments may arise from many different configurations: lying on parallel lines, lying on the same line, touching end-to-end, touching as in a T-junction. Software that is based on discrete decisions involving millions of such intersection tests may very well fail if any one of these tests is computed erroneously. Computational geometry research has put the robust and correct computing of geometric primitives on a solid mathematical foundations.

**Linkage to Discrete Combinatorial Geometry:** The study of new solutions to computational problems has given rise to many new problems in the mathematical field of discrete combinatorial geometry. For example, consider a polygon bounded by $n$ sides in the plane. Such a polygon might be thought of as the top-down view of the walls in an art gallery. As a function of $n$, how many "guarding points" suffice so that every point within the polygon can be seen by at least one of these guards. Such combinatorial questions can have profound implications on the complexity of algorithms.

**Limitations of Computational Geometry:**

**Emphasis on discrete geometry:** There are some fairly natural reasons why computational geometry may never fully address the needs of all these applications areas, and these limitations should be understood before undertaking this course. One is the discrete nature of computational geometry. There are many applications in which objects are of a very continuous nature: computational physics, computational fluid dynamics, motion planning.

**Emphasis on flat objects:** Another limitation is the fact that computational geometry deals primarily with straight or flat objects. To a large extent, this is a consequence of CG'ers interest in discrete geometric complexity, as opposed to continuous mathematics. Another issues is that proving the correctness and efficiency of an algorithm is only possible when all the computations are well defined. Many computations on continuous objects (e.g., solving differential and integral equations) cannot guarantee that their results are correct nor that they converge in specified amount of time. Note that 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.

**Emphasis on low-dimensional spaces:** One more limitation is that computational geometry has focused primarily on 2-dimensional problems, and 3-dimensional problems to a limited extent. The nice thing about 2-dimensional problems is that they are easy to visualize and easy to understand. But many of the daunting

applications problems reside in 3-dimensional and higher dimensional spaces. Furthermore, issues related to topology are much cleaner in 2- and 3-dimensional spaces than in higher dimensional spaces.

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

**Convex Hulls:** Convexity is a very important geometric property. A geometric set is *convex* if for every two points in the set, the line segment joining them is also in the set. One of the first problems identified in the field of computational geometry is that of computing the smallest convex shape, called the *convex hull*, that encloses a set of points (see Fig. 2).

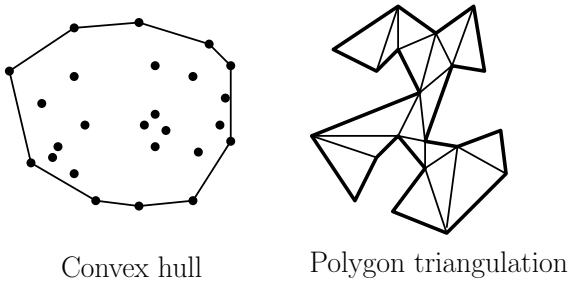



Fig. 2: Convex hulls and polygon triangulation.

**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.

**Voronoi Diagrams and Delaunay Triangulations:** Given a set $S$ of points in space, one of the most important problems is the nearest neighbor problem. Given a point that is not in $S$ which point of $S$ is closest to it? One of the techniques used for solving this problem is to subdivide space into regions, according to which point is closest. This gives rise to a geometric partition of space called a *Voronoi diagram* (see Fig. 3). This geometric structure arises in many applications of geometry. The dual structure, called a *Delaunay triangulation* also has many interesting properties.

**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 (see Fig. 4). We will show that such a structure can be constructed in $O(n^2)$ time.

The reason that this structure is so important is that many problems involving points can be transformed into problems involving lines by a method of *point-line duality*. In the plane, this is a transformation that

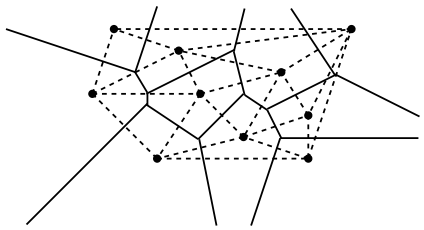


Fig. 3: Voronoi diagram and Delaunay triangulation.

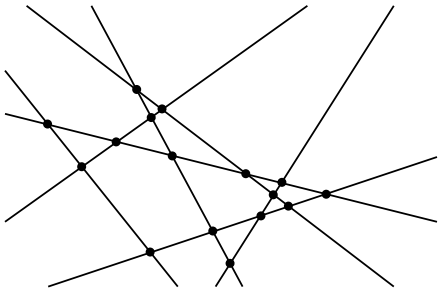


Fig. 4: An arrangement of lines in the plane.

maps lines to points and points to lines (or generally, $(d-1)$-dimensional hyperplanes in dimension $d$ to points, and vice versa). For example, suppose that you want to determine whether any three points of a planar point set are collinear. This could be determined 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 four in the arrangement.

**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, consider the following problem, called *point location*. Given a subdivision of space (e.g., a Delaunay triangulation), determine the face of the subdivision that contains a given query point. Another geometric search problem is the *nearest neighbor problem*: given a set of points, determine the point of the set that is closest to a given query point. Another example is *range searching*: given a set of points and a shape, called a range, either count of report the subset of points lie within the given region. The region may be a rectangle, disc, or polygonal shape, like a triangle.

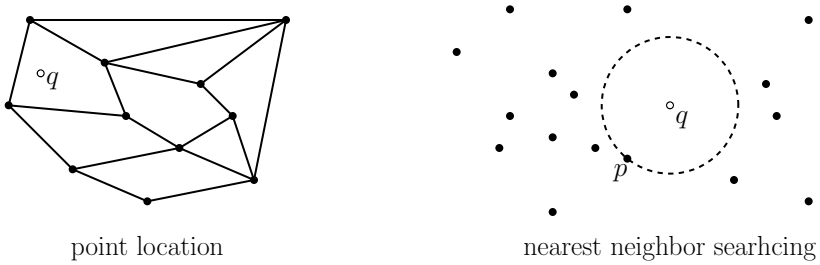



Fig. 5: Geometric search problems. The point-location query determines the triangle containing $q$. The nearest-neighbor query determines the point $p$ that is closest to $q$.

**Approximation:** In many real-world applications geometric inputs are subject to measurement error. In such cases it may not be necessary to compute results exactly, since the input data itself is not exact. Often the ability to produce an approximately correct solution leads to much simpler and faster algorithmic solutions.

Consider for example the problem of computing the diameter (that is, the maximum pairwise distance) among a set of $n$ points in space. In the plane efficient solutions are known for this problem. In higher dimensions it is quite hard to solve this problem exactly in much less than the brute-force time of $O(n^2)$. It is easy to construct input instances in which many pairs of points are very close to the diametrical distance. Suppose however that you are willing to settle for an approximation, say a pair of points at distance at least $(1-\varepsilon)\Delta$, where $\Delta$ is the diameter and $\varepsilon > 0$ is an approximation parameter set by the user. There exist algorithms whose running time is nearly linear in $n$, assuming that $\varepsilon$ is a fixed constant. As $\varepsilon$ approaches zero, the running time increases.

# Lecture 2: Warm-Up Problem: Computing Slope Statistics

**Slope Statistics:** Today, we consider a simple warm-up exercise as an example of a typical problem in computational geometry. To motivate the problem, imagine that a medical experiment is run, where the therapeutic benefits of a certain treatment regimen is being studied. A set of $n$ points in real 2-dimensional space, $\mathbb{R}^2$, is given. We denote this set by $P = \{p_1, \ldots, p_n\}$, where $p_i = (a_i, b_i)$, where $a_i$ indicates the amount of treatment and $b_i$ indicates the therapeutic benefit. The hypothesis is that increasing the amount of treatment by $\Delta a$ units results in an increase in therapeutic benefit of $\Delta b = s(\Delta a)$, where $s$ is an unknown scale factor.

In order to study the properties of $s$, a statistician considers the set of slopes of the lines joining pairs of a points (since each slope represents the increase in benefit for a unit increase in the amount of treatment). For $1 \le i < j \le n$, define

$$s_{i,j} \;=\; \frac{b_j - b_i}{a_j - a_i},$$

(see Fig. 6(a)). So that we don't need to worry about infinite slopes, let us make the simplifying assumption that the $a$-coordinates of the points are pairwise distinct, and to avoid ties, let us assume that the slopes are distinct. Let $S = \{s_{i,j} \mid 1 \le i < j \le n\}$. Clearly $|S| = \binom{n}{2} = n(n-1)/2 = O(n^2)$. Although the set $S$ of slopes is of quadratic size, it is defined by a set of $n$ points. Thus, a natural question is whether we can answer statistical questions about the set $S$ in time $O(n)$ or perhaps $O(n \log n)$, rather than $O(n^2)$.

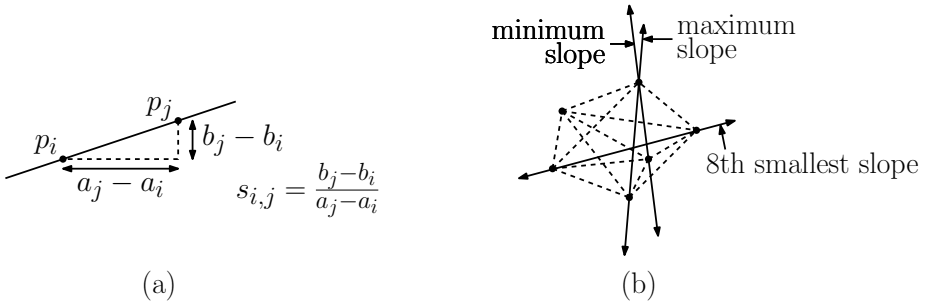



Fig. 6: The slope $s_{i,j}$ and the slope set $S = \{s_{i,j} \mid 1 \le i < j \le n\}$.

Here are some natural questions we might ask about the set $S$ (see Fig. 6(b)):

**Min/Max:** Compute the minimum or maximum slope of $S$.

**$k$-th Smallest:** Compute the $k$-smallest element of $S$, given any $k$, $1 \le k \le \binom{n}{2}$.

**Average:** Compute the average of the elements of $S$.

**Range counting:** Given a pair of reals $s^- \le s^+$, return a count of the number of elements of $S$ that lie in the interval $[s^-, s^+]$.

**Counting Negative Slopes and Inversions:** In this lecture we will consider the last problem, that is, counting the number of slopes that lie within a given interval $[s^-, s^+]$. Before considering the general problem, let us consider a simpler version by considering the case where $s^- = 0$ and $s^+ = +\infty$. In other words, we will count the number of pairs $(i, j)$ where $s_{i,j}$ is nonnegative. This problem is interesting statistically, because it represents the number of instances in which increasing the amount of treatment results in an increase in the therapeutic benefit.

Our approach will be to count the number of pairs such that $s_{i,j}$ is strictly negative. There is no loss of generality in doing this, since we can simply subtract the count from $\binom{n}{2}$ to obtain the number of nonnegative slopes. (The reason for this other formulation is that it will allow us to introduce the concept of inversion counting, which will be useful for the general problem.) It will simplify the presentation to make the assumption that the sets of $a$-coordinates and $b$-coordinates are distinct.

Suppose we begin by sorting the points of $P$ in increasing order by their $a$-coordinates. Let $P = \langle p_1, \ldots, p_n \rangle$ be the resulting ordered sequence, and let $B = \langle b_1, \ldots, b_n \rangle$ be the associated sequence of $b$-coordinates. Observe that, for $1 \le i < j \le n$, $b_i > b_j$ if and only if $s_{i,j}$ is negative. For $1 \le i < j \le n$, we say that the pair $(i, j)$ is an *inversion* for $B$ if $b_i > b_j$. Clearly, our task reduces to counting the number of inversions of $B$ (see Fig. 7(a)).

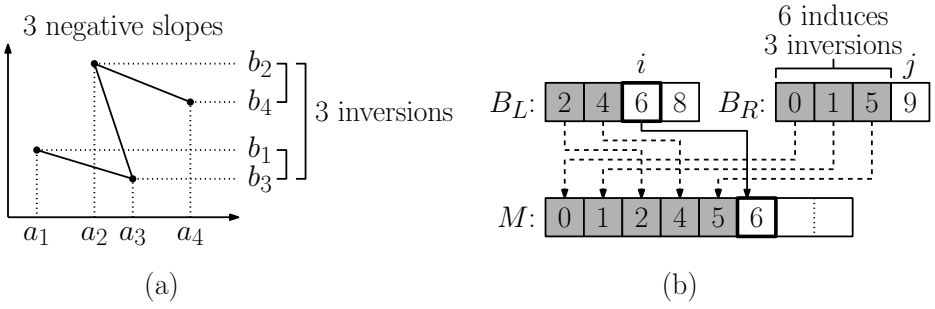



Fig. 7: Inversion counting and application to counting negative slopes.

**Inversion Counting:** Counting the number of inversions in a sequence of $n$ numbers is a simple exercise, which can be solved in $O(n \log n)$ time. Normally, such exercises will be left for you to do, but since this is the first time to present an algorithm, let's do it in full detail.

The algorithm is a simple generalization of the MergeSort algorithm. Recall that MergeSort is a classical example of divide-and-conquer. The sequence is partitioned into a left and right subsequence, denoted $B_L$ and $B_R$, each of size roughly $n/2$. These two subsequences are sorted recursively, and then the resulting sorted sequences are then merged to form the final sorted sequence.

To generalize this to inversion counting, in addition to returning the sorted subsequences, the recursive calls return the counts $I_L$ and $I_R$ of the inversions *within* each of the subsequences. In the merging process we count the inversions $I$ that occur *between* the two subsequences. That is, for each element of $B_L$, we compute the number of smaller elements in $B_R$, and add these to $I$. In the end, we return the total number of inversions, $I_L + I_R + I$.

The algorithm is presented in the code block below. To merge the subsequences, we maintain two indices $i$ and $j$, which indicate the current elements of the respective subsequences $B_L$ and $B_R$. We repeatedly[2] copy the smaller of $B_L[i]$ and $B_R[j]$ to the merged sequence $M$. Because both subsequences are sorted, when we copy $B_L[i]$ to $M$, $B_L[i]$ is inverted with respect to the elements $B_R[1 \ldots j-1]$, whose values are smaller than it (see Fig. 7(b)). Therefore, we add $j - 1$ to the count $I$ of inversions.

The main loop stops either when $i$ or $j$ exceeds the number of elements in its subsequence. When we exit, one of the two subsequences is exhausted. We append the remaining elements of the other subsequence to $M$. In

[2]More formally, we maintain the invariant that $B_L[i] > B_R[j']$ for $1 \le j' \le j-1$ and $B_R[j] \ge B_L[i']$ for $1 \le i' \le i-1$.

particular, if $i \leq |B_L|$, we append the remaining $|B_L| - i + 1$ elements of $B_L$ to $M$. Since these elements are all larger than any element of $B_R$, we add $(|B_L| - i + 1)|B_R|$ to the inversion counter. (When copying the remaining elements from $B_R$, there is no need to modify the inversion counter.) See the code block below for the complete code.

Inversion Counting

**InvCount**$(B)$ [**Input:** a sequence $B$; **Output:** sorted sequence $M$ and inversion count $I$.]

(1) Partition $B$ into disjoint subsets $B_L$ and $B_R$, each of size at most $\lceil n/2 \rceil$, where $n = |B|$;
(2) $(B_L, I_L) \leftarrow$ InvCount$(B_L)$;
$(B_R, I_R) \leftarrow$ InvCount$(B_R)$;
(3) Let $i \leftarrow j \leftarrow 1$; $I \leftarrow 0$; $M \leftarrow \emptyset$;
(4) While $(i \leq |B_L|$ and $j \leq |B_R|)$
  (a) if $(B_L[i] \leq B_R[j])$ append $B_L[i{+}{+}]$ to $M$ and $I \leftarrow I + (j - 1)$;
  (b) else append $B_R[j{+}{+}]$ to $M$;
  On exitting the loop, either $i > |B_L|$ or $j > |B_R|$.
(5) If $i \leq |B_L|$, append $B_L[i \ldots]$ to $M$ and $I \leftarrow I + (|B_L| - i + 1)|B_R|$;
(6) Else (we have $j \leq |B_R|$), append $B_R[j \ldots]$ to $M$;
(7) return $(M, I_L + I_R + I)$;

The running time exactly matches that of MergeSort. It obeys the well known recurrence $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$.

By combining this with the above reduction from slope range counting over negative slopes, we obtain an $O(n \log n)$ time algorithm for counting nonnegative slopes.

**General Slope Range Counting and Duality:** Now, let us consider the general range counting problem. Let $[s^-, s^+]$ be the range of slopes to be counted. It is possible to adapt the above inversion-counting approach, subject to an appropriate notion of "order". In order to motivate this approach, we will apply a geometric transformation that converts the problem into a form where this order is more apparent. This transformation, called *point-line duality* will find many uses later in the semester.

To motivate duality, observe that a point in $\mathbb{R}^2$ is defined by two coordinates, say $(a, b)$. A nonvertical line line in $\mathbb{R}^2$ can also be defined by two parameters, a slope and $y$-intercept. In particular, we associate a point $p = (a, b)$ with the line $y = ax - b$, whose slope is $a$ and whose $y$-intercept is $-b$. This line is called $p$'s *dual* and is denoted by $p^*$. (The reason for the negating the intercept will become apparent shortly.) Similarly, given any nonvertical line in $\mathbb{R}^2$, say $\ell : y = ax - b$, we define its *dual* to be the point $\ell^* = (a, b)$. Note that the dual is a involutory (self-inverse) mapping, in the sense that $(p^*)^* = p$ and $(\ell^*)^* = \ell$.

Later in the semester we will discuss the various properties of the dual transformation. For now, we need only a property. Consider two points $p_i = (a_i, b_i)$ and $p_j = (a_j, b_j)$. The corresponding dual lines are $p_i^* : y = a_i x - b_i$ and $p_j^* : y = a_j x - b_j$, respectively. Assuming that $a_i \neq a_j$ (that is, the lines are not parallel), we can compute the $x$-coordinate of their intersection point by equating the right-hand sides of these two equations, which yields

$$a_i x - b_i \;=\; a_j x - b_j \qquad \Rightarrow \qquad x = \frac{b_j - b_i}{a_j - a_i}.$$

Interestingly, this is just $s_{i,j}$. In other words, we have the following nice relationship: *Given two points, the $x$-coordinate of the intersection of their dual lines is the slope of the line passing through the points* (see Fig. 8). (The reason for negating the $b$ coordinate is now evident. Otherwise, we would get the negation of the slope.)

**Slope Range Counting in the Dual:** Based on the above observations, we see that the problem of counting the slopes of $S$ that lie within the interval $[s^-, s^+]$ can be reinterpreted in the following equivalent form. Given a set of $n$ nonvertical lines in $\mathbb{R}^2$ and given an interval $[s^-, s^+]$, count the pairs of lines whose intersections lie within the vertical *slab* whose left side is $x = s^-$ and whose right side is $s^+$ (see Fig. 9(a)).

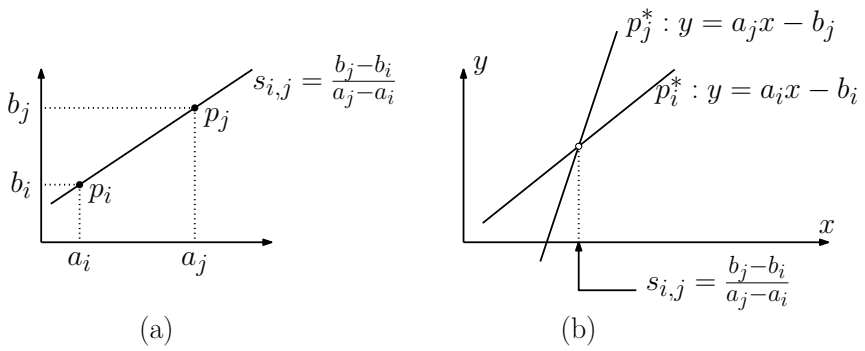



Fig. 8: Point-line duality and the relationship between the slope of a line between two points and the $x$-coordinate of the duals of the two points.

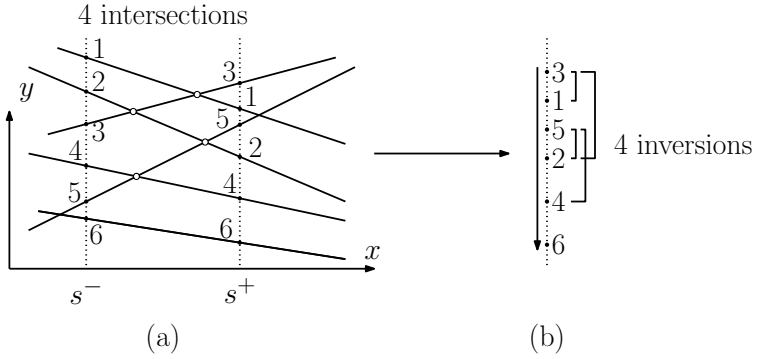



Fig. 9: Intersections in the vertical slab $[s^-, s^+]$ and inversion counting.

How can we count the number of such intersection points efficiently? Again, this can be done through inversion counting. To see this, observe that two lines intersect within the slab if and only if the order of their intersection with the left side of the slab is the inverse of their intersection with the right side of the slab.

We can reduce the problem to inversion counting, therefore, as follows. First, consider the order in which the lines intersect the left side of the slab (taken from top to bottom). In particular, the line $y = a_i x - b_i$ intersects at the point $y = a_i s^- - b_i$. Sort the lines according in decreasing order of these $y$-coordinates, thus obtaining the order from top to bottom, and renumber them from 1 to $n$ according to this order (see Fig. 9(a)). Next, compute the order in which the (renumbered) lines intersect the right side of the slab. In particular, line $i$ is associated with the value $y = a_i s^+ - b_i$. Letting $Y = \langle y_1, \ldots, y_n \rangle$ denote the resulting sequence, it is easy to see that the number of inversions in $-Y$ is equal to the number of pairs of lines that intersect within the slab. The time to compute the intersection along the left side and sort according to this order is $O(n \log n)$, and the time to compute the intersections with the right side and count the inversions is also $O(n \log n)$. Therefore, the total running time is $O(n \log n)$.

**Negative Slope Range Counting Revisited:** By the way, you might wonder what the earlier instance of counting negative slopes maps to in this instance. In this case the interval is $[-\infty, 0]$. Observe that a vertical line at $x = -\infty$ (from top to bottom) intersects the lines in increasing order of slope, or equivalently, in order of $a$-coordinates. Thus, sorting the points from top to bottom order by their intersection with $s^- = -\infty$ is equivalent to the sorting by $a$-coordinates, which is just what we we did in the case of negative slopes.

The right side of the slab is determined by the top-to-bottom order of intersections of the lines with vertical line at $x = 0$. Clearly, line $i$ intersects this vertical at $y = -b_i$. Therefore, counting the inversions of the sequence $-Y = \langle -y_1, \ldots, -y_n \rangle$ is equivalent to the process of counting inversions in the sequence $B = \langle b_1, \ldots, b_n \rangle$, exactly as we did before. Thus, the case of counting negative slopes can indeed be seen to be a special case of this algorithm.

**Review:** In summary, we have seen how an apparently 2-dimensional geometric problem involving $O(n^2)$ (implicitly defined) objects can be solved in $O(n \log n)$ time through reduction to simple 1-dimensional sorting algorithms. Namely, we showed how to solve the slope range counting problem in $O(n \log n)$ time. The problems of computing the minimum and maximum slopes can also be solved in $O(n \log n)$ time. We will leave this problem as an exercise. The problem of computing the $k$-th smallest slope is a considerably harder problem. It is not too hard to devise a randomized algorithm whose running time is $O(n \log^2 n)$. Such an algorithm applies a sort of "randomized binary search" in dual space to locate the intersection point of the desired rank. Improving the expected running time to $O(n \log n)$ time is a nontrivial exercise, and making the algorithm deterministic is even more challenging. I do not know of an efficient solution to the problem of computing the average slope.

The reduction of a geometric problem to 1-dimensional sorting and searching is quite common in computational geometry. We will see other examples of this later in the semester. We have also seen a nice application of the notion of point-line duality, which will be seen many more times this semester.

# Lecture 3: Convex Hulls

**Convexity:** Let us consider a fundamental structure in computational geometry, called the *convex hull*. We will give a more formal definition later, but, given a set $P$ of points in the plane, the convex hull of $P$, denoted $\text{conv}(P)$, can be defined intuitively by surrounding a collection of points with a rubber band and then letting the rubber band "snap" tightly around the points (see Fig. 10).

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. (Other examples include minimum area enclosing rectangles, circles, and ellipses.) It can also be used for approximating more complex shapes. For example, the convex hull of a polygon in the plane or polyhedron in 3-space is the convex hull of its vertices.

Also many algorithms compute the convex hull as an initial stage in their execution or to filter out irrelevant points. For example, the *diameter* of a point set is the maximum distance between any two points of the set. It

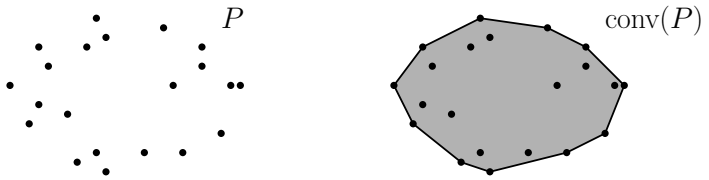



Fig. 10: A point set and its convex hull.

can be shown that the pair of points determining the diameter are both vertices of the convex hull. Also observe that minimum enclosing convex shapes (such as the minimum area rectangle, circle, and ellipse) depend only on the points of the convex hull.

**Convexity:** A set $K$ is *convex* if given any points $p, q \in K$, the line segment $\overline{pq}$ is entirely contained within $K$.

**Boundedness:** A convex body may be bounded, meaning that it can be enclosed within a sphere of a fixed radius or unbounded, meaning that it extends to infinity. Examples of unbounded convex sets in the plane include lines, rays, halfplanes, the region lying to one side of a line, and infinite cones. Given a line $\ell$, the set of points lying entirely to one side of $\ell$ (possibly including $\ell$ itself) is called a *halfplane*.

**Support:** An important property of any convex set $K$ in the plane is that at every point $p$ on the boundary of $K$, there exists a line $\ell$ (or generally in hyperplane in higher dimensions) that passes through $p$ such that $K$ lies entirely in one of the closed halfplanes defined by $\ell$.

**Convex hull:** The *convex hull* of any set $P$ is the intersection of all convex sets that contains $P$, or more intuitively, the smallest convex set that contains $P$. We will denote this $\mathrm{conv}(P)$.

When computing convex hulls, we will usually take $P$ to be a finite set of points. In such a case, $\mathrm{conv}(P)$ will be a convex polygon. Generally $P$ could be an infinite set of points. For example, we could talk about the convex hull of a collection of circles. The boundary of such a shape would consist of a combination of circular arcs and straight line segments.

**Convex Hull Problem:** The (planar) *convex hull problem* is, given a set of $n$ points $P$ in the plane, output a representation of $P$'s convex hull. The convex hull is a closed convex polygon, the simplest representation is a counterclockwise enumeration of the vertices of the convex hull. (Although points of $P$ might lie in the interior of an edge of the boundary of the convex hull, such a point is not considered a vertex. Since we will assume that the points are in *general position*, and in particular, no three are collinear, this issue does not arise.) Although the output consists only of the boundary of the hull, the convex hull of $P$ is a convex polygon, which means that it includes both the boundary and interior of this polygon.

**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 loosely based on a common approach for building geometric structures called *incremental construction*. In such a algorithm object (points here) are added one at a time, and the structure (convex hull here) is updated with each new insertion.

An important issue with incremental algorithms is the order of insertion. If we were to add points in some arbitrary order, we would need some method of testing whether the newly added point is inside the existing hull. It will simplify things to add points in some appropriately sorted order, in our case, in increasing order of $x$-coordinate. This guarantees that each newly added point is outside the current hull. (Note that Graham's original algorithm sorted points in a different way. It found the lowest point in the data set and then sorted points cyclically around this point. Sorting by $x$-coordinate seems to be a bit easier to implement, however.)

Since we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right. As mentioned above, the convex hull is a convex polygon, which can be represented as a cyclic sequence of vertices. It will make matters a bit simpler for us to represent this convex polygon as two chains,

one representing its upper part, called the *upper hull* and one representing the lower part, called the *lower hull* (see Fig. 11(a)).

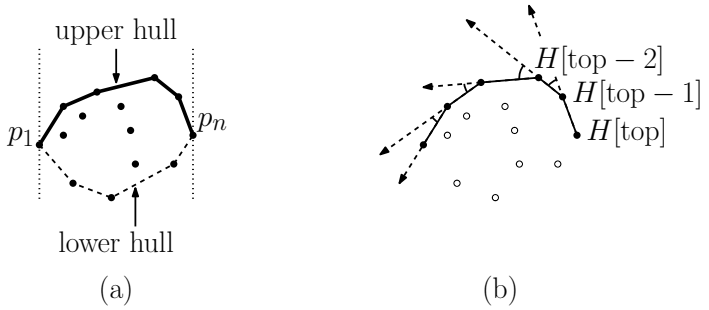



Fig. 11: Upper and lower hulls.

The *break points* common to both hulls will be the leftmost and rightmost vertices of the convex hull, that is, the points of $P$ having the smallest and largest $x$-coordinates, respectively. (By general position, we may assume there are no duplicate $x$-coordinates, and so there will be a unique leftmost point and unique rightmost points.) After building both, the two hulls can be concatenated into a single cyclic counterclockwise list.

Let us just consider how to compute the upper hull, since the lower hull is similar. Recall that the points of $P$ are first sorted in increasing order of their $x$-coordinates, and they will be added one-by-one. We store the vertices of the current upper hull in a stack $H$, where the top of the stack corresponds to the most recently added point of $P$. Let $H[\text{top}]$ denote the top of the stack, and let and $H[\text{top}-1]$ denote the element immediately below the top. Observe that as we read the stack elements from top to bottom (that is, from right to left) consecutive triples of points of the upper hull will make a (strict) "left-hand turn" (see Fig. 11(b)). As we push new points on the stack, we will maintain this property, by popping points off of the stack if they fail to satisfy this property.

**Turning and orientations:** Before proceeding with the presentation of the algorithm, we should first make a short digression to discuss the meaning of "left-hand turn." Given an ordered triple of points $\langle p, q, r\rangle$ in the plane, we say that they have *positive orientation* if they define a counterclockwise oriented triangle (see Fig. 12(a)), *negative orientation* if they define a clockwise oriented triangle (see Fig. 12(b)), and *zero orientation* if they are collinear, which includes as well the case where two or more of the points are identical (see Fig. 12(c)). Note that orientation depends on the order in which the points are given.

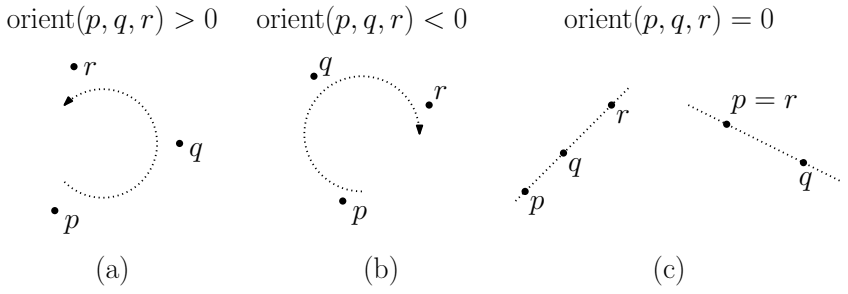



Fig. 12: Orientations of the ordered triple $(p, q, r)$.

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

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

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

Given a sequence of three points $p$, $q$, $r$, we say that the sequence $\langle p, q, r\rangle$ makes a (strict) *left-hand turn* if Orient$(p, q, r) > 0$.

**Graham's algorithm continued:** Let $p_i$ denote the next point to be added in the left-to-right ordering of the points (see Fig. 13(a)). If the triple $\langle p_i, H[\text{top}], H[\text{top} - 1]\rangle$ forms a strict left-hand turn, then we can simply push $p_i$ onto the stack. Otherwise, we can infer that the middle point of the triple $H[\text{top}]$ cannot be on the upper hull, and so we pop it off the stack. We repeat this until reaching a positively oriented triple (see Fig. 13(b)), or there are fewer than two elements on the stack. The popping process ends when $p_i$'s predecessor on the stack is its predecessor on the convex hull (see Fig. 13(c)). The algorithm is presented in the code block below.

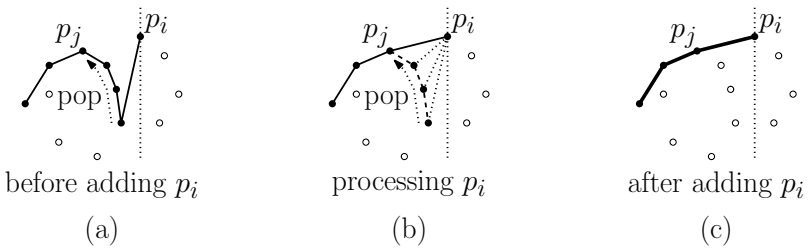



Fig. 13: Graham's scan.

Graham's Scan

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

(2) push $p_1$ and then $p_2$ onto $H$.

(3) for $i \leftarrow 3, \ldots, n$ do:

  (a) while ($|H| \geq 2$ and Orient$(p_i, H[\text{top}], H[\text{top} - 1]) \leq 0$) pop $H$.

  (b) push $p_i$ onto $H$.

**Correctness:** Why is Graham's algorithm correct? We can show inductively that the contents of $H$ at any stage of the algorithm constitute the upper hull of the points that have been processed so far. For the induction basis ($H = \{p_1, p_2\}$) this is trivially true. For the induction step, observe that $p_i$ is the rightmost point among the points processed so far, and therefore it must lie on the upper hull. Let $p_j$ be the neighboring vertex to $p_i$ on the upper hull of the first $i$ points (see Fig. 13(a)). It is easy to see that $p_j$ must be in $H$ prior to the addition of $p_i$. Each point $p_k$ in $H$ that lies between $p_j$ and $p_i$ lies beneath the edge $\overline{p_j p_i}$, and so $p_k$ should not be part of the upper hull after $p_i$ is added. For each such point it is easy to see that Orient$(p_i, p_k, p_j) \leq 0$. It follows that, as each of these points $p_k$ is tested within the while loop, it will be deleted. (We are being a bit sloppy here, because this is not exactly the same orientation test made by the algorithm, since $p_j$ is not necessarily $p_k$'s predecessor on the stack. We'll leave fixing this proof up as an exercise.)

Finally, when $p_j$ reaches the top of the stack either find that $p_j = p_1$, and hence there are less than two elements on the stack, or we find that we finally have the triple that satisfies the orientation test. In either case, the loop terminates and $p_i$ is pushed on the stack, as desired.

The lower hull can be computed by an essentially symmetric algorithm, but working from right to left instead. Once the two hulls are computed, we simply concatenate the two hulls into a single circular list.

**Running-time analysis:** We will show that Graham's algorithm runs in $O(n \log n)$ time. Clearly, it takes this much time for the initial sorting. After this, we will show that $O(n)$ time suffices for the rest of the computation.

Let $d_i$ denote the number of points that are popped (deleted) on processing $p_i$. Because each orientation test takes $O(1)$ time, the amount of time spent processing $p_i$ is $O(d_i + 1)$. (The extra $+1$ is for the last point tested, which is not deleted.) Thus, the total running time is proportional to

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

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

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

Divide-and-Conquer Convex Hull

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

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

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

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

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

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

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

One thing that simplifies the process of computing the tangents is that the two point sets $A$ and $B$ are separated from each other by a vertical line (assuming no duplicate $x$-coordinates). Let's concentrate on the lower tangent, since the upper tangent is symmetric. The algorithm operates by a simple "walking" procedure. We initialize $a$ to be the rightmost point of $H_A$ and $b$ is the leftmost point of $H_B$ (see Fig. 14(a)). These two points can be computed in linear time.

Lower tangency is a condition that can be tested locally by an orientation test involving the two vertices and neighboring vertices on the hull. We iterate the following two loops, which march $a$ and $b$ down, until they reach the points lower tangency (see Fig. 14(a)–(c)). Given a point $a$ on the hull, let $a$.succ and $a$.pred denote its successor and predecessor in CCW order about the hull.

The condition "$ab$ is not the lower tangent of $H_A$" can be implemented with the orientation test Orient$(b, a, a.\text{pred}) \geq 0$, and the other test for $H_B$ is analogous. Proving the correctness of this procedure is a little tricky, but not too

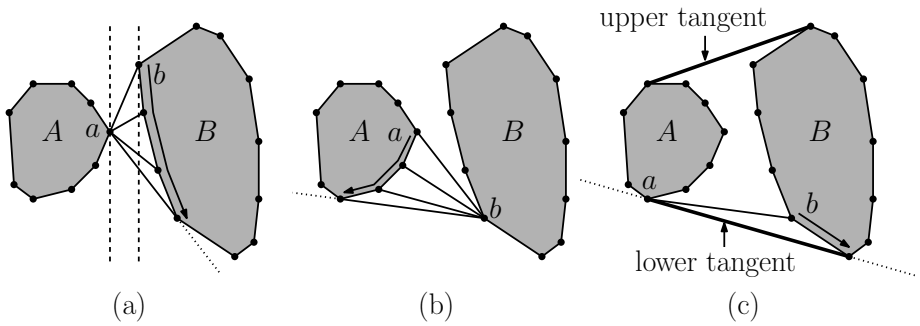



Fig. 14: Computing the lower tangent.

Finding the Lower Tangent

**LowerTangent**$(H_A, H_B)$ :

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

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

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

  (a) While ($ab$ is not a lower tangent to $H_A$) do $a \leftarrow a$.pred (move $a$ clockwise).

  (b) While ($ab$ is not a lower tangent to $H_B$) do $b \leftarrow b$.succ (move $b$ counterclockwise).

(4) Return $ab$.

hard. (The issue is proving that the two inner while loops never go beyond the lower tangent points.) See O'Rourke's book out for a careful proof. The important thing is that each vertex on each hull can be visited at most once by the search, and hence its running time is $O(m)$, where $m = |H_A| + |H_B| \leq |A| + |B|$. This is exactly what we needed to get the overall $O(n \log n)$ running time.

**Gift-Wrapping and Jarvis's March:** The next algorithm that we will consider is a variant on an $O(n^2)$ sorting algorithm called SelectionSort. For sorting, this algorithm repeatedly finds the next element to add to the sorted order from the remaining items. The corresponding convex hull algorithm is called *Jarvis's march*. which builds the hull in $O(nh)$ time by a process called "gift-wrapping". The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the "next" edge on the hull in counterclockwise order. Assuming that $p_k$ and $p_{k-1}$ were the last two points added to the hull, compute the point $q$ that maximizes the angle $\angle p_{k-1} p_k q$ (see Fig. 15). Clearly, we can find the point $q$ in $O(n)$ time.

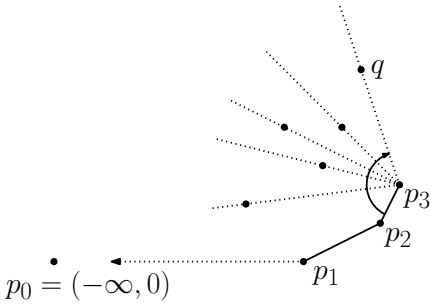



Fig. 15: Jarvis's march.

After repeating this $h$ times, we will return back to the starting point and we are done. Thus, the overall running time is $O(nh)$. Note that if $h$ is $o(\log n)$ (asymptotically smaller than $\log n$) then this is a better method than Graham's algorithm.

One technical detail is that when we to find an edge from which to start. One easy way to do this is to let $p_1$ be the point with the lowest $y$-coordinate, and let $p_0$ be the point $(-\infty, 0)$, which is infinitely far to the right. The point $p_0$ is only used for computing the initial angles, after which it is discarded (see Fig. 15).

# Lecture 4: More on Convex Hulls

**Output Sensitive Convex Hull Algorithms:** We have seen two algorithms for planar convex hull, Graham's algorithm and the divide-and-conquer algorithm, that both run in $O(n \log n)$ time. We have also seen Jarvis's algorithm, which runs in $O(hn)$ time, where $h$ is the number of vertices on the hull.

Traditionally, algorithms are analyzed in terms of their running time as a function of input size alone. However, many geometric algorithms produce outputs whose sizes vary greatly (from a constant up to a large polynomial in $n$). For such problems, it is common to express running time as a function of both the input and the output sizes. Such an algorithm is said to be *output sensitive*. Jarvis's algorithm is such an example.

When $h$ is asymptotically smaller than $\log n$, Jarvis's algorithm is superior to Graham's algorithm. Since neither algorithm is optimal in all cases, it is natural to wonder whether there is some "ultimate" planar convex hull algorithm that is optimal with respect to both $n$ and $h$.

Since the objective is to output the points on the hull in cyclic order, it is pretty easy to see that this requires sorting the points of the hull. It is well known that any comparison-based algorithm for sorting requires $\Omega(n \log n)$ time.[3] If we ignore $h$ and consider the worst case in which all of the points are vertices of the convex hull, then it is pretty easy to prove that the $\Omega(n \log n)$ lower bound cannot be beaten. (We leave the proof of this as an easy exercise. Later in these notes we present an output sensitive lower bound.)

Today, we present a planar convex hull algorithm, called Chan's algorithm, whose running time is $O(n \log h)$, and we show that this is essentially the best possible. While this algorithm is too small an improvement over Graham's algorithm to be practical, it is quite interesting nonetheless from the perspective of the techniques that it uses.

- It is derived based on a combination of two slower algorithms, Graham's and Jarvis's.
- It is based on "knowing" the final number of vertices on the convex hull. Since this number is not known, it adopts an interesting guessing process to determine its value (roughly). It is remarkable that the time to run the guessing version is asymptotically the same as if you had known the number in advance!

**How to Beat Graham and Jarvis:** To motivate Chan's algorithm, observe that the problem with Graham's scan is that it sorts all the points, and hence is doomed to having an $\Omega(n \log n)$ running time, irrespective of the size of the hull. On the other hand, Jarvis's algorithm is not limited in this way. Unfortunately, it is way too slow if there are many points on the hull.So, how can we combine these two insights to produce a faster solution?

The first observation needed for a better approach is that, if we hope to achieve a running time of $O(n \log h)$, we can only afford a log factor depending on $h$. So, if we run Graham's algorithm, we are limited to sorting sets of size at most $h$. (Actually, any polynomial in $h$ will work as well. The reason is that, for any constant $c$, $\log(h^c) = c \log h = O(\log h)$. For example, $\log h$ and $\log(h^2)$ are asymptotically equivalent. This observation will come in handy later on.)

How can we use this observation? Suppose that we partitioned the set into roughly $n/h$ subsets, each of size $h$. We could compute the convex hull of each subset in time $O(h \log h)$, which we'll call a convex *mini-hull*. The total time to compute all the mini-hulls would be $O((n/h)h \log h) = O(n \log h)$. We are within our overall time budget, but of course we would still have to figure out how to merge these mini-hulls into the final global convex hull.

[3] Recall that asymptotic $\Omega$-notation is the lower-bound analog to the $O$-notation upper bound. Formally, we say that a function $f(n)$ is $\Omega(g(n))$ if, as $n$ tends to infinity, the ratio $g(n)/f(n)$ is bounded. That is, $f$ grows at least as fast as $g$. There are faster sorting algorithms that are not comparison based, but they apply to discrete objects such as small integers and strings, not to real numbers.

But wait! We do not know the value of $h$ in advance, so it would seem that we are stuck before we even get started. We will deal with this conundrum later, but, just to get the ball rolling, suppose for now that we had an estimate for $h$, call it $h^*$, whose value is at least as large as $h$, but not too much larger (say $h \leq h^* \leq h^2$). If we run the above partitioning process using $h^*$ rather than $h$, the total running time to compute all the mini-hulls is $O(n \log h^*) = O(n \log h)$.

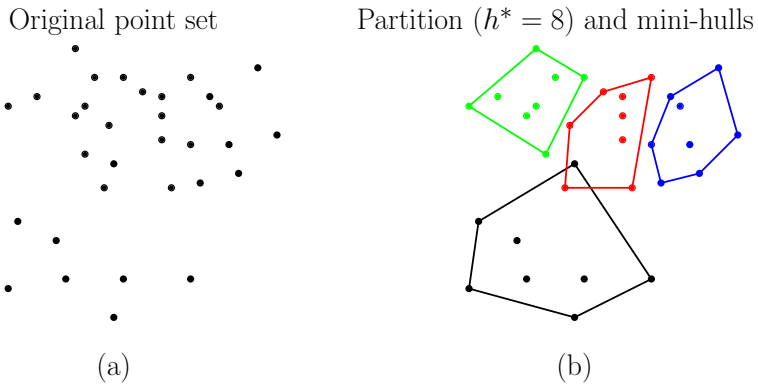



Fig. 16: Partition and mini-hulls.

The partitioning of the points is done by any arbitrary method (e.g., just break the input up into groups of size roughly $h^*$). Of course, the resulting mini-hulls might overlap one another (see Fig. 16(a) and (b)). Although we presume that $h^*$ is a rough approximation to $h$, we cannot infer anything about the numbers of vertices on the various mini-hulls. They could range from 3 up to $h^*$.

**Merging the minis:** The question that remains is how to merge the mini-hulls into a single global hull. The idea is to run Jarvis's algorithm, but we treat each mini-hull as if it is a "fat point". At each step, rather than computing the angle from the current hull vertex to every point of the set, we compute the tangent lines of the current hull vertex to each of the mini-hulls, including the mini-hull containing this vertex. (There are two tangents from a point to a mini-hull, and we need to take care to compute the proper one.) Note that the current vertex is on the global convex hull, so it cannot lie in the interior of any of the mini-hulls. Among all these tangents, we take the one that yields the smallest external angle. (The process is illustrated in Fig. 17(a).) Note that, even though a point can appear only once on the final global hull, a single mini-hull may contribute many points to the final hull.

You might think that, since a mini-hull may have as many as $h^*$ vertices, there is nothing to be saved in computing these tangents over the straightforward method. The key is that each mini-hull is a convex polygon, and hence it has quite a bit more structure than an arbitrary collection of (unsorted) points. In particular, we make use of the following lemma:

**Lemma:** Consider a convex polygon $K$ in the plane and a point $p$ that is external to $K$, such that the vertices of $K$ are stored in cyclic order in an array. Then the two tangents from $p$ to $K$ (more formally, the two supporting lines for $K$ that pass through $p$) can each be computed in time $O(\log m)$, where $m$ is the number of vertices of $K$.

We will leave the proof of this lemma as an exercise, but the key idea is that, since the vertices of the hull form a cyclically sorted sequence, it is possible to adapt binary search to find the desired points of tangency with $p$ (Fig. 17(b)). Using the above lemma, it follows that we can compute the tangent from an arbitrary point to a single mini-hull in time $O(\log h^*) = O(\log h)$.

The final "restricted algorithm" (since we assume we have the estimate $h^*$) is presented in the code block below. (The $k$th stage is illustrated in Fig. 17(c).) Since we do not generally know what the value of $h$ is, it is possible that our restricted algorithm may be run with a value of $h^*$ that is not within the prescribed range, $h \leq h^* \leq h^2$.

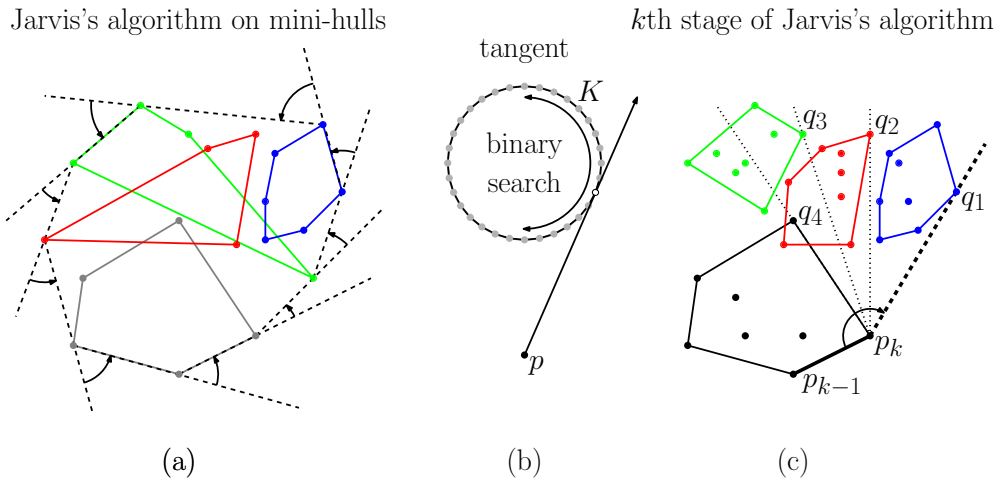



Fig. 17: Using Jarvis's algorithm to merge the mini-hulls.

(In particular, our final algorithm will maintain the guarantee that $h^* \leq h^2$, but the lower bound of $h$ may not hold.) If $h^* < h$, when we are running the Jarvis phase, we will discover the error as soon as we encounter more than $h^*$ vertices on the hull. If this happens, we immediately terminate the algorithm and announce the algorithm has "failed". If we succeed in completing the hull with $h^*$ points or fewer, we return the final hull.

Chan's Algorithm for the Restricted Hull Problem

**RestrictedHull**$(P, h^*)$ :

(1) Let $r \leftarrow \lceil n/h^* \rceil$.

(2) Partition $P$ into disjoint subsets $P_1, P_2, \ldots, P_r$, each of size at most $h^*$.

(3) For ($i \leftarrow 1$ to $r$)

compute Hull$(P_i)$ using Graham's scan and store the vertices in an ordered array.

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

(5) For ($k \leftarrow 1$ to $h^*$)

(a) For ($i \leftarrow 1$ to $r$)

compute point tangent $q_i \in$ Hull$(P_i)$, that is, the vertex of Hull$(P_i)$ that maximizes the angle $\angle p_{k-1} p_k q_i$.

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

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

(6) (Unable to complete the hull after $h^*$ iterations.) Return "Failure: $h^*$ is too small."

The upshots of this are: (1) the Jarvis phase never performs for more than $h^*$ stages, and (2) if $h \leq h^*$, the algorithm succeeds in finding the hull. To analyze its running time, recall that each partition has roughly $h^*$ points, and so there are roughly $n/h^*$ mini-hulls. Each tangent computation takes $O(\log h^*)$ time, and so each stage takes a total of $O((n/h^*) \log h^*)$ time. By (1) the number of Jarvis stages is at most $h^*$, so the total running time of the Jarvis phase is $O(h^*(n/h^*) \log h^*) = O(n \log h^*)$.

Combining this with the fact that the Graham phase takes $O(n \log h^*)$ time, the total time of the restricted algorithm is $O(n \log h^*)$. If we maintain the condition that $h^* \leq h^2$ then, irrespective of success or failure, the running time will be $O(n \log h)$.

**Guessing the Hull's Size:** The only question remaining is how do we know what value to give to $h^*$? Remember that, if $h^* \geq h$, the algorithm will succeed in computing the hull, and if $h^* \leq h^2$, the running time of the restricted algorithm is $O(n \log h)$. Clearly we do not want to try a value of $h^*$ that is way too high, or we are doomed to having an excessively high running time. So, we should start our guess small, and work up to larger values until

we achieve success. Each time we try a test value $h^* < h$, the restricted hull procedure may tell us we have failed, and so we need to increase the value if $h^*$.

As a start, we could try $h^* = 1, 2, 3, \ldots, i$, until we luck out as soon as $h^* = h$. Unfortunately, this would take way too long. (Convince yourself that this would result in a total time of $O(nh \log h)$, which is even worse than Jarvis's march.)

The next idea would be to perform a *doubling search*. That is, let's try $h^* = 1, 2, 4, 8, \ldots, 2^i$. When we first succeed, we might have overshot the value of $h$, but not by more than a factor of 2, that is $h \leq h^* \leq 2h$. The convex hull will have at least three points, and clearly for $h \geq 3$, we have $2h \leq h^2$. Thus, this value of $h^*$ will satisfy our requirements. Unfortunately, it turns out that this is still too slow. (You should do the analysis yourself and convince yourself that it will result in a running time of $O(n \log^2 h)$. Better but still not the best.)

So if doubling is not fast enough, what is next? Recall that we are allowed to overshoot the actual value of $h$ by as much as $h^2$. Therefore, let's try repeatedly squaring the previous guess. In other words, let's try $h^* = 2, 4, 16, \ldots, 2^{2^i}$. Clearly, as soon as we reach a value for which the restricted algorithm succeeds, we have $h \leq h^* \leq h^2$. Therefore, the running time for this stage will be $O(n \log h)$. But what about the total time for all the previous stages?

To analyze the total time, consider the $i$th guess, $h_i^* = 2^{2^i}$. The $i$th trial takes time $O(n \log h_i^*) = O\big(n \log 2^{2^i}\big) = O(n2^i)$. We know that we will succeed as soon as $h_i^* \geq h$, that is if $i = \lceil \lg \lg h \rceil$. (Throughout the semester, we will use $\lg$ to denote logarithm base 2 and $\log$ when the base does not matter.[4] ) Thus, the algorithm's total running time (up to constant factors) is

$$T(n, h) \;=\; \sum_{i=1}^{\lg \lg h} n2^i \;=\; n \sum_{i=1}^{\lg \lg h} 2^i.$$

This is a geometric series. Let us use the well known fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$. We obtain a total running time of

$$T(n, h) \;<\; n \cdot 2^{1+\lg \lg h} \;=\; n \cdot 2 \cdot 2^{\lg \lg h} \;=\; 2n \lg h \;=\; O(n \log h),$$

which is just what we want. In other words, by the "miracle" of the geometric series, the total time to try all the previous failed guesses is asymptotically the same as the time for the final successful guess. The final algorithm is presented in the code block below.

Chan's Complete Convex Hull Algorithm

**Hull**$(P)$ :

(1) $h^* \leftarrow 2$. $L \leftarrow$ fail.
(2) while ($L \neq$ fail)
  (a) Let $h^* \leftarrow \min((h^*)^2, n)$.
  (b) $L \leftarrow$ RestrictedHull$(P, h^*)$.
(3) Return $L$.

**Lower Bound (Optional):** Next we will show that Chan's result is asymptotically optimal in the sense that any algorithm for computing the convex hull of $n$ points with $h$ points on the hull requires $\Omega(n \log h)$ time. The proof is a generalization of the proof that sorting a set of $n$ numbers requires $\Omega(n \log n)$ comparisons.

If you recall the proof that sorting takes at least $\Omega(n \log n)$ comparisons, it is based on the idea that any sorting algorithm can be described in terms of a *decision tree*. Each comparison has at most 3 outcomes ($<$, $=$, or $>$). Each such comparison corresponds to an internal node in the tree. The execution of an algorithm can be viewed as a traversal along a path in the resulting 3-ary tree. The height of the tree is a lower bound on the worst-case running time of the algorithm. There are at least $n!$ different possible inputs, each of which must be reordered

[4] When $\log n$ appears as a factor within asymptotic big-O notation, the base of the logarithm does not matter provided it is a constant. This is because $\log_a n = \log_b n / \log_b a$. Thus, changing the base only alters the constant factor.

differently, and so you have a 3-ary tree with at least $n!$ leaves. Any such tree must have $\Omega(\log_3(n!))$ height. Using Stirling's approximation for $n!$, this solves to $\Omega(n \log n)$ height. (For further details, see the algorithms book by Cormen, Leiserson, Rivest, and Stein.)

We will give an $\Omega(n \log h)$ lower bound for the convex hull problem. In fact, we will give an $\Omega(n \log h)$ lower bound on the following simpler decision problem, whose output is either yes or no.

**Convex Hull Size Verification Problem (CHSV):** Given a point set $P$ and integer $h$, does the convex hull of $P$ have $h$ distinct vertices?

Clearly if this takes $\Omega(n \log h)$ time, then computing the hull must take at least as long. As with sorting, we will assume that the computation is described in the form of a decision tree. The sorts of decisions that a typical convex hull algorithm will make will likely involve orientation primitives. Let's be even more general, by assuming that the algorithm is allowed to compute *any* algebraic function of the input coordinates. (This will certainly be powerful enough to include all the convex hull algorithms we have discussed.) The result is called an *algebraic decision tree*.

The input to the CHSV problem is a sequence of $2n = N$ real numbers. We can think of these numbers as forming a vector in real $N$-dimensional space, that is, $(z_1, z_2, \ldots, z_N) = \vec{z} \in \mathbb{R}^N$, which we will call a *configuration*. Each node of the decision tree is associated with a multivariate algebraic formula of degree at most $d$, where $d$ is any fixed constant. For example,

$$f(\vec{z}) = z_1 z_4 - 2 z_3 z_6 + 5 z_6^2,$$

would be an algebraic function of degree 2. The node branches in one of three ways, depending on whether the result is negative, zero, or positive. Each leaf of the resulting tree corresponds to a possible answer that the algorithm might give.

For each input vector $\vec{z}$ to the CHSV problem, the answer is either "yes" or "no". The set of all "yes" points is just a subset of points $Y \subset \mathbb{R}^N$, that is a region in this space. Given an arbitrary input $\vec{z}$ the purpose of the decision tree is to tell us whether this point is in $Y$ or not. This is done by walking down the tree, evaluating the functions on $\vec{z}$ and following the appropriate branches until arriving at a leaf, which is either labeled "yes" (meaning $\vec{z} \in Y$) or "no". An abstract example (not for the convex hull problem) of a region of configuration space and a possible algebraic decision tree (of degree 1) is shown in the following figure. (We have simplified it by making it a binary tree.) In this case the input is just a pair of real numbers.

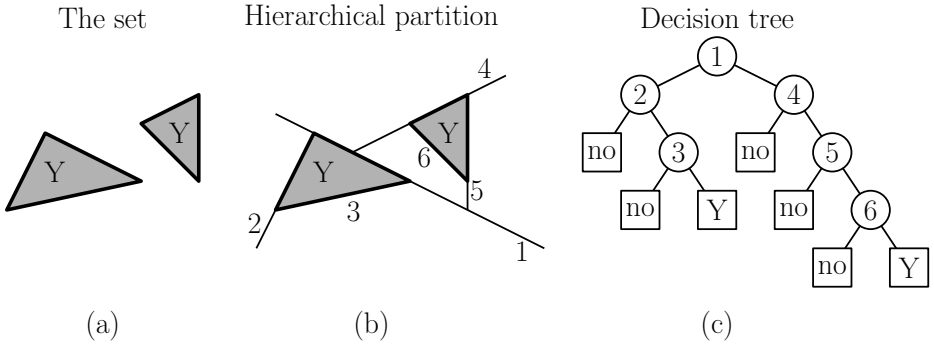



Fig. 18: The geometric interpretation of an algebraic decision tree.

We say that two points $\vec{u}, \vec{v} \in Y$ are in the same *connected component* of $Y$ if there is a path in $\mathbb{R}^N$ from $\vec{u}$ to $\vec{v}$ such that all the points along the path are in the set $Y$. (There are two connected components in the figure.) We will make use of the following fundamental result on algebraic decision trees, due to Ben-Or. Intuitively, it states that if your set has $M$ connected components, then there must be at least $M$ leaves in any decision tree for the set, and the tree must have height at least the logarithm of the number of leaves.

**Theorem:** Let $Y \in \mathbb{R}^N$ be any set and let $T$ be any $d$-th order algebraic decision tree that determines membership in $W$. If $W$ has $M$ disjoint connected components, then $T$ must have height at least $\Omega((\log M) - N)$.

We will begin our proof with a simpler problem.

**Multiset Size Verification Problem (MSV):** Given a multiset of $n$ real numbers and an integer $k$, confirm that the multiset has exactly $k$ distinct elements.

**Lemma:** The MSV problem requires $\Omega(n \log k)$ steps in the worst case in the $d$-th order algebraic decision tree

**Proof:** In terms of points in $\mathbb{R}^n$, the set of points for which the answer is "yes" is

$$Y = \{(z_1, z_2, \ldots, z_n) \in \mathbb{R}^n \ : \ |\{z_1, z_2, \ldots, z_n\}| = k\}.$$

It suffices to show that there are at least $k!k^{n-k}$ different connected components in this set, because by Ben-Or's result it would follow that the time to test membership in $Y$ would be

$$\Omega(\log(k!k^{n-k}) - n) \ = \ \Omega(k \log k + (n-k) \log k - n) \ = \ \Omega(n \log k).$$

Consider the all the tuples $(z_1, \ldots, z_n)$ with $z_1, \ldots z_k$ set to the distinct integers from 1 to $k$, and $z_{k+1} \ldots z_n$ each set to an arbitrary integer in the same range. Clearly there are $k!$ ways to select the first $k$ elements and $k^{n-k}$ ways to select the remaining elements. Each such tuple has exactly $k$ distinct items, but it is not hard to see that if we attempt to continuously modify one of these tuples to equal another one, we must change the number of distinct elements, implying that each of these tuples is in a different connected component of $Y$.

To finish the lower bound proof, we argue that any instance of MSV can be reduced to the convex hull size verification problem (CHSV). Thus any lower bound for MSV problem applies to CHSV as well.

**Theorem:** The CHSV problem requires $\Omega(n \log h)$ time to solve.

**Proof:** Let $Z = (z_1, \ldots, z_n)$ and $k$ be an instance of the MSV problem. We create a point set $\{p_1, \ldots, p_n\}$ in the plane where $p_i = (z_i, z_i^2)$, and set $h = k$. (Observe that the points lie on a parabola, so that all the points are on the convex hull.) Now, if the multiset $Z$ has exactly $k$ distinct elements, then there are exactly $h = k$ points in the point set (since the others are all duplicates of these) and so there are exactly $h$ points on the hull. Conversely, if there are $h$ points on the convex hull, then there were exactly $h = k$ distinct numbers in the multiset to begin with in $Z$.

Thus, we cannot solve CHSV any faster than $\Omega(n \log h)$ time, for otherwise we could solve MSV in the same time.

The proof is rather unsatisfying, because it relies on the fact that there are many duplicate points. You might wonder, does the lower bound still hold if there are no duplicates? Kirkpatric and Seidel actually prove a stronger (but harder) result that the $\Omega(n \log h)$ lower bound holds even you assume that the points are distinct.

# Lecture 5: Line Segment Intersection

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

- In solid modeling complex shapes are constructed by applying various boolean operations (intersection, union, and difference) to simple primitive shapes. The process is called *constructive solid geometry* (CSG). Computing intersections of model surfaces is an essential part of the process.
- In robotics and motion planning it is important to know when two objects intersect for *collision detection* and *collision avoidance*.

- In geographic information systems it is often useful to *overlay* two subdivisions (e.g. a road network and county boundaries to determine where road maintenance responsibilities lie). Since these networks are formed from collections of line segments, this generates a problem of determining intersections of line segments.
- In computer graphics, *ray shooting* is an important method for rendering scenes. The computationally most intensive part of ray shooting is determining the intersection of the ray with other objects.

**Line segment intersection:** The problem that we will consider is, given a set $S$ of $n$ line segments in the plane, report (that is, output) 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 zero and as many as $\binom{n}{2} = O(n^2)$ different intersection points. We could settle for an $O(n^2)$ time 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 design an *output sensitive algorithm*, that is, one whose running time depends not only on the input size, but also on the output size.

Given a set $S$ of $n$ line segments, let $I = I(S)$ denote the number of intersections. We will express the running time of our algorithm in terms of both $n$ and $I$. As usual, we will assume that the line segments are in general position. In particular, we assume:

(1) The $x$-coordinates of the endpoints and intersection points are all distinct. (This implies that no line segment is vertical.)

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

(3) No three line segments intersect in a common point.

Generalizing the algorithm to handle degeneracies efficiently is an interesting exercise. (See our book for more discussion of this.)

**Plane Sweep Algorithm:** Let us now consider the algorithm for reporting the segment intersections. Let $S = \{s_1, \ldots, s_n\}$ denote the line segments whose intersections we wish to compute. The method, called *plane sweep*, is a fundamental technique in computational geometry. We solve a 2-dimensional problem by simulating the process of sweeping a 1-dimensional line across the plane. The intersections of the sweep line with the segments defines a collection of points along the sweep line. We will store these points in a data structure, which we call the *sweep-line status*.

Although we might visualize the sweeping process as a continuous one, there is a discrete set of *event points* where important things happen. As the line sweeps from left to right, points are inserted, deleted, and may swap order along the sweep line. Thus, we reduce a static 2-dimensional problem to a dynamic 1-dimensional problem.

There are three basic elements that are maintained at any time in any plane-sweep algorithm: (1) the partial solution that has already been constructed to the left of the sweep line, (2) the current status of objects along the sweep line itself, and (3) a (sub)set of the future events to be processed (see Fig. 19).

The key to designing an efficient plane-sweep algorithm involves determining the best way to store and update these three elements as each new event is process. Let's consider each of these elements in greater detail in the context of line-segment intersection.

**Sweep line status:** We will simulate the sweeping of a vertical line $\ell$ from left to right. The sweep-line status will consist of the line segments that intersect the sweep line sorted, say, from top to bottom. In order to maintain this set dynamically, we will store them in a data structure, which will be described below.

Note that each time the sweep line moves, all the $y$-coordinates of the intersection points change as well. It will be too inefficient to continually update all the $y$-coordinates each time the sweep line moves. We exploit the fact that it is not the actual $y$-coordinates that we really care about, just their *order*. To do this, rather than storing

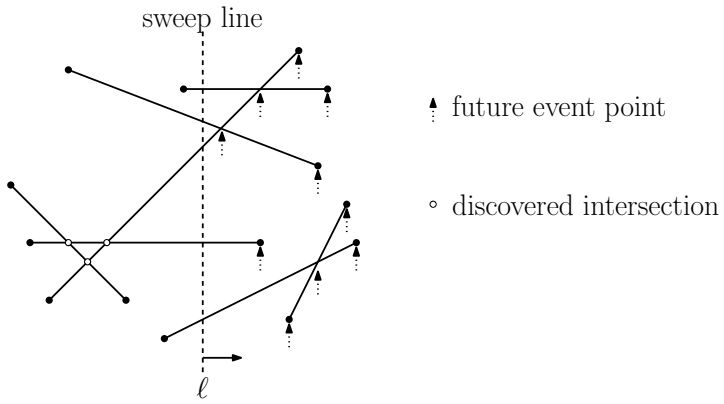



Fig. 19: Plane sweep.

$y$-coordinates, for each line segment $s_i$ that intersects the sweep line, we store the coefficients $(a_i, b_i)$ of the equation of the line, e.g., $y = a_i x + b_i$. (These coefficients can easily be derived from the segment endpoints.) In this way, whenever the sweep line arrives at a new $x$-coordinate, say $x = x_0$, we can determine the current $y$-coordinate at which segment $s_i$ intersects the sweep line as $y(x_0) = a_i x_0 + b_i$ (see Fig. 20). As we shall see, only a constant number of such intersections need to be evaluated at each event point.

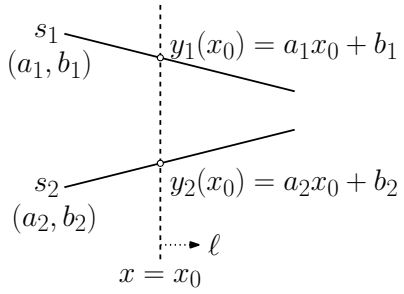



Fig. 20: The sweep-line status stores coefficients of the line equations, and the $y$-coordinates of the intersections are computed as needed.

**Events and Detecting Intersections:** It suffices to process events only when there is a change in the sweep-line status. These $x$-coordinates are called *event points*. For our application, we have three types of event points, corresponding to when the sweep line encounters (1) the left endpoint of a segment, (2) the right endpoint of a segment, and (3) an intersection point between two segments.

Note that endpoint events can be presorted before the sweep runs. In contrast, intersection events will be discovered as the sweep executes. 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 occurring to the right of the sweep line. If so, we will add this new event to a priority queue of future events. This priority queue will be sorted in left-to-right order by $x$-coordinates.

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:** Consider a set $S$ of line segments in general position, and consider two segments $s_i, s_j \in S$ that intersect in some point $p = (p_x, p_y)$. 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.

**Proof:** By general position, it follows that no three lines intersect in a common point. Therefore if we consider a placement of the sweep line that is infinitesimally to the left of the intersection point, the line segments $s_i$ and $s_j$ will be adjacent along this sweep line. Consider the event point $q$ with the largest $x$-coordinate that is strictly less than $p_x$. Since there are no events between $q_x$ and $p_x$, there can be no segment intersections within the vertical slab bounded by $q$ on the left and $p$ on the right (the shaded region of Fig. 20), and therefore the order of lines along the sweep line after processing $q$ will be identical the order of the lines along the sweep line just prior $p$. Therefore, $s_i$ and $s_j$ are adjacent immediately after processing event $q$.

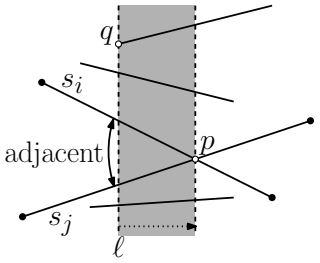



Fig. 21: Correctness of the "adjacent segment rule".

When two formerly adjacent segments cease to be adjacent (e.g., because a new segment is discovered between them), we will delete the event from the queue. While this is not formally necessary, it keeps us from inserting the same event point over and over again, and hence we do not need to worry about the existence of duplicate events from the priority queue.

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

**Event queue:** This holds the set of future events, sorted by increasing $x$-coordinate. Each event in this set contains the auxiliary information of what type of event this is (left-endpoint, right-endpoint, or intersection) and which segment(s) are involved. The operations that this data structure should support are:

- insert a new event with a given $x$-coordinate
- extract the event with the smallest $x$-coordinate
- delete an existing event

A typical priority queue data structure (e.g., a binary heap sorted on $x$) is adequate for performing the first two operations, but deletion is a problem. Instead, we store the events in a sorted dictionary (e.g., either a balanced binary tree or a skip list) sorted by $x$-coordinates. Each of the above operations can be performed in $O(\log m)$ time, where $m$ is the current number of events.

The number of events is never more than $O(n)$, since there are at most $n$ left endpoints, $n$ right endpoints, and $n - 1$ pairs of adjacent segments on the sweep line. Therefore, each event-queue operation can be performed in time $O(\log n)$.

**Sweep-line status:** To store the sweep-line status, we maintain an ordered dictionary (e.g., a balanced binary tree or skip-list) which contains the lines that intersect the sweep line sorted from top to bottom. As mentioned earlier, each entry stores the coefficients of the line equation, not the actual intersection point. (You may want to take a moment to convince yourself that the operations of maintaining the dictionary can be performed "on the fly" given the $x$-coordinate of the current sweep line.)

This data structure needs to support the following operations, given the $x$-coordinate of the current sweep line:

- insert a new line segment (whose left endpoint coincides with $x$).
- delete an existing line segment (whose right endpoint coincides with $x$).
- swap two adjacent entries (whose intersection point coincides with $x$).

- determine the segment immediately above or below any given segment on the sweep line.

Since there are at most $n$ segments on the sweep line at any time, the dictionary contains at most $n$ elements, and so these operations can be performed in $O(\log n)$ time each.

**Processing Events:** All that remains is explaining how to process the events. This is presented in the code block below. (See our text for a more careful implementation.) The various cases are illustrated in Fig. 21.

Line Segment Intersection Reporting

(1) Insert all of the endpoints of the line segments of $S$ into the event queue. The initial sweep-line 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:

**Left endpoint:**

(a) Insert this line segment $s$ into the sweep-line status, based on the $y$-coordinate of this endpoint.

(b) Let $s'$ and $s''$ be the segments immediately above and below $s$ on the sweep line. If there is an event associated with this pair, remove it from the event queue.

(c) Test for intersections between $s$ and $s'$ and between $s$ and $s''$ to the right of the sweep line. If so, add the corresponding event(s) to the event queue.

**Right endpoint:**

(a) Let $s'$ and $s''$ be the segments immediately above and below $s$ on the sweep line.

(b) Delete segment $s$ from the sweep-line status.

(c) Test for intersections between $s'$ and $s''$ to the right of the sweep line. If so, add the corresponding event to the event queue.

**Intersection:**

(a) Report this intersection.

(b) Let $s'$ and $s''$ be the two intersecting segments. Swap these two line segments in the sweep-line status (they must be adjacent to each other).

(c) As a result, $s'$ and $s''$ have changed which segments are immediately above and below them. Remove any old events due to adjacencies that have ended and insert any new intersection events from adjacencies that have been created.

Observe that our algorithm is very careful about storing intersection events only for adjacent elements in the priority queue. For example, consider two segments $s$ and $s'$ that intersect at a segment $p$, such that, when the two are initially added to the sweep-line status, they are adjacent. Therefore, the intersection point $p$ is added to event queue (see Fig. 23). As intervening segments are seen between them, they successfully become non-adjacent and then adjacent again. Because our algorithm is careful about deleting intersections between non-adjacent entries in the sweep-line status, the event $p$ is repeated deleted and reinserted. If we had not done this, we would have many duplicate events in the queue.

**Analysis:** Altogether, there are $2n+I$ events processed. Each event involves a constant amount of work and a constant number of accesses to our data structures. As mentioned above, each access to either of the data structures takes $O(\log n)$ time. Therefore, the total running time is $O((2n + I)\log n) = O(n \log n + I \log n)$.

Is this the best possible? There is an algorithm that achieves a running time of $O(n \log n + I)$. It can be shown that this is asymptotically optimal. Clearly $\Omega(I)$ time is needed to output the intersections. The lower bound of $\Omega(n \log n)$ results from a reduction from the element uniqueness problem. Given a list of $n$ numbers $\langle x_1, \ldots, x_n\rangle$ the *element uniqueness problem* asks whether these numbers are all distinct. Element uniqueness is known to have a lower bound of $\Omega(n \log n)$ in the algebraic decision tree model of computation. (It can be solved in $O(n)$ time using hashing, but the algebraic decision tree model does not allow integer division, which is needed by hashing.)

The reduction is as follows. Convert each $x_i$ into a vertical segment passing through the point $(x_i, 0)$, clearly two segments intersect if and only if two elements of the list are identical. You might complain that this lower-bound example violates our general position assumptions, but note that if you were to apply an very tiny random rotation to each line segment, the segments would now be in general position.

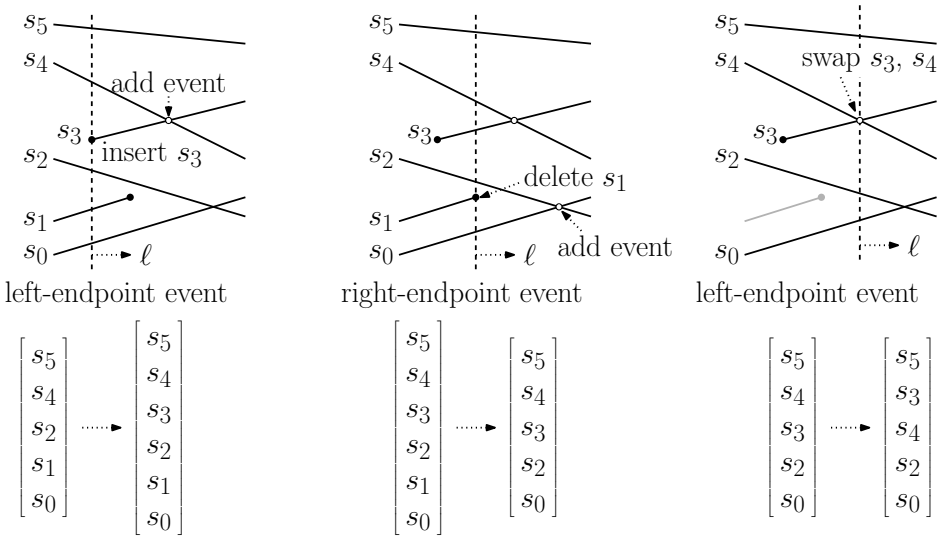



Fig. 22: Plane-sweep algorithm event processing.

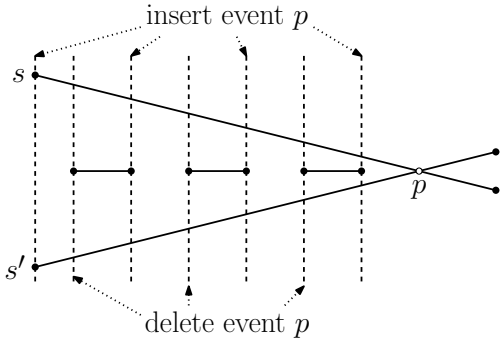



Fig. 23: An intersection event that is repeatedly inserted and deleted from the event queue

**Computing Segment Intersections (Optional):** We have assumed that the primitive of computing the intersection point of two line segments can be performed exactly in $O(1)$ time. Let us see how to do this. Let $\overline{ab}$ and $\overline{cd}$ be two line segments in the plane, given by their endpoints, for example $a = (a_x, a_y)$. First observe that it is possible to determine *whether* these line segments intersect, simply by applying an appropriate combination of orientation tests. (We will leave this as an exercise.) However, this alone is not sufficient for the plane-sweep algorithm.

One way to determine the point at which the segments intersect is to use a *parametric representation* of the segments. Any point on the line segment $\overline{ab}$ can be written as a convex combination involving a real parameter $s$:

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

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

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

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

$$(1 - s)a_x + sb_x \;=\; (1 - t)c_x + td_x \qquad \text{and} \qquad (1 - s)a_y + sb_y \;=\; (1 - t)c_y + td_y.$$

The coordinates of the points are all known, so it is just a simple exercise in linear algebra to solve for $s$ and $t$. In general, such a linear system could be solved using Gauss elimination and floating-point computations. If the denominator of the result is 0, the line segments are either parallel or collinear. These special cases can be dealt with some care. If the denominator is nonzero, then we obtain values for $s$ and $t$ as rational numbers (the ratio of two integers). Once the values of $s$ and $t$ have been computed all that is needed is to check that both are in the interval $[0, 1]$.

**Exact Computation (Optional):** The above approach is fine for producing a floating-point representation of the final result. Floating-point calculations are intrinsically approximate, and so the question arises of whether the algorithm is formally correct.

It is noteworthy that our plane-sweep algorithm does not actually require computing the coordinates of the intersection points. Two discrete primitives suffice: (1) the ability to compare the $x$-coordinates of two intersection points (for ordering intersection events) and (2) the ability to compare the $y$-coordinates of the intersection points of two segments with the vertical sweep line (for ordering segments on the plane-sweep status).

If the input coordinates are integers, it is possible to perform rational number calculations and comparisons exactly using multiple-precision integer arithmetic. In particular, each rational number $q/r$ is maintained as a pair $(q, r)$, by explicitly storing the numerator and denominator as integers. It is possible add, subtract, multiply and divide rational numbers in this form, by purely integer operations. (For example, $q_1/r_1 + q_2/r_2 = (q_1r_2 + q_2r_1)/r_1r_2$.) In this way, we never need to perform divisions. We can compute the solutions to the above system of linear equations applying Cramer's rule, which expresses the solution as a ratio of two determinants with integer coordinates. Thus, the comparisons required by the algorithm can be computed exactly, if desired. The price we pay is the need to implement some form of multiple precision integer arithmetic.

# Lecture 6: Polygon Triangulation

**The Polygon Triangulation Problem:** Triangulation is the general problem of subdividing a spatial domain into simplices, which in the plane means triangles. In its simplest form, a simple polygon is given (that is, a planar region that is defined by a closed, simple polygonal curve), and the objective is to subdivide the polygon into triangles (see Fig. 24). Such a subdivision is not necessarily unique, and there may be other criteria to be optimized in computing the triangulation.

Triangulating simple polygons is important for many reasons. This operation useful, for example, whenever it is needed to decompose a complex shapes a set of disjoint simpler shapes. Note that in some applications it is

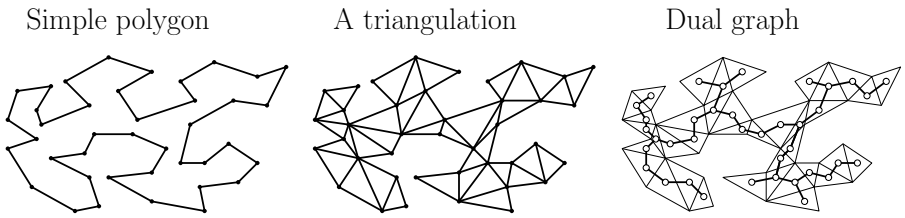



Fig. 24: Polygon triangulation.

desirable to produce "fat" (nearly equilateral) triangles, but we will not worry about this issue in this lecture. A triangulation provides a simple graphical representation of the polygon's interior, which is useful for algorithms that operate on polygons. In particular, consider a graph whose vertices are the triangles of the triangulation and two vertices of this graph are adjacent if the associated triangles are adjacent (see Fig. 24(c)). This is called the *dual graph* of the triangulation. It is easy to show that such a graph is a *free tree*, that is, it is an acylic, connected graph.

This simple problem has been the focus of a remarkably large number of papers in computational geometry spanning a number of years. There is a simple naive polynomial-time algorithm for the planar case (as opposed to possibly nonconvex polyhedra in higher dimensions). The idea is based on repeatedly adding "diagonals." We say that two points on the boundary of the polygon are *visible* if the interior of the line segment joining them lies entirely within the interior of the polygon. Define a *diagonal* of the polygon to be the line segment joining any pair of visible vertices. Observe that the addition of a diagonal splits the polygon into two polygons of smaller size. In particular, if the original polygon has $n$ vertices, the diagonal splits the polygon into two polygons with $n_1$ and $n_2$ vertices, respectively, where $n_1, n_2 < n$, and $n_1 + n_2 = n + 2$. Any simple polygon with at least four vertices has at least one diagonal. (This seemingly obvious fact is not that easy to prove. You might try it.) A simple induction argument shows that the final number of diagonals is $n - 3$ and the final number of triangles is $n - 2$.

The naive algorithm operates by repeatedly adding diagonals. Unfortunately, this algorithm is not very efficient (unless the polygon has special properties, for example, convexity) because of the complexity of the visibility test.

There are very simple $O(n \log n)$ algorithms for this problem that have been known for many years. A long-standing open problem was whether there exists an $O(n)$ time algorithm. (Observe that the input polygon is presented as a cyclic list of vertices, and hence the data is in some sense "pre-sorted", which precludes an $\Omega(n \log n)$ lower bound.) The problem of a linear time polygon triangulation was solved by Bernard Chazelle in 1991, but the algorithm is so amazingly complicate. Unless other properties of the triangulation are desired, the $O(n \log n)$ algorithm that we will present in this lecture is quite practical and probably preferable in practice to any of the "theoretically" faster algorithms.

Our approach is based on a two-step process (although with a little cleverness, both steps could be combined into one algorithm).

- First, the simple polygon is decomposed into a collection of simpler polygons, called *monotone polygons*. This step takes $O(n \log n)$ time.
- Second, each of the monotone polygons is triangulated separately, and the result are combined. This step takes $O(n)$ time.

The triangulation results in a planar subdivision. Such a subdivision could be stored as a planar graph or simply as a set of triangles, but there are representations that are more suited to representing planar subdivisions. One of these is called *double-connect edge list* (or DCEL). This is a linked structure whose individual entities consist of the vertices (0-dimensional elements), edges (1-dimensional elements), triangular faces (2-dimensional elements). Each entity is joined through links to its neighboring elements. For example, each edge stores the two vertices that form its endpoints and the two faces that lie on either side of it.

We refer the reader to Chapter 2 of our text for a more detailed description of the DCEL structure. Henceforth, we will assume that planar subdivisions are stored in a manner than allows local traversals of the structure to be performed $O(1)$ time.

**Monotone Polygons:** Let's begin with a few definitions. A *polygonal curve* is a collection of line segments, joined end-to-end. If the last endpoint is equal to the first endpoint, the polygonal curve is said to be *closed*. The line segments are called *edges*. The endpoints of the edges are called the *vertices* of the polygonal curve. Each edge is *incident* to two vertices (its endpoints), and each vertex is incident (to up) two edges. A polygonal curve is said to be *simple* if no two nonincident elements intersect each other. A closed simple polygonal curve decomposes the plane into two parts, its *interior* and *exterior*. Such a polygonal curve is called a *simple polygon*. When we say "polygon" we mean simple polygon.

A polygonal chain $C$ is *monotone* with respect to $\ell$ if each line that is orthogonal to $\ell$ intersects $C$ in a single connected component. (It may intersect, not at all, at a single point, or along a single line segment.) A polygonal chain $C$ is said to be *strictly monotone* with respect to a given line $\ell$, if any line that is orthogonal to $\ell$ intersects $C$ in at most one point. A simple polygon $P$ is said to be *monotone* with respect to a line $\ell$ if its boundary, (sometimes denoted bnd$(P)$ or $\partial P$), can be split into two chains, each of which is monotone with respect to $\ell$ (see Fig. 25(a)).

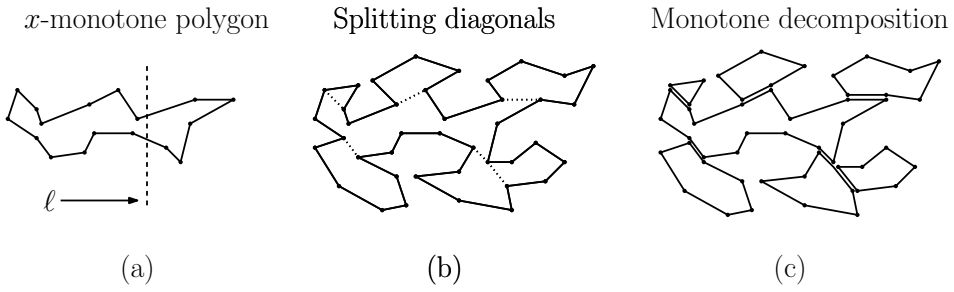



Fig. 25: Monotonicity.

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

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

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

(As an exercise, consider the problem of determining whether a polygon is monotone in *any* direction. This can be done in $O(n)$ time.)

**Triangulation of Monotone Polygons:** We begin by showing how to triangulate a monotone polygon by a simple variation of the plane-sweep method. We will return to the question of how to decompose a polygon into monotone components later.

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

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

Consider the example shown in Fig. 26. 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

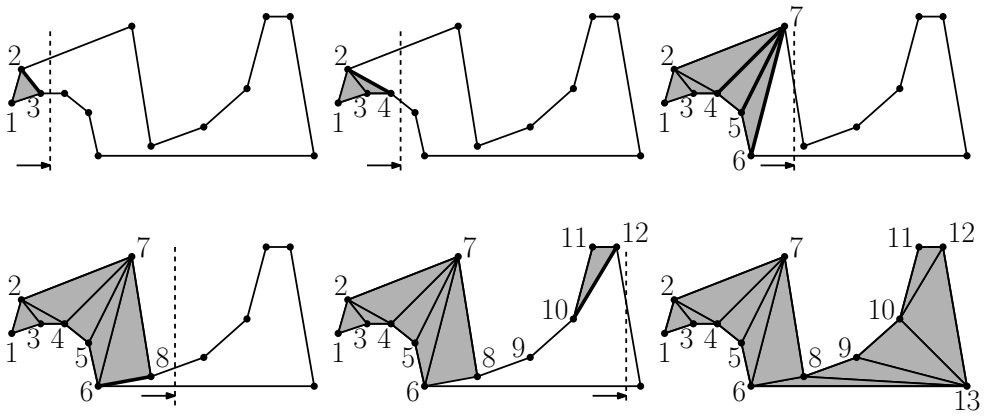



Fig. 26: Triangulating a monotone polygon.

diagonal to vertex 2. However, vertices 5 and 6 are not visible to any other nonadjacent vertices so no new diagonals can be added. When we get to vertex 7, it can be connected to 4, 5, and 6. The process continues until reaching the final vertex.

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

**Lemma:** (*Main Invariant*) For $i \geq 2$, let $v_i$ be the vertex just processed by the triangulation algorithm. The untriangulated region lying to the left of $v_i$ consists of two $x$-monotone chains, a lower chain and an upper chain each containing at least one edge. If the chain from $v_i$ to $u$ has two or more edges, then these edges form a reflex chain (that is, a sequence of vertices with interior angles all at least 180 degrees). The other chain consists of a single edge whose left endpoint is $u$ and whose right endpoint lies to the right of $v_i$ (see Fig. 27(a)).

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 complete the proof, 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) $u$ (see Fig. 27(b)). Note that all of these vertices are visible from $v_i$. Certainly $u$ is visible to $v_i$. Because the chain is reflex, $x$-monotone, and lies to the left of $v_i$ it follows that the chain itself cannot block the visibility from $v_i$ to some other vertex on the chain. Finally, the fact that the polygon is $x$-monotone implies that the unprocessed portion of the polygon (lying to the right of $v_i$) cannot "sneak back" and block visibility to the chain.

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

**Case 2:** $v$ is on the same chain as $v_{i-1}$. There are two subcases to be considered:

**Case 2(a):** The vertex $v_{i-1}$ is a nonreflex vertex (that is, its interior angle is less than 180 degrees): We walk back along the reflex chain adding diagonals joining $v_i$ to prior vertices until we find the last vertex $v_j$ of the chain that is visible to $v_i$. As can be seen in Fig. 27(c), this will involve connecting $v_i$ to one or more vertices of the chain. Remove these vertices from $v_{i-1}$ back to, but not including $v_j$

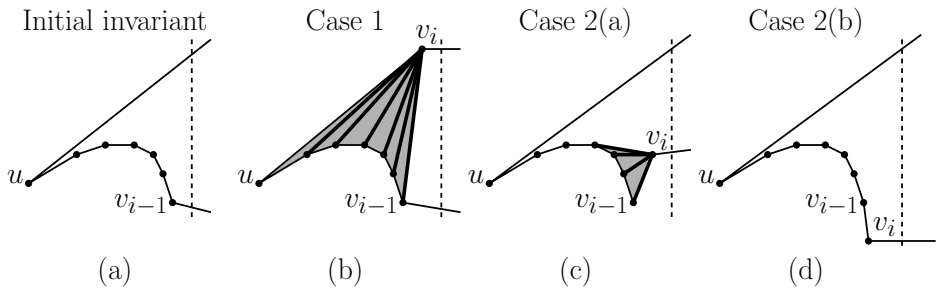



Fig. 27: Triangulation cases.

from the reflex chain. Add $v_i$ to the end of reflex chain. (You might observe a similarity between this step and the inner loop of Graham's scan.)

**Case 2(b):** The vertex $v_{i-1}$ is a reflex vertex. In this case $v_i$ cannot see any other vertices of the chain. In this case, we simply add $v_i$ to the end of the existing reflex chain (see Fig. 27(d)).

In either case, when we are done 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 the diagonal can be added in constant time. Since the number of diagonals is $n - 3$, the total time is $O(n)$.

**Monotone Subdivision:** In order to run the above triangulation algorithm, we first need to subdivide an arbitrary simple polygon $P$ into monotone polygons. This is also done by a plane-sweep approach. We will add a set of nonintersecting diagonals that partition the polygon into monotone pieces (recall Fig. 25).

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

Our approach will be to apply a left-to-right plane sweep (see Fig. 28(a)), which will add diagonals to all the split and merge vertices. We add a diagonal to each split vertex as soon as we reach it. We add a diagonal to each merge vertex when we encounter the next visible vertex to its right.

The key is storing enough information in the sweep-line status to allow us to determine where this diagonal will go. When a split vertex $v$ is encountered in the sweep, there will be an edge $e_a$ of the polygon lying above and an edge $e_b$ lying below. We might consider attaching the split vertex to left endpoint of one of these two edges, but it might be that neither endpoint is visible to the split vertex. Instead, we need to maintain a vertex that is visible to any split vertex that may arise between $e_a$ and $e_b$. To do this, imagine a sweeping a vertical segment between $e_a$ and $e_b$ to the left until it hits a vertex. Called this helper$(e_a)$ (see Fig. 28(b)).

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

Observe that *helper*$(e_a)$ is defined with respect to the current location of the sweep line. As the sweep line moves, its value changes. The helper is defined only for those edges intersected by the sweep line. Our approach will be to join each split vertex to *helper*$(e_a)$, where $e_a$ is the edge of $P$ immediately above the split vertex.

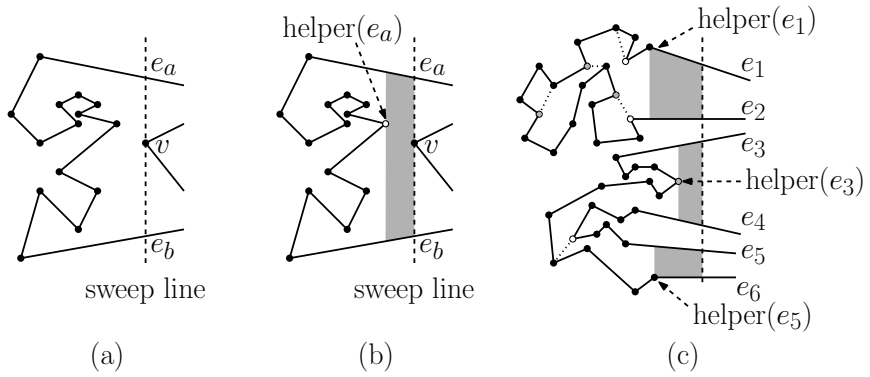



Fig. 28: Split vertices, merge vertices, and helpers.

(Note that it is possible that the helper is the left endpoint of $e_a$.) When we hit a merge vertex, we cannot add a diagonal right away. Instead, our approach is to take note of any time a helper is a merge vertex. The diagonal will be added when the very next visible vertex is processed.

**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 interior of 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), so that the operations of insert, delete, find, predecessor and successor can be evaluated in $O(\log n)$ time each.

**Event processing:** There are six event types based on a case analysis of the local structure of edges around each vertex. Let $v$ be the current vertex encountered by the sweep (see Fig. 29). Recall that, whenever we see a split vertex, we add a diagonal to the helper of the edge immediately above it. We defer adding diagonals to merge vertices until the next opportunity arises. To help with this, we define a common action called "Fix-up." It is given a vertex $v$ and an edge $e$ (either above $v$ or incident to its left). Fix-up adds a diagonal to helper($e$), if helper($e$) is a merge vertex.

**Fix-up**$(v, e)$**:** If helper($e$) is a merge vertex, add a diagonal from $v$ to this merge vertex.

**Split vertex**$(v)$**:** 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 $v$ into the sweep line status. Let $e'$ be the lower of these two edges. Make $v$ the helper of both $e$ and $e'$.

**Merge vertex**$(v)$**:** Find the two edges incident to this vertex in the sweep line status (they must be adjacent). Let $e'$ be the lower of the two. Delete them both. Let $e$ be the edge lying immediately above $v$. Fix-up$(v, e)$ and Fix-up$(v, e')$.

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

**End vertex**$(v)$**:** (Both edges lie to the left of $v$, but the interior angle is less than 180 degrees.) Let $e$ be the upper of the two edges. Fix-up$(v, e)$. Delete both edges from the sweep line status.

**Upper-chain vertex**$(v)$**:** (One edge is to the left, and one to the right, and the polygon interior is below.) Let $e$ be the edge just to the left of $v$. Fix-up$(v, e)$. Replace the edge to $v$'s left with the edge to its right in the sweep line status. Make $v$ the helper of the new edge.

**Lower-chain vertex**$(v)$**:** (One edge is to the left, and one to the right, and the polygon interior is above.) Let $e$ be the edge immediately above $v$. Fix-up$(v, e)$. Replace the edge to $v$'s left with the edge to its right in the sweep line status. Make $v$ the helper of the new edge.

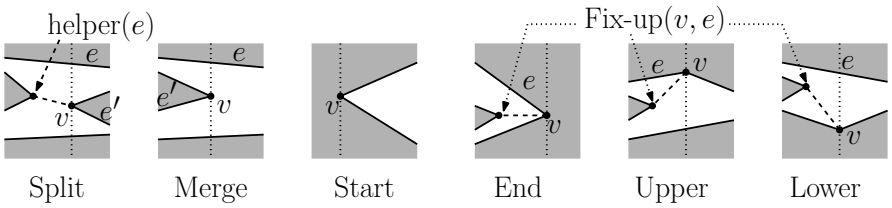



Fig. 29: Plane sweep cases, where $v$ is the vertex being swept. The label $e$ denotes the edge such that ***helper***$(e) \leftarrow v$.

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 7: Linear Programming

**Linear Programming:** One of the most important computational problems in science and engineering is linear programming, or LP for short. LP is a special case of *multi-dimensional constrained optimization problems*. In constrained optimization, the objective is to find a point in $d$-dimensional space that minimizes (or maximizes) some function, subject to various constraints on the set of allowable solutions. Linear programming is perhaps the simplest example of such a problem, since the constraints and the objective function are all linear. In spite of this apparent limitation, linear programming is a very powerful way of modeling optimization problems. Typically, linear programming is performed in spaces of very high dimension (hundreds to thousands or more), but because the focus of this course is on algorithms for low-dimensional geometric problems, we will assume that the dimension $d$ is a constant, independent of the number of constraints.

Formally, in *linear programming* we are given a set of linear inequalities, called *constraints*, in real $d$-dimensional space $\mathbb{R}^d$. Given a point $(x_1, \ldots, x_d) \in \mathbb{R}^d$, we can express such a constraint as $a_1x_1 + \ldots + a_dx_d \leq b$, by specifying the coefficient $a_i$ and $b$. (Note that there is no loss of generality in assuming that the inequality relation is $\leq$, since we can convert a $\geq$ relation to this form by simply negating the coefficients on both sides.) Geometrically, each constraint defines a closed halfspace in $\mathbb{R}^d$. The intersection of these halfspaces intersection defines a (possibly empty or possibly unbounded) polyhedron in $\mathbb{R}^d$, called the *feasible polytope*[5] (see Fig. 30(a)).

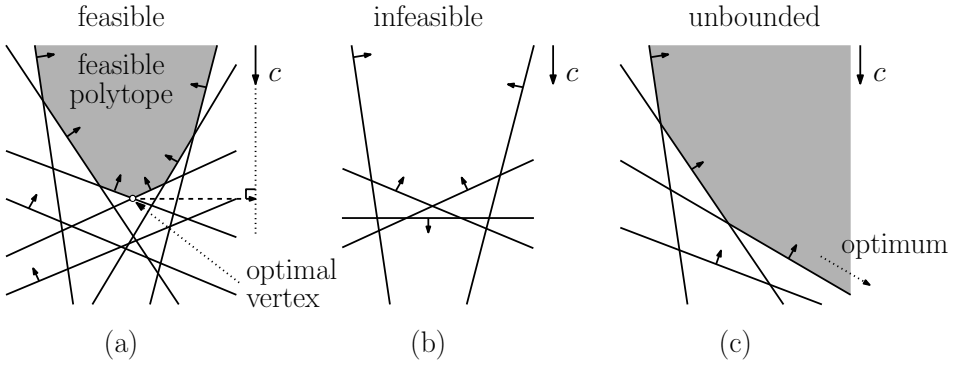



Fig. 30: 2-dimensional linear programming.

We are also given a linear *objective function*, which is to be minimized or maximized subject to the given constraints. We can express such as function as $c_1x_1 + \ldots c_dx_d$, by specifying the coefficients $c_i$. (Again, there

[5]To some geometric purists this an abuse of terminology, since a polytope is often defined to be a closed, bounded convex polyhedron, and feasible polyhedra need not be bounded.

is no essential difference between minimization and maximization, since we can simply negate the coefficients to simulate the other.) We will assume that the objective is to maximize the objective function. If we think of $(c_1, \ldots, c_d)$ as a vector in $\mathbb{R}^d$, the value of the objective function is just the projected length of the vector $(x_1, \ldots, x_d)$ onto the direction defined by the vector $c$ (see Fig. 30(a)). It is not hard to see that (assuming general position), if a solution exists, it will be achieved by a vertex of the feasible polytope, called the *optimal vertex*.

In general, a $d$-dimensional linear programming problem can be expressed as:

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

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

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

where $c$ and $x$ are $d$-vectors, $b$ is an $n$-vector and $A$ is an $n \times d$ matrix. Note that $c$ should be a nonzero vector, and $n$ should be at least as large as $d$ and may generally be much larger.

There are three possible outcomes of a given LP problem:

**Feasible:** The optimal point exists (and assuming general position) is a unique vertex of the feasible polytope (see Fig. 30(a)).

**Infeasible:** The feasible polytope is empty, and there is no solution (see Fig. 30(b)).

**Unbounded:** The feasible polytope is unbounded in the direction of the objective function, and so no finite optimal solution exists (see Fig. 30(c)).

In our figures (in case we don't provide arrows), we will assume the feasible polytope is the intersection of upper halfspaces. Also, we will usually take the objective vector $c$ to be a vertical vector pointing down. (It can point in any direction, but, if we wished, we could rotate space to make it point any direction we want.) In this setting, the problem is just that of finding the lowest vertex (minimum $y$-coordinate) of the feasible polytope.

**Linear Programming in High Dimensional Spaces:** As mentioned earlier, typical instances of linear programming may involve hundreds to thousands of constraints in very high dimensional space. It can be proved that the combinatorial complexity (total number of faces of all dimensions) of a polytope defined by $n$ halfspaces can be as high as $\Omega(n^{\lfloor d/2 \rfloor})$. In particular, the number of vertices alone might be this high. Therefore, building a representation of the entire feasible polytope is not an efficient approach (except perhaps in the plane).

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 polytope, then walking edge by edge downwards until reaching a local minimum. (By convexity, any local minimum is the global minimum.) It has been long known that there are instances where the simplex algorithm runs in exponential time, but in practice it is quite efficient.

The question of whether linear programming is even solvable in polynomial time was unknown until Khachiyan's ellipsoid algorithm (late 70's) and Karmarkar's more practical interior-point algorithm (mid 80's). Both algorithms are polynomial in the total number of bits needed to describe the input. This is called a *weakly polynomial time* algorithm. It is not known whether there is a strongly polynomial time algorithm, that is, one whose running time is polynomial in both $n$ and $d$, irrespective of the number of bits used for the input coefficients.

**Solving LP in Spaces of Constant Dimension:** There are a number of interesting optimization problems that can be posed as a low-dimensional linear programming problem. This means that the number of variables (the $x_i$'s) is constant, but the number of constraints $n$ may be arbitrarily large.

The algorithms that we will discuss for linear programming are based on a simple method called *incremental construction*. Incremental construction is among the most common design techniques in computational geometry, and this is another important reason for studying the linear programming problem.

**Deterministic Incremental Algorithm:** Recall our geometric formulation of the LP problem. We are given $n$ halfspaces $\{h_1, \ldots, h_d\}$ in $\mathbb{R}^d$ and an objective vector $c$, and we wish to compute the vertex of the feasible polytope that is most extreme in direction $c$. Our incremental approach will be based on starting with an initial solution to the LP problem for a small set of constraints, and then we will successively add one new constraint and update the solution.

In order to get the process started, we need to assume (1) that the LP is bounded and (2) we can find a set of $d$ halfspaces that provide us with an initial feasible point. Getting to this starting point is actually not trivial.[6] For the sake of focusing on the main elements of the algorithm, we will skip this part and just assume that the first $d$ halfspaces define a bounded feasible polytope (actually it will be a polyhedral cone). The the unique point where all $d$ bounding hyperplanes, $h_1, \ldots, h_d$, intersect will be our initial feasible solution. We denote this vertex as $v_d$ (see Fig. 31).

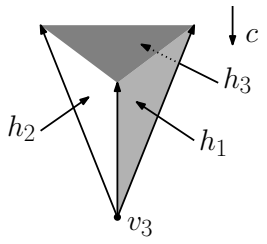



Fig. 31: Starting point of the incremental construction in $\mathbb{R}^3$.

We will then add halfspaces one by one, $h_{d+1}, h_{d+2}, \ldots$, and with each addition we update the current optimum vertex, if necessary. Let $v_i$ denote the optimal feasible vertex after the addition of $\{h_1, h_2, \ldots, h_i\}$. Notice that with each new constraint, the feasible polytope generally becomes smaller, and hence the value of the objective function at optimum vertex can only decrease. (In terms of our illustrations, the $y$-coordinate of the feasible vertex increases.)

There are two cases that can arise when $h_i$ is added. In the first case, $v_{i-1}$ lies within the halfspace $h_i$, and so it already satisfies this constraint (see Fig. 32(a)). If so, then it is easy to see that the optimum vertex does not change, that is $v_i = v_{i-1}$. In the second case $v_{i-1}$ violates constraint $h_i$. In this case we need to find a new optimum vertex (see Fig. 32(b)). Let us consider this case in greater detail.

**Updating the Optimum Vertex:** The important observation is that (assuming that the feasible polytope is not empty) the new optimum vertex must lie on the $(d-1)$-dimensional hyperplane that bounds $h_i$. Our book presents a formal proof of this fact.[7] In general, the problem can be reduced to an LP problem in one lower dimension. First, project the objective vector $c$ onto $\ell_i$, letting $c'$ be the resulting vector (see Fig. 32(c)). Next, intersect each of the halfspaces $\{h_1, \ldots, h_{i-1}\}$ with $\ell_i$. Each intersection is a $(d-1)$-dimensional halfspace that lies on $\ell_i$. We then recursively solve the $(d-1)$-dimensional LP involving these $i-1$ halfspaces with respect to $c'$. The resulting optimum vertex $v_i$ is the desired solution.

[6] Our text book explains how to overcome these assumptions in $O(n)$ additional time.

[7] Here is an intuitive argument. Let $\ell_i$ denote the bounding hyperplane. Suppose that the new optimum vertex does not lie on $\ell_i$. Draw a line segment from $v_{i-1}$ to the new optimum. Observe (1) that, by linearity, as you walk along this segment the value of the objective function decreases monotonically, and (2) that this segment must cross $\ell_i$ (because it goes from being infeasible with respect to $h_i$ to being feasible). Thus, the objective function is maximized at the crossing point, which lies on $\ell_i$.

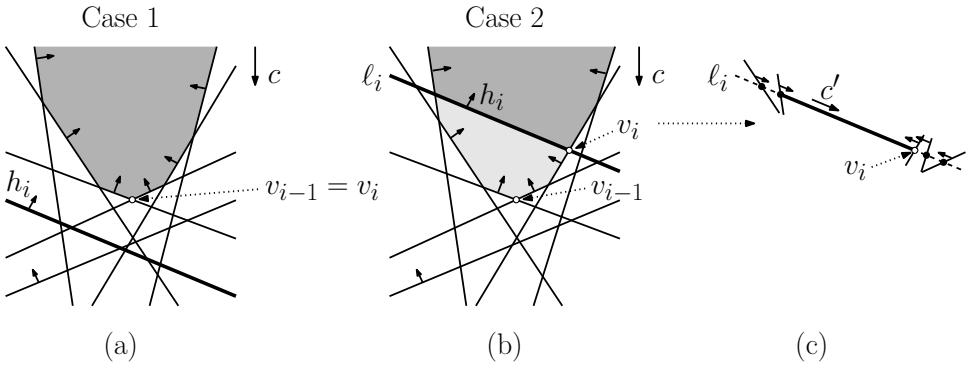



Fig. 32: Incremental construction.

Suppose for the sake of illustration that $d = 2$. In this case $\ell_i$ is a line (see Fig. 32(c)). The projected objective vector $c'$ is a vector pointing one way or the other on $\ell_i$. The intersection of each halfspace with $\ell_i$ is a ray, which can be thought of as an interval on the line that is bounded on one side and unbounded on the other. Computing the intersection of a collection of intervals on a line, is very easy and can be done in linear time, that is, $O(i-1)$ time in this case. (This interval is the heavy solid line in Fig. 32(c).) We return whichever vertex of this interval is extreme in the direction of $c'$ as the desired vertex $v_i$. If the interval is empty, then it follows that the feasible polytope is also empty, and we may terminate the algorithm immediately and report that there is no solution. Because, by assumption, the original LP is bounded, it follows that the $(d-1)$-dimensional LP is also bounded.

**Worst-Case Analysis:** What is the running time of this algorithm? Ignoring the initial $d$ halfspaces, there are $n-d$ halfspace insertions performed. In step $i$, we may find that the current optimum vertex is feasible. This takes $O(d)$ time. The alternative is that we need to solve a $(d-1)$-dimensional LP with $i-1$ constraints. It takes $O(d(i-1))$ to intersect each of the constraints with $\ell_i$ and $O(d)$ time to project $c$ onto $\ell_i$. If we let $T_d(n)$ denote the time to run this algorithm in dimension $d$ with $n$ constraints. In this case the time is $O(di + T_{d-1}(i-1))$. Since there are two alternatives, the running time is the maximum of the two. Ignoring constant factors, the running time can be expressed by the following recurrence formula:

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \Big( \max\big(d, di + T_{d-1}(i-1)\big) \Big).$$

Since $d$ is a constant, we can simplify this to:

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \big(i + T_{d-1}(i-1)\big).$$

The basis case of the recurrence occurs when $d = 1$, and we just solve the interval intersection problem described above in $O(n)$ time by brute force. Thus, we have $T_1(n) = n$.

Unfortunately, this recurrence solves to $T_d(n) = O(n^d)$, which is not very efficient. We can see this by induction. In particular, let's try to prove that, for some constant $\alpha$, we have $T_d(n) \le \alpha n^d$. We'll skip the basis case (which is easy). In general, for $d \ge 2$, we have

$$T_d(n) \;=\; \sum_{i=d+1}^{n} \big(i + T_{d-1}(i-1)\big) \;\le\; \sum_{i=d+1}^{n} \big(i + \alpha(i-1)^{d-1}\big) \;\le\; \sum_{i=1}^{n} \alpha n^{d-1} \;\le\; \alpha n^d.$$

(Although this analysis is quite crude, it can be shown to be asymptotically tight.)

Notice that this worst-case analysis is based on the rather pessimistic assumption that the current vertex is *always infeasible*. Although there may exist insertion orders for which this might happen, we might wonder whether we can arrange the insertion order so this worst case does not occur. We'll consider this alternative next.

**Randomized Algorithm:** Suppose that we apply the above algorithm, but we insert the halfspaces in *random order* (except for the first $d$, which need to be chosen to provide an initial feasible vertex.) This is an example of a general class of algorithms called *randomized incremental algorithms*. There is only one difference between this algorithm and the deterministic one, namely, just prior to running the incremental algorithm, we call a procedure that randomly permutes the initial input list (excluding the first $d$ halfspaces). A description is given in the code block below.

---

Randomized Incremental $d$-Dimensional Linear Programming

Input: Let $H$ be a set of $n$ $(d-1)$-dimensional halfspaces, such that the first $d$ define an initial feasible vertex $v_d$, and let $c$ be the objective function vector.

(1) Let $v_d$ be the intersection point of the hyperplanes bounding $h_1, \ldots, h_d$, which we assume define an initial feasible vertex. Randomly permute the remaining halfspaces, and let $\langle h_{d+1}, \ldots, h_n \rangle$ denote the resulting sequence.

(2) For $i = d+1$ to $n$ do:

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

  (b) Otherwise, intersect $\{h_1, h_2, \ldots, h_{i-1}\}$ with the $(d-1)$-dimensional hyperplane $\ell_i$ that bounds $h_i$. Let $c'$ be the projection of $c$ onto $\ell_i$. Solve the resulting $(d-1)$-dimensional LP recursively. (When the dimension falls to 1, we can just solve the problem brute force by intersecting up to $n$ intervals.)

    (i) If the $(d-1)$-dimensional LP is infeasible, terminate and report that the LP is infeasible.

    (ii) Otherwise, let $v_i$ be the solution to the $(d-1)$-dimensional LP.

(3) Return $v_n$ as the final solution.

---

What is the expected case running time of this randomized incremental algorithm? Note that the expectation is over the random permutation of the insertion order. We make *no assumptions* about the distribution of the input. (Thus, the analysis is in the worst-case with respect to the input, but in the expected case with respect to random choices.)

The number of random permutations is $(n-d)!$, but it will simplify things to pretend that we permute all the halfspaces, and so there are $n!$ permutations. Each permutation has an equal probability of $1/n!$ of occurring, and an associated running time. However, presenting the analysis as sum of $n!$ terms does not lead to something that we can easily simplify. We will apply a technique called *backwards analysis*, which is quite useful.

**Warm-Up Exercise for Backwards Analysis:** To motivate how backwards analysis works, let us consider a much simpler example, namely the problem of computing the minimum of a set of $n$ distinct numbers. We permute the numbers and inspect them one-by-one. We maintain a variable that holds the minimum value seen so far. If we see a value that is smaller than the current minimum, then we update the minimum. The question we will consider is, on average how many times is the minimum value updated? Below are three sequences that illustrate that the minimum may updated once (if the numbers are given in increasing order), $n$ times (if given in decreasing order). Observe that in the third sequence, which is random, the minimum does not change very often at all.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\underline{0}$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| $\underline{14}$ | $\underline{13}$ | $\underline{12}$ | $\underline{11}$ | $\underline{10}$ | $\underline{9}$ | $\underline{8}$ | $\underline{7}$ | $\underline{6}$ | $\underline{5}$ | $\underline{4}$ | $\underline{3}$ | $\underline{2}$ | $\underline{1}$ | $\underline{0}$ |
| $\underline{5}$ | 9 | $\underline{4}$ | 11 | $\underline{2}$ | 6 | 8 | 14 | $\underline{0}$ | 3 | 13 | 12 | 1 | 7 | 10 |

Let $p_i$ denote the probability that the minimum value changes on inspecting the $i$th number of the random permutation. Thus, with probability $p_i$ the minimum changes (and we add 1 to the counter for the number of changes) and with probability $1 - p_i$ it does not (and we add 0 to the counter for the number of changes). The

total expected number of changes is

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

It suffices to compute $p_i$. We might be tempted to reason as follows. Let us consider a random subset of the first $i-1$ values, and then consider all the possible choices for the $i$th value. However, this leads to a complicated analysis involving conditional probabilities. Let us instead consider an alternative approach, in which we work backwards. In particular, let us consider a random set of $i$ values, and consider the probability the *last value added to this set resulted in a change in the minimum*.

To make this more formal, let $S_i$ be an arbitrary subset of $i$ numbers from our initial set of $n$. (In theory, the probability is conditional on the fact that the elements of $S_i$ represent the first $i$ elements to be chosen, but since the analysis will not depend on the particular choice of $S_i$, it follows that the probability that we compute will hold unconditionally.) Among all $i!$ permutations of the elements of $S_i$, in how many of these does the minimum change in the transition from the $(i-1)$-st stage to the $i$th stage? The key observation is that the minimum only changes for those sequences in which the minimum element was the last ($i$th) element of the sequence. Since the minimum item appears with equal probability in each of the $i$ positions of a random sequence, the probability that it appears last is exactly $1/i$. Thus, $p_i = 1/i$. From this we have

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

This summation $\sum_i (1/i)$ is called the *Harmonic series* and the fact that it is nearly equal to $\ln n$ is a well known fact. (See any text on probability theory.)

This is called a *backwards analysis* because the analysis works by considering the possible random transitions that brought us to $S_i$ from $S_{i-1}$, as opposed to working forward from $S_{i-1}$ to $S_i$. Of course, the probabilities are no different whether we consider the random sequence backwards rather than forwards, so this is a perfectly accurate analysis. It's arguably simpler and easier to understand.

**Backwards Analysis for Randomized LP:** Let us apply this same approach to the analysis of the running time of the randomized incremental linear programming algorithm. We will do the analysis in $d$-dimensional space. Let $T_d(n)$ denote the expected running time of the algorithm on a set of $n$ halfspaces in dimension $d$. We will prove by induction that $T_d(n) \le \gamma d!\, n$, where $\gamma$ is some constant that does not depend on dimension. It will make the proof simpler if we start by proving that $T_d(n) \le \gamma_d d!\, n$, where $\gamma_d$ does depend on dimension, and later we will eliminate this dependence.

For $d+1 \le i \le n$, let $p_i$ denote the probability that the insertion of the $i$th hyperplane in the random order results in a change in the optimum vertex.

**Case 1:** With probability $(1-p_i)$ there is no change. It takes us $O(d)$ time to determine that this is the case.

**Case 2:** With probability $p_i$, there is a change to the optimum. First we project the objective vector onto $\ell_i$ (which takes $O(d)$ time), next we intersect the existing $i-1$ halfspaces with $\ell_i$ (which takes $O(d(i-1))$ time). Together, these last two steps take $O(di)$ time. Finally we invoke a $(d-1)$-dimensional LP on a set of $i-1$ halfspaces in dimension $d-1$. By the induction hypothesis, the running time of this recursive call is $T_{d-1}(i-1)$.

Combining the two cases, up to constant factors (which don't depend on dimension), we have a total expected running time of

$$T_d(n) \;\le\; \sum_{i=d+1}^{n} \Big((1-p_i)d + p_i\big(di + T_{d-1}(i)\big)\Big) \;\le\; \sum_{i=d+1}^{n} \big(d + p_i\big(di + T_{d-1}(i)\big)\big).$$

It remains is to determine what $p_i$ is. To do this, we will apply the same backward-analysis technique as above. Let $S_i$ denote an arbitrary subset consisting of $i$ of the original halfspaces. Again, it will simplify things to assume that all the $i$ hyperplanes are being permuted (not just the last $i - d$). Among all $i!$ permutations of $S_i$, in how many does the optimum vertex change with the $i$th step? Let $v_i$ denote the optimum vertex for these $i$ halfspaces. It is important to note that $v_i$ only depends on the set $S_i$ and not on the order of their insertion. (You might think about why this is important.)

Assuming general position, there are $d$ halfspaces whose intersection defines $v_i$. (For example, in Fig. 33(a), we label these halfspaces as $h_4$ and $h_7$.)

- If none of these $d$ halfspaces were the last to be inserted, then $v_i = v_{i-1}$, and there is no change. (As is the case in Fig. 33(b), where $h_5$ is the last to be inserted.)
- On the other hand, if any of them were the last to be inserted, then $v_i$ did not exist yet, and hence the optimum must have changed as a result of this insertion. (As is the case in Fig. 33(c), where $h_7$ is the last to be inserted.)

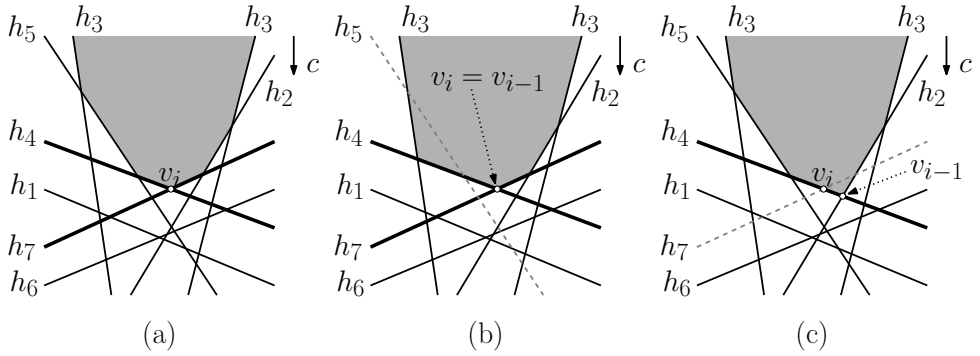



Fig. 33: Backwards analysis for the randomized LP algorithm.

Thus, the optimum changes if and only if either one of the $d$ defining halfspaces was the last halfspace inserted. Since all of the $i$ halfspaces are equally likely to be last, this happens with probability $d/i$. Therefore, $p_i = d/i$.

This probabilistic analysis has been conditioned on the assumption that $S_i$ was the subset of halfspace seen so far, but since the final probability does not depend on any properties of $S_i$ (just on $d$ and $i$), the probabilistic analysis applies unconditionally to all subsets of size $i$.

Returning to our analysis, since $p_i = d/i$, and applying the induction hypothesis that $T_{d-1}(i) = \gamma_{d-1}(d-1)!\, i$, we have

$$\begin{aligned} T_d(n) &\leq \sum_{i=d+1}^{n} \big(d + p_i\big(di + T_{d-1}(i)\big)\big) \leq \sum_{i=d+1}^{n} \left(d + \frac{d}{i}\big(di + \gamma_{d-1}(d-1)!\, i\big)\right) \\ &\leq \sum_{i=d+1}^{n} (d + d^2 + \gamma_{d-1} d!) \leq (d + d^2 + \gamma_{d-1} d!)n. \end{aligned}$$

To complete the proof, we just need to select $\gamma_d$ so that the right hand side is at most $\gamma_d d!$. To achieve this, it suffices to set

$$\gamma_d = \frac{d + d^2}{d!} + \gamma_{d-1}.$$

Plugging this value into the above formula yields

$$T_d(n) \leq (d + d^2 + \gamma_{d-1} d!)n \leq \left(\frac{d + d^2}{d!} + \gamma_{d-1}\right) d!\, n \leq \gamma_d d!\, n,$$

as desired.

As mentioned above, we don't like the fact that the "constant" $\gamma_d$ changes with the dimension. To remedy this, note that because $d!$ grows so rapidly compared to either $d$ or $d^2$, it is easy to show that $(d + d^2)/d! \leq 1/2^d$ for almost all sufficiently large values of $d$. Because the geometric series $\sum_{d=1}^{\infty} 1/2^d$, converges, it follows that there is a constant $\gamma$ (independent of dimension) such that $\gamma_d \leq \gamma$ for all $d$. Thus, we have that $T_d(n) \leq O(d!\, n)$, where the constant factor hidden in the big-Oh does not depend on dimension.

In summary, we have presented a simple and elegant randomized incremental algorithm for solving linear programming problems. The algorithm runs in $O(n)$ time in expectation. (Remember that expectation does *not* depend on the input, only on the random choices.) Unfortunately, our assumption that the dimension $d$ is a constant is crucial. The factor $d!$ grows so rapidly (and it seems to be an unavoidable part of the analysis) that this algorithm is limited to fairly low dimensional spaces.

You might be disturbed by the fact that the algorithm is not deterministic, and that we have only bounded the expected case running time. Might it not be the case that the algorithm takes ridiculously long, degenerating to the $O(n^d)$ running time, on very rare occasions? The answer is, of course, yes. In his original paper, Seidel proves that the probability that the algorithm exceeds its running time by a factor $b$ is $O((1/c)^{b\,d!})$, for any fixed constant $c$. For example, he shows that in 2-dimensional space, the probability that the algorithm takes more than 10 times longer than its expected time is at most $0.0000000000065$. You would have a much higher probability be being struck by lightning *twice* in your lifetime!

# Lecture 8: Halfplane Intersection and Point-Line Duality

**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 and the topic of convex hulls. Any line in the plane splits the plane into two regions, one lying on either side of the line. Each such region is called a *halfplane*. (In $d$-dimensional space the corresponding notion is a *halfspace*, which consists of the space lying to one side of a $(d-1)$-dimensional hyperplane.) We say that a halfplane is either *closed* or *open* depending on whether or not it contains the line itself. For this lecture we will be dealing entirely with closed halfplanes.

How do we represent lines and halfplanes? For our purposes (since, by general position, we may assume we are dealing only with nonvertical lines), it will suffice to represent lines in the plane using the following equation:

$$y = ax - b,$$

where $a$ denotes the slope and $b$ denotes the negation of the $y$-intercept. (We will see later why it is convenient to negate the intercept value.) Note that this is not fully general, since it cannot handle vertical lines (which have infinite slope). Each nonvertical line defines two closed *halfplanes*, consisting of the points on or below the line and the points on or above the line:

$$\text{lower (closed) halfplane: } y \leq ax - b \qquad \text{upper (closed) halfplane: } y \geq ax - b.$$

**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. (Fig. 34 illustrates the intersection of a collection of upper halfspaces.) Unlike the convex hull problem, the intersection of $n$ halfplanes may generally be empty or even unbounded. A natural output representation might be to list the lines bounding the intersection in counterclockwise order.

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 halfplanes? As with the convex hull problem, it can be shown, through a suitable reduction from sorting, that the problem has a lower bound of $\Omega(n \log n)$.

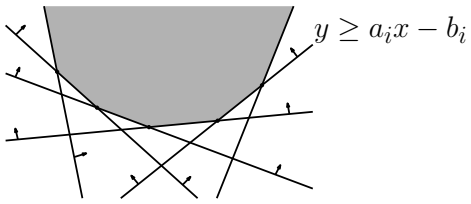



Fig. 34: Halfplane intersection.

Who cares about this problem? Halfplane intersection and halfspace intersection in higher dimensions are also used in generating convex shape approximations. For example, in robotics and computer graphics, rather than computing collisions with a complex shape, it is easier to first check for collisions with a enclosing convex approximation to the shape. Also, many optimization problems can be expressed as minimization problems over a convex domain, and these domains are represented by the intersection of halfspaces.

Solving the halfspace intersection problem in higher dimensions is quite a bit more challenging than in the plane. In general, the worst-case total combinatorial complexity the intersection of $n$ halfspaces in $\mathbb{R}^d$ can be as high as $\Theta(n^{\lfloor d/2 \rfloor})$. For example, the boundary of the intersection of halfspaces in dimension $d$ is a $(d-1)$-dimensional cell complex, and would require an appropriate data structure for storing such objects.

We will discuss two algorithms for the halfplane intersection problem. The first is given in the text, and involves an interesting combination of two techniques we have discussed for geometric problems, geometric divide-and-conquer and plane sweep. 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 $K_1$ and $K_2$ be the results.

(4) Intersect the convex polygons $K_1$ and $K_2$ (which might be unbounded) into a single convex polygon $K$, and return $K$.

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:

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

where $M(n)$ is the time required to merge the two results, that is, to compute the intersection of two convex polygons whose total complexity is $n$. We will show below that $M(n) = O(n)$, and so it follows by standard results in recurrences that the overall running time $T(n)$ is $O(n \log n)$. (See any standard algorithms textbook.)

**Intersecting Two Convex Polygons:** The only nontrivial part of the process is implementing an algorithm that intersects two convex polygons, $K_1$ and $K_2$, 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. (As an exercise, try to prove this.) This would given $O(n \log n)$ algorithm for computing the intersection. This is too slow, however, and would result in an overall time of $O(n \log^2 n)$ for $T(n)$.

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

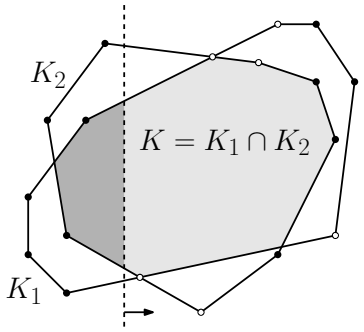



Fig. 35: Intersecting two convex polygons by plane sweep.

We perform a left-to-right plane sweep to compute the intersection (see Fig. 35). We begin by breaking the boundaries of the convex polygons into their upper and lower chains. (This can be done in $O(n)$ time.) By convexity, the sweep line intersects the boundary of each convex polygon $K_i$ in at most two points, and hence, there are at most four points in the sweep line status at any time. Thus, we do not need a ordered dictionary for storing the sweep line status—a simple 4-element list suffices. Also, our event queue need only be of constant size. At any point there are at most 8 possible candidates for the next event, namely, the right endpoints of the four edges stabbed by the sweep line and the (up to four) intersection points of these upper and lower edges of $K_1$ with the upper and lower edges of $K_2$. Since there are only a constant number of possible events, and each can be handled in $O(1)$ time, the total running time is $O(n)$.

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

The problem to consider is called the *lower envelope* problem, and it is a special case of the halfplane intersection problem. We are given a set of $n$ lines $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ where $\ell_i$ 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 (see Fig. 36). The *upper envelope* is defined symmetrically.

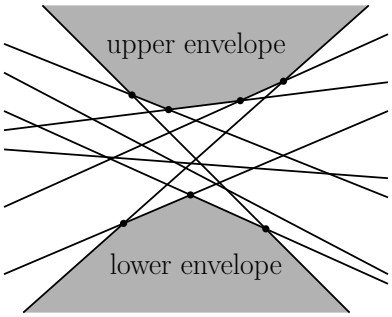



Fig. 36: Lower and upper envelopes.

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.

We will see that solving the lower envelope problem is very similar to solving the upper convex hull problem. In fact, they are so similar 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 interpret the inputs and the outputs.

**Lines, Points, and Incidences:** In order to motivate duality, let us discuss the representation of 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 $b$. (Why $-b$ rather than $+b$? The distinction is unimportant, but it will simplify some of the notation defined below.) We cannot represent vertical lines in this way, and for now we will just ignore them.

Therefore, in order to describe a line in the plane, you need only give its two coefficients $(a, b)$. Thus, lines in the plane can be thought of as points in a new 2-dimensional space, in which the coordinate axes are labeled $(a, b)$, rather than $(x, y)$. For example, the line $\ell : y = 2x + 1$ corresponds to the point $(2, -1)$ in this space, which we denote by $\ell^*$. Conversely, each point $p = (a, b)$ in this space of "lines" corresponds to a nonvertical line, $y = ax - b$ in the original plane, which we denote by $p^*$. We will call the original $(x, y)$-plane the *primal plane*, and the new $(a, b)$-plane the *dual plane*.

This insight would not be of much use unless we could say something about how geometric relationships in one space relate to the other. The connection between the two involves incidences between points and line. Two lines determine a point through intersection. Two points determine a line, by taking their affine combination. Later, we'll show that these relationships are preserved by duality. For example, consider the two lines $\ell_1 : y = 2x+1$ and the line $\ell_2 : y = -\frac{x}{2} + 6$ (see Fig. 37(a)). These two lines intersect at the point $p = (2, 5)$. The duals of these two lines are $\ell_1^* = (2, -1)$ and $\ell_2^* = \left(-\frac{1}{2}, -6\right)$. The line in the $(a, b)$ dual plane passing through these two points is easily verified to be $b = 2a - 5$. Observe that this is exactly the dual of the point $p$ (see Fig. 37(b)). (As an exercise, prove this for two general lines.)

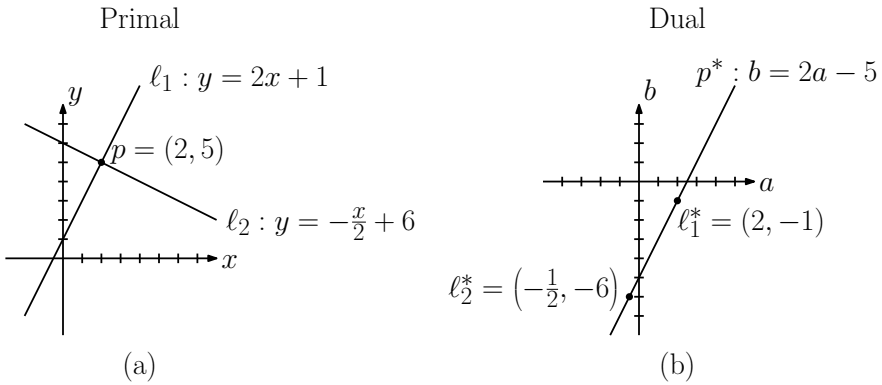



Fig. 37: The primal and dual planes.

**Point-Line Duality:** Let us explore this dual transform more formally. Duality (or more specifically *point-line duality*) is a transformation that maps points in the plane to lines and lines to point. (More generally, it maps points in $d$-space to hyperplanes dimension $d$.) We denote this transformation using a asterisk ($*$) as a superscript. Thus, given point $p$ and line $\ell$ in the primal plane we define $\ell^*$ and $p^*$ to be a point and line, respectively, in the

dual plane defined as follows.[8]

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

It is convenient to define the dual transformation so that it is its own inverse (that is, it is an involution). In particular, it maps points in the dual plane to lines in the primal, and vice versa. For example, given a point $p = (p_a, p_b)$ in the dual plane, its dual is the line $y = p_a x - p_b$ in the primal plane, and is denoted by $p^*$. It follows that $p^{**} = p$ and $\ell^{**} = \ell$.

**Properties of Point-Line Duality:** 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 $p$ is above/on/below line $\ell$ in the primal plane if and only if line $p^*$ is below/on/above point $\ell^*$ in the dual plane, respectively (see Fig. 38).

**Intersection preserving:** Lines $\ell_1$ and $\ell_2$ intersect at point $p$ if and only if the dual line $p^*$ passes through points $\ell_1^*$ and $\ell_2^*$.

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

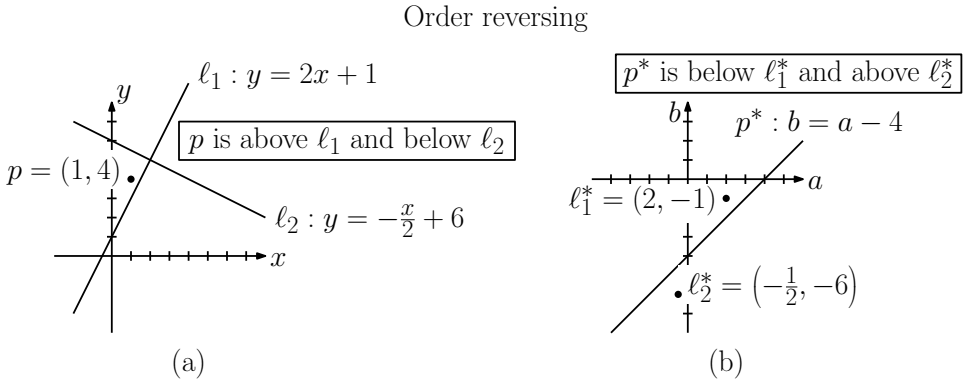



Fig. 38: The order-reversing property.

The self inverse property was already established (essentially by definition). To verify the order reversing property, consider any point $p$ and any line $\ell$.

$$p \text{ is on or above } \ell \iff p_y \geq \ell_a p_x - \ell_b \iff \ell_b \geq p_x \ell_a - p_y \iff p^* \text{ is on or below } \ell^*$$

(From this is should be apparent why we chose to negate the $y$-intercept when dualizing points to lines.) The other two properties (intersection preservation and collinearity/coincidence are direct consequences of the order reversing property.)

**Convex Hulls and Envelopes:** Let us return now to the question of the relationship between convex hulls and the lower/upper envelopes of a collection of lines in the plane. The following lemma demonstrates the, under the duality transformation, the convex hull problem is dually equivalent to the problem of computing lower and upper envelopes.

[8]Duality can be generalized to higher dimensions as well. In $\mathbb{R}^d$, let us identify the $y$ axis with the $d$-th coordinate vector, so that an arbitrary point can be written as $p = (x_1, \ldots, x_{d-1}, y)$ and a $(d-1)$-dimensional hyperplane can be written as $h : y = \sum_{i=1}^{d-1} a_i x_i - b$. The dual of this hyperplane is $h^* = (a_1, \ldots, a_{d-1}, -b)$ and the dual of the point $p$ is $p^* : b = \sum_{i=1}^{d-1} x_i a_i - y$. All the properties defined for point-line relationships generalize naturally to point-hyperplane relationships.

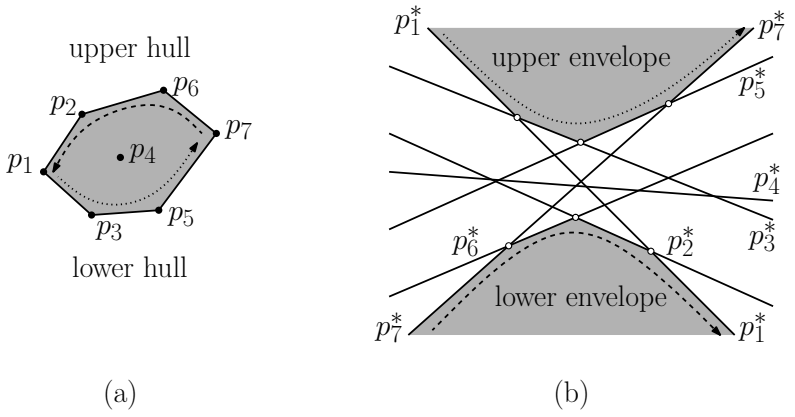



Fig. 39: Equivalence of hulls and envelopes.

**Lemma:** Let $P$ be a set of points in the plane. The counterclockwise order of the points along the upper (lower) convex hull of $P$ (see Fig. 39(a)), is equal to the left-to-right order of the sequence of lines on the lower (upper) envelope of the dual $P^*$ (see Fig. 39(b)).

**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.

Consider a pair of points $p_i$ and $p_j$ that are consecutive vertices on the upper convex hull. This is equivalent to saying that all the other points of $P$ lie beneath the line $\ell_{ij}$ that passes through both of these points.

Consider the dual lines $p_i^*$ and $p_j^*$. By the incidence preserving property, the dual point $\ell_{ij}^*$ is the intersection point of these two lines. (By general position, we may assume that the two points have different $x$-coordinates, and hence the lines have different slopes. Therefore, they are not parallel, and the intersection point exists.)

By the order reversing property, all the dual lines of $P^*$ pass above point $\ell_{ij}^*$. This is equivalent to saying the $\ell_{ij}^*$ lies on the lower envelope of $P^*$.

To see how the order of points along the hulls are represented along the lower envelope, observe that as we move counterclockwise along the upper hull (from right to left), 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 cryptic 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. The places where the two envelopes wraps around correspond to the vertical lines (having infinite slope) passing through the left and right endpoints of the hull. (As an exercise, can you see which is which?)

# Lecture 9: Trapezoidal Maps

**Trapezoidal Map:** Many techniques in computational geometry are based on generating some sort of organizing structure to an otherwise unorganized set of geometric objects. We have seen triangulations as one example, where the interior of a simple polygon is subdivided into triangles. Today, we will consider a considerably more general method of defining a subdivision of the plane into simple regions. It works not only for simple polygons but for much more general inputs as well.

Let $S = \{s_1, \ldots, s_n\}$ be a set of line segments in the plane, such that the segments do not intersect one another, except where the endpoint of one segment intersect the endpoint of another segment. (Note that any planar

straight-line subdivision could be expressed in this form.) Let us assume that no two segment endpoints share the same $x$-coordinate (except when two or more segments share a common endpoint). This implies that there are no vertical segments.

We wish to produce a subdivision of space that "respects" these line segments. To do so, we start by enclosing all the segments within a large bounding rectangle (see Fig. 40(a)). This is mostly a convenience, so we don't have to worry about unbounded regions. Next, imagine shooting a *bullet path* vertically upwards and downwards from the endpoints of each segment of $S$ until it first hits another segment of $S$ or the top or bottom of the bounding rectangle. The combination of the original segments and these vertical bullet paths defines a subdivision of the bounding rectangle called the *trapezoidal map* of $S$ (see Fig. 40(b)).

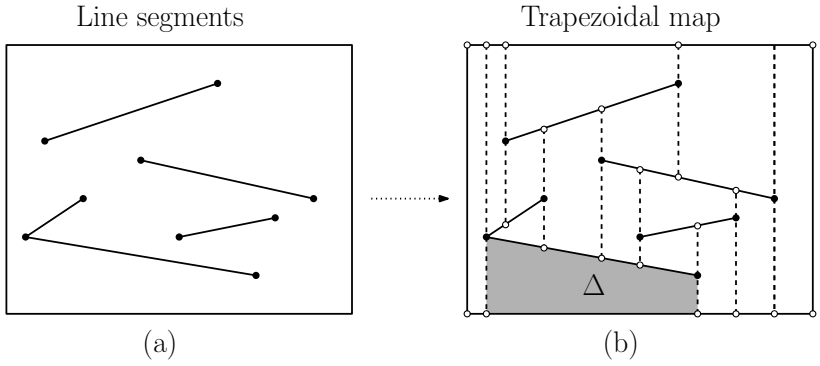



Fig. 40: A set of segments and the associated trapezoidal map.

The faces of the resulting subdivision are generally trapezoids with vertical sides, but they may degenerate to triangles in some cases. The vertical sides are sometimes called *walls*. Also observe that it is possible that the nonvertical side of a trapezoid may have multiple vertices along the interior of its top or bottom side. (See the trapezoid labeled $\Delta$ in Fig. 40.)

We claim that the process of converting an arbitrary polygonal subdivision into a trapezoidal decomposition increases its size by at most a constant factor. We derive the exact expansion factor in the next claim.

**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. The remaining four come from the bounding rectangle.

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

An important fact to observe about each trapezoid is that it is *defined* (that is, its existence is determined) by exactly four entities from the original subdivision: a segment on top, a segment on the bottom, a bounding vertex on the left, and a bounding vertex on the right. The bounding vertices may be endpoints of the upper or lower segments, or they may below to completely different segments. This simple observation will play an important role later in the analysis.

**Construction:** We could construct the trapezoidal map easily by plane sweep. (This should be an easy exercise by this point. You might think about how you would do it.) Instead, we will build the trapezoidal map by a different

approach, namely a randomized incremental algorithm.[9] Later, when we discuss the point-location problem, we will see the advantages of this approach.

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 this update, we need to know which trapezoid the left endpoint of the segment lies in. We will address this question later when we discuss point location. We then trace the line segment from left to right, by "walking" it through the existing trapezoidal map (see Fig. 41). Along the way, we discover which existing trapezoids it intersects. We go back to these trapezoids and "fix them up". There are two things that are involved in fixing process.

- The left and right endpoints of the new segment need to have bullets fired from them.
- One of the earlier created walls might hit the new line segment. When this happens the wall is trimmed back. (We store which vertex shot the bullet path for this wall, so we know which side of the wall to trim.)

The process is illustrated in Fig. 41.

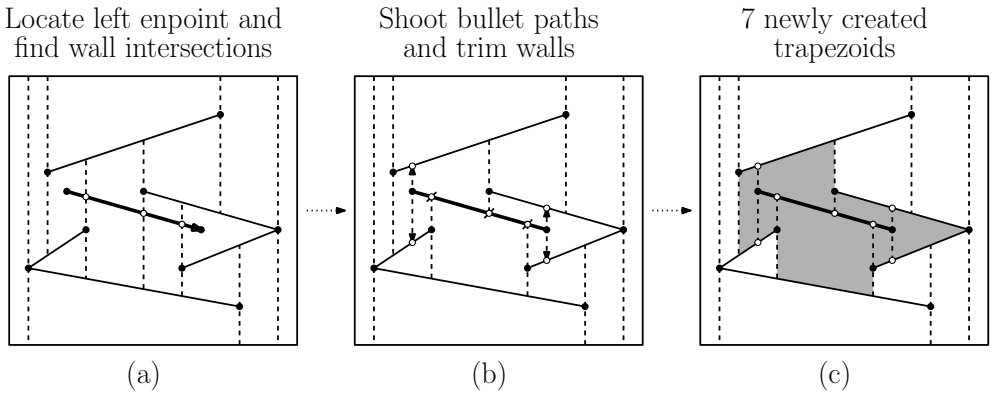



Fig. 41: Incremental update.

Observe that the structure of the trapezoidal decomposition does *not* depend on the order in which the segments are added. (This is why trimming back walls is so important.) 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 $i$th 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 $i$th segment, and let $K$ denote the number of existing walls that this segment intersects. We need to shoot four bullets (two from each endpoint) and then trim each of the $K$ walls, for a total of $K + 4$ operations that need to be performed. If the new segment did not cross any of the walls, then we would get exactly four new trapezoids. For each of the $K$ walls 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 as a planar subdivision, for example, a doubly connected edge list (DCEL).

[9]Historically, the randomized incremental algorithm arose as a method for computing the intersection of a collection of line segments. Given $n$ line segments that have $I$ intersections, this algorithm runs in $O(I + n \log n)$ time, which is superior to the plane sweep algorithm we discussed earlier.

**Analysis:** We will analyze the expected time to build the trapezoidal map, assuming that segments are inserted in random order. Clearly, the running time depends on how many walls are trimmed with each intersection. In the worst case, each newly added segment could result in $\Omega(n)$ walls being trimmed, and this would imply an $\Omega(n^2)$ running time. We will show, however, that the expected running time is much smaller, in fact, we will show the rather remarkable fact that, each time we insert a new segment, the expected number of wall trimmings is just $O(1)$. (This is quite surprising at first. If many of the segments are long, it might seem that every insertion would cut through $O(n)$ trapezoids. What saves us is that, although a long segment might cut through many trapezoids, it shields later segments from cutting through many trapezoids.) As was the case in our earlier lecture on linear programming, we will make use of a backwards analysis to establish this result.

There are two things that we need to do when each segment is inserted. First, we need to determine which cell of the current trapezoidal map contains its left endpoint. We will not discuss this issue today, but in our next lecture, we will show that the expected time needed for this operation is $O(n \log n)$. Second, we need to trim the walls that are intersected by the new segment. The remainder of this lecture will focus on this aspect of the running time.

From the previous claim, we know that it suffices to count the number of new trapezoids created with each insertion. The main result that drives the analysis is presented in the next lemma.

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

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

Let $\mathcal{T}_i$ denote the trapezoidal map resulting after the insertion of the $i$th 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.

We say that a trapezoid $\Delta$ of the existing map *depends* on an segment $s$, if $s$ would have caused $\Delta$ to be created, had $s$ been added last (see Fig. 42). 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 $\Delta$ depends on $s$, and 0 otherwise, then the expected value is

$$E(k_i) \;=\; \frac{1}{i}\sum_{s \in S_i} (\text{no. of trapezoids that depend on } s) \;=\; \frac{1}{i}\sum_{s \in S_i}\sum_{\Delta \in \mathcal{T}_i} \delta(\Delta, s).$$

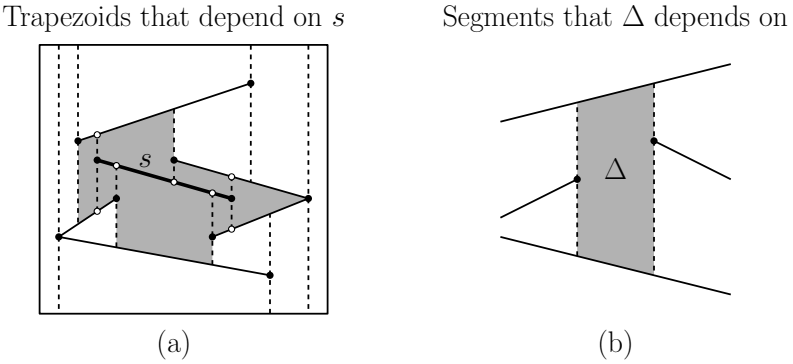



Fig. 42: Trapezoid-segment dependencies.

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. (This is the old combinatorial trick of reversing the order of summation.) In other words we can express the above quantity as:

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

This quantity is much easier to analyze. In particular, each trapezoid is bounded by at most four sides. (The reason it is "at most" is that degenerate trapezoids are possible which may have fewer 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.[10]

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

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

Since the expected number of new trapezoids created with each insertion is $O(1)$, it follows that the total number of trapezoids that are created in the entire process is $O(n)$. This fact is important in bounding the total time needed for the randomized incremental algorithm.

The only question that we have not considered in the construction is how to locate the trapezoid that contains left endpoint of each newly added segment. We will consider this question, and the more general question of how to do point location in our next lecture.

# Lecture 10: Trapezoidal Maps and Planar Point Location

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

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

**Point Location Data Structure:** The point location data structure is based on a rooted directed acyclic graph (DAG). Each node will have either zero or two outgoing edges. Nodes with zero outgoing edges are called *leaves*. The leaves will be in 1–1 correspondence with the trapezoids of the map. The other nodes are called *internal nodes*, and they are used to guide the search to the leaves. This DAG can be viewed as a variant of a binary tree, where subtrees may be shared between different nodes. (This sharing is important for keeping the space to $O(n)$.)

There are two types of internal nodes, *$x$-nodes* and *$y$-nodes*. Each $x$-node contains the point $p$ (an endpoint of one of the segments), and its two children correspond to the points lying to the left and to the right of the vertical

[10] There is a bit of a subtlety here. What if multiple segments share the endpoint? Note that the trapezoid is only dependent on the first such segment to be added, since this is the segment that caused the vertex to come into existence. Also note that the same segment that forms the top or bottom side might also provide the left or right endpoint. These considerations only decrease the number of segments on which a trapezoid depends.

line passing through $p$ (see Fig. 44(a)). Each $y$-node contains a pointer to a line segment of the subdivision, and the left and right children correspond to whether the query point is above or below the line containing this segment, respectively (see Fig. 44(b)). (Don't be fooled by the name—$y$-node comparisons depend on both the $x$ and $y$ values of the query point.) Note that the search will reach a $y$-node only if we have already verified that the $x$-coordinate of the query point lies within the vertical slab that contains this segment.

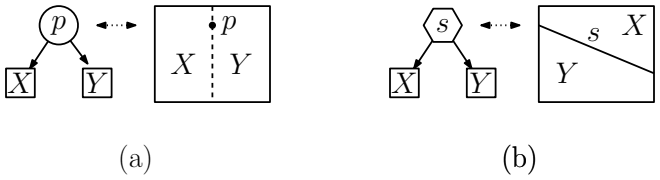



Fig. 43: (a) $x$-node and (b) $y$-node.

Our construction of the point location data structure mirrors the incremental construction of the trapezoidal map. In particular, if we freeze the construction just after the insertion of any segment, the current structure will be a point location structure for the current trapezoidal map.

In Fig. 44 below we show a simple example of what the data structure looks like for two line segments. For example, if the query point is in trapezoid $D$, we would first detect that it is to the right of enpoint $p_1$ (right child), then left of $q_1$ (left child), then below $s_1$ (right child), then right of $p_2$ (right child), then above $s_2$ (left child).

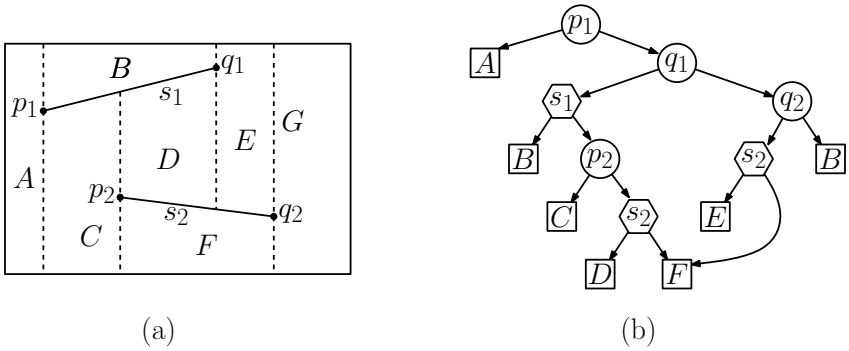



Fig. 44: Trapezoidal map point location data structure.

**Incremental Construction:** The question is how do we build this data structure incrementally? First observe that when a new line segment is added, we only need to adjust the portion of the tree that involves the trapezoids that have been deleted as a result of this new addition. Each trapezoid that is deleted will be replaced with a search structure that determines the newly created trapezoid that contains it.

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

**Single (left or right) endpoint:** A single trapezoid $A$ is replaced by three trapezoids, denoted $X$, $Y$, and $Z$. Letting $p$ denote the endpoint, we create an $x$-node for $p$, and one child is a leaf node for the trapezoid $X$ that lies outside vertical projection of the segment. For the other child, we create a $y$-node whose children are the trapezoids $Y$ and $Z$ lying above and below the segment, respectively (see Fig. 45(a)).

**Two segment endpoints:** This happens when the segment lies entirely inside the trapezoid. In this case one trapezoid $A$ is replaced by four trapezoids, $U$, $X$, $Y$, and $Z$. Letting $p$ and $q$ denote the left and right

endpoints of the segment, we create two $x$-nodes, one for $p$ and the other for $q$. We create a $y$-node for the line segment, and join everything together (see Fig. 45(b)).

**No segment endpoints:** This happens when the segment cuts completely through a trapezoid. A single trapezoid is replaced by two trapezoids, one above and one below the segment, denoted $Y$ and $Z$. We replace the leaf node for the original trapezoid with a $y$-node whose children are leaf nodes associated with $Y$ and $Z$ (see Fig. 45(c)).

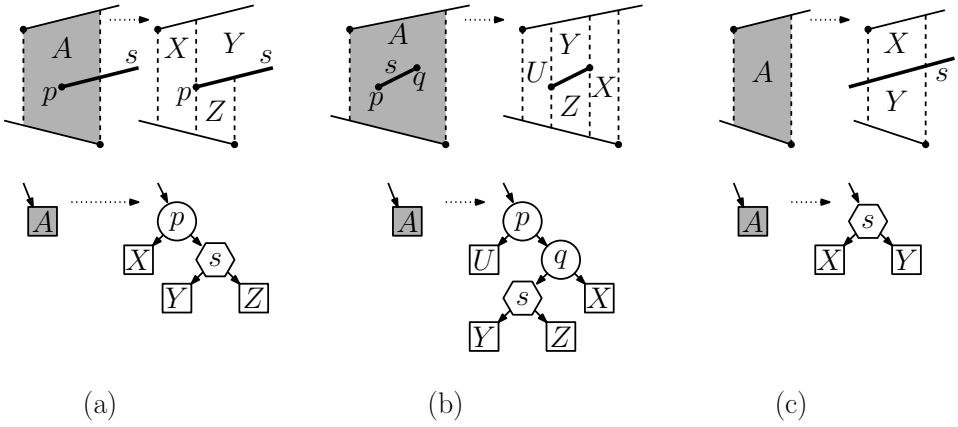



Fig. 45: Line segment insertion and updates to the point location structure. The single-endpoint case (left) and the two-endpoint case (right). The no-endpoint case is not shown.

It is important to notice that (through sharing) each trapezoid appears exactly once as a leaf in the resulting structure. An example showing the complete transformation to the data structure after adding a single segment is shown in Fig. 46 below.

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

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

Analyzing the query time is a little subtler. In a normal probabilistic analysis of data structures we think of the data structure as being fixed, and then compute expectations over random queries. Here the approach will be to imagine that we have exactly one query point to handle. The query point can be chosen arbitrarily (imagine an adversary that tries to select the worst-possible query point) but this choice is made without knowledge of the random choices the algorithm makes. We will show that, given a fixed query point $q$, the expected search path length for $q$ is $O(\log n)$, where the expectation is over all segment insertion orders. (Note that this does not imply that the expected maximum depth of the tree is $O(\log n)$. We will discuss this issue later.)

Let $q$ denote the query point. Rather than consider the search path for $q$ in the final search structure, we will consider how $q$ moves incrementally through the structure with the addition of each new line segment. Let $\Delta_i$ 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 $i$th 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 Fig. 46 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 $i$th segment caused $q$'s trapezoid to be replaced by a different one. As a result, $q$ must now perform some additional comparisons to locate itself with respect to the newly

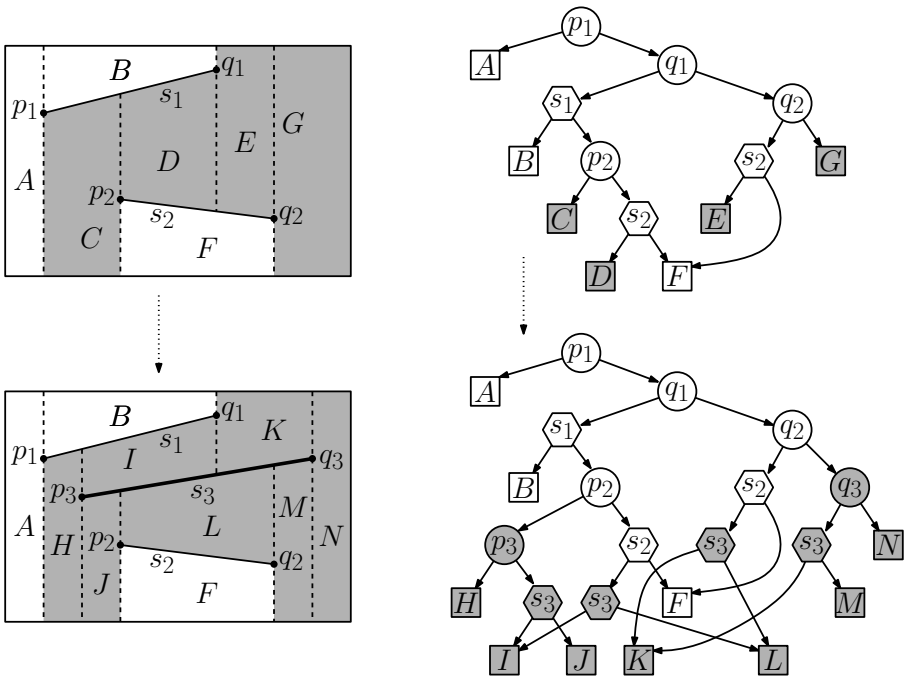



Fig. 46: Line segment insertion.

created trapezoids that overlap $\Delta_{i-1}$. 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 descend at most three levels in the search tree after the insertion. The worst case occurs in the two-endpoint case, where the query point falls into one of the trapezoids lying above or below the segment (see Fig. 45(b)).

Since a point can descend at most 3 levels with each change of its containing trapezoid, the expected length of the search path to $q$ is at most 3 times the number of times that $q$ changes its trapezoid as a result of each insertion. For $1 \leq i \leq n$, let $X_i(q)$ denote the random event that $q$ changes its trapezoid after the $i$th insertion, and let Prob$(X_i(q))$ denote the probability of this event. Letting $D(q)$ denote the average depth of $q$ in the final search tree, we have

$$D(q) \;\leq\; 3\sum_{i=1}^{n} \text{Prob}(X_i(q)).$$

What saves us is the observation that, as $i$ becomes larger, the more trapezoids we have, and the smaller the probability that any random segment will affect a given trapezoid. In particular, we will show that Prob$(X_i(q)) \leq 4/i$. We do this through a backwards analysis. In particular, consider the trapezoid $\Delta_i$ that contained $q$ *after* the $i$th insertion. Recall from the previous lecture that each trapezoid is dependent on at most four segments, which define the top and bottom edges, and the left and right sides of the trapezoid. Clearly, $\Delta_i$ would have changed as a result of insertion $i$ if any of these four segments had been inserted last. Since, by the random insertion order, each segment is equally likely to be the last segment to have been added, the probability that one of $\Delta_i$'s dependent segments was the last to be inserted is at most $4/i$. Therefore, Prob$(X_i(q)) \leq 4/i$.

From this, it follows that the expected path length for the query point $q$ is at most

$$D(q) \;\leq\; 3\sum_{i=1}^{n} \frac{4}{i} \;=\; 12\sum_{i=1}^{n} \frac{1}{i}.$$

Recall that $\sum_{i=1}^{n} \frac{1}{i}$ is the Harmonic series, and for large $n$, its value is very nearly $\ln n$. Thus we have

$$D(q) \;\leq\; 12 \ln n \;=\; O(\log n).$$

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

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

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

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

Nonetheless, this itself is enough to lead to variant of the algorithm for which $O(\log n)$ time is guaranteed. Rather than just running the algorithm once and taking what it gives, instead run it repeatedly and keep track of the structure's depth as you go. As soon as the depth exceeds $c \log n$ for some suitably chosen $c$, then stop and start over again with a new random sequence. For a suitable $c$, the above lemma implies that such a failure will occur with at most some very small constant probability. Therefore, after a constant number of trials, we will succeed in constructing a data structure of the desired depth bound. 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 $O(\log n)$.

**Line Segment Intersection Revisited:** Earlier this semester we presented a plane-sweep algorithm for computing line segment intersection. The algorithm had a running time of $O((n + I) \log n)$, where $I$ is the number of intersection points. It is interesting to note that the randomized approach we discussed today can be adapted to deal with intersecting segments as well. In particular, whenever a segment is added, observe that in addition to it stabbing vertical segments, it may generally cross over one of the existing segments. When this occurs, the algorithm must determine the trapezoid that is hit on the other side of the segment, and then continue the process of walking the segment. Note that the total size of the final decomposition is $O(n+I)$, which would suggest that the running time might be the same as the plane-sweep algorithm. It is remarkable, therefore, that the running time is actually better. Intuitively, the reason is that the $O(\log n)$ factor in the randomized algorithm comes from the point location queries, which are applied only to the left endpoint of each of the $n$ segments. With a bit of additional work, it can be shown that the adaptation of the randomized algorithm to general (intersecting) segments runs in $O(I + n \log n)$ time, thus removing the log factor from the $I$ term.

# Lecture 11: Voronoi Diagrams and Fortune's Algorithm

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

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

where $\|pq\| = \left(\sum_{i=j}^{d} (p_j - q_j)^2\right)^{1/2}$ denotes the Euclidean distance between points $p$ and $q$. The Voronoi diagram can be defined over any metric and in any dimension, but we will concentrate on the planar, Euclidean case here.

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,

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

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 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 (see Fig. 47).

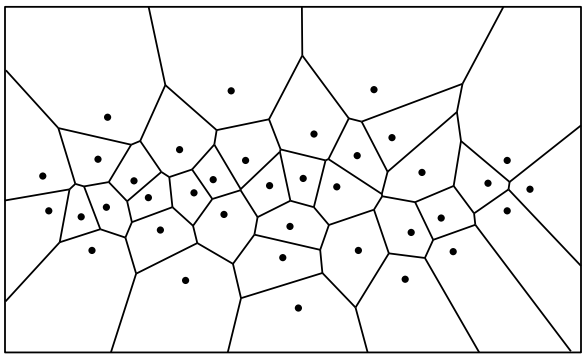


Fig. 47: Voronoi diagram

Voronoi diagrams have a huge number of important applications in science and engineering. These include answering nearest neighbor queries, computational morphology and shape analysis, clustering and data mining, facility location, multi-dimensional interpolation.

**Properties of the Voronoi diagram:** Here are some properties of the Voronoi diagrams in the plane.

**Voronoi complex:** Clearly the diagram is a cell complex whose faces are (possibly unbounded) convex polygons. 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 (see Fig. 48(a)).

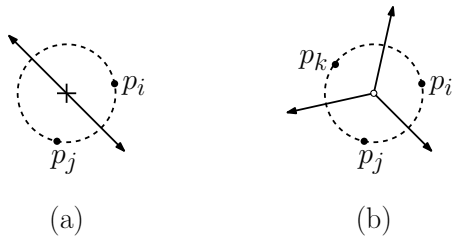



Fig. 48: Properties of the Voronoi diagram.

**Voronoi vertices:** It follows that the vertex at which three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ intersect, called a *Voronoi vertex* is equidistant from all sites (see Fig. 48(b)). Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior.

**Degree:** Generally three points in the plane define a unique circle. If we make the general position assumption that no four sites are cocircular, then the vertices of the Voronoi diagram all have degree three.

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

**Size:** If $n$ denotes the number of sites, then the Voronoi diagram is a planar graph (if we imagine all the unbounded edges as going to a common vertex infinity) with exactly $n$ faces. It follows from Euler's formula[11] that the number of Voronoi vertices is roughly $2n$ and the number of edges is roughly $3n$. (See the text for details. In higher dimensions the diagram's combinatorial complexity ranges from $O(n)$ up to $O(n^{\lceil d/2 \rceil})$.)

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

Historically, $O(n^2)$ algorithms for computing Voronoi diagrams were known for many years (based on incremental constructions). When computational geometry came along, a more complex, but asymptotically superior $O(n \log n)$ algorithm was discovered. This algorithm was based on divide-and-conquer. But it was rather complex, and somewhat difficult to understand. Later, Steven Fortune discovered 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 a variant of this algorithm later when we talk about the dual structure, called the Delaunay triangulation.

**Fortune's Algorithm:** Before discussing Fortune's algorithm, it is interesting to consider why this algorithm was not invented much earlier. In fact, it is quite a bit trickier than any plane sweep algorithm we have seen so far. The key to any plane sweep algorithm is the ability to discover all upcoming events in an efficient manner. For example, in the line segment intersection algorithm we considered all pairs of line segments that were adjacent in the sweep-line status, and inserted their intersection point in the queue of upcoming events. The problem with the Voronoi diagram is that of predicting when and where the upcoming events will occur.

To see the problem, suppose that you are designing a plane sweep algorithm. Behind the sweep line you have constructed the Voronoi diagram based on the points that have been encountered so far in the sweep. The difficulty is that a site that lies *ahead* of the sweep line may generate a Voronoi vertex that lies *behind* the sweep line. How could the sweep algorithm know of the existence of this vertex until it sees the site. But by the time it sees the site, it is too late. It is these *unanticipated events* that make the design of a plane sweep algorithm challenging (see Fig. 49).

**The Beach Line:** The sweeping process will involve sweeping two different object. First, there will be a horizontal sweep line, moving from top to bottom. We will also maintain an $x$-monotonic curve called a *beach line*. (It is so named because it looks like waves rolling up on a beach.) The beach line lags behind the sweep line in such a way that it is unaffected by sites that have yet to be seen. Thus, there are no unanticipated events on the beach line. The sweep-line status will be based on the manner in which the Voronoi edges intersect the beach line, not the actual sweep line.

Let's make these ideas more concrete. We subdivide the halfplane lying above the sweep line into two regions: those points that are closer to some site $p$ above the sweep line than they are to the sweep line itself, and those points that are closer to the sweep line than any site above the sweep line.

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

[11] Euler's formula for planar graphs states that a planar graph with $v$ vertices, $e$ edges, and $f$ faces satisfies $v - e + f = 2$. There are $n$ faces, and since each vertex is of degree three, we have $3v = 2e$, from which we infer that $v - (3/2)v + n = 2$, implying that $v = 2n - 4$. A similar argument can be used to bound the number of edges.

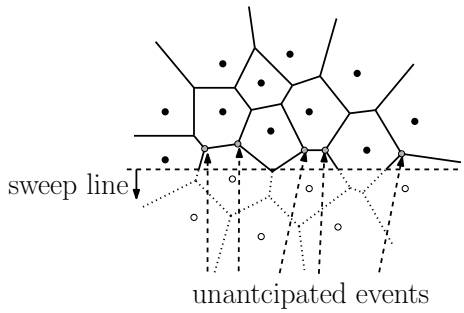



Fig. 49: Plane sweep for Voronoi diagrams. Note that the position of the indicated vertices depends on sites that have not yet been encountered by the sweep line, and hence are unknown to the algorithm. (Note that the sweep line moves from top to bottom.)

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

What does the beach line look like? Recall from high school geometry that the set of points that are equidistant from a point (in this case a site) and a line (in this case the sweep line) is a parabola (see Fig. 50(a)). Clearly the parabola's shape changes continuously as the sweep line moves continuously. With a little analytic geometry, it is easy to show that the parabola becomes "skinnier" when the site is closer to the line and becomes "fatter" as the sweep line moves farther away. In the degenerate case when the line contains the site the parabola degenerates into a vertical ray shooting up from the site. (You should work through the distance equations to see why this is so.)

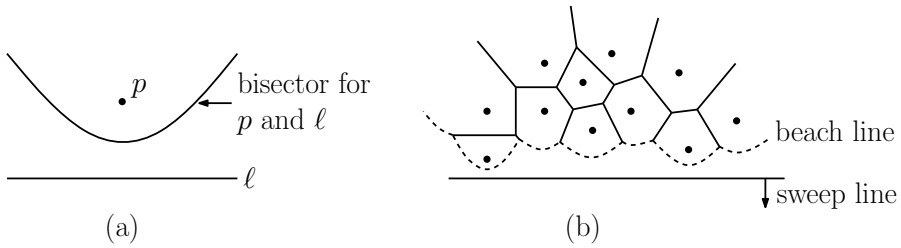



Fig. 50: The beach line. Notice that only the portion of the Voronoi diagram that lies above the beach line is computed. The sweep-line status maintains the intersection of the Voronoi diagram with the beach line.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site (see Fig. 50(b)). Note that the parabola of some sites above the beach line will not touch the lower envelope and hence will not contribute to the beach line. Because the parabolas are $x$-monotone, so is the beach line. Also observe that the point where two arcs of the beach line intersect, which we call a *breakpoint*, is equidistant from two sites and the sweep line, and hence must lie on some Voronoi edge. In particular, if the beach line arcs corresponding to sites $p_i$ and $p_j$ share a common breakpoint on the beach line, then this breakpoint lies on the Voronoi edge between $p_i$ and $p_j$. From this we have the following important characterization.

**Lemma:** The beach line is an $x$-monotone curve made up of parabolic arcs. The breakpoints (that is, vertices) of the beach line lie on Voronoi edges of the final diagram.

Fortune's algorithm consists of simulating the growth of the beach line as the sweep line moves downward, and in particular tracing the paths of the breakpoints as they travel along the edges of the Voronoi diagram. Of course, as the sweep line moves, the parabolas forming the beach line change their shapes continuously. As with

all plane-sweep algorithms, we will maintain a sweep-line status and we are interested in simulating the discrete event points where there is a "significant event", that is, any event that changes the topological structure of the Voronoi diagram or the beach line.

**Sweep-Line Status:** The algorithm maintains the current location ($y$-coordinate) of the sweep line. It stores, in left-to-right order the sequence of sites that define the beach line. (We will say more about this later.) **Important:** The algorithm does *not* store the parabolic arcs of the beach line. They are shown solely for conceptual purposes.

**Events:** There are two types of events:

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

**Voronoi vertex events:** (What our text calls *circle events*.) When the length of an arc of the beach line 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 Voronoi vertex events, it will be an easy matter to update the diagram as we go (assuming any reasonable representation of this planar cell complex), and so to link the entire diagram together. Let us consider the two types of events that are encountered.

**Site events:** A site event is generated whenever the horizontal sweep line passes over a site $p_i$. 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 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.) Let $p_j$ denote the site generating this arc. We then split this arc in two by inserting a new entry at this point in the sweep-line status. (Initially this corresponds to a infinitesimally small arc along the beach line, but as the sweep line sweeps on, this arc will grow wider. Thus, the entry for $\langle \ldots, p_j, \ldots \rangle$ on the sweep-line status is replaced by the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$ (see Fig. 51).

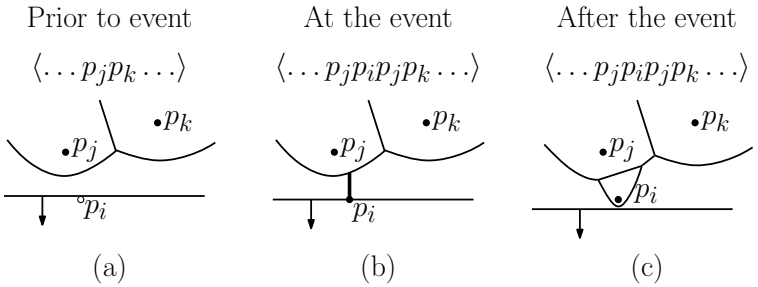



Fig. 51: Site event.

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, it follows that the maximum number of arcs on the beach line can be at most $2n - 1$, since each new point can result in creating one new arc, and splitting an existing arc, for a net increase of two arcs per point (except the first). Note that a point may generally contribute more than one arc to the beach line. (As an exercise you might consider what is the maximum number of arcs a single site can contribute.)

The nice thing about site events is that they are all known in advance. Thus, the sites can be presorted by the $y$-coordinates and inserted as a batch into the event priority queue.

**Voronoi vertex events:** In contrast to site events, Voronoi vertex events are generated dynamically as the algorithm runs. As with the line segment intersection algorithm, the important idea is that each such event is generated

by objects that are *adjacent* on the beach line (and thus, can be found efficiently). However, unlike the segment intersection where pairs of consecutive segments generated events, here triples of points generate the events.

In particular, consider any three consecutive sites $p_i$, $p_j$, and $p_k$ whose arcs appear consecutively on the beach line from left to right (see Fig. 52(a). 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. Thus, the triple of consecutive sites $p_i, p_j, p_k$ on the sweep-line status is replaced with $p_i, p_k$ (see Fig. 52).

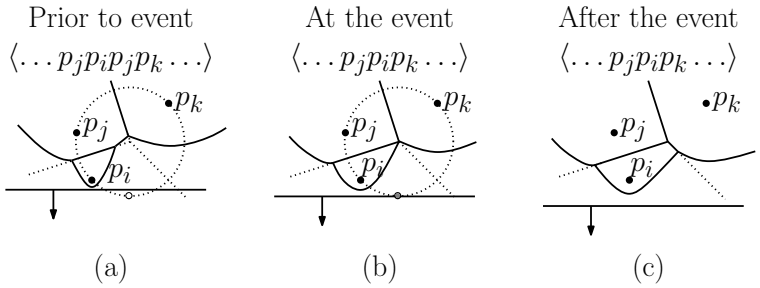



Fig. 52: Voronoi vertex event.

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

**(Partial) Voronoi diagram:** The partial Voronoi diagram that has been constructed so far will be stored in any reasonable data structure for storing planar subdivisions, for example, a doubly-connected edge list. There is one technical difficulty caused by the fact that the diagram contains unbounded edges. This can be handled by enclosing everything within a sufficiently large bounding box. (It should be large enough to contain all the Voronoi vertices, but this is not that easy to compute in advance.) An alternative is to create an imaginary Voronoi vertex "at infinity" and connect all the unbounded edges to this imaginary vertex.

**Beach line:** The beach line consists of the sorted sequence of sites whose arcs form the beach line. It is represented using a dictionary (e.g. a balanced binary tree or skip list). As mentioned above, we *do not* explicitly store the parabolic arcs. They are just there for the purposes of deriving the algorithm. Instead for each parabolic arc on the current beach line, we store the site that gives rise to this arc.

The key search operation is that of locating the arc of the beach line that lies directly above a newly discovered site. (As an exercise, before reading the next paragraph you might think about how you would design a binary search to locate this arc, given that you only have the sites, not the actual arcs.)

Between each consecutive pair of sites $p_i$ and $p_j$, there is a breakpoint. Although the breakpoint moves as a function of the sweep line, observe that it is possible to compute the exact location of the breakpoint as a function of $p_i$, $p_j$, and the current $y$-coordinate of the sweep line. In particular, the breakpoint is the center of a circle that passes through $p_i$, $p_j$ and is tangent to the sweep line. (Thus, as with beach lines, *we do not explicitly store breakpoints*. Rather, we compute them only when we need them.) Once the breakpoint is computed, we can then determine whether a newly added site is to its left or right. Using the sorted ordering of the sites, we use this primitive comparison to drive a binary search for the arc lying above the new site.

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

**Search:** Given the current $y$-coordinate of the sweep line and a new site $p_i$, determine the arc of the beach line lies immediately above $p_i$. Let $p_j$ denote the site that contributes this arc. Return a reference to this beach line entry.

**Insert and split:** Insert a new entry for $p_i$ within a given arc $p_j$ of the beach line (thus effectively replacing the single arc $\langle \ldots, p_j, \ldots \rangle$ with the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$. Return a reference to the newly added beach line entry (for future use).

**Delete:** Given a reference to an entry $p_j$ on the beach line, delete this entry. This replaces a triple $\langle \ldots, p_i, p_j, p_k, \ldots \rangle$ with the pair $\langle \ldots, p_i, p_k, \ldots \rangle$.

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. All operations can be implemented in $O(\log n)$ time assuming that the priority queue is stored as an ordered dictionary.

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 Voronoi 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. The algorithm starts by inserting the topmost vertex into the sweep-line status. 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 in somewhat greater detail.

**Site event:** Let $p_i$ be the new site (see Fig. 51 above).

(1) Advance the sweep line so that it passes through $p_i$. Apply the above search operation to determine the beach line arc that lies immediately above $p_i$. Let $p_j$ be the corresponding site.

(2) Applying the above insert-and-split operation, inserting a new entry for $p_i$, thus replacing $\langle \ldots, p_j, \ldots \rangle$ with $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$.

(3) Create a new (dangling) edge in the Voronoi diagram, which lies on the bisector between $p_i$ and $p_j$.

(4) Some old triples that involved $p_j$ may need to be deleted and some new triples involving $p_i$ will be inserted, based on the change of neighbors on the beach line. (The straightforward details are omitted.) Note that the newly created beach-line triple $p_j, p_i, p_j$ does not generate an event because it only involves two distinct sites.

**Voronoi vertex event:** Let $p_i$, $p_j$, and $p_k$ be the three sites that generated this event, from left to right (see Fig. 52 above).

(1) Delete the entry for $p_j$ from the beach line status. (Thus eliminating its associated arc.)

(2) Create a new vertex in the Voronoi diagram (at the circumcenter of $\{p_i, p_j, p_k\}$) and join the two Voronoi edges for the bisectors $(p_i, p_j)$, $(p_j, p_k)$ to this vertex.

(3) Create a new (dangling) edge for the bisector between $p_i$ and $p_k$.

(4) Delete any events that arose from triples involving the arc of $p_j$, and generate new events corresponding to consecutive triples involving $p_i$ and $p_k$. (There are two of them. The straightforward details are omitted.)

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

# Lecture 12: Delaunay Triangulations: General Properties

**Delaunay Triangulations:** Last time we discussed the topic of Voronoi diagrams. Today we consider the related structure, called the *Delaunay triangulation* (DT). The Voronoi diagram of a set of sites in the plane is a planar subdivision, that is, a cell complex. The *dual* of such subdivision is another subdivision that is defined as follows. For each face of the Voronoi diagram, we create a vertex (corresponding to the site). For each edge of the Voronoi diagram lying between two sites $p_i$ and $p_j$, we create an edge in the dual connecting these two vertices. Finally, each vertex of the Voronoi diagram corresponds to a face of the dual.

The resulting dual graph is a planar subdivision. Assuming general position, the vertices of the Voronoi diagram have degree three, it follows that the faces of the resulting dual graph (excluding the exterior face) are triangles. Thus, the resulting dual graph is a triangulation of the sites, called the *Delaunay triangulation* (see Fig. 53.)

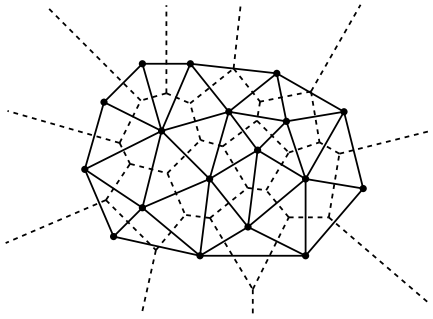


Fig. 53: The Delaunay triangulation of a set of points (solid lines) and the Voronoi diagram (broken lines).

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

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

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

**Proof:** This is because the center of this circle is the corresponding dual Voronoi vertex, and by definition of the Voronoi diagram, the three sites defining this vertex are its nearest neighbors.

**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 $p_j$.

**Proof:** If two sites $p_i$ and $p_j$ are neighbors in the Delaunay triangulation, then their cells are neighbors in the Voronoi diagram, and so for any point on the Voronoi edge between these sites, a circle centered at this point passing through $p_i$ and $p_j$ cannot contain any other point (since they must be closest). Conversely, if there is an empty circle passing through $p_i$ and $p_j$, then the center $c$ of this circle is a point on the edge of the Voronoi diagram between $p_i$ and $p_j$, because $c$ is equidistant from each of these sites and there is no closer site. Thus the Voronoi cells of two sites are adjacent in the Voronoi diagram, implying that there edge is in the Delaunay triangulation.

**Closest pair property:** The closest pair of sites in $P$ are neighbors in the Delaunay triangulation.

**Proof:** Suppose that $p_i$ and $p_j$ are the closest sites. The circle having $p_i$ and $p_j$ as its diameter cannot contain any other site, since otherwise such a site would be closer to one of these two points, violating the hypothesis that these points are the closest pair. Therefore, the center of this circle is on the Voronoi edge between these points, and so it is an empty circle.

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

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

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

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

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

**Proof:** Let $T$ be the MST for $P$, let $w(T)$ denote the total weight of $T$. Let $a$ and $b$ be any two sites such that $ab$ is an edge of $T$. Suppose to the contrary that $ab$ is not an edge in the Delaunay triangulation. This implies that there is no empty circle passing through $a$ and $b$, and in particular, the circle whose diameter is the segment $\overline{ab}$ contains a site, call it $c$ (see Fig. 54.)

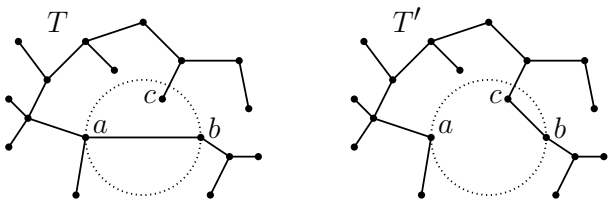



Fig. 54: The Delaunay triangulation and MST.

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

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

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

By the way, this suggests another interesting question. Among all triangulations, we might ask, does the Delaunay triangulation minimize the total edge length? The answer is no (and there is a simple four-point counterexample). However, this claim was made in a famous paper on Delaunay triangulations, and you may still hear it quoted from time to time. The triangulation that minimizes total edge weight is called the *minimum weight triangulation*. Recently it was proved that this problem is NP-hard. (This problem has been open for many years, dating back to the original development of the theory of NP-completeness back in the 1970's.)

**Spanner Properties:** A natural observation about Delaunay triangulations is that its edges would seem to form a resonable transporation road network between the points. On inspecting a few examples, it is natural to conjecture that the length of the shortest path between two points in a planar Delaunay triangulation is not significantly longer than the straight-line distance between these points.

This is closely related to the theory of geometric spanners, that is, geometric graphs whose shortest paths are not too long. Consider any point set $P$ and a straight-line graph $G$ whose vertices are the points of $P$. For any two points $p, q \in P$, let $\delta_G(p, q)$ denote the length of the shortest path from $p$ to $q$ in $G$, where the weight of each edge is its Euclidean length. Given any parameter $t \geq 1$, we say that $G$ is a *$t$-spanner* if for any two points $p, q \in P$, the shortest path length between $p$ and $q$ in $G$ is at most a factor $t$ longer than the Euclidean distance between these points, that is

$$\delta_G(p, q) \leq t\|pq\|$$

Observe that when $t = 1$, the graph $G$ must be the complete graph, consisting of $\binom{n}{2} = O(n^2)$ edges. Of interest is whether there exist spanners having $O(n)$ edges.

It can be proved that the edges of the Delaunay triangulation form a spanner (see Fig. 55). We will not prove the following result, which is due to Keil and Gutwin.

**Theorem:** Given a set of points $P$ in the plane, the Delaunay triangulation of $P$ is a $t$-spanner for $t = 4\pi\sqrt{3}/9 \approx 2.4$.

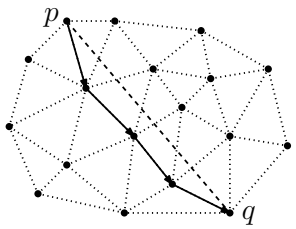



Fig. 55: Spanner property of the Delaunay Triangulation.

In fact, it is conjectured that the Delaunay triangulation is a $(\pi/2)$-spanner, but this has never been proved (and it seems to be a hard problem).

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

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

**Theorem:** Among all triangulations of a given planar point set, the Delaunay triangulation has the lexicographically largest angle sequence, and in particular, it maximizes the minimum angle.

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. Thus, in Fig. 56(a) below, we have $\theta_1 > \theta_2 > \theta_3$.

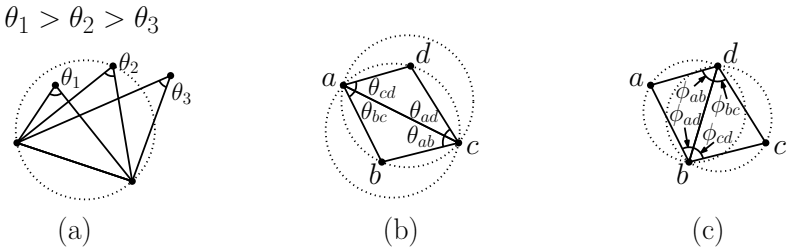



Fig. 56: Angles and edge flips.

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 $\triangle abc$. (Note that this implies that $b$ lies inside the circumcircle of $\triangle cda$.) If we flip the edge it will follow that the two circumcircles of the two resulting triangles, $\triangle abd$ and $\triangle bcd$ are now empty (relative to these four points), and the observation above about circles and angles proves that the minimum angle increases at the same time. In particular, in Fig. 56(b) and (c), we have

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

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

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

Note that the process of edge-flipping can be generalized to simplicial complexes in higher dimensions. However, the process does not generally replace a fixed number of triangles with the same number, as it does in the plane (replacing two old triangles with two new triangles). For example, in 3-space, the most basic flip can replace two adjacent tetrahedra with three tetrahedra, and vice versa. Although it is known that in the plane any triangulation can be converted into any other through a judicious sequence of edge flips, this is not known in higher dimensions.

# Lecture 13: Delaunay Triangulations: Incremental Construction

**Constructing the Delaunay Triangulation:** We will present a simple randomized $O(n \log n)$ expected time algorithm for constructing Delaunay triangulations for $n$ sites in the plane. The algorithm is remarkably similar in

spirit to the randomized algorithm for trapezoidal map algorithm in that not only builds the triangulation but also provides a point-location data structure as well. We will not discuss the point-location data structure in detail, but the details are easy to fill in.

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

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

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

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

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

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

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

and similarly for the other three points, $b$, $c$, and $d$. If we let $X_1$, $X_2$, $X_3$ and $X_4$ denote the columns of the above matrix (e.g., $X_1 = (a_x, b_x, c_x, d_x)^{\mathsf{T}}$) we have

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

Since there is a linear combination of these vectors that sums to 0, it follows that these vector are linearly dependent. From standard linear algebra, we know that the columns of a matrix are linearly dependent if and only if the determinant of the matrix is zero. We will leave the completion of the proof (involving inside and outside) as an exercise.

**Incremental update:** When we add the next site, $p_i$, the problem is to convert the current Delaunay triangulation into a new Delaunay triangulation containing this site. This will be done by creating a non-Delaunay triangulation containing the new site, and then incrementally "fixing" this triangulation to restore the Delaunay properties. The basic changes are:

- Joining a site in the interior of some triangle to the triangle's vertices (see Fig. 57(a)).

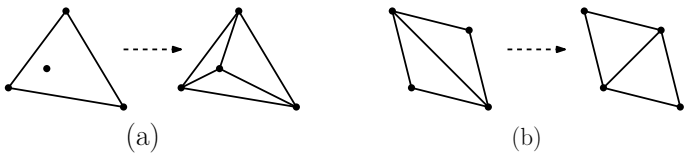



Fig. 57: Basic triangulation changes.

- Performing an *edge flip* (see Fig. 57(b)).

Both of these operations can be performed in $O(1)$ time, assuming that the triangulation is maintained in any reasonable way, say, as a double-connected edge list.

The algorithm that we will describe has been known for many years, but was first analyzed by Guibas, Knuth, and Sharir. The algorithm starts within an initial triangulation such that all the points lie in the convex hull. This can be done by enclosing the points in a suitably large triangle.[12] Our book suggests a symbolic alternative, which is more reliable. Generate a triangle that contains all the points, but then *modify* the incircle test so that the vertices of this enclosing triangle *behave* as if they are infinitely far away.

The sites are added in random order. When a new site $p$ is added, we find the triangle $\triangle abc$ of the current triangulation that contains this site (we will see how later), insert the site in this triangle, and join this site to the three surrounding vertices. This creates three new triangles, $\triangle pab$, $\triangle pbc$, and $\triangle pca$, each of which may or may not satisfy the empty-circle condition. How do we test this? For each of the triangles that have been added, we check the vertex of the triangle that lies on the opposite side of the edge that does not include $p$. (If there is no such vertex, because this edge is on the convex hull, then we are done.) If this vertex fails the incircle test (that is, if it is inside the circumcircle), then we swap the edge (creating two new triangles that are adjacent to $p$). This replaces one triangle that was incident to $p$ with two new triangles. We repeat the same test with these triangles. An example is shown in Fig. 58.

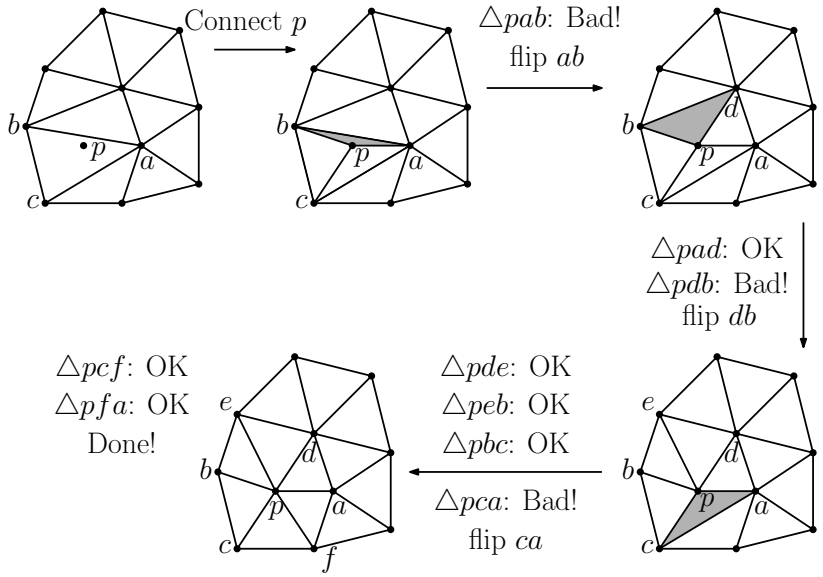



Fig. 58: Point insertion.

The algorithm for the incremental algorithm is shown below, and an example is presented in Fig. 58. The current

[12] Some care must be taken in the construction of this enclosing triangle. It is not sufficient that it simply contains all the points. It should be so large that the vertices of the triangle do not lie in the circumcircles of any of the triangles of the final triangulation.

triangulation is kept in a global data structure. The edges in the following algorithm should be thought of as pointers to an reasonable representation of the simplicial complex.

Randomized Incremental Delaunay Triangulation Algorithm

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

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

As you can see, the algorithm is very simple. There are only two elements that have not been shown are the implementation. The first is the update operations on the data structure for the simplicial complex. These can be done in $O(1)$ time each on any reasonable representation. The other issue is locating the triangle that contains $p$. We will discuss this below.

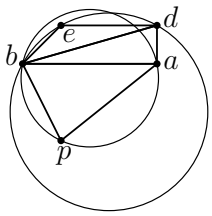
**Correctness:** There is one major issue in establishing the correctness of the algorithm. When we performed empty-circle tests, we only tested the empty circle tests for the newly created triangles containing the site $p$, and then only for sites that lay on the opposite side of an edge of each such triangle.

This is related to an important issue in Delaunay triangulations. We say that a triangulation is *locally Delaunay* if for each triangle the vertices lying on the opposite side of each edge of the (up to) three neighboring triangles satisfy the empty-circle condition. But to establish complete correctness of the triangulation, we need to show that the triangulation is *globally Delaunay*, meaning that empty-circle condition is satisfied for all triangles, and all points of $P$. That is, it suffices to show that if a triangulation is locally Delaunay, then it is globally Delaunay. This important theorem (called *Delaunay's Theorem*), and we will sketch a proof of this theorem below for this special context.

First, to see that it suffices to consider only triangles that are incident to $p$, observe that $p$ is the only newly added site, and hence it is the only site that can cause a violation of the empty-circle condition.

To finish the argument, it suffices to see why "locally Delaunay" implies "globally Delaunay." Consider a triangle $\triangle pab$ that contains $p$ and consider the vertex $d$ belonging to the triangle that lies on the opposite side of edge $ab$. We argue that if $d$ lies outside the circumcircle of $pab$, then no other point of the point set can lie within this circumcircle. A complete proof of this takes some effort, but here is a simple justification. What could go wrong? It might be that $d$ lies outside the circumcircle, but there is some other site, say, a vertex $e$ of a triangle adjacent to $d$, that lies inside the circumcircle (see Fig. 59). 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 the geometry of circles.) However, this would violate the assumption that the initial triangulation (before the insertion of $p$) was a Delaunay triangulation.

**Point Location:** The point location can be accomplished by one of two means. Our text discusses the idea of building a history graph point-location data structure, just as we did in the trapezoid map case. A simpler approach is based on the idea of maintaining the uninserted sites in a set of *buckets*. Think of each triangle of the current




if $e$ violates the circumcircle condition for $\triangle pab$
then $a$ violates the condition with respect to $\triangle bde$.

Fig. 59: Proof of sufficiency of testing neighboring sites.

triangulation as a *bucket* that holds the sites that lie within this triangle and have yet to be inserted. Whenever an edge is flipped, or when a triangle is split into three triangles through point insertion, some old triangles are destroyed and are replaced by a constant number of new triangles. When this happens, we lump together all the sites in the buckets corresponding to the deleted triangles, create new buckets for the newly created triangles, and reassign each site into its new bucket. Since there are a constant number of triangles created, this process requires $O(1)$ time per site that is rebucketed.

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

**Structural Changes:** We argue first that the expected number of edge changes with each insertion is $O(1)$ by a simple application of backwards analysis. First observe that (assuming general position) the structure of the Delaunay triangulation is independent of the insertion order of the sites so far. Thus, any of the existing sites is equally likely to have been the last site to be added to the structure.

Suppose that some site $p$ was the last to have been added. How much work was needed to insert $p$? Observe that the initial insertion of $p$ involved the creation of three new edges, all incident to $p$. Also, whenever an edge swap is performed, a new edge is added to $p$. These are the only changes that the insertion algorithm can make. Therefore the total number of changes made in the triangulation for the insertion of $p$ is proportional to the degree of $p$ *after* the insertion is complete. Thus the work needed to insert $p$ is proportional to $p$'s degree after the insertion.

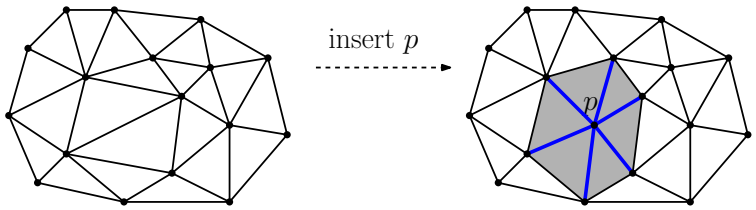



Fig. 60: Number of structural changes is equal to $p$'s degree after insertion (three initial edges and three edge flips).

To perform the backwards analysis, we consider the situation after the insertion of the $i$th site. Since the diagram's structure does not depend on the order of insertion, every one of the $i$ sites appearing in the diagram was equally likely to be the last one added. Thus, by a backwards analysis, the expected time to insert the $i$th site is equal to the average degree of a vertex in the triangulation of $i$ sites. (The only exception are the three initial vertices at infinity, which must be the first sites to be inserted.)

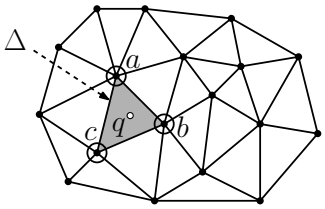
By Euler's formula we know that the average degree of a vertex in any planar graph is at most 6. (Recall that a planar graph with $n$ vertices can have at most $3n$ edges, and the sum of vertex degrees is equal to twice the number of edges, which is at most $6n$.) Thus, irrespective of the stage number, the expected number of edge changes is proportional to the expected vertex degree, which is $O(1)$. Summing over all $n$ insertions, the total

number of structural changes is $O(n)$. Recall that each structural change (new edges and edge flips) can be performed in $O(1)$ time.

**Rebucketing:** Next we argue that the total expected time spent in rebucketing points is $O(n \log n)$. From this it will follow that the overall expected running time is dominated by the rebucketing process, and so is $O(n \log n)$.

To do this, we will show that the expected number of times that any site is rebucketed (as to which triangle it lies in) is $O(\log n)$. Again this is done by a standard application of backwards analysis. Let us fix a site $q \in P$. Consider the situation just after the insertion of the $i$th site. If $q$ has already been inserted, then it is not involved in the rebucketing process, so let us assume that $q$ has not yet been inserted. As above we make use of the fact that any of the existing sites is equally likely to be the last site inserted.

We assert that the probability that $q$ was rebucketed as a result of the last insertion is at most $3/i$. To see this, let $\Delta$ be the triangle containing $q$ after the $i$th insertion. As observed above, after we insert the $i$th site all of the newly created triangles are now incident to this new site. $\Delta$ would have come into existence as a result of the last insertion if and only one of its three vertices was the last to be added (see Fig. 61). Since $\Delta$ is incident to exactly three sites, and every site is equally likely to be the last inserted, it follows that the probability that $\Delta$ came into existence is $3/i$. (We are cheating a bit here by ignoring the three initial sites at infinity.) Thus, the probability that $q$ required rebucketing after the last insertion is at most $3/i$.




$q$ would have been rebucketed only if one of $a$, $b$, or $c$ was the last to be inserted

Fig. 61: Probability of rebucketing.

After stage $i$ there are $n - i$ points that might be subject to rebucketing, and each has probability $3/i$ of being rebucketed. Thus, the expected number of points that require rebucketing as part of the last insertion is $(n-i)3/i$. By the linearity of expectation, to obtain the total number of rebucketings, we sum these up over all stages, yielding

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

(where as usual we have applied the bound on the Harmonic series.) Thus, the total expected time spent in rebucketing is $O(n \log n)$, as desired.

There is one place in the proof that we were sloppy. (Can you spot it?) We showed that the number of points that required rebucketing is $O(n \log n)$, but notice that when a point is inserted, each rebucketed point may change buckets many times (one for the initial insertion and one for each additional edge flip). We will not give a careful analysis of the total number of individual rebucketing operations per point, but it is not hard to show that the expected total number of individual rebucketing operations will not be larger by more than a constant factor. The reason is that (as argued above) each new insertion only results in a constant number of edge flips, and hence the number of individual rebucketings per insertion is also a constant. But a careful proof should consider this. Such a proof is given in our textbook.

# Lecture 14: Line Arrangements and the Zone Theorem

**Line Arrangements:** So far we have studied a few of the most important structures in computational geometry: convex hulls, Voronoi diagrams and Delaunay triangulations. The next most important structure is that of a *line arrangement*.

Consider a finite set $L$ of lines in the plane.[13] These lines naturally subdivide the plane into a cell complex, which is called the *arrangement* of $L$, and is denoted $\mathcal{A}(L)$ (see Fig. 62(a)). The points where two lines intersect form the vertices of the complex, the segments between two consecutive intersection points form its edges, and the polygonal regions between the lines form the faces. Although an arrangement contains unbounded edges and faces, as we did with Voronoi diagrams (from a purely topological perspective) it is possible to add a vertex at infinity and attach all these edges to this vertex to form a proper planar graph. An arrangement can be represented using any standard data structure for cell complexes, a DCEL for example.

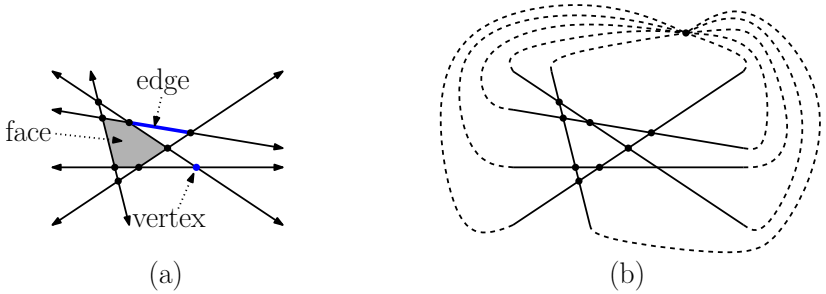



Fig. 62: Arrangement of lines; (a) the basic elements of an arrangement and (b) adding a vertex at infinity to form a proper planar graph.

As we shall see, arrangements have many applications in computational geometry. Through the use of point-line duality, many of these applications involve sets of points. We will begin by discussing the basic geometric and combinatorial properties of arrangements and an algorithm for constructing them. Later we will discuss applications of arrangements to other problems in computational geometry.

**Combinatorial Properties:** The *combinatorial complexity* of an arrangement is the total number of vertices, edges, and faces in the arrangement. An arrangement is said to be *simple* if no three lines intersect at a common point. Through our usual general position assumption that no three lines intersect in a single point, it follows that all our arrangements are simple. The following lemma shows that all of these quantities are $\Theta(n^2)$ for simple planar line arrangements.

**Lemma:** Let $\mathcal{A}(L)$ be a simple arrangement of $n$ lines $L$ in the plan. Then:

(i) the number of vertices (not counting the vertex at infinity) in $\mathcal{A}(L)$ is $\binom{n}{2}$

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

(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 (one line and one 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 follows from Euler's formula, $v - e + f = 2$. To form a cell complex, recall that we added an additional vertex at infinity. Thus, we have $v = 1 + \binom{n}{2}$ and $e = n^2$. Therefore, the number of faces is

$$\begin{aligned} f &= 2 - v + e = 2 - \left(1 + \tbinom{n}{2}\right) + n^2 = 2 - \left(1 + \tfrac{n(n-1)}{2}\right) + n^2 \\ &= 1 + \tfrac{n^2}{2} + \tfrac{n}{2} = 1 + \tfrac{n(n-1)}{2} + n = \tbinom{n}{2} + n + 1, \end{aligned}$$

as desired.

[13] In general, it is possible to define arrangements in $\mathbb{R}^d$ by considering a finite collection of $(d-1)$-dimensional hyperplanes. In such a case the arrangement is a polyhedral cell complex that subdivides $\mathbb{R}^d$.

By the way, this generalizes to higher dimensions as well. The combinatorial complexity of an arrangement of $n$ hyperplanes in $\mathbb{R}^d$ is $\Theta(n^d)$. Thus, these structures are only practical in spaces of relatively low dimension when $n$ is not too large.

**Incremental Construction:** Arrangements are used for solving many problems in computational geometry. But in order to use an arrangement, we first must be able to construct it.[14] We will present a simple incremental algorithm, which builds an arrangement by adding lines one at a time. Unlike the other incremental algorithms we have seen so far, this one is *not randomized*. Its worst-case asymptotic running time, which is $O(n^2)$, holds irrespective of the insertion order. This is asymptotically optimal, since this is the size of the arrangement. The algorithm will also require $O(n^2)$ space, since this is the amount of storage needed to store the final result.

Let $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ denote the set of lines. We will add lines one by one and update the resulting arrangement, and we will show that the $i$-th line can be inserted in $O(i)$ time (irrespective of the insertion order). Summing over $i$, $1 \leq i \leq n$, yields $O(n^2)$ total time.

Suppose that the first $i-1$ lines have already been inserted, and consider the process of adding $\ell_i$. We start by determining the leftmost (unbounded) face of the arrangement that contains this line. Observe that at $x = \infty$, the lines are sorted from top to bottom in increasing order of their slopes. In $O(n)$ time, we can determine where the slope of $\ell_i$ falls in this order, and this determines the leftmost face of the arrangement that contains this line.

The newly inserted line cuts through a sequence of $i-1$ edges and $i$ faces of the existing arrangement. In order to process the insertion, we need to determine which edges are cut by $\ell_i$, and then we split each such edge and update the DCEL for the arrangement accordingly.

In order to determine which edges are cut by $\ell_i$, we "walk" this line through the current arrangement, from one face to the next. Whenever we enter a face, we need to determine through which edge $\ell_i$ exits this face. We answer the question by a very simple strategy. We walk along the edges of the face, say in a counterclockwise direction until we find the exit edge, that is, the other edge that $\ell_i$ intersects. We then jump to the face on the other side of this edge and continue the trace with the neighboring face. This is illustrated in Fig. 63(a). (The DCEL data structure supports such local traversals in time linear in the number of edges traversed.)

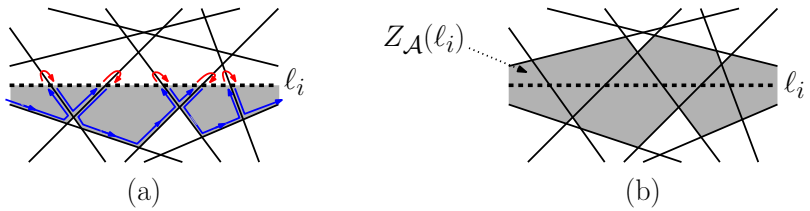



Fig. 63: Adding the line $\ell_i$ to the arrangement; (a) traversing the arrangement and (b) the zone of a line $\ell_i$. (Note that only a portion of the zone is shown in the figure.)

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 $\ell$ that is not in $L$, the *zone* of $\ell$ in $\mathcal{A}(\ell)$, denoted $Z_{\mathcal{A}}(\ell)$, is the set of faces whose closure intersects

[14] This is not quite accurate. For some applications, it suffices to perform a plane-sweep of the arrangement. If we think of each line as an infinitely long line segment, the line segment intersection algorithm that was presented in class leads to an $O(n^2 \log n)$ time and $O(n)$ space solution. There exists a special version of plane sweep for planar line arrangements, called *topological plane sweep*, which runs in $O(n^2)$ time and $O(n)$ space.

$\ell$. (Fig. 63(b) illustrates a zone for the line $\ell$.) For the purposes of the above construction, we are only interested in the edges of the zone that lie below $\ell_i$, but if we bound the total complexity of the zone, then this will be an upper bound on the number of edges traversed in the above algorithm. The combinatorial complexity of a zone (as argued above) is at most $O(n^2)$. The Zone theorem states that the complexity is actually much smaller, only $O(n)$.

**Theorem: (Zone Theorem)** Given an arrangement $\mathcal{A}(L)$ of $n$ lines in the plane, and given any line $\ell$ in the plane, the total number of edges in all the cells of the zone $Z_{\mathcal{A}}(\ell)$ 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 $\ell$. 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. For the sake of illustration, let us assume that $\ell$ is horizontal. By general position, we may assume that none of the lines of $L$ is parallel to $\ell$. 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 bound faces from the left (see Fig. 64(a)). A symmetrical argument applies to the right-bounding edges. (Note that an edge of the zone that crosses $\ell$ itself contributes only twice to the complexity of the zone, once as a left-bounding edge and once as a right-bounding edge. The book's proof counts each such edge four times because it distinguishes not only left and right, but it counts separately the part of the edge that lies above $\ell$ from the part that lies below $\ell$. Thus, they obtain a higher bound of $8n$. Note that we ignore the edges of the bounding box.)

For the base case, when $n = 1$, then there is exactly one left bounding edge in $\ell$'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 $\ell$. Call this $\ell_1$. (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 $\ell$.

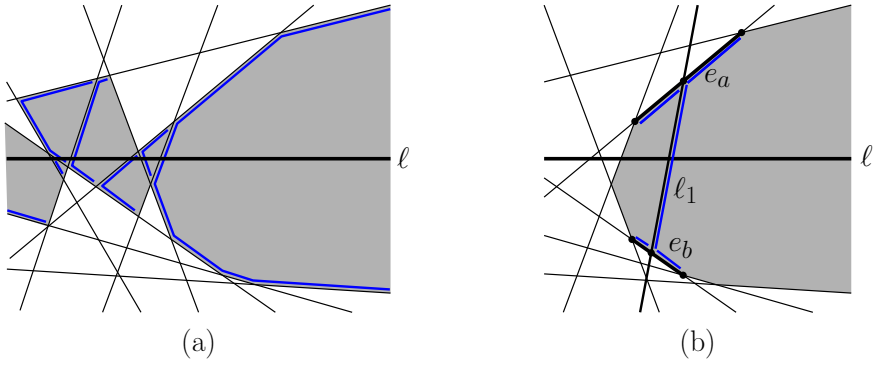



Fig. 64: Proof of the Zone Theorem.

Now let us add back $\ell_1$ and see how many more left-bounding edges result. Consider the rightmost face of the arrangement of $n-1$ lines. (Shaded in Fig. 64(b).) Note that all of its edges are left-bounding edges. Line $\ell_1$ intersects $\ell$ within this face. By convexity, $\ell_1$ intersects the boundary of this face in two edges, denoted $e_a$ and $e_b$, where $e_a$ is above $\ell$ $e_b$ is below. The insertion of $\ell_1$ creates a new left bounding edge along $\ell_1$ 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 $\ell_1$ 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 $\ell_1$'s visibility

from $\ell$. (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.

# Lecture 15: Applications of Arrangements

**Applications of Arrangements and Duality:** Last time we introduced the concept of an arrangement of lines in the plane, and we showed how to construct such an arrangement in $O(n^2)$ time. Line arrangements, when combined with the dual transformation, make it possible to solve a number of geometric computational problems. A number of examples are given below. Unless otherwise stated, all these problems can be solved in $O(n^2)$ time and $O(n^2)$ space by constructing a line arrangement. Alternately, they can be solved in $O(n^2 \log n)$ time and $O(n)$ space by applying plane sweep to the arrangement.

**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 (see Fig. 65(a)).

**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 (see Fig. 65(b)).

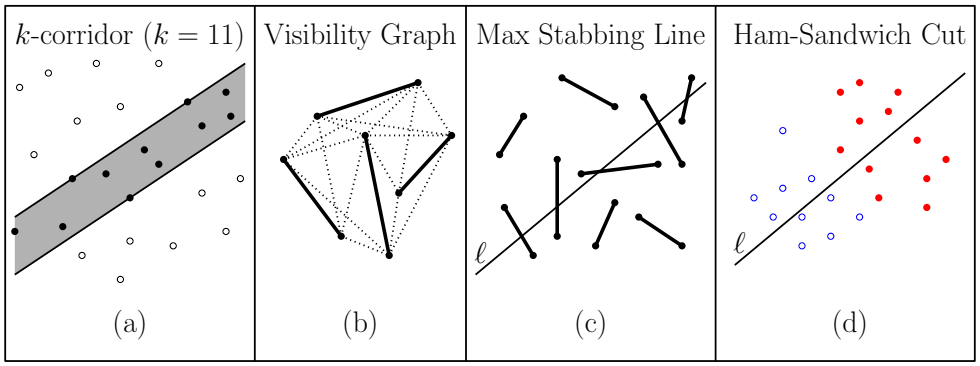



Fig. 65: Applications of arrangements.

**Maximum stabbing line:** Given a set of $n$ line segments in the plane, compute the line $\ell$ that stabs (intersects) the maximum number of these line segments (see Fig. 65(c)).

**Ham Sandwich Cut:** Given $n$ red points and $m$ blue points, find a single line $\ell$ 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 two point sets are separable by a line (that is, the red convex hull and the blue convex hull do not intersect), then this can be solved in time $O(n+m)$ (see Fig. 65(d)).

In the remainder of the lecture, we'll see how problems like these can be solved through the use of arrangements.

**Sweeping Arrangements:** Since an arrangement of $n$ lines is of size $\Theta(n^2)$, we cannot expect to solve problems through the explicit use of arrangements in less than quadratic time. Most applications involve first constructing the arrangement, and then traversing it in some manner. In many instances, the most natural traversal to use is

based on a plane-sweep. (This is not the only way however. Since a planar arrangement is a graph, methods such as depth-first and breadth-first search can be used.)

If an arrangement is to be built just so it can be swept, then maybe you don't need to construct the arrangement at 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 $x = -\infty$ (which means sorting by slope). The sweep line status maintains the lines in, say, bottom to top order according to their intersection with the sweep line. The events are the vertices of the arrangement.

Note that the sweep-line status always contains exactly $n$ entries. Whenever an intersection event occurs, all that happens is that two line exchange positions within the status. Thus, rather than using a general ordered dictionary (e.g., binary search tree) for the sweep-line status, it suffices to store the lines in a simple $n$-element array, sorted from bottom to top, say.

Sweeping an arrangement in this manner takes $O(n^2 \log n)$ time, and $O(n)$ space. Because it is more space-efficient, this is often an attractive alternative to constructing the entire subdivision.

There is a somewhat more "relaxed" version of plane sweep, which works for line arrangements in the plane. (It does not apply to arbitrary line segments.) It is called *topological plane sweep*. You are *not* responsible for knowing how this algorithm works. It runs in $O(n^2)$ time (thus, eliminating a log factor) and uses $O(n)$ space. Although I will not present any justification of this, it is applicable to all the problems we will discuss in today's lecture.

**Sorting all angular sequences:** Here is a natural application of duality and arrangements that turns out to be important for the problem of computing visibility graphs. Consider a set of $n$ points in the plane. For each point $p$ in this set we want to perform an angular sweep, say in counterclockwise order, visiting the other $n-1$ points of the set. For each point, it is possible to compute the angles between this point and the remaining $n-1$ points and then sort these angles. This would take $O(n \log n)$ time per point, and $O(n^2 \log n)$ time overall.

With arrangements we can speed this up to $O(n^2)$ total time, getting rid of the extra $O(\log n)$ factor. Here is how. Recall the point-line dual transformation. The dual of a point $p = (a, b)$ is the line $p^* : y = ax - b$. The dual of a line $\ell : y = ax - b$ is the point $\ell^* = (a, b)$. Recall that $p$ lies above $\ell$ (by distance $h$) if and only if $p^*$ lies below $\ell^*$ (also by distance $h$).

Suppose that $p$ is the point around which we want to sort, and let $\langle p_1, \ldots, p_n \rangle$ be the points in final angular order about $p$ (see Fig. 66(a)). Consider the arrangement defined by the dual lines $p_i^*$. How does this order manifest itself in the arrangement?

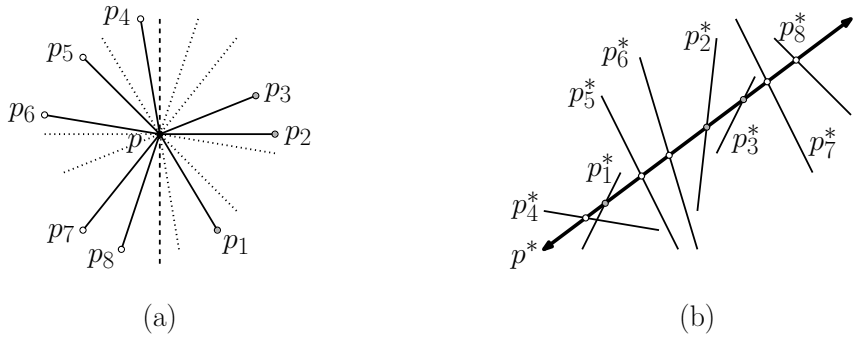



Fig. 66: Arrangements and angular sequences.

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. It follows from the definition of our dual transformation that the $a$-coordinate of each of these vertices in the dual arrangement is the slope of some line of

the form $\overline{pp_i}$ in the primal plane. Thus, the sequence in which the vertices appear on the line is a *slope ordering* of the points about $p_i$, not an *angular ordering*.

However, given this slope ordering, we can simply test which primal points lie to the left of $p$ (that is, have a smaller $x$-coordinate in the primal plane), and separate them from the points that lie to the right of $p$ (having a larger $x$-coordinate). We partition the vertices into two sorted sequences, and then concatenate these two sequences, with the points on the right side first, and the points on the left side later. The resulting is an angular sequence starting with the angle $-90$ degrees and proceeding up to $+270$ degrees.

Thus, once the arrangement has been constructed, we can reconstruct each of the angular orderings in $O(n)$ time, for a total of $O(n^2)$ time. (Since the output size is $\Omega(n^2)$, there no real benefit to be achieved by using plane sweep.)

**Narrowest $k$-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 not difficult 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 make the usual general position assumptions that no three points of $P$ are collinear and no two points have the same $x$-coordinate. This implies that the narrowest corridor contains exactly three points and has strictly positive height.

If we dualize the points of $P$, then in dual space we have a set $L$ of $n$ lines, $\{\ell_1, s \ldots, \ell_n\}$. 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 $\ell_a$ and $\ell_b$ in primal space (see Fig. 67(a)). Their duals $\ell_a^*$ and $\ell_b^*$ 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 height 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 $\ell_b$ and below $\ell_a$. Thus, by the order reversing property, in the dual plane, there are three dual lines that pass both below point $\ell_b^*$ and above $\ell_a^*$. Combining all these observations it follows that the dual formulation of the narrowest 3-corridor problem is the following (see Fig. 67(b)):

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

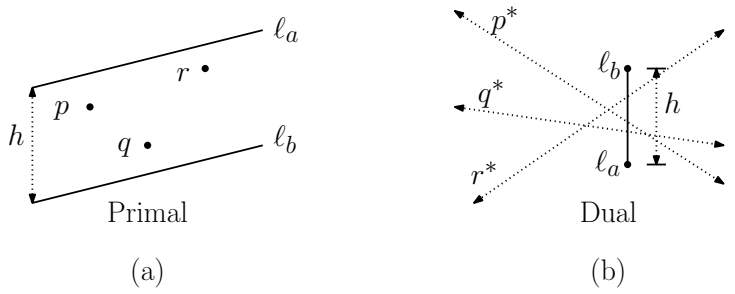



Fig. 67: A 3-corridor in the (a) primal and (b) dual form. (Note that the corridor is not as narrow as possible.)

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 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.

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 (see Fig. 68). We can solve this problem by plane sweep in $O(n^2 \log n)$ time and $O(n)$ space. (By using topological plane sweep, the extra $\log n$ factor can be removed.)

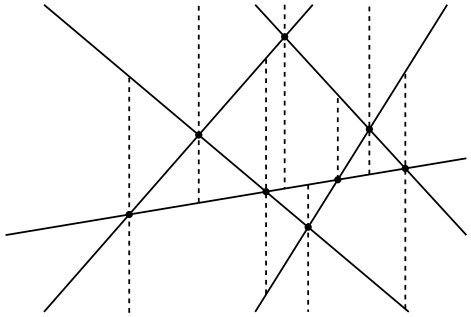


Fig. 68: The critical line segments used in computing the narrowest 3-corridor.

**Halfplane Discrepancy:** Next we consider a problem derived from computer graphics and sampling. Suppose that we are given a collection of $n$ points $P$ 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 $\mu_P(h) = |P \cap h|/|P|$, then we would like $\mu(h) \approx \mu_P(h)$ 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 $P$ with respect to a halfplane $h$ to be

$$\Delta_P(h) = |\mu(h) - \mu_P(h)|.$$

For example, in Fig. 69(a), the area of $h \cap U$ is $\mu(h) = 0.625$, and there are 7 out of 13 points in $h$, thus $\mu_P(h) = 7/13 = 0.538$. Thus, the discrepancy of $h$ is $|0.625 - 0.538| = 0.087$. Define the *halfplane discrepancy* of $P$ to be the maximum (or more properly the supremum, or least upper bound) of this quantity over all halfplanes:

$$\Delta(P) = \sup_h \Delta_P(h).$$

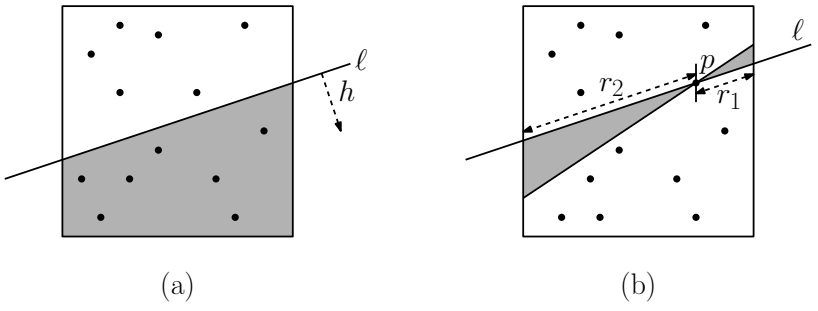



Fig. 69: Discrepancy of a point set.

Since there are an uncountably infinite number of halfplanes, it is important to derive some sort of *finiteness criterion* on the set of halfplanes that might produce the greatest discrepancy.

**Lemma:** Let $h$ denote the halfplane that generates the maximum discrepancy with respect to $P$, and let $\ell$ denote the line that bounds $h$. Then either (i) $\ell$ passes through at least two points of $P$, or (ii) $\ell$ passes through one point of $P$, and this point is the midpoint of the line segment $\ell \cap U$.

**Remark:** If a line passes through one or more points of $P$, then should this point be included in $\mu_P(h)$? For the purposes of computing the maximum discrepancy, the answer is to either include or omit the point, whichever will generate the larger discrepancy. The justification is that it is possible to perturb $h$ infinitesimally so that it includes none or all of these points without altering $\mu(h)$.

**Proof:** If $\ell$ does not pass through any point of $P$, then (depending on which is larger $\mu(h)$ or $\mu_P(h)$) we can move the line up or down without changing $\mu_P(h)$ and increasing or decreasing $\mu(h)$ to increase their difference. If $\ell$ passes through a point $p \in P$, but is not the midpoint of the line segment $\ell \cap U$, then we claim that we can rotate this line about $p$ and hence increase or decrease $\mu(h)$ without altering $\mu_P(h)$, to increase their difference.

To establish the claim, consider Fig. 69(b). Suppose that the line $\ell$ passes through point $p$ and let $r_1 < r_2$ denote the two lengths along $\ell$ from $p$ to the sides of the square. Observe that if we rotate $\ell$ through a small angle $\theta$, then to a first order approximation, the gain due to area of the triangle on the right is $r_1^2\theta/2$, since this triangle can be approximated by an angular sector of a circle of radius $r_1$ and angle $\theta$. The loss due to the area of the triangle on the left is $r_2^2\theta/2$. Thus, since $r_1 < r_2$ this rotation will decrease the area of region lying below $h$ infinitesimally. A rotation in the opposite increases the area infinitesimally. Since the number of points bounded by $h$ does not change as a function of $\theta$, the discrepancy cannot be achieved as long as such a rotation is possible.

Call the lines satisfying (i) as *type-1* and the lines satisfying (ii) as *type-2*. We will show that the discrepancy for each set of lines can be computed in $O(n^2)$ time.

Since for each point $p \in P$ there are only a constant number of lines $\ell$ (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 $O(n)$ type-1 lines, and hence the discrepancy of all of these lines can be tested by brute force in $O(n^2)$ time.

**Type-2 Discrepancies and Levels:** Computing the discrepancies of the type-2 lines will involve arrangements. In the primal plane, a line $\ell$ that passes through two points $p_i, p_j \in P$, is mapped in the dual plane to a point $\ell^*$ at which the lines $p_i^*$ and $p_j^*$ intersect. This is just a vertex in the arrangement of the dual lines for $P$. 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 $P$ lying below $\ell$, for $\ell$'s lower halfspace, and the number of points lying above it, for $\ell$'s upper halfspace. (As indicated in the above remark, we take the two points lying on $\ell$ as being above or below, whichever makes the discrepancy higher.) In the dual plane, this corresponds to determining the number of dual lines that lie above each vertex in the arrangement and the number of lines that lie below it. If we know the number of dual lines that lie above each vertex in the arrangement, then it is trivial to compute the number of lines that lie below by subtraction.

In order to count the number of lines lying above/below a vertex of the arrangement, it will be useful to the notion of a level in an arrangements. We say that a point is at *level* $k$, denoted $\mathcal{L}_k$, in an arrangement if there are at most $k-1$ lines above this point and at most $n-k$ lines below this point. The $k$-th level of an arrangement is an $x$-monotone polygonal curve (see Fig. 70(a)). For example, the upper envelope of the lines is level 1 of the arrangement, and the lower envelope is level $n$. Note that (assuming general position) each vertex of the arrangement is generally on two levels. (Beware: Our definition of level is exactly one greater than our text's definition.)

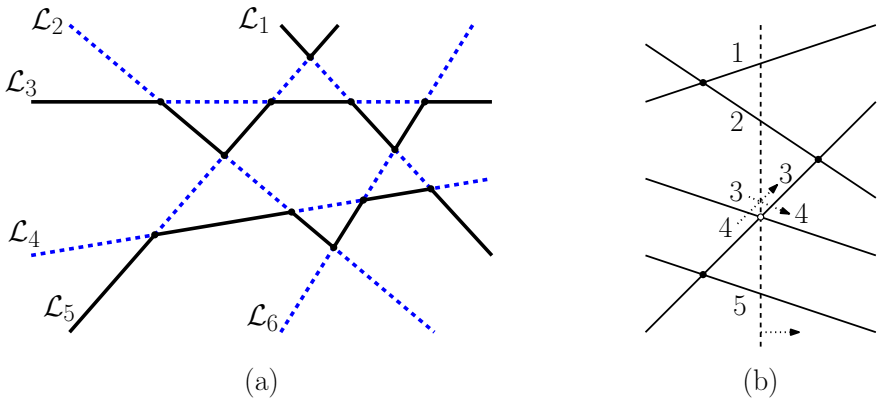



Fig. 70: Examples of levels in an arrangement.

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. Whenever we sweep over a vertex, we swap the level numbers associated with the two lines (see Fig. 70(b)).

Thus, by using plane sweep, in $O(n^2 \log n)$ time and $O(n)$ space, we can determine 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 one indicates the number of primal points that lie strictly below the corresponding primal line and the maximum level number is the number of primal 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 discrepancies of all the type-2 lines in $O(n^2 \log n)$ time and $O(n)$ space, through plane sweep. (Through the use of topological plane sweep, the extra factor of $\log n$ can be eliminated.)

# Lecture 16: Orthogonal Range Searching and kd-Trees

**Geometric Retrieval:** We will shift our focus from algorithm problems to data structures for the next few lectures. We will consider the following class of problems. Given a collection of objects, preprocess them (storing the results in a data structure of some variety) so that queries of a particular form can be answered efficiently. Generally we measure data structures in terms of two quantities, the time needed to answer a query and the amount of space needed by the data structure. Often there is a trade-off 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 or $O(n \log n)$ preprocessing time.

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

**Range Queries:** In a *range queries* we are given a set $P$ of points and region $Q$ in space (e.g., a rectangle, polygon, halfspace, or disk) and are asked to provide some information about the points of $P$ lying within $Q$. Examples of the types of information include the following:

**Range reporting:** Return a list of all the points of $P$ that lie within $Q$

**Range counting:** Return a count of all the points of $P$ that lie within $Q$. There are a number of variations.

**Weights:** Each point $p \in P$ is associated with a numeric weight $w(p)$. Return the sum of weights of the points of $P$ lying within $Q$

**Semigroup weights:** The weights need not be numbers and the operation need not be addition. In general, the weights of $P$ are drawn from any commutative semigroup. A commutative semigroup is pair $(\Sigma, \circ)$, where $\Sigma$ is a set, and $\circ : \Sigma \times \Sigma \to \Sigma$ is a commutative and associative binary operator on $\Sigma$. The objective is to return the "sum" of the weights of the elements of $P \cap Q$, where "$\circ$" takes the role of addition.

For example, if we wanted to compute the maximum weight of a set of real values, we could use the semigroup $(\mathbb{R}, \max)$. If we wanted to know the parity of the number of points of $P$ in $Q$, we could take the semigroup $(\{0, 1\}, \oplus)$, where $\oplus$ denotes exclusive-or (or equivalently, addition modulo 2).

**Group weights:** A group is a special case of a semigroup, where inverses exist. (For example, the semigroup of reals under addition $(\mathbb{R}, +)$ is a group (where subtraction plays the role of inverse), but the semigroup $(\mathbb{R}, \max)$ is *not* a group (since the max operator does not have inverses).

If it is known that the semigroup is, in fact, a group, the data structure may take advantage of this to speed-up query processing. For example, the query processing algorithm has the flexibility to both "add" and "subtract" weights.

To achieve the best possible performance, range searching data structures are tailored to the particular type of query ranges and the properties of the semigroup involved. On the other hand, a user may prefer to sacrifice efficiency for a data structure that is more general and can answer a wide variety of range searching problems.

**Range Spaces and VC-Dimension:** An important concept underlying geometric range searching is that the subsets that can be formed by simple geometric ranges (such as rectangles, discs, triangles, half-spaces) are typically much more restrictive than the set of all possible subsets, which is called the *power set*, of $P$.

We can characterize any range search problem abstractly as follows. A *range space* is defined to be a pair $(X, \mathcal{R})$ where $X$ is an arbitrary set and $\mathcal{R}$ is a subset of the power set of $X$. For example, $X$ might be the real 2-dimensional plane and $\mathcal{R}$ might be the set of all closed, bounded triangles. Given a set $P \subseteq X$, define

$$\Pi_{\mathcal{R}}(P) \;=\; \{P \cap Q \mid Q \in \mathcal{R}\}.$$

That is, $\Pi_{\mathcal{R}}(P)$ is the collection of subsets of $P$ that can be formed by intersecting $P$ with the ranges of the range space.

For example consider the range space consisting of axis-parallel rectangles in $\mathbb{R}^2$. Fig. 71 illustrates a number of the subsets of $P$ that constitute $\Pi_{\mathcal{R}}(P)$. Note that not all subsets of $P$ are in $\Pi_{\mathcal{R}}(P)$. For example, the sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be formed by intersecting $P$ with axis-parallel rectangular ranges.

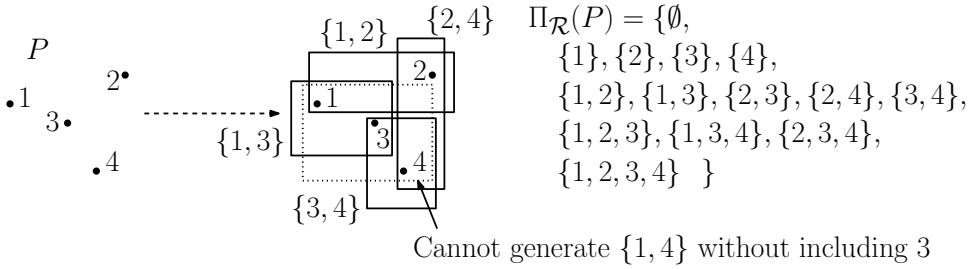



Fig. 71: A 4-point set and the range space of axis-parallel rectangles. Note that sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be generated.

Suppose that we are given a set $P$ of $n$ points in the plane and $\mathcal{R}$ consists of axis parallel rectangles. How large might $\Pi_{\mathcal{R}}(P)$ be? If we take any axis-parallel rectangle that encloses some subset of $P$, and we shrink it as much as possible without altering the points contained within, we see that such a rectangle is determined by four points of $P$, that is, the points that lie on the rectangle's top, bottom, left, and right sides. It is easy to see, therefore, that, for this particular range space, we have $\Pi_{\mathcal{R}}(P) = O(n^4)$.

How complex is an arbitrary range space? A useful concept is the notion of *VC dimension*, which is short for *Vapnik-Chervonenkis dimension*.[15] Given an arbitrary range space $(X, \mathcal{R})$ and point set $P$, we say that $\mathcal{R}$ *shatters* $P$ is $\Pi_{\mathcal{R}}(P)$ is equal to the power set of $P$, that is, we can form any of the $2^{|P|}$ subsets of $P$ by taking intersections with the ranges of $\mathcal{R}$. For example, the point set shown in Fig. 71 is *not* shattered by the range space of axis-parallel rectangles. However, the four-element point set $P'$ shown in Fig. 72 is shattered by this range space.

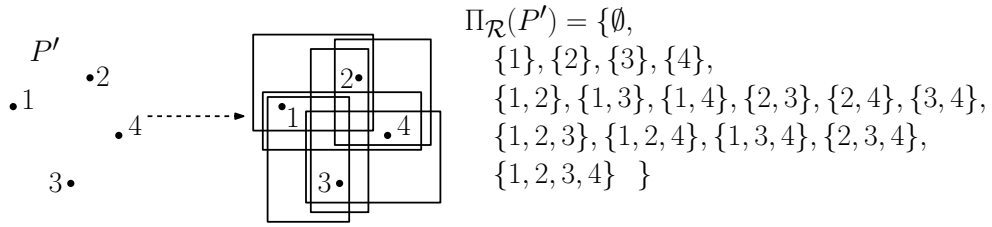



Fig. 72: A 4-point set that is shattered by the range space of axis-parallel rectangles. (We show only the 2-element point sets in the drawing.)

The *VC-dimension* of a range space $(X, \mathcal{R})$ is defined to be the size of the *largest* point set that is shattered by the range space. In Fig. 72 we have shown that the four-element point set $P'$ is shattered by the range space of axis-parallel rectangles. It is not hard to show, however, that no 5-element point set of $\mathbb{R}^2$ can be shattered by this same range space. (We will leave this as an exercise.) Therefore, the VC-dimension of the range space of 2-dimensional axis-parallel rectangles is four.

The VC-dimension of a range space provides useful information as to the complexity of answering range queries for such a space. An important result in this area is *Sauer's Lemma*, which states that, if $(X, \mathcal{R})$ has VC dimension $d$, then $|\Pi_{\mathcal{R}}(P)| = O(n^d)$, where $n = |P|$. This is consistent with the observation that we made earlier for the case of axis-parallel rectangles.

**Canonical Subsets:** A common approach used in solving almost all range queries is to represent $P$ as a collection of *canonical subsets* $\{P_1, P_2, \ldots, P_k\}$, each $P_i \subseteq P$ (where $k$ is generally a function of $n$ and the type of ranges), such that any set can be formed as the disjoint union of canonical subsets. Note that these subsets may generally overlap each other.

There are many ways to select canonical subsets, and the choice affects the space and time complexities. For example, the canonical subsets might be chosen to consist of $n$ singleton sets, each of the form $\{p_i\}$. This would be very space efficient, since we need only $O(n)$ total space to store all the canonical subsets, but in order to answer a query involving $k$ objects we would need $k$ sets. (This might not be bad for reporting queries, but it would be too long for counting queries.) At the other extreme, we might let the canonical subsets be all the sets of the range space $\mathcal{R}$. Thus, any query could be answered with a single canonical subset (assuming we could determine which one), but we would have $|\mathcal{R}|$ different canonical subsets to store, which is typically a higher ordered polynomial in $n$, and may be too high to be of practical value. The goal of a good range data structure is to strike a balance between the total number of canonical subsets (space) and the number of canonical subsets needed to answer a query (time).

Perhaps the most common way in which to define canonical subsets is through the use of a *partition tree*. A partition tree is a rooted (typically binary) tree, whose leaves correspond to the points of $P$. Each node $u$ of such a tree is naturally associated with a subset of $P$, namely, the points stored in the leaves of the subtree rooted at $u$. We will see an example of this when we discuss one-dimensional range queries.

**One-dimensional range queries:** Before we consider how to solve general range queries, let us consider how to answer 1-dimension range queries, or *interval queries*. Let us assume that we are given a set of points $P =$

[15] The concept of VC-dimension was first developed in the field of probability theory in the 1970's. The topic was discovered to be very relevant to the fields of machine learning and computational geometry in late 1980's.

$\{p_1, p_2, \ldots, p_n\}$ on the line, which we will preprocess into a data structure. Then, given an interval $[x_{lo}, x_{hi}]$, the goal is to count or report all the points lying within the interval. Ideally, we would like to answer counting queries in $O(\log n)$ time, and we would like to answer reporting queries in time $O((\log n) + k)$ time, where $k$ is the number of points reported.

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 enumerate (or count) all the points between. This works fine in dimension 1, but does not generalize readily to any higher dimensions. Also, it does not work when dealing with the weighted version, unless the weights are drawn from a group.

Let us consider a different approach, which will generalize to higher dimensions. Sort the points of $P$ in increasing order and store them in the leaves of a balanced binary search tree. Each internal node of the tree is labeled with the largest key appearing in its left child. We can associate each node of this tree (implicitly or explicitly) with the subset of points stored in the leaves that are descendants of this node. This gives rise to the $O(n)$ *canonical subsets*. In order to answer reporting queries, the canonical subsets do *not* need to be stored explicitly with each node of the tree. The reason is that we can enumerate each canonical subset in time proportional to its size by simply traversing the subtree and reporting the points lying in its leaves. This is illustrated in Fig. 73. For range counting, we associate each node with the total weight of points in its subtree.

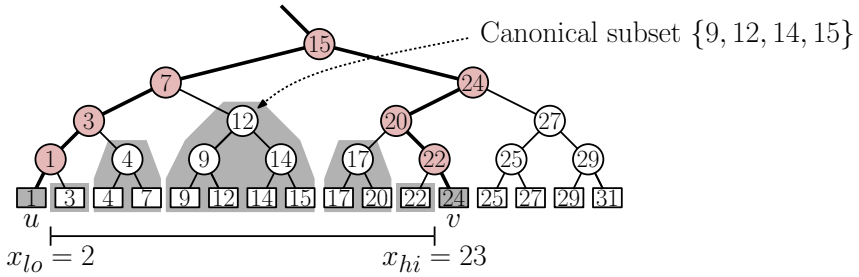



Fig. 73: Canonical sets for interval queries. For range reporting, canonical subsets are generated as needed by traversing the subtree.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the rightmost leaf $u$ whose key is less than $x_{lo}$ and the leftmost leaf $v$ whose key is greater than $x_{hi}$. (To make this possible for all ranges, we could add two sentinel points with values of $-\infty$ and $+\infty$ to form the leftmost and rightmost leaves.) Clearly all the leaves between $u$ and $v$ constitute the points that lie within the range. To form these canonical subsets, we take the subsets of all the *maximal subtrees* lying between the paths from the root $u$ and $v$.

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

As mentioned earlier, to answer a range reporting query we simply traverse the canonical subtrees, reporting the points of their leaves. To answer a range counting query we return the sum of weights associated with the nodes of the canonical subtrees.

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

In summary, 1-dimensional range queries can be answered in $O(\log n)$ (counting) or $((\log n) + k)$ (reporting) time, using $O(n)$ storage. This concept of finding maximal subtrees that are contained within the range is

fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Let see next how can we extend this to higher dimensional range queries.

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

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

A kd-tree is an example of a partition tree. For each node, we subdivide space either by splitting along the $x$-coordinates or along the $y$-coordinates of the points. Each internal node $t$ of the kd-tree is associated with the following quantities:

| | |
|---|---|
| $t$.*cut-dim* | the cutting dimension (e.g., $x = 0$ and $y = 1$) |
| $t$.*cut-val* | the cutting value (a real number) |
| $t$.*weight* | the number (or generally, total weight) of points in $t$'s subtree |

In dimension $d$, the cutting dimension may be represented as in integer ranging from 0 to $d - 1$. If the cutting dimension is $i$, then all points whose $i$th coordinate is less than or equal to $t$.*cut-val* are stored in the left subtree and the remaining points are stored in the right subtree. (See Fig. 74.) If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side. This is done to allow us to balance the number of points in the left and right subtrees if there are many equal coordinate values. When a single point remains (or more generally a small constant number of points), we store it in a leaf node, whose only field $t$.*point* is this point.

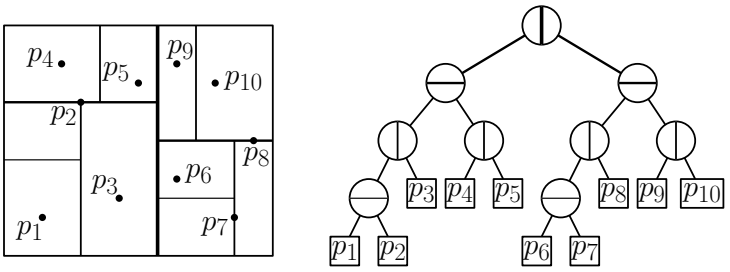



Fig. 74: A kd-tree and the associated spatial subdivision.

The cutting process has a geometric interpretation. Each node of the tree is associated implicitly with a rectangular region of space, called a *cell*. (In general these rectangles may be unbounded, but in many applications it is common to restrict ourselves to some bounded rectangular region of space before splitting begins, and so all these rectangles are bounded.) The cells are nested in the sense that a child's cell is contained within its parent's cell. Hence, these cells define a *hierarchical decomposition* of space. This is illustrated on the left side of Fig. 74.

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

**How is the cutting dimension chosen?** The simplest method is to cycle through the dimensions one by one. (This method is shown in Fig. 74.) Since the cutting dimension depends only on the level of a node in the

tree, one advantage of this rule is that the cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

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

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

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

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

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

**Theorem:** Given $n$ points, it is possible to build a kd-tree of height $O(\log n)$ and space $O(n)$ in time $O(n \log n)$ time.

**Range Searching in kd-trees:** Let us consider how to answer orthogonal range counting queries. Range reporting queries are an easy extension. Let $Q$ denote the desired range, and $u$ denote the current node in the kd-tree. We assume that each node $u$ is associated with its rectangular cell, denoted $u.cell$. (Alternately, this can be computed on the fly, as the algorithm is running.) The search algorithm is presented in the code block below.

kd-tree Range Counting Query

```
int range-count(Range Q, KDNode u)
(1) if (u is a leaf)
    (a) if (u.point ∈ Q) return u.weight,
    (b) else return 0 /* or generally, the semigroup identity element */
(2) else /* u is internal */
    (a) if (u.cell ∩ Q = ∅) return 0 /* the query does not overlap u's cell */
    (b) else if (u.cell ⊆ Q) return u.weight /* u's cell is contained within query range */
    (c) else, return range-count(Q, u.left) + range-count(Q, u.right).
```

The search algorithm traverses the tree recursively. If it arrives at a leaf cell, we check to see whether the associated point, $u.point$, lies within $Q$ in $O(1)$ time, and if so we count it. Otherwise, $u$ is an internal node. If $u.cell$ is disjoint from $Q$ (which can be tested in $O(1)$ time since both are rectangles), then we know that no point in the subtree rooted at $u$ is in the query range, and so there is nothing to count. If $u.cell$ is entirely contained within $Q$ (again testable in $O(1)$ time), then every point in the subtree rooted at $u$ can be counted.

(These points constitute a canonical subset.) Otherwise, $u$'s cell partially overlaps $Q$. In this case we recurse on $u$'s two children and update the count accordingly.

Fig. 75 shows an example of a range search. Blue shaded nodes contribute to the search result and red shaded nodes do not. The red shaded subtrees are not visited. The blue-shaded subtrees are not visited for the sake of counting queries. Instead, we just access their total weight.

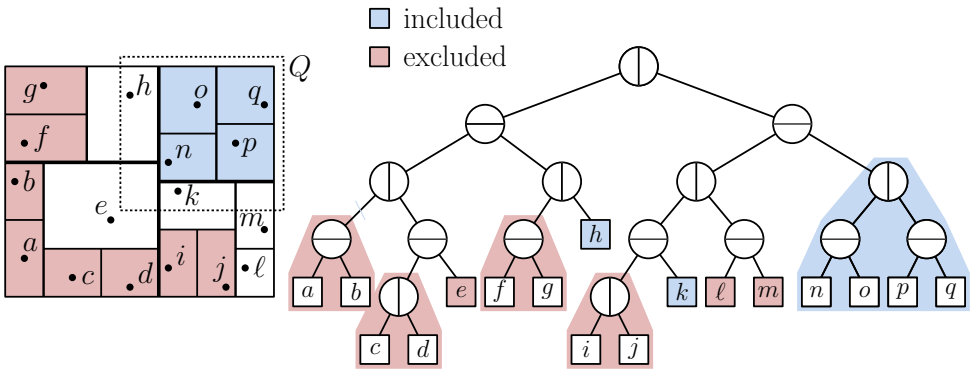



Fig. 75: Range search in a kd-tree. (Note: This particular tree was not generated by the algorithm described above.)

**Analysis of query time:** How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree. Rather than counting visited nodes, we will count nodes that are *expanded*. We say that a node is expanded if it is visited and both its children are visited by the recursive range count algorithm.

A node is expanded if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are expanded in the search, it suffices to bound the number of nodes whose cells are stabbed.

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

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

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

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

This can be expressed more formally as the following recurrence. Let $T(n)$ denote the number of nodes stabbed for a subtree containing $n$ points. We have

$$T(n) \leq \begin{cases} 2 & \text{if } n \leq 4, \\ 1 + 2T\left(\frac{n}{4}\right) & \text{otherwise.} \end{cases}$$

We can solve this recurrence by appealing to the Master theorem for solving recurrences, as presented in the book by Cormen, Leiserson, Rivest and Stein. To keep the lecture self-contained, let's solve it by repeated expansion.

$$\begin{aligned} T(n) &\leq 1 + 2T\left(\frac{n}{4}\right) \\ &\leq 1 + 2\left(1 + 2T\left(\frac{n/4}{4}\right)\right) \;=\; (1+2) + 4T\left(\frac{n}{16}\right) \\ &\leq (1+2) + 4\left(1 + 2T\left(\frac{n/16}{4}\right)\right) \;=\; (1+2+4) + 8T\left(\frac{n}{64}\right) \\ &\leq \cdots \\ &\leq \sum_{i=0}^{k-1} 2^i + 2^k T\left(\frac{n}{4^k}\right). \end{aligned}$$

To get to the basis case ($T(1)$) let's set $k = \log_4 n$, which means that $4^k = n$. Observe that $2^{\log_4 n} = 2^{(\log_2 n)/2} = n^{1/2} = \sqrt{n}$. Since $T(1) \leq 2$, we have

$$T(n) \;\leq\; (2^{\log_4 n} - 1) + 2^{\log_4 n} T(1) \;\leq\; 3\sqrt{n} \;=\; O(\sqrt{n}).$$

This completes the proof.

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

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

# Lecture 17: Orthogonal Range Trees

**Orthogonal Range Trees:** Last time we saw that kd-trees could be used to answer orthogonal range queries in the plane in $O(\sqrt{n})$ time for counting and $O(\sqrt{n} + k)$ time for reporting. It is natural to wonder whether we can replace the $O(\sqrt{n})$ term with something closer to the ideal query time of $O(\log n)$. Today we consider a data structure, which is more highly tuned to this particular problem, called an *orthogonal range tree*. Recall that we are given a set $P$ of $n$ points in $\mathbb{R}^2$, and our objective is to preprocess these points so that, given any axis-parallel rectangle $Q$, we can count or report the points of $P$ that lie within $Q$ efficiently.

An orthogonal range tree is a data structure which, in the plane uses $O(n \log n)$ space and can answer range reporting queries in $O(\log n + k)$ time, where $k$ is the number of points reported. In general in dimension $d \geq 2$, it uses $O(n \log^{(d-1)} n)$ space, and can answer orthogonal rectangular range queries in $O(\log^{(d-1)} n + k)$ time. The preprocessing time is the same as the space bound. 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 generalization to higher dimensions will be straightforward.

**Multi-level Search Trees:** The orthogonal range-tree data structure is a nice example of a more general concept, called a *multi-level search tree*. In this method, a complex search is decomposed into a constant number of simpler range searches. Recall that a range space is a pair $(X, \mathcal{R})$ consisting of a set $X$ and a collection $\mathcal{R}$ of subsets of $X$, called *ranges*. Given a range space $(X, \mathcal{R})$, suppose that we can decompose it into two (or generally a small number of) range subspaces $(X, \mathcal{R}_1)$ and $(X, \mathcal{R}_2)$ such that any query $Q \in \mathcal{R}$ can be expressed as $Q_1 \cap Q_2$, for $Q_i \in \mathcal{R}_i$. (For example, an orthogonal range query in the plane, $[x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$, can be

expressed as the intersection of a vertical strip and a horizontal strip, in particular, the points whose $x$-coordinates are in the range $Q_1 = [x_{lo}, x_{hi}] \times \mathbb{R}$ and the points whose $y$-coordinates are in the range $Q_2 = \mathbb{R} \times [y_{lo}, y_{hi}]$.) The idea is to then "cascade" a number of search structures, one for each range subspace, together to answer a range query for the original space.

Let's see how to build such a structure for a given point set $P$. We first construct an appropriate range search structure, say, a partition tree, for $P$ for the *first* range subspace $(X, \mathcal{R}_1)$. Let's call this tree $T$ (see Fig. 76). Recall that each node $u \in T$ is implicitly associated with a *canonical subset* of points of $P$, which we will denote by $P_u$. In the case that $T$ is a partition tree, this is just the set of points lying in the leaves of the subtree rooted at $u$. (For example, in Fig. 76, $P_{u_6} = \{p_5, \ldots, p_8\}$.) For each node $u \in T$, we construct an *auxiliary search tree* for the points of $P_u$, but now over the *second* range subspace $(X, \mathcal{R}_2)$. Let $T_u$ denote the resulting tree (see Fig. 76). The final data structure consists of the primary tree $T$, the auxiliary search trees $T_u$ for each $u \in T$, and a link from each node $u \in T$ to the corresponding auxiliary search tree $T_u$. The total space is the sum of space requirements for the primary tree and all the auxiliary trees.

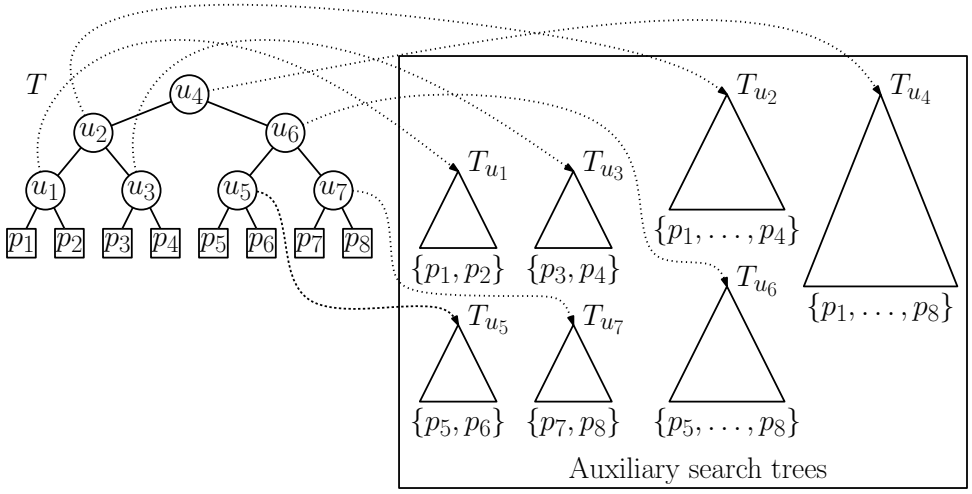



Fig. 76: Multi-level search trees.

Now, given a query range $Q = Q_1 \cap Q_2$, where $Q_i \in \mathcal{R}_i$, we answer queries as follows. Recall from our earlier lecture that, the partition tree $T$ allows us to express the answer to the query $P \cap Q_1$ as a disjoint union $\bigcup_u P_u$ for an appropriate (and ideally small) subset of nodes $u \in T$. Call this subset $U(Q_1)$. In order to complete the query, for each $u \in U(Q_1)$, we access the corresponding auxiliary search tree $T_u$ in order to determine the subset of points $P_u$ that lie within the query range $Q_2$. To see why this works, observe that

$$P \cap Q \;=\; (P \cap Q_1) \cap Q_2 \;=\; \left( \bigcup_{u \in U(Q_1)} P_u \right) \cap Q_2 \;=\; \left( \bigcup_{u \in U(Q_1)} P_u \cap Q_2 \right).$$

Therefore, once we have computed the answers to all the auxiliary ranges $P_u \cap Q_2$ for all $u \in U(Q_1)$, all that remains is to combine the results (e.g., by summing the counts or concatenating all the lists, depending on whether we are counting or reporting, respectively). The query time is equal to the sum of the query times over all the trees that were accessed.

**A Multi-Level Approach to Orthogonal Range Searching:** Now, let us consider how to apply the abstract framework of a multi-level search tree to the problem of 2-dimensional orthogonal range queries. First, we assume that we have preprocessed the data by building a range tree for the first range query, which in this case is just a 1-dimensional range tree for the $x$-coordinates. Recall that this is just a balanced binary tree $T$ whose leaves are the points of $P$ sorted by $x$-coordinate. Each node $u$ of this binary tree is implicitly associated with a canonical

subset $P_u \subseteq P$ consisting of the points lying within the leaves in $u$'s subtree. Next, for each node $u \in T$, we build a 1-dimensional range tree for $P_u$, sorted this time by $y$-coordinates. The resulting tree is called $T_u$.

The final data structure, called a *2-dimensional range tree* consists of two levels: an $x$-range tree $T$, where each node $u \in T$ points to auxiliary $y$-range search tree $T_u$. (For $d$-dimensional range trees, we will have $d$-levels of trees, one for each coordinate.)

Queries are answered as follows. Consider an orthogonal range query $Q = [x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$. Let $Q_1 = [x_{lo}, x_{hi}] \times \mathbb{R}$ and $Q_2 = \mathbb{R} \times [y_{lo}, y_{hi}]$. First, we query $T$ to determine the subset $U(Q_1)$ of $O(\log n)$ nodes $u$ such that $\bigcup_{u \in U(Q_1)} P_u$ forms a disjoint cover of the points of $P$ whose $x$-coordinate lies within $[x_{lo}, x_{hi}]$. (These are the roots of the shaded subtrees in the top half of Fig. 77.) For each $u \in U(Q_1)$, we access the auxiliary tree $T_u$ and perform a 1-dimensional range search (based on $y$-coordinates) to determine the subset of $P_u$ that lies within $Q_2$, that is, the points whose $y$-coordinates lie within $[y_{lo}, y_{hi}]$ (see Fig.77).

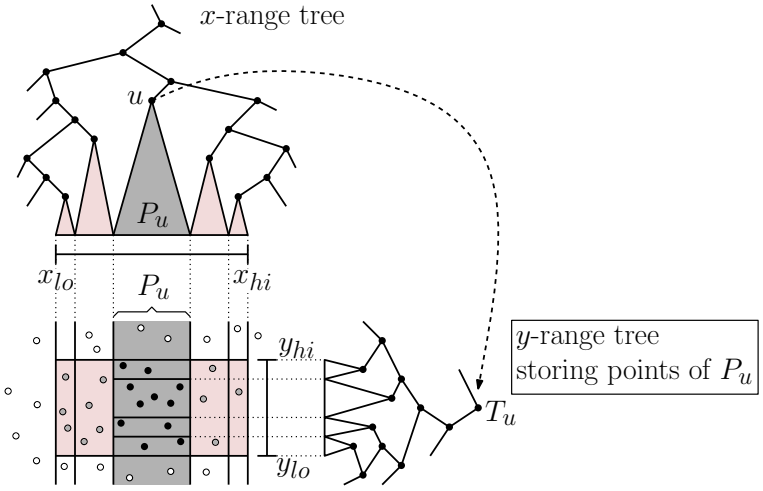



Fig. 77: Orthogonal range tree search.

What is the query time? Recall that it takes $O(\log n)$ time to locate the nodes representing the canonical subsets for the 1-dimensional range query over the $x$-coordinates, and there are $O(\log n)$ nodes $u \in U(Q_1)$. For each such node, we invoke a 1-dimensional range search over the $y$-coordinates on the canonical subset $P_u$, which will result in the generation of $O(\log |P_u|) \leq O(\log n)$ canonical sets. Thus, (ignoring constant factors) the total number of canonical subsets accessed by the algorithm is

$$\sum_{u \in U(Q_1)} \log |P_u| \;\leq\; |U(Q_1)| \cdot \log n \;\leq\; \log^2 n.$$

As before, listing the elements of these sets can be performed in additional $O(k)$ time by just traversing the subtrees corresponding to the canonical subsets of the auxiliary search trees that contribute the final result. Counting queries can be answered by precomputing the subtree sizes for each node of each auxiliary search tree, and just adding up all those that contribute to the query. Therefore, reporting queries can be answered in $O((\log^2 n) + k)$ time and counting queries can be answered in $O(\log^2 n)$ time. It is easy to see that we can generalize this to orthogonal range searching in $\mathbb{R}^d$ by cascading $d$ levels of 1-dimensional search trees. The log factor in the resulting query time would be $\log^d n$.

**Space and Preprocessing Time:** To derive a bound on the total space used, we sum the sizes of all the trees. The primary search tree $T$ for the $x$-coordinates requires only $O(n)$ storage. For each node $u \in T$, the size of the auxiliary search tree $T_u$ is clearly proportional to the number of points in this tree, which is the size of the

associated canonical subset, $|P_u|$. Thus, up to constant factors, the total space is

$$n + \sum_{u \in T} |P_u|.$$

To bound the size of the sum, observe that each point of $P$ appears in the set $P_u$ for each ancestor of this leaf. Since the tree $T$ is balanced, its depth is $O(\log n)$, and therefore, each point of $P$ appears in $O(\log n)$ of the canonical subsets. Since each of the $n$ points of $P$ contributes $O(\log n)$ to the sum, it follows that the sum is $O(n \log n)$.

In summary, the space required by the orthogonal range tree is $O(n \log n)$. Observe that for the purposes of reporting, we could have represented each auxiliary search tree $T_u$ as an array containing the points of $P_u$ sorted by the $y$-coordinates. The advantage of using a tree structure is that it makes it possible to answer counting queries over general semigroups, and it makes efficient insertion and deletion possible as well.

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.

**Improved Query Times through Fractional Cascading:** Can we improve on the $O(\log^2 n)$ query time? We would like to reduce the query time to $O(\log n)$. (In general, this approach will shave a factor of $\log n$ from the query time, which will lead to a query time of $O(\log^{d-1} n)$ in $\mathbb{R}^d$).

What is the source of the extra log factor? As we descend the search the $x$-interval tree, for each node we visit, we need to search the corresponding auxiliary search tree based on the query's $y$-coordinates $[y_{lo}, y_{hi}]$. It is this combination that leads to the squaring of the logarithms. If we could search each auxiliary in $O(1)$ time, then we could eliminate this annoying log factor.

There is a clever trick that can be used to eliminate the additional log factor. Observe that we are repeatedly searching different lists (in particular, these are subsets of the canonical subsets $P_u$ for $u \in U(Q_1)$) but always with the *same* search keys (in particular, $y_{lo}$ and $y_{hi}$). How can we exploit the fact that the search keys are static to improve the running times of the individual searches?

The idea to rely on economies of scale. Suppose that we merge all the different lists that we need to search into a single master list. Since $\bigcup_u P_u = P$ and $|P| = n$, we can search this master list for any key in $O(\log n)$ time. We would like to exploit the idea that, if we know where $y_{lo}$ and $y_{hi}$ lie within the master list, then it should be easy to determine where they are located in any canonical subset $P_u \subseteq P$. Ideally, after making one search in the master list, we would like to be able to answer all the remaining searches in $O(1)$ time each. Turning this intuition into an algorithm is not difficult, but it is not trivial either.

In our case, the master list on which we will do the initial search is the entire set of points, sorted by $y$-coordinate. We will assume that each of the auxiliary search trees is a sorted array. (In dimension $d$, this assumption implies that we can apply this only to the last level of the multi-level data structure.) Call these the *auxiliary lists*.

Here is how we do this. Let $v$ be an arbitrary internal node in the range tree of $x$-coordinates, and let $v'$ and $v''$ be its left and right children. Let $A$ be the sorted auxiliary list for $v$ and let $A'$ and $A''$ be the sorted auxiliary lists for its respective children. Observe that $A$ is the disjoint union of $A'$ and $A''$ (assuming no duplicate $y$-coordinates). For each element in $A$, we store two pointers, one to the item of equal or larger value in $A'$ and the other to the item of equal or larger value in $A''$. (If there is no larger item, the pointer is null.) Observe that once we know the position of an item in $A$, then we can determine its position in either $A'$ or $A''$ 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 $v'$ and $v''$ denote its left and right children. Suppose that (in increasing order of $y$-coordinates) the associated nodes within this range

are: $\langle p_1, p_2, p_3, p_4, p_5, p_6 \rangle$, and suppose that in $v'$ we store the points $\langle p_2, p_4, p_5 \rangle$ and in $v''$ we store $\langle p_1, p_3, p_6 \rangle$ (see Fig. 78(a)). For each point in the auxiliary list for $v$, we store a pointer to the lists $v'$ and $v''$, 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 (see Fig. 78(b)).

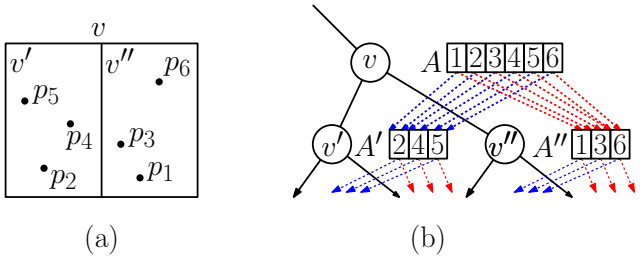



Fig. 78: Cascaded search in range trees.

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, the running time is $O(\log n)$, rather than $O(log^2 n)$. By applying this to the last level of the auxiliary search structures, we save one log factor, which gives us the following result.

**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 space $O(n \log^{(d-1)} n)$ which can be constructed in $O(n \log^{(d-1)} n)$ time.

This technique is special case of a more general data structures technique called *fractional cascading*. The idea is that information about the search the results "cascades" from one level of the data structure down to the next.

The result can be applied to range counting queries as well, but under the provision that we can answer the queries using a sorted array representation for the last level of the tree. For example, if the weights are drawn from a group, then the method is applicable, but if the the weights are from a general semigroup, it is not possible. (For general semigroups, we need to sum the results for individual subtrees, which implies that we need a tree structure, rather than a simple array structure.)

# Lecture 18: Well Separated Pair Decompositions

**Approximation Algorithms in Computational Geometry:** Although we have seen many efficient techniques for solving fundamental problems in computational geometry, there are many problems for which the complexity of finding an exact solution is unacceptably high. Geometric approximation arises as a useful alternative in such cases. Approximations arise in a number of contexts. One is when solving a hard optimization problem. A famous example is the *Euclidean traveling salesman problem*, in which the objective is to find a minimum length path that visits each of $n$ given points (see Fig. 79(a)). (This is an NP-hard problem, but there exists a polynomial time algorithm that achieves an approximation factor of $1 + \varepsilon$ for any $\varepsilon > 0$.) Another source arises when approximating geometric structures. For example, early this semester we mentioned that the convex hull of $n$ points in $\mathbb{R}^d$ could have combinatorial complexity $\Omega(n^{\lfloor d/2 \rfloor})$. Rather than computing the exact convex hull, it may be satisfactory to compute a convex polytope, which has much lower complexity, and whose boundary is within a small distance $\varepsilon$ from the actual hull (see Fig. 79(b)).

Another important motivations for geometric approximations is that geometric inputs are typically the results of sensed measurements, which are subject to limited precision. There is no good reason to solve a problem to a degree of accuracy that exceeds the precision of the inputs themselves.

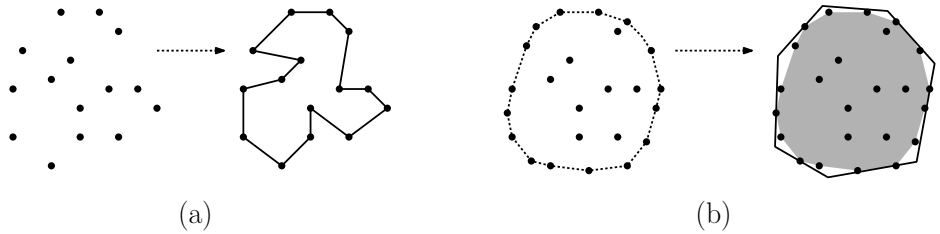



Fig. 79: Geometric approximations: (a) Euclidean traveling salesman, (b) approximate convex hull.

**Motivation: The $n$-Body Problem:** We begin our discussion of approximation algorithms in geometry with a simple and powerful example. To motivate this example, consider an application in physics involving the simulation of the motions of a large collection of bodies (e.g., planets or stars) subject to their own mutual gravitational forces. In physics, such a simulation is often called the *$n$-body problem*. Exact analytical solutions are known to exist in only extremely small special cases. Even determining a good numerical solution is relative costly. In order to determine the motion of a single object in the simulation, we need to know the gravitational force induced by the other $n-1$ bodies of the system. In order to compute this force, it would seem that at a minimum we would need $\Omega(n)$ computations per point, for a total of $\Omega(n^2)$ total computations. The question is whether there is a way to do this faster?

What we seek is a structure that allows us to encode the distance information of $\Omega(n^2)$ pairs in a structure of size only $O(n)$. While this may seem to be an impossible task, a clever approximate answer to this question was discovered by Greengard and Rokhlin in the mid 1980's, and forms the basis of a technique called the *fast multipole method*[16] (or FMM for short). We will not discuss the FMM, since it would take us out of the way, but will instead discuss the geometric structure that encodes much of the information that made the FMM such a popular technique.

**Well Separated Pairs:** A set of $n$ points in space defines a set of $\binom{n}{2} = \Theta(n^2)$ distinct pairs. To see how to encode this set approximately, let us return briefly to the $n$-body problem. Suppose that we wish to determine the gravitational effect of a large number of stars in a one galaxy on the stars of distant galaxy. Assuming that the two galaxies are far enough away from each other relative to their respective sizes, the individual influences of the bodies in each galaxy can be aggregated into a single physical force. If there are $n_1$ and $n_2$ points in the respective galaxies, the interactions due to all $n_1 \cdot n_2$ pairs can be well approximated by a single *interaction pair* involving the centers of the two galaxies.

To make this more precise, assume that we are given an $n$-element point set $P$ in $\mathbb{R}^d$, and a separation factor $s > 0$. We say that two disjoint sets of $A$ and $B$ are *$s$-well separated* if the the sets $A$ and $B$ can be enclosed within two Euclidean balls of radius $r$ such that the closest distance between these balls is at least $sr$ (see Fig. 80).

Observe that if a pair of points is $s$-well separated, it is also $s'$-well separated for all $s' < s$. Of course, since any point lies within a (degenerate) ball of radius 0, it follows that a pair of singleton sets, $\{\{a\}, \{b\}\}$, for $a \neq b$, is well-separated for any $s > 0$.

**Well Separated Pair Decomposition:** Okay, distant galaxies are well separated, but if you were given an *arbitrary* set of $n$ points in $\mathbb{R}^d$ (which may not be as nicely clustered as the stars in galaxies) and a fixed separation factor $s > 0$, can you concisely approximate all $\binom{n}{2}$ pairs? We will show that such a decomposition exists, and its size is $O(n)$. The decomposition is called a *well separated pair decomposition*. Of course, we would expect the complexity to depend on $s$ and $d$ as well. The constant factor hidden by the asymptotic notion grows as $O(s^d)$.

Let's make this more formal. Given arbitrary sets $A$ and $B$, define $A \otimes B$ to be the set of all distinct (unordered)

[16] As an indication of how important this algorithm is, it was listed among the top-10 algorithms of the 20th century, along with quicksort, the fast fourier transform, and the simplex algorithm for linear programming.

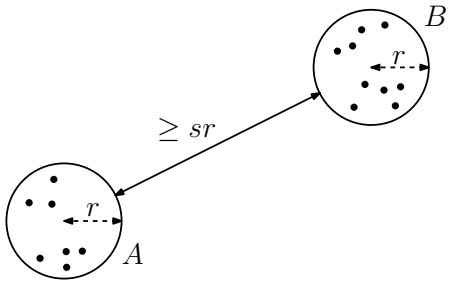



Fig. 80: A well separated pair with separation factor $s$.

pairs from these sets, that is

$$A \otimes B \;=\; \{\{a, b\} \mid a \in A,\; b \in B,\; a \neq b\}\,.$$

Observe that $A \otimes A$ consists of all the $\binom{n}{2}$ distinct pairs of $A$. Given a point set $P$ and separation factor $s > 0$, we define an *$s$-well separated pair decomposition* ($s$-WSPD) to be a collection of pairs of subsets of $P$, denoted $\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}\}$, such that

(1) $A_i, B_i \subseteq P$, for $1 \leq i \leq m$

(2) $A_i \cap B_i = \emptyset$, for $1 \leq i \leq m$

(3) $\bigcup_{i=1}^{m} A_i \otimes B_i = P \otimes P$

(4) $A_i$ and $B_i$ are $s$-well separated, for $1 \leq i \leq m$

Conditions (1)–(3) assert we have a cover of all the unordered pairs of $P$, and (4) asserts that the pairs are well separated. Although these conditions alone do not imply that every unordered pair from $P$ occurs in a unique pair $A_i \otimes B_i$, our construction will have this further property. An example is shown in Fig. 81. (Although there appears to be some sort of hierarchical structure here, note that the pairs are not properly nested within one another.)

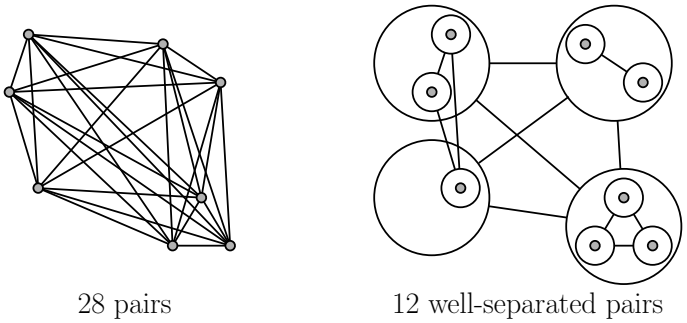



Fig. 81: A point set and a well separated pair decomposition for separation $s = 1$.

Trivially, there exists a WSPD of size $O(n^2)$ by setting the $\{A_i, B_i\}$ pairs to each of the distinct pair singletons of $P$. Our goal is to show that, given an $n$-element point set $P$ in $\mathbb{R}^d$ and any $s > 0$, there exists a $s$-WSPD of size $O(n)$ (where the constant depends on $s$ and $d$). Before doing this, we must make a brief digression to discuss the quadtree data structure, on which our construction is based.

**Quadtrees:** A *quadtree* is a hierarchical subdivision of space into regions, called *cells*, that are hypercubes. The decomposition begins by assuming that the points of $P$ lie within a bounding hypercube. For simplicity we may assume that $P$ has been scaled and translated so it lies within the unit hypercube $[0, 1]^d$.

The initial cell, associated with the *root* of the tree, is the unit hypercube. The following process is then repeated recursively. Consider any unprocessed cell and its associated node $u$ in the current tree. If this cell contains either zero or one point of $P$, then this is declared a leaf node of the quadtree, and the subdivision process terminates for this cell. Otherwise, the cell is subdivided into $2^d$ hypercubes whose side lengths are exactly half that of the original hypercube. For each of these $2^d$ cells we create a node of the tree, which is then made a child of $u$ in the quadtree. (The process is illustrated in Fig. 82. The points are shown in Fig. 82(a), the node structure in Fig. 82(b), and the final tree in Fig. 82(c).)

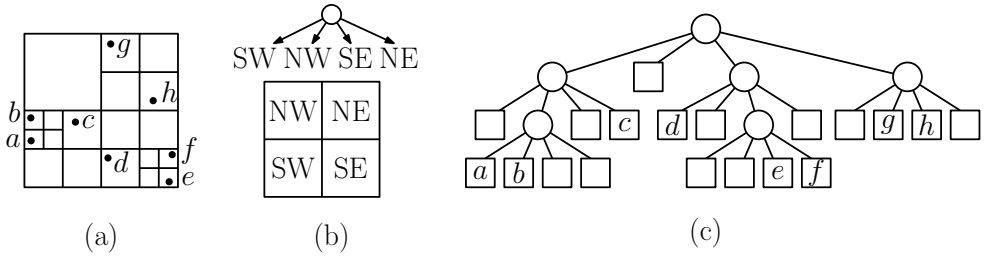



Fig. 82: The quadtree for a set of eight points.

Although in practice, quadtrees as described above tend to be reasonably efficient in fairly small dimensions, there are a number of important issues in their efficient implementation in the worst case. The first is that a quadtree containing $n$ points may have many more than $O(n)$ nodes. The reason is that, if a group of points are extremely close to one another relative to their surroundings, there may be an arbitrarily long *trivial path* in the tree leading to this cluster, in which only one of the $2^d$ children of each node is an internal node (see Fig. 83(a)).

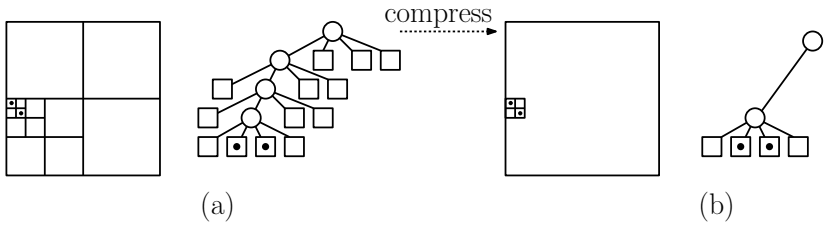



Fig. 83: Compressed quadtree: (a) The original quadtree, (b) after path compression.

This issue is easily remedied by a process called *path compression*. Every such trivial path is compressed into a single link. This link is labeled with the coordinates of the smallest quadtree box that contains the cluster (see Fig. 83(b)). The resulting data structure is called a *compressed quadtree*. Observe that each internal node of the resulting tree separates at least two points into separate subtrees. Thus, there can be no more than $n-1$ internal nodes, and hence the total number of nodes is $O(n)$.

A second issue involves the efficient computation of the quadtree. It is well known that the tree can be computed in time $O(hn)$, where $h$ is the height of the tree. However, even for a compressed quadtree the tree height can be as high as $n$, which would imply an $O(n^2)$ construction time. We will not discuss it here, but it can be shown that in any fixed dimension it is possible to construct the quadtree of an $n$-element point set in $O(n \log n)$ time. (The key is handling uneven splits efficiently. Such splits arise when one child contains almost all of the points, and all the others contain only a small constant number.)

The key facts that we will use about quadtrees below are:

(a) Given an $n$-element point set $P$ in a space of fixed dimension $d$, a compressed quadtree for $P$ of size $O(n)$ can be constructed in $O(n \log n)$ time.

(b) Each internal node has a constant number ($2^d$) children.

(c) The cell associated with each node of the quadtree is a $d$-dimensional hypercube, and as we descend from the parent to a child (in the uncompressed quadtree), the size (side length) of the cells decreases by a factor of 2.

(d) The cells associated with any level of the tree (where tree levels are interpreted relative to the uncompressed tree) are of the same size and all have pairwise disjoint interiors.

An important consequence stemming from (c) and (d) is the following lemma, which provides an upper bound on the number of quadtree disjoint quadtree cells of size at least $x$ that can overlap a ball of radius $r$.

**Packing Lemma:** Consider a ball $b$ of radius $r$ in any fixed dimension $d$, and consider any collection $X$ of pairwise disjoint quadtree cells of side lengths at least $x$ that overlap $b$. Then

$$|X| \leq \left(1 + \left\lceil \frac{2r}{x} \right\rceil\right)^d \leq O\left(\max\left(2, \frac{r}{x}\right)^d\right)$$

**Proof:** We may assume that all the cells of $X$ are of side length exactly equal to $x$, since making cells larger only reduces the number of overlapping cells (see Fig. 84(b)).

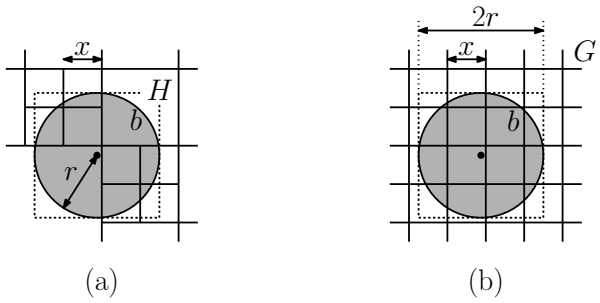



Fig. 84: Proof of the Packing Lemma.

By the nature of a quadtree decomposition, the cells of side length $x$ form a hypercube grid $G$ of side length $x$. Consider a hypercube $H$ of side length $2r$ that encloses $b$ (see Fig. 84). Clearly every cell of $X$ overlaps this hypercube. Along each dimension, the number of cells of $G$ that can overlap an interval of side length $2r$ is at most $1 + \lceil 2r/x \rceil$. Thus, the number of grid cubes of $G$ that overlap $H$ is at most $(1 + \lceil 2r/x \rceil)^d$. If $2r < x$, this quantity is at most $2^d$, and otherwise it is $O((r/x)^d)$.

For the construction of the WSPD, we need to make a small augmentation to the quadtree structure. We wish to associate each node of the tree, both leaves and internal nodes, with a point that lies within its cell (if such a point exists). Given a node $u$, we will call this point $u$'s *representative* and denote this as $\mathrm{rep}(u)$. We do this recursively as follows. If $u$ is a leaf node that contains a point $p$, then $\mathrm{rep}(u) = \{p\}$. If $u$ is a leaf node that contains no point, then $\mathrm{rep}(u) = \emptyset$. Otherwise, if $u$ is an internal node, then it must have at least one child $v$ that is not an empty leaf. (If there are multiple nonempty children, we may select any one.) Set $\mathrm{rep}(u) = \mathrm{rep}(v)$.

Given a node $u$ in the tree, let $P_u$ denote the points that lie within the subtree rooted at $u$. We will assume that each node $u$ is associated with its *level* in the tree, denoted $\mathrm{level}(u)$. Assuming that the original point set lies within a unit hypercube, the side lengths of the cells are of the form $1/2^i$, for $i \geq 0$. We define $\mathrm{level}(u)$ to be $-\log_2 x$, where $x$ is the side length of $u$'s cell. Thus, $\mathrm{level}(u)$ is just the depth of $u$ in the (uncompressed) quadtree, where the root has depth 0. The key feature of level is that $\mathrm{level}(u) \leq \mathrm{level}(v)$ holds if and only if the sidelength of $u$'s cell at least as large as that of $v$'s cell.

**Constructing a WSPD:** We now have the tools needed to to show that, given an $n$-element point set $P$ in $\mathbb{R}^d$ and any $s > 0$, there exists a $s$-WSPD of size $O(s^d n)$, and furthermore, this WSPD can be computed in time that is roughly proportional to its size. In particular, the construction will take $O(n \log n + s^d n)$ time. We will

show that the final WSPD can be encoded in $O(s^d n)$ total space. Under the assumption that $s$ and $d$ are fixed (independent of $n$) then the space is $O(n)$ and the construction time is $O(n \log n)$.

The construction operates as follows. Recall the conditions (1)–(4) given above for a WSPD. We will maintain a collection of sets that satisfy properties (1) and (3), but in general they may violate conditions (2) and (4), since they may not be disjoint and may not be well separated. When the algorithm terminates, all the pairs will be well-separated, and this will imply that they are disjoint. Each set $\{A_i, B_i\}$ of the pair decomposition will be encoded as a pair of nodes $\{u, v\}$ in the quadtree. Implicitly, this pair represents the pairs $P_u \otimes P_v$, that is, the set of pairs generated from all the points descended from $u$ and all the points descended from $v$. This is particularly nice, because it implies that the total storage requirement is proportional to the number of pairs in the decomposition.

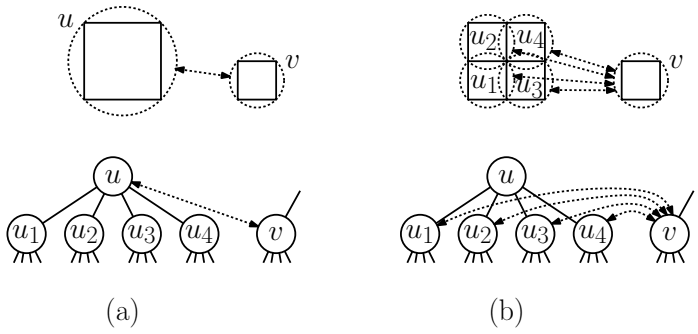



Fig. 85: WSPD recursive decomposition step.

The algorithm is based on a recursive subdivision process. Consider a pair of nodes $\{u, v\}$ that arise in the decomposition process. First, let us assume that $u$'s cell is least as large as $v$'s. That is, $u$'s level number is not greater than $v$'s. Consider the two smallest Euclidean balls of equal radius that enclose $u$'s cell and $v$'s cell (see Fig. 85(a)). If these balls are well separated, then we can report $\{u, v\}$ as (the encoding of) a well separated pair. Otherwise, we subdivide $u$ by considering its children, and apply the procedure recursively to the pairs $\{u_i, v\}$, for each child of $u_i$ of $u$ (see Fig. 85(b)).

A more formal presentation of the algorithm is presented in the following code block. The procedure is called ws-pairs$(u, v, s)$, where $u$ and $v$ are the current nodes of a compressed quadtree for the point set, and $s$ is the separation factor. The procedure returns a set node pairs, encoding the well separated pairs of the WSPD. The initial call is ws-pairs$(u_0, u_0, s)$, where $u_0$ is the root of the compressed quadtree.

Construction of a Well Separated Pair Decomposition

```
ws-pairs(u, v, s) {
    if (rep(u) or rep(v) is empty) return ∅;      // no pairs to report
    else if (u and v are s-well separated)        // (see remark below)
        return {{u, v}};                          // return the WSP {P_u, P_v}
    else {                                        // subdivide
        if (level(u) > level(v)) swap u and v; // swap so that u's cell is at least as large as v's
        Let u_1, ..., u_m denote the children of u;
        return ⋃_{i=1}^{m} ws-pairs(u_i, v, s);   // recurse on children
    }
}
```

How do we test whether two nodes $u$ and $v$ are $s$ well separated? For each internal node, consider the smallest Euclidean balls enclosing the associated quadtree box. For each leaf node, consider a degenerate ball of radius zero that contains the point. In $O(1)$ time, we can determine whether these balls are $s$ well separated. Note that a pair of leaf cells will always pass this test (since the radius is zero), so the algorithm will eventually terminate.

Note that, due to its symmetry, this procedure will generally produce duplicate pairs $\{P_u, P_v\}$ and $\{P_v, P_u\}$. A simple disambiguation rule can be applied to eliminate one of them.

**Analysis:** How many pairs are generated by this recursive procedure? It will simplify our proof to assume that the quadtree is not compressed (and yet it has size $O(n)$). This allows us to assume that the children of each node all have cell sizes that are exactly half the size of their parent's cell. (We leave the general case as an exercise.)

From this assumption, it follows that whenever a call is made to ws-pairs, the sizes of the cells of the two nodes $u$ and $v$ differ by at most a factor of two (because we always split the larger of the two cells). It will also simplify the proof to assume that $s \geq 1$ (if not, replace all occurrences of $s$ below with $\max(s, 1)$).

To evaluate the number of well separated pairs, we will count calls to ws-pairs. We say that a call to ws-pairs is *terminal* if it does not make it to the final "else" clause. Each terminal call generates at most one new well separated pair, and so it suffices to count the number of terminal calls to ws-pairs. In order to do this, we will instead bound the number of nonterminal calls. Each nonterminal call generates at most $2^d$ recursive calls (and this is the only way that terminal calls may arise). Thus, the total number of well separated pairs is at most $2^d$ times the number of nonterminal calls to ws-pairs.

To count the number of nonterminal calls to ws-pairs, we will apply a charging argument to the nodes of the compressed quadtree. Each time we make it to the final "else" clause and split the cell $u$, we assign a charge to the "unsplit" cell $v$. Recall that $u$ is generally the larger of the two, and thus the smaller node receives the charge. We assert that the total number of charges assigned to any node $v$ is $O(s^d)$. Because there are $O(n)$ nodes in the quadtree, the total number of nonterminal calls will be $O(s^d n)$, as desired. Thus, to complete the proof, it suffices to establish this assertion about the charging scheme.

A charge is assessed to node $v$ only if the call is nonterminal, which implies that $u$ and $v$ are not $s$-well separated. Let $x$ denote the side length of $v$'s cell and let $r_v = x\sqrt{d}/2$ denote the radius of the ball enclosing this cell. As mentioned earlier, because we are dealing with an uncompressed quadtree, and the construction always splits the larger cell first, we may assume that $u$'s cell has a side length of either $x$ or $2x$. Therefore, the ball enclosing $u$'s cell is of radius $r_u \leq 2r_v$. Since $u$ and $v$ are not well separated, it follows that the distance between their enclosing balls is at most $s \cdot \max(r_u, r_v) \leq 2sr_v = sx\sqrt{d}$. The centers of their enclosing balls are therefore within distance

$$r_v + r_u + sx\sqrt{d} \;\leq\; \left(\frac{1}{2} + 1 + s\right) x\sqrt{d} \;\leq\; 3sx\sqrt{d} \qquad \text{(since } s \geq 1\text{)},$$

which we denote by $R_v$ (see Fig. 86(a)).

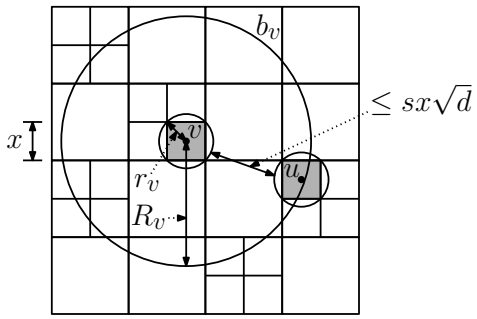



Fig. 86: WSPD analysis.

Let $b_v$ be a Euclidean ball centered at $v$'s cell of radius $R_v$. Summarizing the above discussion, we know that the set of quadtree nodes $u$ that can assess a charge to $v$ have cell sizes of either $x$ or $2x$ and overlap $b_v$. Clearly the cells of side length $x$ are disjoint from one another and the cells of side length $2x$ are disjoint from one another.

Thus, by the Packing Lemma, the total number of nodes that can assess a charge to node $v$ is at most $C$, where

$$\begin{aligned} C &\leq \left(1+\left\lceil\frac{2R_v}{x}\right\rceil\right)^d + \left(1+\left\lceil\frac{2R_v}{2x}\right\rceil\right)^d \leq 2\left(1+\left\lceil\frac{2R_v}{x}\right\rceil\right)^d \\ &\leq 2\left(1+\left\lceil\frac{6sx\sqrt{d}}{x}\right\rceil\right)^d \leq 2(1+6s\sqrt{d})^d \leq O(s^d), \end{aligned}$$

as desired.

Putting this all together, we recall that there are $O(n)$ nodes in the compressed quadtree and $O(s^d)$ charges assigned to any node of the tree, which implies that there are a total of $O(s^d n)$ total nonterminal calls to ws-pairs. As observed earlier, the total number of well separated pairs is larger by a factor of $O(2^d)$, which is just $O(1)$ since $d$ is a constant. Together with the $O(n \log n)$ time to build the quadtree, this gives an overall running time of $O((n \log n) + s^d n)$ and $O(s^d n)$ total well separated pairs. In summary we have the following result.

**Theorem:** Given a point set $P$ in $\mathbb{R}^d$, and a fixed separation factor $s \geq 1$, in $O(n \log n + s^d n)$ time it is possible to build an $s$-WSPD for $P$ consisting of $O(s^d n)$ pairs.

As mentioned earlier, if $0 < s < 1$, then replace $s$ with $\max(s, 1)$. Next time we will consider applications of WSPDs to solving a number of geometric approximation problems.

# Lecture 19: Applications of WSPDs

**Review:** Recall that given a parameter $s > 0$, we say that two sets of $A$ and $B$ are *$s$-well separated* if the sets can be enclosed within two spheres of radius $r$ such that the closest distance between these spheres is at least $sr$. Given a point set $P$ and separation factor $s > 0$, recall that an *$s$-well separated pair decomposition* ($s$-WSPD) is a collection of pairs of subsets of $P$ $\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}\}$ such that

(1) $A_i, B_i \subseteq P$, for $1 \leq i \leq m$
(2) $A_i \cap B_i = \emptyset$, for $1 \leq i \leq m$
(3) $\bigcup_{i=1}^{n} A_i \otimes B_i = P \otimes P$
(4) $A_i$ and $B_i$ are $s$-well separated, for $1 \leq i \leq m$,

where $A \otimes B$ denotes the set of all unordered pairs from $A$ and $B$.

Last time we showed that, given $s \geq 2$, there exists an $s$-WSPD of size $O(s^d n)$, which can be constructed in time $O(n \log n + s^d n)$. (The algorithm works for any $s > 0$, and the $s^d$ term is more accurately stated as $\max(2, s)^d$.) The WSPD is represented as a set of unordered pairs of nodes of a compressed quadtree decomposition of $P$. It is possible to associate each nonempty node $u$ of the compressed quadtree with a *representative point*, denoted $\mathrm{rep}(u)$, chosen from its descendants. We will make use of this fact in some of our constructions below.

Today we discuss a number of applications of WSPDs.

**Approximating the Diameter:** Recall that the *diameter* of a point set is defined to be the maximum distance between any pair of points of the set. (For example, the points $x$ and $y$ in Fig. 87(a) define the diameter.)

The diameter can be computed exactly by brute force in $O(n^2)$ time. For points in the plane, it is possible to compute the diameter[17] in $O(n \log n)$ time. Generalizing this method to higher dimensions results in an $O(n^2)$ running time, which is no better than brute force search.

Using the WSPD construction, we can easily compute an $\varepsilon$-approximation to the diameter of a point set $P$ in linear time. Given $\varepsilon$, we let $s = 4/\varepsilon$ and construct an $s$-WSPD. As mentioned above, each pair $(P_u, P_v)$ in our WSPD construction consists of the points descended from two nodes, $u$ and $v$, in a compressed quadtree.

[17] This is nontrivial, but is not much harder than a homework exercise. In particular, observe that the diameter points must lie on the convex hull. After computing the hull, it is possible to perform a rotating sweep that finds the diameter.

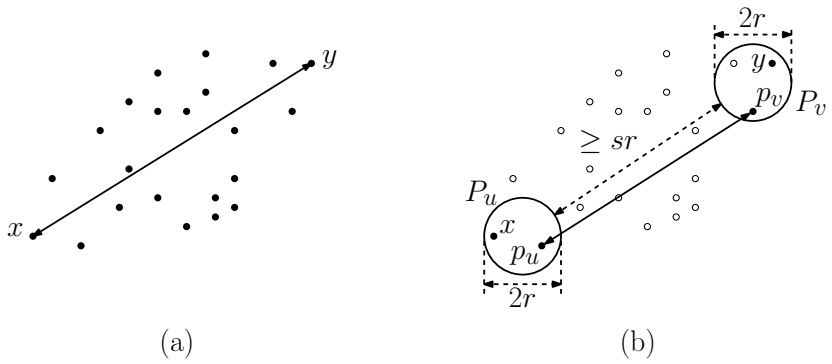



Fig. 87: Approximating the diameter.

Let $p_u = \text{rep}(u)$ and $p_v = \text{rep}(v)$ denote the representative points associated with $u$ and $v$, respectively. For every well separated pair $\{P_u, P_v\}$, we compute the distance $\|p_u p_v\|$ between their representative, and return the largest such distance.

To prove correctness, let $x$ and $y$ be the points of $P$ that realize the diameter. Let $\{P_u, P_v\}$ be the well separated pair containing these points, and let $p_u$ and $p_v$ denote their respective representatives. By definition of well separatedness, we know that $P_u$ and $P_v$ can be enclosed in balls of radius $r$ that are separated by distance at least $sr$ (see Fig. 87(b)). Therefore, by the triangle inequality we have

$$\|xy\| \;\leq\; \|p_u p_v\| + 2r + 2r \;=\; \|p_u p_v\| + 4r.$$

Also, by the WSPD separation properties, we have $\|p_u p_v\| \geq sr$ implying that $r \leq \|p_u p_v\|/s$. Combining these we have

$$\begin{aligned} \|xy\| &\leq \|p_u p_v\| + 4r \;\leq\; \|p_u p_v\| + \frac{4}{s}\|p_u p_v\| \\ &= \left(1 + \frac{4}{s}\right)\|p_u p_v\| \;=\; (1+\varepsilon)\|p_u p_v\|, \end{aligned}$$

Clearly, $\|p_u p_v\| \leq \|xy\|$, and therefore we have

$$\frac{\|xy\|}{1+\varepsilon} \;\leq\; \|p_u p_v\| \;\leq\; \|xy\|,$$

which implies that the output is an $\varepsilon$-approximation. The running time is dominated by the size of the WSPD, which is $O(s^d n) = O(n/\varepsilon^d)$. If we treat $\varepsilon$ as a constant, this is $O(n)$.

**Closest Pair:** The same sort of approach could be used to produce an $\varepsilon$-approximation to the closest pair as well, but surprisingly, there is a much better solution. If we were to generalize the above algorithm, we would first compute an $s$-WSPD for an appropriate value of $s$, and for each well separated pair $\{P_u, P_v\}$ we would compute the distance $\|p_u p_v\|$, where $p_u = \text{rep}(u)$ and $p_v = \text{rep}(v)$, and return the smallest such distance. As before, we would like to argue that (assuming $s$ is chosen properly) this will yield an approximation to the closest pair. It is rather surprising to note that, if $s$ is chosen carefully, this approach yields the *exact* closest pair, not just an approximation.

To see why, consider a point set $P$, let $x$ and $y$ be the closest pair of points and let $p_u$ and $p_v$ be the representatives from their associated well separated pair. If it were the case that $x = p_u$ and $y = p_v$, then the representative-based distance would be exact. Suppose therefore that either $x \neq p_u$ or $y \neq p_v$. But wait! If the separation factor is high enough, this would imply that either $\|xp_u\| < \|xy\|$ or $\|yp_v\| < \|xy\|$, either of which contradicts the fact that $x$ and $y$ are the closest pair.

To make this more formal, let us assume that $\{x, y\}$ is the closest pair and that $s > 2$. We know that $P_u$ and $P_v$ lie within balls of radius $r$ that are separated by a distance of at least $sr > 2r$. If $p_u \neq x$, then we have

$$\|p_u x\| \;\leq\; 2r \;<\; sr \;\leq\; \|xy\|,$$

yielding a contradiction. Therefore $p_u = \text{rep}(u) = x$. By a symmetrical argument $p_v = \text{rep}(v) = y$. Since the representative was chosen arbitrarily, it follows that the $P_u = \{x\}$ and $P_v = \{y\}$. Therefore, the closest representatives are in fact, the *exact* closest pair.

Since $s$ can be chosen to be arbitrarily close to 2, the running time is $O(n \log n + 2^d n) = O(n \log n)$, since we assume that $d$ is a constant. Although this is not a real improvement over our existing closest-pair algorithm, it is interesting to note that there is yet another way to solve this problem.

**Spanner Graphs:** Recall that a set $P$ of $n$ points in $\mathbb{R}^d$ defines a complete weighted graph, called the *Euclidean graph*, in which each point is a vertex, and every pair of vertices is connected by an edge whose weight is the Euclidean distance between these points. This graph is *dense*, meaning that it has $\Theta(n^2)$ edges. It would be nice to have a *sparse* graph having only $O(n)$ edges that approximates the Euclidean graph in some sense.

One such notion is to approximate the distances (length of the shortest path) between all pairs of vertices. A subgraph of a graph that approximates all shortest paths is called a *spanner*. In the geometric context, suppose that we are given a set $P$ and a parameter $t \geq 1$, called the *stretch factor*. We define a *$t$-spanner* to be a weighted graph $G$ whose vertex set is $P$ and, given any pair of points $x, y \in P$ we have

$$\|xy\| \;\leq\; \delta_G(x, y) \;\leq\; t \cdot \|xy\|,$$

where $\delta_G(x, y)$ denotes the length of the shortest path between $x$ and $y$ in $G$.

**WSPD-based Spanner Construction:** Do sparse geometric spanners exist? Remarkably, we have actually already seen one. It can be proved that the planar Delaunay triangulation is a $t$-spanner, for some $t$, where $1.5932 \leq t \leq 1.998$. The tightest value of $t$ is not known.[18]

There are many different ways of building sparse spanners. Here we will discuss a straightforward method based on a WSPD of the point set. The idea is to create one edge for each well-separated pair. More formally, suppose that we are given a point set $P$ and stretch factor $t > 1$. We begin by computing a WSPD for an appropriate separation factor $s$ depending on $t$. (We will prove later that the separation value $s = 4(t+1)/(t-1)$ will do the job). For each well-separated pair $\{P_u, P_v\}$ associated with the nodes $u$ and $v$ of the quadtree, let $p_u = \text{rep}(u)$ and let $p_v = \text{rep}(v)$. Add the undirected edge $\{p_u, p_v\}$ to our graph. Let $G$ be the resulting undirected weighted graph (see Fig. 88). We claim that $G$ is the desired spanner. Clearly the number of edges of $G$ is equal to the number of well-separated pairs, which is $O(s^d n)$, and can be built in the same $O(n \log n + s^d n)$ running time as the WSPD construction.

**Correctness:** To establish the correctness of our spanner construction algorithm, it suffices to show that for all pairs $x, y \in P$, we have

$$\|xy\| \;\leq\; \delta_G(x, y) \;\leq\; t \cdot \|xy\|.$$

Clearly, the first inequality holds trivially, because (by the triangle inequality) no path in any graph can be shorter than the distance between the two points. To prove the second inequality, we apply an induction based on the number of edges of the shortest path in the spanner.

For the basis case, observe that, if $x$ and $y$ are joined by an edge in $G$, then clearly $\delta_G(x, y) = \|xy\| < t \cdot \|xy\|$ for all $t > 1$.

If, on the other hand, there is no direct edge between $x$ and $y$, we know that $x$ and $y$ must lie in some well-separated pair $\{P_u, P_v\}$ defined by the pair of nodes $\{u, v\}$ in the quadtree. let $p_u = \text{rep}(u)$ and $p_v = \text{rep}(v)$ be

[18] The lower bound of $1.5932$ appears in "Toward the Tight Bound of the Stretch Factor of Delaunay Triangulations," by G. Xia and L. Zhang, *Proc. CCCG*, 2011. The upper bound of $1.998$ appears in "Improved Upper Bound on the Stretch Factor of Delaunay Triangulations," by G. Xia, *Proc. SoCG*, 2011.

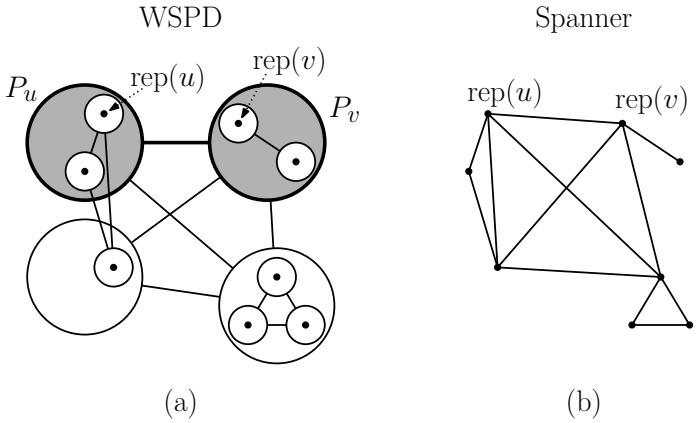



Fig. 88: A WSPD and its associated spanner.

the respective representative representative. (It might be that $p_u = x$ or $p_v = y$, but not both.) Let us consider the length of the path from $x$ to $p_u$ to $p_v$ to $y$. Since the edge $\{p_u, p_v\}$ is in the graph, we have

$$\begin{aligned}\delta_G(x,y) &\leq \delta_G(x,p_u) + \delta_G(p_u,p_v) + \delta_G(p_v,y)\\ &\leq \delta_G(x,p_u) + \|p_u p_v\| + \delta_G(p_v,y).\end{aligned}$$

(See Fig. 89.)

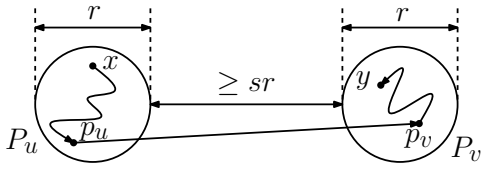



Fig. 89: Proof of the spanner bound.

Since the paths from $x$ to $p_u$ and $p_v$ to $y$ are subpaths, and hence shorter than the overall path, we may apply the induction hypothesis, which yields $\delta_G(x,p_u) \leq t\|xp_u\|$ and $\delta_G(p_v,y) \leq t\|p_v y\|$, yielding

$$\delta_G(x,y) \;\leq\; t(\|xp_u\| + \|p_v y\|) + \|p_u p_v\|. \tag{1}$$

Let $s$ denote the separation factor for the WSPD. Since $P_u$ and $P_v$ are $s$-well separated, we know that each of these point sets can be enclosed within a ball of radius $r$ such that the two balls are separated by distance at least $sr$. Thus, we have $\max(\|xp_u\|, \|p_v y\|) \leq 2r$, and $\|xy\| \geq sr$. From the second inequality we have $r \leq \|xy\|/s$. By the triangle inequality, we have

$$\|p_u p_v\| \;\leq\; \|p_u x\| + \|xy\| + \|yp_v\| \;\leq\; 2r + \|xy\| + 2r \;\leq\; 4r + \|xy\|.$$

Combining these observations with Eq. (1) we obtain

$$\delta_G(x,y) \;\leq\; t(2r + 2r) + (4r + \|xy\|) \;\leq\; 4r(t+1) + \|xy\|.$$

From the fact that $r \leq \|xy\|/s$ we have

$$\delta_G(x,y) \;\leq\; \frac{4(t+1)}{s}\|xy\| + \|xy\| \;\leq\; \left(1 + \frac{4(t+1)}{s}\right)\|xy\|.$$

To complete the proof, observe that it suffices to select $s$ so that $1 + 4(t+1)/s \leq t$. We easily see that this is true if $s$ is chosen so that

$$s \;=\; 4\left(\frac{t+1}{t-1}\right).$$

Since we assume that $t > 1$, this is possible for any $t$. Thus, substituting this value of $s$, we have

$$\delta_G(x,y) \;\leq\; \left(1 + \frac{4(t+1)}{4(t+1)/(t-1)}\right)\|xy\| \;=\; (1+(t-1))\|xy\| \;=\; t\cdot\|xy\|,$$

which completes the correcness proof.

The number of edges in the spanner is $O(s^d n)$. Since spanners are most interesting for small stretch factors, let us assume that $t \leq 2$. If we express $t$ as $t = 1+\varepsilon$ for $\varepsilon \leq 1$, we see that the size of the spanner is

$$O(s^d n) \;=\; O\left(\left(4\frac{(1+\varepsilon)+1}{(1+\varepsilon)-1}\right)^d n\right) \;\leq\; O\left(\left(\frac{12}{\varepsilon}\right)^d n\right) \;=\; O\left(\frac{n}{\varepsilon^d}\right).$$

In conclusion, we have the following theorem:

**Theorem:** Given a point set $P$ in $\mathbb{R}^d$ and $\varepsilon > 0$, a $(1+\varepsilon)$-spanner for $P$ containing $O(n/\varepsilon^d)$ edges can be computed in time $O(n\log n + n/\varepsilon^d)$.

**Approximating the Euclidean MST:** We will now show that with the above spanner result, we can compute an $\varepsilon$-approximation to the minimum spanning tree. Suppose we are given a set $P$ of $n$ points in $\mathbb{R}^d$, and we wish to compute the Euclidean minimum spanning tree (MST) of $P$. Given a graph with $v$ vertices and $e$ edges, it is well known that the MST can be computed in time $O(e + v\log v)$. It follows that we can compute the MST of a set of points in any dimension by first constructing the Euclidean graph and then computing its MST, which takes $O(n^2)$ time. To compute the approximation to the MST, we first construct a $(1+\varepsilon)$-spanner, call it $G$, and then compute and return the MST of $G$ (see Fig. 90). This approach has an overall running time of $O(n\log n + s^d n)$.

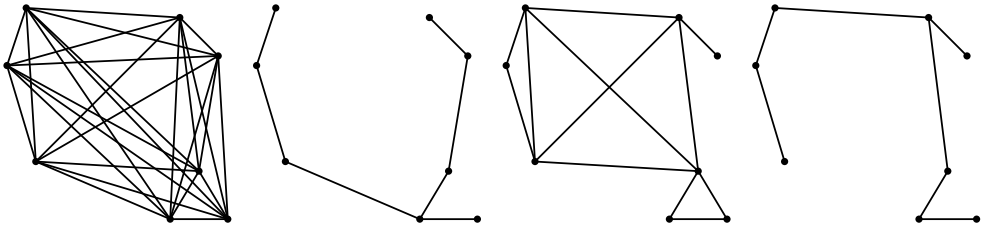



Fig. 90: Approximating the Euclidean MST.

To see why this works, for any pair of points $\{x, y\}$, let $w(x,y) = \|xy\|$ denote the weight of the edge between them in the complete Euclidean graph. Let $T$ denote the edges of the Euclidean minimum weight spanning tree, and $w(T)$ denote the total weight of its edges. For each edge $\{x,y\} \in T$, let $\pi_G(x,y)$ denote the shortest path (as a set of edges) between $x$ and $y$ in the spanner, $G$. Since $G$ is a spanner, we have

$$w(\pi_G(x,y)) \;=\; \delta_G(x,y) \;\leq\; (1+\varepsilon)\|xy\|.$$

Now, consider the subgraph $G' \subseteq G$ formed by taking the union of all the edges of $\pi_G(x,y)$ for all $\{x,y\} \in T$. That is, $G$ and $G'$ have the same vertices, but each edge of the MST is replaced by its spanner path. Clearly, $G'$

is connected (but it may not be a tree). We can bound the weight of $G'$ in terms of the weight of the Euclidean MST:

$$\begin{aligned} w(G') &= \sum_{\{x,y\}\in T} w(\pi_G(x,y)) \le \sum_{\{x,y\}\in T} (1+\varepsilon)\|xy\| \\ &= (1+\varepsilon)\sum_{\{x,y\}\in T} \|xy\| = (1+\varepsilon)w(T). \end{aligned}$$

However, because $G$ and $G'$ share the same vertices, and the edge set of $G'$ is a subset of the edge set of $G$, it follows that $w(\mathrm{MST}(G) \le w(\mathrm{MST}(G'))$. (To see this, observe that if you have fewer edges from which to form the MST, you may generally be forced to use edges of higher weight to connect all the vertices.) Combining everything we have

$$w(\mathrm{MST}(G)) \le w(\mathrm{MST}(G')) \le w(G') \le (1+\varepsilon)w(T),$$

yielding the desired approximation bound.

# Lecture 20: Coresets for Directional Width

**Coresets:** One of the issues that arises when dealing with very large geometric data sets, especially in multi-dimensional spaces, is that the computational complexity of many geometric optimization problems grows so rapidly that it is not feasible to solve the problem exactly. In the previous lecture, we saw how the concept of a well-separated pair decomposition can be used to approximate a quadratic number of objects (all pairs) by a smaller linear number of objects (the well separated pairs). Another approach for simplifying large data sets is to apply some sort of sampling. The idea is as follows. Rather than solve an optimization problem on some (large) set $P \subset \mathbb{R}^d$, we will extract a relatively small subset $Q \subseteq P$, and then solve the problem exactly on $Q$.

The question arises, how should the set $Q$ be selected and what properties should it have in order to guarantee a certain degree of accuracy? Consider the following example from geometric statistics. A set $P$ of $n$ points in $\mathbb{R}^2$ defines $O(n^3)$ triangles whose vertices are drawn from $P$. Suppose that you wanted to estimate the *average* area of these triangles. You could solve this naively in $O(n^3)$ time, but the central limit theorem from probability theory states that the average of a sufficiently large random sample will be a reasonable estimate to the average. This suggests that a good way to select $Q$ is to take a random sample of $P$.

Note, however, that random sampling is not always the best approach. For example, suppose that you wanted to approximate the minimum enclosing ball (MEB) for a point set $P$ (see Fig. 91(a)). A random subset may result in a ball that is much smaller than the MEB. This will happen, for example, if $P$ is densely clustered but with a small number of distant outlying points (see Fig. 91(b)). In such a case, the sampling method should favor points that are near the extremes of $P$'s distribution (see Fig. 91(c)).

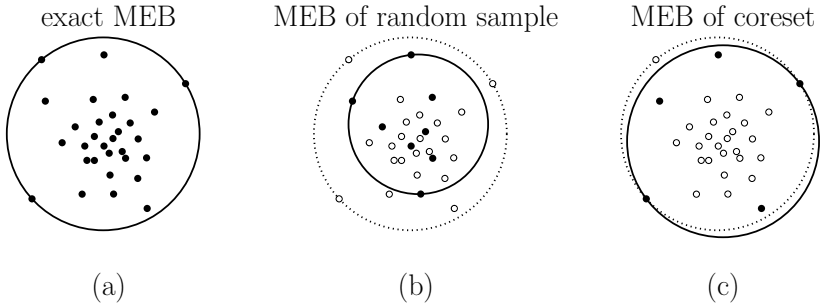



Fig. 91: Approximating the minimum enclosing ball (MEB): (a) exact solution, (b) MEB of a random sample, (c) MEB of a possible coreset.

Abstractly, consider any optimization problem on point sets. For a point set $P$, let $f^*(P)$ denote the value of the optimal solution. Given $\varepsilon > 0$, we say that subset $Q \subseteq P$ is an *$\varepsilon$-coreset* for this problem if, the relative error committed by solving the problem on $Q$ is at most $\varepsilon$, that is:

$$1 - \varepsilon \;\leq\; \frac{f^*(Q)}{f^*(P)} \;\leq\; 1 + \varepsilon.$$

For a given optimization problem, the relevant questions are: (1) does a small coreset exist? (2) if so, how large must the coreset be to guarantee a given degree of accuracy? (3) how quickly can such a coreset be computed? Ideally, the coreset should be significantly smaller than $n$. For many optimization problems, the coreset size is actually independent of $n$ (but does depend on $\varepsilon$).

In this lecture, we will present algorithms for computing coresets for a problem called the *directional width*. This problem can be viewed as a way of approximating the convex hull of a point set.

**Directional Width and Coresets:** Consider a set $P$ of points in real $d$-dimensional space $\mathbb{R}^d$. Given vectors $\vec{u}, \vec{v} \in \mathbb{R}^d$, let $(\vec{v} \cdot \vec{u})$ denote the standard inner (dot) product in $\mathbb{R}^d$. From basic linear algebra we know that, given any vector $\vec{u}$ of unit length, for any vector $\vec{v}$, $(\vec{v} \cdot \vec{u})$ is the length of $\vec{v}$'s orthogonal projection onto $\vec{u}$. The *directional width* of $P$ in direction $\vec{u}$ is defined to be the minimum distance between two hyperplanes, both orthogonal to $\vec{u}$, that has $P$ "sandwiched" between them. More formally, if we think of each point $p \in P$ as a vector $\vec{p} \in \mathbb{R}^d$, the directional width can be formally defined to be

$$W_P(\vec{u}) \;=\; \max_{p \in P}(\vec{p} \cdot \vec{u}) - \min_{p \in P}(\vec{p} \cdot \vec{u})$$

(see Fig. 92(a)). Note that this is a signed quantity, but we are typically interested only in its magnitude.

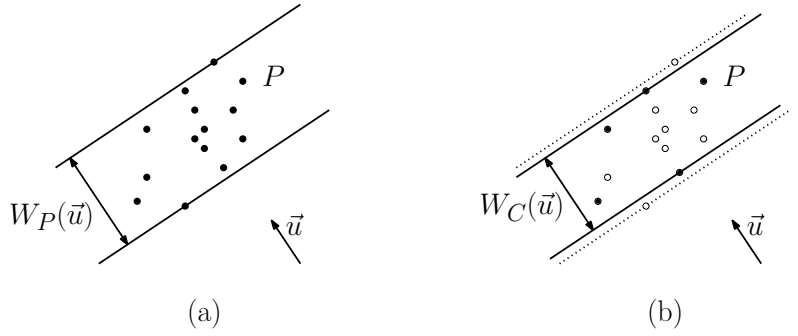



Fig. 92: Directional width and coresets. In (b) the points of $C$ are shown as black points.

The directional width has a number of nice properties. For example, it is invariant under translation and it scales linearly if $P$ is uniformly scaled.

Note that the only points of $P$ that are relevant to the directional width are the points of the convex hull of $P$, that is, conv$(P)$. Although we can compute conv$(P)$ in $O(n \log n)$ time in $\mathbb{R}^2$, the combinatorial complexity of the hull may be as large as $\Omega(n^{\lfloor d/2 \rfloor})$ in $\mathbb{R}^d$. We seek a more space efficient solution, but we will allow for an approximation error.

Given $0 < \varepsilon < 1$, we say that a subset $C \subseteq P$ is an *$\varepsilon$-coreset for directional width* if, for any unit vector $\vec{u}$,

$$W_C(u) \;\geq\; (1 - \varepsilon) W_P(u).$$

That is, the perpendicular width of the minimum slab orthogonal to $\vec{u}$ for $Q$ is smaller than that of $P$ by a factor of only $(1 - \varepsilon)$ (see Fig. 92(b)). We will show that, given an $n$-element point set $P$ in $\mathbb{R}^d$, it is possible to compute an $\varepsilon$-coreset for directional width of size $O(1/\varepsilon^{(d-1)/2})$. For the rest of this lecture, the term "coreset" will mean "coreset for directional width," and if not specified, the approximation parameter is $\varepsilon$.

Note that coresets combine nicely. In particular, it is easy to prove the following:

**Chain Property:** If $X$ is an $\varepsilon$-coreset of $Y$ and $Y$ is an $\varepsilon'$-coreset of $Z$, then $X$ is an $(\varepsilon + \varepsilon')$ coreset of $Z$.
**Union Property:** If $X$ is an $\varepsilon$-coreset of $P$ and $X'$ is an $\varepsilon$-coreset of $P'$, then $X \cup X'$ is an $\varepsilon$-coreset of $P \cup P'$.

**Quick-and-Dirty Construction:** Let's begin by considering a very simple, but not very efficient, coreset for directional widths. We will apply the a utility lemma, which states that it is possible to reduce the problem of computing a coreset for directional widths to one in which the convex hull of the point set is "fat".

Before giving the lemma, let us give a definition. Let $B$ denote a $d$-dimensional unit ball, and for any scalar $\lambda > 0$, let $\lambda B$ be a scaled copy of $B$ by a factor $\lambda$. Given $\alpha \leq 1$, we say that a convex body $K$ in $\mathbb{R}^d$ is $\alpha$-*fat* if there exist two positive scalars $\lambda_1$ and $\lambda_2$, such that $K$ lies within a translate of $\lambda_2 B$, $K$ contains a translate of $\lambda_1 B$, and $\lambda_1/\lambda_2 = \alpha$ (see Fig. 93(a)). Observe that any Euclidean ball is 1-fat. A line segment is 0-fat. It is easy to verify that a $d$-dimensional hypercube is $(1/\sqrt{d})$-fat. We say that a point set $P$ is $\alpha$-fat if its convex hull, conv$(P)$, is $\alpha$-fat (see Fig. 93(b)).

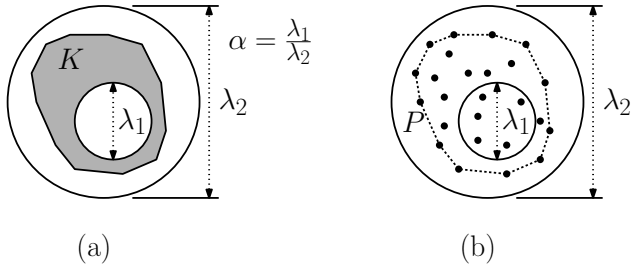



Fig. 93: The definition of $\alpha$-fatness for: (a) a convex body $K$ and (b) for a point set $P$.

**Lemma 1:** Given an $n$-element point set $P \subset \mathbb{R}^d$, there exists a linear transformation $T$ such that $TP$ is contained within a unit ball and is $\alpha$-fat, where $\alpha$ is a constant depending only on the dimension. Also, a subset $C \subseteq P$ is a directional-width $\varepsilon$-coreset for $P$ if and only if $TC$ is a directional-width $\varepsilon$-coreset. The transformation $T$ can be computed in $O(n)$ time.

**Proof:** (Sketch) Let $K = \text{conv}(P)$. If computation time is not an issue, it is possible to use a famous fact from the theory of convexity. This fact, called *John's Theorem*, states that if $E$ is a maximum volume ellipsoid contained within $K$, then (subject to a suitable translation) $K$ is contained within $dE$, where $dE$ denotes a scaled copy of $E$ by a factor of $d$ (the dimension). Take $T$ to be the linear transformation that stretches $dE$ into a unit ball (see Fig. 94(a)–(b)). (For example, through an appropriate rotation, we can align the principal axes of $E$ with the coordinate axes and then apply a scaling factor to each of the coordinate axes so that each principal axis of is of length $1/d$. The expanded ellipse will be mapped to a unit ball, and we have $\alpha = 1/d$.)

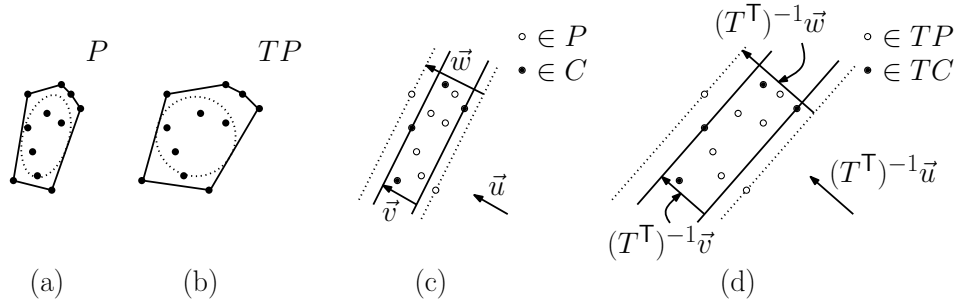



Fig. 94: Proof of Lemma 1.

The resulting transformation will not generally preserve directional widths, but for our purposes, it suffices that it preserves the *ratios* of directional widths. (More formally, through basic linear algebra, we can show

that for any unit vector $\vec{u}$ the ratio of the widths two sets $C$ and $P$ along $\vec{u}$ is equal to the ratio of the widths of $TC$ and $TP$ relative to the transformed direction $(T^{\mathsf{T}})^{-1}\vec{u}$ (see Fig. 94(c)–(d)). We will omit the simple proof.) The maximum ratio of directional widths (over all unit vectors $\vec{u}$) is therefore preserved, which implies that the coreset condition is also preserved.

To obtain the $O(n)$ running time, it suffices to compute a constant factor approximation to the John ellipsoid. Such a construction has been given by Barequet and Har-Peled.

Armed with the above lemma, we may proceed as follows to compute our quick-and-dirty coreset. First, we assume that $P$ has been fattened, by the above procedure. $P$ is contained within a unit ball $B$ and that conv$P$ contains a translate of the shrunken ball $\alpha B$. Because $P$ is sandwiched between $\alpha B$ and $B$, it follows that the width of $P$ along any direction is at least $2\alpha$ and at most 2. Since no width is smaller than $2\alpha$, in order to achieve a relative error of $\varepsilon$, it suffices to approximate any width to an absolute error of at most $2\alpha\varepsilon$, which we will denote by $\varepsilon'$.

Let $H = [-1, +1]^d$ be a hypercube that contains $B$. Subdivide $H$ into a grid of hypercubes whose diameters are at most $\varepsilon'/2$ (see Fig. 95(a)). Each edge of $H$ will be subdivided into $O(1/\varepsilon') = O(1/\varepsilon)$ intervals. Thus, the total number of hypercubes in the grid is $O(1/\varepsilon^d)$. For each such hypercube, if it contains a point of $P$, add any one such point to $C$. The resulting number of points of $C$ cannot exceed the number of hypercubes, which is $O(1/\varepsilon^d)$.

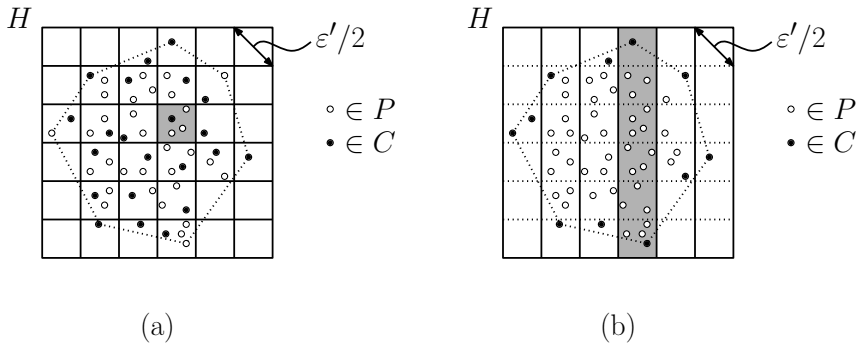



Fig. 95: The quick-and-dirty coreset construction: (a) of size $O(1/\varepsilon^d)$ and (b) the improved construction of of size $O(1/\varepsilon^{d-1})$.

We can do this efficiently by hashing each point according to the index of the hypercube it lies within. We retain one point from each nonempty hash bucket. This can be done in $O(n)$ time.

**Theorem 2:** Given an $n$-element point set $P \subset \mathbb{R}^d$, in $O(n)$ time it is possible to compute an $\varepsilon$-coreset of size $O(1/\varepsilon^d)$ for directional width.

**Proof:** It suffices the establish the correctness of the above construction. For each point $p \in P$ there is a point of $C$ within distance $\varepsilon'/2$. Therefore, given any direction $\vec{u}$, if $p_1$ and $p_2$ are the two points of $P$ that determine the extremes of the width along this direction, then we can find two points $q_1$ and $q_2$ in $C$ that are within distance $\varepsilon'/2$ of each, implying that the resulting width is within (absolute) distance $2(\varepsilon'/2) = \varepsilon'$ of the true width. As established above, since the width in any direction is at least $2\alpha$, the relative error is at most

$$\frac{\varepsilon'}{2\alpha} = \frac{2\alpha\varepsilon}{2\alpha} = \varepsilon,$$

as desired.

**Improved Construction:** It is possible make a small improvement in the size of the quick-and-dirty coreset. Observe from Fig. 95(a) that we may select many points from the interior of conv$(P)$, which clearly can play no useful role in the coreset construction. Rather than partition $H$ into small hypercubes, we can instead partition the upper

$(d-1)$-dimensional facet of $H$ into $O(1/\varepsilon^{d-1})$ cubes of diameter $\varepsilon'/2$, and then extrude each into a "column" that passes through $H$. For each column, take the highest and lowest point to add to $C$ (see Fig. 95(b)). We leave it as an easy geometric exercise to show that this set of points suffices.

**Smarter Coreset Construction:** The above coreset construction has the advantage of simplicity, but, as shall see next, it is possible to construct much smaller coresets for directional widths. We will reduce the size from $O(1/\varepsilon^{d-1})$ to $O(1/\varepsilon^{(d-1)/2})$, thus reducing the exponential dependency by half.

Our general approach will be similar to the one taken above. First, we will assume that the point set $P$ has been "fattened" so that it lies within a unit ball, and its convex hull contains a ball of radius at least $\alpha$, where $\alpha \leq 1$ is a constant depending on dimension. As observed earlier, since the width of $P$ in any direction is at least $2\alpha$, in order to achieve a relative error of $\varepsilon$, it suffices to compute a coreset whose absolute difference in width along any direction is at most $\varepsilon' = 2\alpha\varepsilon$.

A natural approach to solving this problem would involve uniformly sampling a large number (depending on $\varepsilon$) of different directions $\vec{u}$, computing the two extreme points that maximize and minimize the inner product with $\vec{u}$ and taking these to be the elements of $C$. It is noteworthy, that this construction does not result in the best solution. In particular, it can be shown that the angular distance between neighboring directions may need to be as small as $\varepsilon$, and this would lead to $O(1/\varepsilon^{d-1})$ sampled directions, which is asymptotically the same as the (small improvement to) the quick-and-dirty method. The approach that we will take is similar in spirit, but the sampling process will be based not on computing extreme points but instead on computing nearest neighbors.

We proceed as follows. Recall that $P$ is contained within a unit ball $B$. Let $S$ denote the sphere of radius 2 that is concentric with $B$. (The expansion factor 2 is not critical. Any constant factor expansion works, but the constants in the analysis will need to be adjusted.) Let $\delta = \sqrt{\varepsilon\alpha/4}$. (The source of this "magic number" will become apparent later.) On the sphere $S$, construct a $\delta$-dense set of points, denoted $Q$ (see Fig. 96). This means that, for every point on $S$, there is a point of $Q$ within distance $\delta$. The surface area of $S$ is constant, and since the sphere is a manifold of dimension $d-1$, it follows that $|Q| = O(1/\delta^{d-1}) = O(1/\varepsilon^{(d-1)/2})$. For each point of $Q$, compute its nearest neighbor in $P$.[19] Let $C$ denote the resulting subset of $P$. We will show that $C$ is the desired coreset.

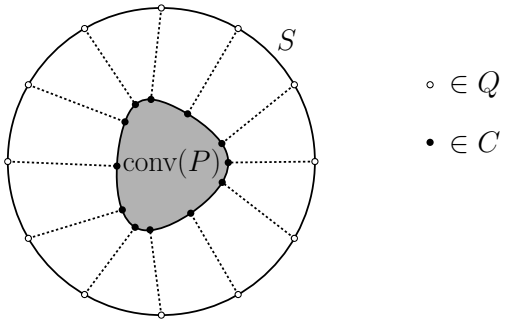



Fig. 96: Smarter coreset construction. (Technically, the points of $Q$ are connected to the closest point of $P$, not conv$(P)$.)

In the figure we have connected each point of $Q$ to its closest point on conv$(P)$. It is a bit easier to conceptualize the construction as sampling points from conv$(P)$. (Recall that the coreset definition requires that the coreset is a subset of $P$.) There are a couple of aspects of the construction that are noteworthy. First, observe that the construction tends to sample points of $P$ that lie close to regions where the curvature of $P$'s convex hull is higher (see Fig. 96). This is useful, because areas of high curvature need more points to approximate them well.

[19] This clever construction was discovered in the context of polytope approximation independently by E. M. Bronstein and L. D. Ivanov, "The approximation of convex sets by polyedra," *Siber. Math J.*, 16, 1976, 852–853 and R. Dudley, "Metric entropy of some classes of sets with differentiable boundaries," *J. Appr. Th.*, 10, 1974, 227–236.

Also, because the points on $S$ are chosen to be $\delta$-dense on $S$, it can be shown that they will be at least this dense on $P$'s convex hull. Before presenting the proof of correctness, we will prove a technical lemma.

**Lemma 2:** Let $0 < \delta \leq 1/2$, and let $q, q' \in \mathbb{R}^d$ such that $\|q\| \geq 1$ and $\|q' - q\| \leq \delta$ (see Fig. 97). Let $B(q')$ be a ball centered at $q'$ of radius $\|q'\|$. Let $\vec{u}$ be a unit length vector from the origin to $q$. Then

$$\min_{p' \in B(q')} (p' \cdot \vec{u}) \geq -\delta^2.$$

**Proof:** (Sketch) We will prove the lemma in $\mathbb{R}^2$ and leave the generalization to $\mathbb{R}^d$ as an exercise. Let $o$ denote the origin, and let $\ell = \|q\|$ be the distance from $q$ to the origin. Let us assume (through a suitable rotation) that $\vec{u}$ is aligned with the $x$-coordinate axis. The quantity $(p' \cdot \vec{u})$ is the length of the projection of $p'$ onto the $x$-axis, that is, it is just the $x$-coordinate of $p'$. We want to show that this coordinate cannot be smaller than $-\delta^2$.

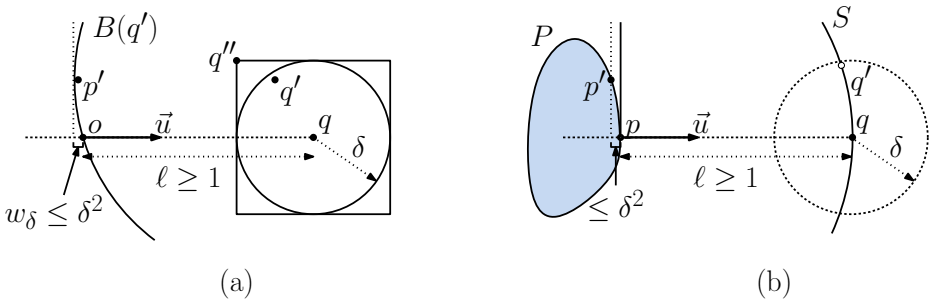



Fig. 97: Analysis of the coreset construction.

We will prove a slightly stronger version of the above. In particular, let us assume that $q'$ is contained within a square of side length $2\delta$ centered at $q$. This suffices because this square contains all points that lie within distance $\delta$ of $q$. Observe that the boundary of the ball $B(q')$ passes through the origin. We wish to bound how far such a ball might protrude over the $(-x)$-axis. Its easy to see that worst case arises when $q'$ is placed in the upper left corner of the square (see Fig. 97(a)). Call this point $q''$.

The distance between $q''$ and the origin is $\sqrt{(\ell - \delta)^2 + \delta^2}$. Therefore, the amount by which the ball of radius $\|q''\|$ centered at $\|q''\|$ may protrude over the $(-x)$-axis is at most

$$\sqrt{(\ell - \delta)^2 + \delta^2} - (\ell - \delta)$$

which we will denote by $w_\delta$. Since $p$ lies in this ball, to complete the proof it suffices to show that $w_\delta \leq \delta^2$. To simplify this, we multiply by a fraction whose numerator and denominator are both $\sqrt{(\ell - \delta)^2 + \delta^2} + (\ell - \delta)$. It is easily verified that $\sqrt{(\ell - \delta)^2 + \delta^2} \geq \ell - \delta$. Using this and the fact that $\ell \geq \delta$, we have

$$\begin{aligned} w_\delta &= \frac{((\ell - \delta)^2 + \delta^2) - (\ell - \delta)^2}{\sqrt{(\ell - \delta)^2 + \delta^2} + (\ell - \delta)} \leq \frac{2(\ell - \delta)\delta + \delta^2}{(\ell - \delta) + (\ell - \delta)} = \frac{2\ell\delta - \delta^2}{2(\ell - \delta)} \\ &\leq \frac{\delta^2}{2(\ell - \delta)} \leq \delta^2, \end{aligned}$$

as desired.

To establish the correctness of the construction, consider any direction $\vec{u}$. Let $p \in P$ be the point that maximizes $(p \cdot \vec{u})$. We will show that there is a point $p' \in C$ such that $(p \cdot \vec{u}) - (p' \cdot \vec{u}) \leq \varepsilon'/2$. In particular, let us translate the coordinate system so that $p$ is at the origin, and let us rotate space so that $\vec{u}$ is horizontal (see Fig. 97(b)). Let $q$ be the point at which the extension of $\vec{u}$ intersects the sphere $S$. By our construction, there exists a point

$q' \in Q$ that lies within distance $\delta$ of $q$, that is $\|q' - q\| \leq \delta$. Let $p'$ be the nearest neighbor of $P$ to $q'$. Again, by our construction $p'$ is in the coreset. Since $q$ lies on a sphere of radius 2 and $P$ is contained within the unit ball, it follows that $\|q\| \geq 1$. Thus, we satisfy the conditions of Lemma 2. Therefore, $(p' \cdot \vec{u}) \geq -\delta^2 = \varepsilon\alpha/4 \leq \varepsilon'/2$. Thus, the absolute error in the inner product is at most $\varepsilon'/2$, and hence (combining both the maximum and minimum sides) the total absolute error is at most $\varepsilon'$. By the remarks made earlier, this implies that the total relative error is $\varepsilon$, as desired.

# Lecture 21: Geometric Basics

**Geometry Basics:** As we go through the semester, we will introduce much of the geometric facts and computational primitives that we will be needing. For the most part, we will assume that any geometric primitive involving a constant number of elements of constant complexity can be computed in $O(1)$ time, and we will not concern ourselves with how this computation is done. (For example, given three non-collinear points in the plane, compute the unique circle passing through these points.) Nonetheless, for a bit of completeness, let us begin with a quick review of the basic elements of affine and Euclidean geometry.

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

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

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

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

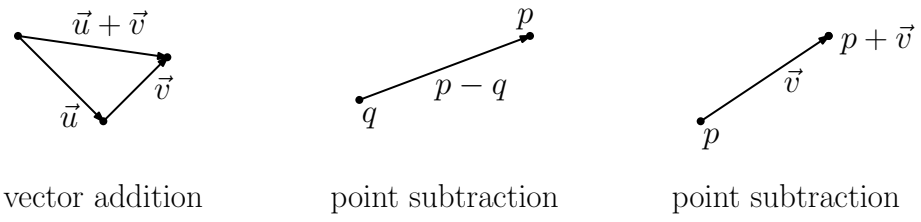



Fig. 98: Affine operations.

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

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

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

Note that the middle term of the above equation is not legal given our list of operations. But this is how the affine combination is typically expressed, namely as the weighted average of two points. The right-hand side (which is easily seen to be algebraically equivalent) is legal. An important observation is that, if $p_0 \neq p_1$, then the point $\text{aff}(p_0, p_1; \alpha_0, \alpha_1)$ lies on the line joining $p_0$ and $p_1$. As $\alpha_1$ varies from $-\infty$ to $+\infty$ it traces out all the points on this line.

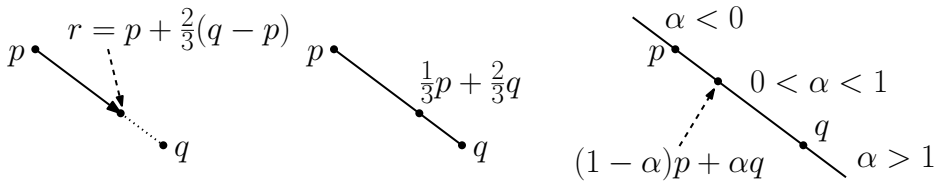



Fig. 99: Affine combination.

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

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

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

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

**Euclidean Geometry:** In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*, which maps two real vectors (not points) into a nonnegative real. One important example of an inner product is the *dot product*, defined as follows. Suppose that the $d$-dimensional vectors $\vec{u}$ and $\vec{v}$ are represented by the (nonhomogeneous) coordinate vectors $(u_1, u_2, \ldots, u_d)$ and $(v_1, v_2, \ldots, v_d)$. Define

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

The dot product is useful in computing the following entities.

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

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

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

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

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

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

This is easy to derive from the law of cosines.

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

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

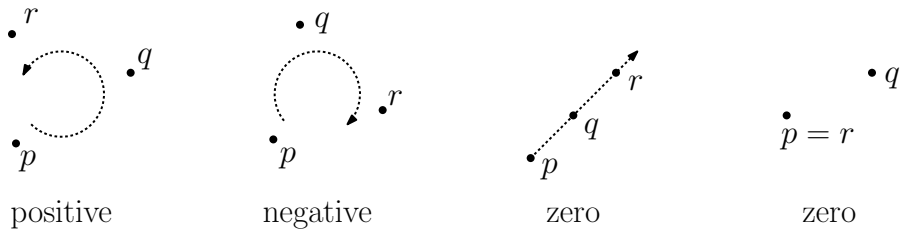



Fig. 100: Orientations of the ordered triple $(p, q, r)$.

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

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

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

This generalizes readily to higher dimensions. For example, given an ordered 4-tuple points in 3-space, we can define their orientation as being either positive (forming a right-handed screw), negative (a left-handed screw), or zero (coplanar). It can be computed as the sign of the determinant of an appropriate $4 \times 4$ generalization of the above determinant. This can be generalized to any ordered $(d + 1)$-tuple of points in $d$-space.

**Areas and Angles:** The orientation determinant, together with the Euclidean norm can be used to compute angles in the plane. This determinant $\text{Orient}(p, q, r)$ is equal to twice the signed area of the triangle $\triangle pqr$ (positive if CCW and negative otherwise). Thus the area of the triangle can be determined by dividing this quantity by 2. In general in dimension $d$ the area of the simplex spanned by $d + 1$ points can be determined by taking this determinant and dividing by $d! = d \cdot (d-1) \cdots 2 \cdot 1$. Given the capability to compute the area of any triangle (or simplex in higher dimensions), it is possible to compute the volume of any polygon (or polyhedron), given an

appropriate subdivision into these basic elements. (Such a subdivision does not need to be disjoint. The simplest methods that I know of use a subdivision into overlapping positively and negatively oriented shapes, such that the signed contribution of the volumes of regions outside the object cancel each other out.)

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

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

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

**Topology Terminology:** Although we will not discuss topology with any degree of formalism, we will need to use some terminology from topology. These terms deserve formal definitions, but we are going to cheat and rely on intuitive definitions, which will suffice for the simple, well behaved geometric objects that we will be dealing with. Beware that these definitions are not fully general, and you are referred to a good text on topology for formal definitions.

For our purposes, for $r > 0$, define the *$r$-neighborhood* of a point $p$ to be the set of points whose distance to $p$ is strictly less than $r$, that is, it is the set of points lying within an open ball of radius $r$ centered about $p$. Given a set $S$, a point $p$ is an *interior point* of $S$ if for some radius $r$ the neighborhood about $p$ of radius $r$ is contained within $S$. A point is an *exterior point* if it lies in the interior of the complement of $S$. A points that is neither interior nor exterior is a *boundary point*. A set is *open* if it contains none of its boundary points and *closed* if its complement is open. If $p$ is in $S$ but is not an interior point, we will call it a *boundary point*.

We say that a geometric set is *bounded* if it can be enclosed in a ball of finite radius. A set is *compact* if it is both closed and bounded.

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

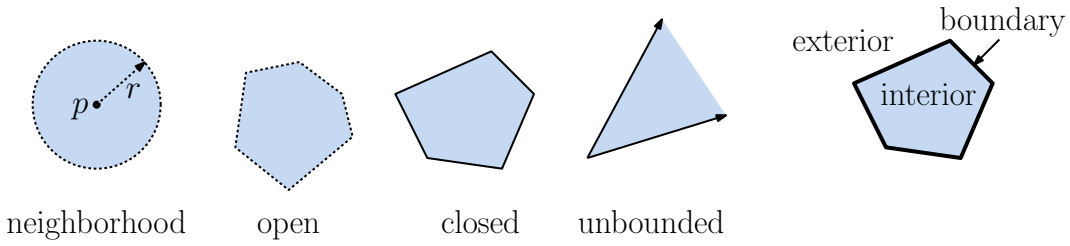



Fig. 101: Terminology.

# Lecture 22: DCELs and Subdivision Intersection

**Doubly-connected Edge List:** We consider the question of how to represent plane straight-line graphs (or PSLG). The DCEL is a common *edge-based representation*. Vertex and face information is also included for whatever geometric application is using the data structure. There are three sets of records one for each element in the PSLG: *vertex records*, a *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*.

We will make a simplifying assumption that faces do not have holes inside of them. This assumption can be satisfied by introducing some number of *dummy edges* joining each hole either to the outer boundary of the face, or to some other hole that has been connected to the outer boundary in this way. With this assumption, it may be assumed that the edges bounding each face form a single cyclic list.

**Vertex:** Each vertex stores its coordinates, along with a pointer to any incident directed edge that has this vertex as its origin, `v.inc_edge`.

**Edge:** Each undirected edge is represented as two directed edges. Each edge has a pointer to the oppositely directed edge, called its *twin*. Each directed edge has an *origin* and *destination* vertex. Each directed edge is associate with two faces, one to its left and one to its right.

We store a pointer to the origin vertex `e.org`. (We do not need to define the destination, `e.dest`, since it may be defined to be `e.twin.org`.)

We store a pointer to the face to the left of the edge `e.left` (we can access the face to the right from the twin edge). This is called the dent face. We also store the next and previous directed edges in counterclockwise order about the incident face, `e.next` and `e.prev`, respectively.

**Face:** Each face $f$ stores a pointer to a single edge for which this face is the incident face, `f.inc_edge`. (See the text for the more general case of dealing with holes.)

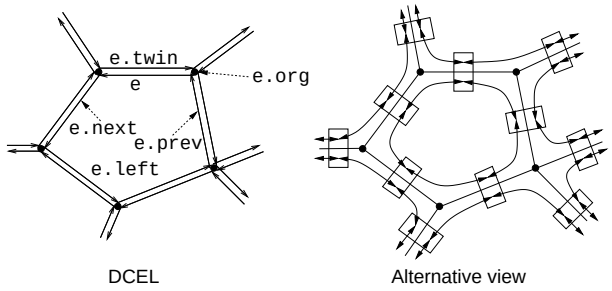



Fig. 102: Doubly-connected edge list.

The figure shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternative way of viewing the data structure that gives a better sense of the connectivity structure is based on covering each edge with a two element block, one for $e$ and the other for its twin. The next and prev pointers provide links around each face of the polygon. The next pointers are directed counterclockwise around each face and the prev pointers are directed clockwise.

Of course, in addition the data structure may be enhanced with whatever application data is relevant. In some applications, it is not necessary to know either the face or vertex information (or both) at all, and if so these records may be deleted. See the book for a complete example.

For example, suppose that we wanted to enumerate the vertices that lie on some face $f$. Here is the code:

Vertex enumeration using DCEL

```
enumerate_vertices(Face f) {
    Edge start = f.inc_edge;
    Edge e = start;
    do {
        output e.org;
        e = e.next;
    } while (e != start);
}
```

**Merging subdivisions:** Let us return to the applications problem that lead to the segment intersection problem. Suppose that we have two planar subdivisions, $S_1$ and $S_2$, and we want to compute their overlay. In particular, this is a subdivision whose vertices are the union of the vertices of each subdivision and the points of intersection of the line segments in the subdivision. (Because we assume that each subdivision is a planar graph, the only new

vertices that could arise will arise from the intersection of two edges, one from $S_1$ and the other from $S_2$.) Suppose that each subdivision is represented using a DCEL. Can we adapt the plane-sweep algorithm to generate the DCEL of the overlaid subdivision?

The answer is yes. The algorithm will destroy the original subdivisions, so it may be desirable to copy them before beginning this process. The first part of the process is straightforward, but perhaps a little tedious. This part consists of building the edge and vertex records for the new subdivision. The second part involves building the face records. It is more complicated because it is generally not possible to know the face structure at the moment that the sweep is advancing, without looking "into the future" of the sweep to see whether regions will merge. (You might try to convince yourself of this.) The entire subdivision is built first, and then the face information is constructed and added later. We will skip the part of updating the face information (see the text).

For the first part, the most illustrative case arises when the sweep is processing an intersection event. In this case the two segments arise as two edges $a_1$ and $b_1$ from the two subdivisions. We will assume that we select the half-edges that are directed from left to right across the sweep-line. The process is described below (and is illustrated in the figure below). It makes use of two auxiliary procedures. $\text{Split}(a_1, a_2)$ splits an edge $a_1$ at its midpoint into two consecutive edges $a_1$ followed by $a_2$, and links $a_2$ into the structure. $\text{Splice}(a_1, a_2, b_1, b_2)$ takes two such split edges and links them all together.

Merge two edges into a common subdivision

**Merge**$(a_1, b_1)$ :

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

(2) Split each of the two intersecting edges, by adding a vertex at the common intersection point. Let $a_2$ and $b_2$ be the new edge pieces. They are created by the calls $a_2 = \text{Split}(a_1)$ and $b_2 = \text{Split}(b_1)$ given below.

(3) Link the four edges together by invoking $\text{Splice}(a_1, a_2, b_1, b_2)$, given below.

The splitting procedure creates the new edge, links it into place. After this the edges have been split, but they are not linked to each other. The edge constructor is given the origin and destination of the new edge and creates a new edge and its twin. The procedure below initializes all the other fields. Also note that the destination of $a_1$, that is the origin of $a_1$'s twin must be updated, which we have omitted. The splice procedure interlinks four edges around a common vertex in the counterclockwise order $a_1$ (entering), $b_1$ (entering), $a_2$ (leaving), $b_2$ (leaving).

Split an edge into two edges

```
Split(edge &a1, edge &a2) {              // a2 is returned
    a2 = new edge(v, a1.dest());         // create edge (v,a1.dest)
    a2.next  = a1.next;   a1.next.prev  = a2;
    a1.next  = a2;        a2.prev  = a1;
    a1t = a1.twin;        a2t = a2.twin; // the twins
    a2t.prev = a1t.prev;  a1t.prev.next = a2t;
    a1t.prev = a2t;       a2t.next  = a1t;
}
```

Splice four edges together

```
Splice(edge &a1, edge &a2, edge &b1, edge &b2) {
    a1t = a1.twin;   a2t = a2.twin;      // get the twins
    b1t = b1.twin;   b2t = b2.twin;
    a1.next  = b2;   b2.prev  = a1;      // link the edges together
    b2t.next = a2;   a2.prev  = b2t;
    a2t.next = b1t;  b1t.prev = a2t;
    b1.next  = a1t;  a1t.prev = b1;
}
```

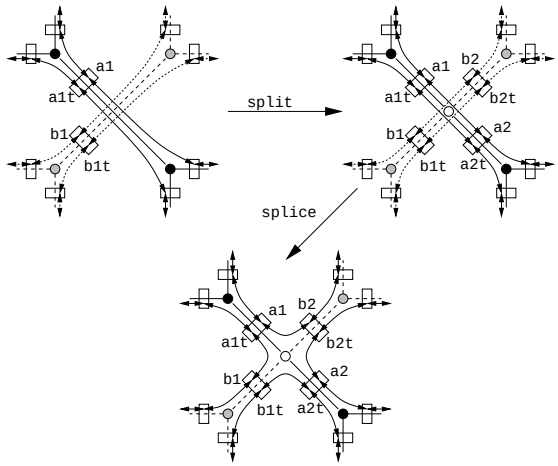



Fig. 103: Updating the DCEL.

# Lecture 23: Smallest Enclosing Disk

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

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

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

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

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

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

This immediately suggests a simple $O(n^4)$ time algorithm. In $O(n^3)$ time we can enumerate all triples of points and then for each we generate the resulting circle and test whether it encloses all the points in $O(n)$ additional

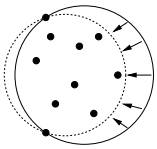


Fig. 104: Contact points for a minimum enclosing disk.

time, for an $O(n^4)$ time algorithm. You might make a few observations to improve this a bit (e.g. by using only triples of points on the convex hull). But even so a reduction from $O(n^4)$ to $O(n)$ is quite dramatic.

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

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

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

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

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

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

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

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

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

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

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

To start we randomly permute the points. We select any two points and compute the unique circle with these points as diameter. (We could have started with three just as easily.) Let $D_{i-1}$ denote the minimum disk after the insertion of the first $i-1$ points. For point $p_i$ we determine in constant time whether the point lies within $D_{i-1}$. If so, then we set $D_i = D_{i-1}$ and go on to the next stage. If not, then we need to update the current disk to contain $p_i$, letting $D_i$ denote the result. When the last point is inserted we output $D_n$.

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

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

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

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

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

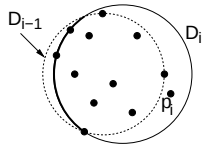



Fig. 105: Why $p_i$ must lie on the boundary of $D_i$.

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

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

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

# Lecture 24: Interval Trees

**Segment Data:** So far we have considered geometric data structures for storing points. However, there are many others types of geometric data that we may want to store in a data structure. Today we consider how to store orthogonal (horizontal and vertical) line segments in the plane. We assume that a line segment is represented by giving its pair of *endpoints*. The segments are allowed to intersect one another.

As a basic motivating query, we consider the following *window query*. Given a set of orthogonal line segments $S$, which have been preprocessed, and given an orthogonal query rectangle $W$, count or report all the line segments of $S$ that intersect $W$. We will assume that $W$ is closed and solid rectangle, so that even if a line segment lies entirely inside of $W$ or intersects only the boundary of $W$, it is still reported. For example, given the window below, the query would report the segments that are shown with solid lines, and segments with broken lines would not be reported.

**Window Queries for Orthogonal Segments:** We will present a data structure, called the *interval tree*, which (combined with a range tree) can answer window counting queries for orthogonal line segments in $O(\log^2 n)$ time,

**MinDisk**$(P)$ :

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

**MinDiskWith1Pt**$(P, q)$ :

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

**MinDiskWith2Pts**$(P, q_1, q_2)$ :

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

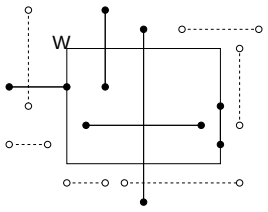



Fig. 106: Window Query.

where $n$ is the number line segments. It can report these segments in $O(k+\log^2 n)$ time, where and $k$ is the total number of segments reported. The interval tree uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.

We will consider the case of range reporting queries. (There are some subtleties in making this work for counting queries.) We will derive our solution in steps, starting with easier subproblems and working up to the final solution. To begin with, observe that the set of segments that intersect the window can be partitioned into three types: those that have no endpoint in $W$, those that have one endpoint in $W$, and those that have two endpoints in $W$.

We already have a way to report segments of the second and third types. In particular, we may build a range tree just for the $2n$ endpoints of the segments. We assume that each endpoint has a cross-link indicating the line segment with which it is associated. Now, by applying a range reporting query to $W$ we can report all these endpoints, and follow the cross-links to report the associated segments. Note that segments that have both endpoints in the window will be reported twice, which is somewhat unpleasant. We could fix this either by sorting the segments in some manner and removing duplicates, or by marking each segment as it is reported and ignoring segments that have already been marked. (If we use marking, after the query is finished we will need to go back an "unmark" all the reported segments in preparation for the next query.)

All that remains is how to report the segments that have no endpoint inside the rectangular window. We will do this by building two separate data structures, one for horizontal and one for vertical segments. A horizontal segment that intersects the window but neither of its endpoints intersects the window must pass entirely through the window. Observe that such a segment intersects any vertical line passing from the top of the window to the bottom. In particular, we could simply ask to report all horizontal segments that intersect the left side of $W$. This is called a *vertical segment stabbing query*. In summary, it suffices to solve the following subproblems (and remove duplicates):

**Endpoint inside:** Report all the segments of $S$ that have at least one endpoint inside $W$. (This can be done using a range query.)

**Horizontal through segments:** Report all the horizontal segments of $S$ that intersect the left side of $W$. (This reduces to a vertical segment stabbing query.)

**Vertical through segments:** Report all the vertical segments of $S$ that intersect the bottom side of $W$. (This reduces to a horizontal segment stabbing query.)

We will present a solution to the problem of vertical segment stabbing queries. Before dealing with this, we will first consider a somewhat simpler problem, and then modify this simple solution to deal with the general problem.

**Vertical Line Stabbing Queries:** Let us consider how to answer the following query, which is interesting in its own right. Suppose that we are given a collection of horizontal line segments $S$ in the plane and are given an (infinite) vertical query line $\ell_q : x = x_q$. We want to report all the line segments of $S$ that intersect $\ell_q$. Notice that for the purposes of this query, the $y$-coordinates are really irrelevant, and may be ignored. We can think of each horizontal line segment as being a closed *interval* along the $x$-axis. We show an example in the figure below on the left.

As is true for all our data structures, we want some balanced way to decompose the set of intervals into subsets. Since it is difficult to define some notion of order on intervals, we instead will order the endpoints. Sort the interval endpoints along the $x$-axis. Let $\langle x_1, x_2, \ldots, x_{2n} \rangle$ be the resulting sorted sequence. Let $x_{med}$ be the median of these $2n$ endpoints. Split the intervals into three groups, $L$, those that lie strictly to the left of $x_{\text{med}}$, $R$ those that lie strictly to the right of $x_{\text{med}}$, and $M$ those that contain the point $x_{\text{med}}$. We can then define a binary tree by putting the intervals of $L$ in the left subtree and recursing, putting the intervals of $R$ in the right subtree and recursing. Note that if $x_q < x_{\text{med}}$ we can eliminate the right subtree and if $x_q > x_{\text{med}}$ we can eliminate the left subtree. See the figure right.

But how do we handle the intervals of $M$ that contain $x_{\text{med}}$? We want to know which of these intervals intersects the vertical line $\ell_q$. At first it may seem that we have made no progress, since it appears that we are back to the

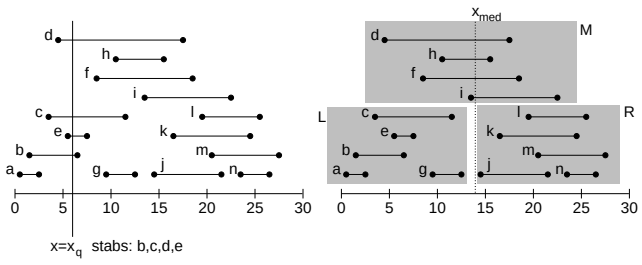



Fig. 107: Line Stabbing Query.

same problem that we started with. However, we have gained the information that all these intervals intersect the vertical line $x = x_{\text{med}}$. How can we use this to our advantage?

Let us suppose for now that $x_q \leq x_{\text{med}}$. How can we store the intervals of $M$ to make it easier to report those that intersect $\ell_q$. The simple trick is to sort these lines in increasing order of their left endpoint. Let $M_L$ denote the resulting sorted list. Observe that if some interval in $M_L$ does not intersect $\ell_q$, then its left endpoint must be to the right of $x_q$, and hence none of the subsequent intervals intersects $\ell_q$. Thus, to report all the segments of $M_L$ that intersect $\ell_q$, we simply traverse the sorted list and list elements until we find one that does not intersect $\ell_q$, that is, whose left endpoint lies to the right of $x_q$. As soon as this happens we terminate. If $k'$ denotes the total number of segments of $M$ that intersect $\ell_q$, then clearly this can be done in $O(k' + 1)$ time.

On the other hand, what do we do if $x_q > x_{\text{med}}$? This case is symmetrical. We simply sort all the segments of $M$ in a sequence, $M_R$, which is sorted from right to left based on the right endpoint of each segment. Thus each element of $M$ is stored twice, but this will not affect the size of the final data structure by more than a constant factor. The resulting data structure is called an *interval tree*.

**Interval Trees:** The general structure of the interval tree was derived above. Each node of the interval tree has a left child, right child, and itself contains the median $x$-value used to split the set, $x_{\text{med}}$, and the two sorted sets $M_L$ and $M_R$ (represented either as arrays or as linked lists) of intervals that overlap $x_{\text{med}}$. We assume that there is a constructor that builds a node given these three entities. The following high-level pseudocode describes the basic recursive step in the construction of the interval tree. The initial call is `root = IntTree(S)`, where $S$ is the initial set of intervals. Unlike most of the data structures we have seen so far, this one is not built by the successive insertion of intervals (although it would be possible to do so). Rather we assume that a set of intervals $S$ is given as part of the constructor, and the entire structure is built all at once. We assume that each interval in $S$ is represented as a pair $(x_{\text{lo}}, x_{\text{hi}})$. An example is shown in the following figure.

We assert that the height of the tree is $O(\log n)$. To see this observe that there are $2n$ endpoints. Each time through the recursion we split this into two subsets $L$ and $R$ of sizes at most half the original size (minus the elements of $M$). Thus after at most $\lg(2n)$ levels we will reduce the set sizes to 1, after which the recursion bottoms out. Thus the height of the tree is $O(\log n)$.

Implementing this constructor efficiently is a bit subtle. We need to compute the median of the set of all endpoints, and we also need to sort intervals by left endpoint and right endpoint. The fastest way to do this is to presort all these values and store them in three separate lists. Then as the sets $L$, $R$, and $M$ are computed, we simply copy items from these sorted lists to the appropriate sorted lists, maintaining their order as we go. If we do so, it can be shown that this procedure builds the entire tree in $O(n \log n)$ time.

The algorithm for answering a stabbing query was derived above. We summarize this algorithm below. Let $x_q$ denote the $x$-coordinate of the query line.

This procedure actually has one small source of inefficiency, which was intentionally included to make code look more symmetric. Can you spot it? Suppose that $x_q = t.x_{\text{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that are strictly to the right of $x_{\text{med}}$ and so is a waste of effort. However it does not affect the asymptotic running time.

```
IntTreeNode IntTree(IntervalSet S) {
    if (|S| == 0) return null                          // no more

    xMed = median endpoint of intervals in S           // median endpoint

    L = {[xlo, xhi] in S | xhi < xMed}                 // left of median
    R = {[xlo, xhi] in S | xlo > xMed}                 // right of median
    M = {[xlo, xhi] in S | xlo <= xMed <= xhi}         // contains median
    ML = sort M in increasing order of xlo             // sort M
    MR = sort M in decreasing order of xhi

    t = new IntTreeNode(xMed, ML, MR)                  // this node
    t.left  = IntTree(L)                               // left subtree
    t.right = IntTree(R)                               // right subtree
    return t
}
```

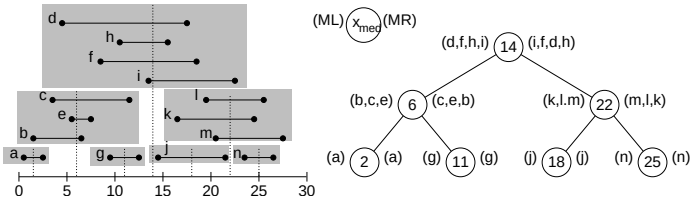



Fig. 108: Interval Tree.



```
stab(IntTreeNode t, Scalar xq) {
    if (t == null) return                          // fell out of tree
    if (xq < t.xMed) {                             // left of median?
        for (i = 0; i < t.ML.length; i++) {        // traverse ML
            if (t.ML[i].lo <= xq) print(t.ML[i])// ..report if in range
            else break                             // ..else done
        }
        stab(t.left, xq)                           // recurse on left
    }
    else {                                         // right of median
        for (i = 0; i < t.MR.length; i++) {        // traverse MR
            if (t.MR[i].hi >= xq) print(t.MR[i])// ..report if in range
            else break                             // ..else done
        }
        stab(t.right, xq)                          // recurse on right
    }
}
```

As mentioned earlier, the time spent processing each node is $O(1 + k')$ where $k'$ is the total number of points that were recorded at this node. Summing over all nodes, the total reporting time is $O(k + v)$, where $k$ is the total number of intervals reported, and $v$ is the total number of nodes visited. Since at each node we recurse on only one child or the other, the total number of nodes visited $v$ is $O(\log n)$, the height of the tree. Thus the total reporting time is $O(k + \log n)$.

**Vertical Segment Stabbing Queries:** Now let us return to the question that brought us here. Given a set of horizontal line segments in the plane, we want to know how many of these segments intersect a vertical line segment. Our approach will be exactly the same as in the interval tree, except for how the elements of $M$ (those that intersect the splitting line $x = x_{\text{med}}$) are handled.

Going back to our interval tree solution, let us consider the set $M$ of horizontal line segments that intersect the splitting line $x = x_{\text{med}}$ and as before let us consider the case where the query segment $q$ with endpoints $(x_q, y_{\text{lo}})$ and $(x_q, y_{\text{hi}})$ lies to the left of the splitting line. The simple trick of sorting the segments of $M$ by their left endpoints is not sufficient here, because we need to consider the $y$-coordinates as well. Observe that a segment of $M$ stabs the query segment $q$ if and only if the left endpoint of a segment lies in the following semi-infinite rectangular region.

$$\{(x, y) \mid x \le x_q \text{ and } y_{\text{lo}} \le y \le y_{\text{hi}}\}.$$

This is illustrated in the figure below. Observe that this is just an orthogonal range query. (It is easy to generalize the procedure given last time to handle semi-infinite rectangles.) The case where $q$ lies to the right of $x_{\text{med}}$ is symmetrical.

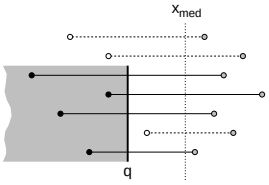



Fig. 109: The segments that stab $q$ lie within the shaded semi-infinite rectangle.

So the solution is that rather than storing $M_L$ as a list sorted by the left endpoint, instead we store the left endpoints in a 2-dimensional range tree (with cross-links to the associated segments). Similarly, we create a range tree for the right endpoints and represent $M_R$ using this structure.

The segment stabbing queries are answered exactly as above for line stabbing queries, except that part that searches $M_L$ and $M_R$ (the for-loops) are replaced by searches to the appropriate range tree, using the semi-infinite range given above.

We will not discuss construction time for the tree. (It can be done in $O(n \log n)$ time, but this involves some thought as to how to build all the range trees efficiently). The space needed is $O(n \log n)$, dominated primarily from the $O(n \log n)$ space needed for the range trees. The query time is $O(k + \log^3 n)$, since we need to answer $O(\log n)$ range queries and each takes $O(\log^2 n)$ time plus the time for reporting. If we use the spiffy version of range trees (which we mentioned but never discussed) that can answer queries in $O(k + \log n)$ time, then we can reduce the total time to $O(k + \log^2 n)$.

# Lecture 25: Hereditary Segment Trees and Red-Blue Intersection

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

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

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

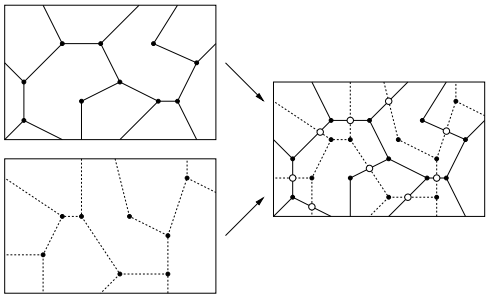


Fig. 110: Red-blue line segment intersection. The algorithm outputs the white intersection points between segments of different colors. The segments of each color are pairwise disjoint (except possibly at their endpoints).

Let $N = n + m$ denote the total input size and let $k$ denote the total number of bichromatic intersecting pairs. We will present an algorithm for this problem that runs in $O(k + N \log^2 N)$ time for the reporting problem and $O(N \log^2 N)$ time for the counting problem. Both algorithms use $O(N \log N)$ space. Although we will not discuss it (but the original paper does) it is possible to remove a factor of $\log n$ from both the running time and space, using a somewhat more sophisticated variant of the algorithm that we will present.

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

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

We associate a segment $s$ with a set of nodes of the tree. A segment is said to *span* interval $I_u$ if its projection covers this interval. We associate a segment $s$ with a node $u$ if $s$ spans $I_u$ but $s$ does not span $I_p$, where $p$ is $u$'s parent. (See the figure above, right.)

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

Each node $u$ is also associated with a list $B^*_u$, called the *blue hereditary list*, which is the union of the $B_v$ for all proper descendents $v$ or $u$. The red hereditary list $R^*_u$ is defined analogously. (Even though a segment may

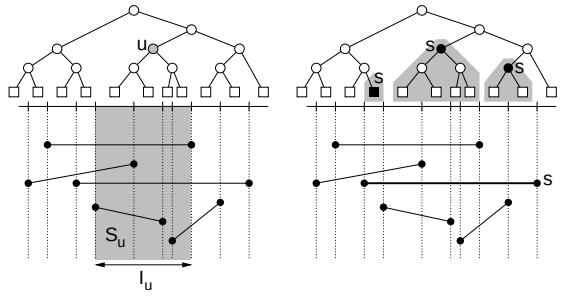



Fig. 111: Hereditary Segment Tree: Intervals, slabs and the nodes associated with a segment.

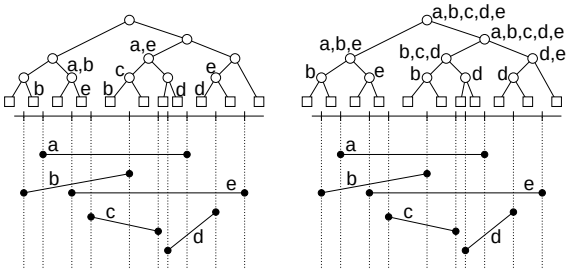



Fig. 112: Hereditary Segment Tree with standard lists (left) and hereditary lists (right).

occur in the standard list for many descendents, there is only one copy of each segment in the hereditary lists.) The segments of $R_u$ and $B_u$ are called the *long segments*, since they span the entire interval. The segments of $R_u^*$ and $B_u^*$ are called the *short segments*, since they do not span the entire interval.

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

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

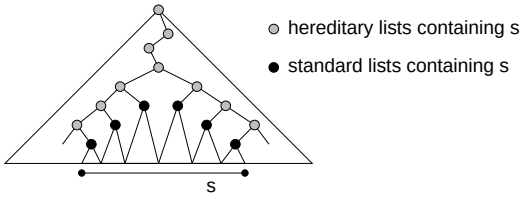



Fig. 113: Standard and hereditary lists containing a segment $s$.

The tree can be built in $O(N \log N)$ time. In $O(N \log N)$ time we can sort the $2N$ segment endpoints. Then for each segment, we search for its left and right endpoints and insert the segment into the standard and hereditary

lists for the appropriate nodes and we descend each path in $O(1)$ time for each node visited. Since each segment appears in $O(\log N)$ lists, this will take $O(\log N)$ time per segment and $O(N \log N)$ time overall.

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

**Long-long intersections:** Sort each of the lists $B_u$ and $R_u$ of long segments in ascending order by $y$-coordinate. (Since the segments of each set are disjoint, this order is constant throughout the interval for each set.) Let $\langle b_1, b_2, \ldots, b_{m_u} \rangle$ and $\langle r_1, r_2, \ldots, r_{n_u} \rangle$ denote these ordered lists. Merge these lists twice, once according to their order along the left side of the slab and one according to their order along the right side of the slab.

Observe that for each blue segment $b \in B_u$, this allows us to determine two indices $i$ and $j$, such that $b$ lies between $r_i$ and $r_{i+1}$ along the left boundary and between $r_j$ and $r_{j+1}$ along the right boundary. (For convenience, we can think of segment 0 as an imaginary segment at $y = -\infty$.)

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

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

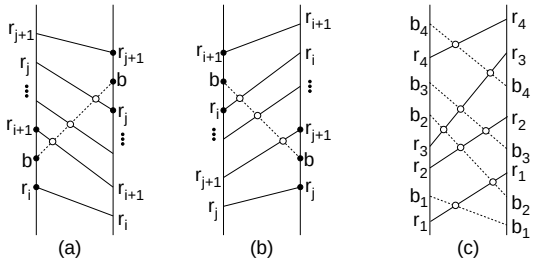



Fig. 114: Red-blue intersection counting/reporting. Long-long intersections.

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

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

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

Thus, after $O(\log n_u)$ time for the binary searches, the segments of $R_u$ intersecting $b$ can be counted in $O(1)$ time, for a total time of $O(m_u^* \log n_u)$. Reporting can be done in time proportional to the number of intersections reported. Adding this to the time for the long blue and short red case, we have a total time complexity of $O(m_u^* \log n_u + n_u^* \log m_u)$.

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

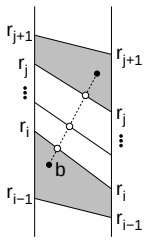



Fig. 115: Red-blue intersection counting/reporting: Long-short intersections.

complexity of

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

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

# Lecture 26: Kirkpatrick's Planar Point Location

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

It is remarkable that although this seems like such a simple and natural problem, it took quite a long time to discover a method that is optimal with respect to both query time and space. It has long been known that there are data structures that can perform these searches reasonably well (e.g. quad-trees and kd-trees), but for which no good theoretical bounds could be proved. There were data structures of with $O(\log n)$ query time but $O(n \log n)$ space, and $O(n)$ space but $O(\log^2 n)$ query time.

The first construction to achieve both $O(n)$ space and $O(\log n)$ query time was a remarkably clever construction due to Kirkpatrick. It turns out that Kirkpatrick's idea has some large embedded constant factors that make it less attractive practically, but the idea is so clever that it is worth discussing, nonetheless. Later we will discuss a more practical randomized method that is presented in our text.

**Kirkpatrick's Algorithm:** Kirkpatrick's idea starts with the assumption that the planar subdivision is a triangulation, and further that the outer face is a triangle. If this assumption is not met, then we begin by triangulating all the faces of the subdivision. The label associated with each triangular face is the same as a label for the original face that contained it. For the outer face is not a triangle, first compute the convex hull of the polygonal subdivision, triangulate everything inside the convex hull. Then surround this convex polygon with a large triangle (call the vertices $a$, $b$, and $c$), and then add edges from the convex hull to the vertices of the convex hull. It may sound like we are adding a lot of new edges to the subdivision, but recall from earlier in the semester that the number of edges and faces in any straight-line planar subdivision is proportional to $n$, the number of vertices. Thus the addition only increases the size of the structure by a constant factor.

Note that once we find the triangle containing the query point in the augmented graph, then we will know the original face that contains the query point. The triangulation process can be performed in $O(n \log n)$ time by a plane sweep of the graph, or in $O(n)$ time if you want to use sophisticated methods like the linear time polygon triangulation algorithm. In practice, many straight-line subdivisions, may already have convex faces and these can be triangulated easily in $O(n)$ time.

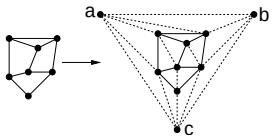



Fig. 116: Triangulation of a planar subdivision.

Let $T_0$ denote the initial triangulation. What Kirkpatrick's method does is to produce a sequence of triangulations, $T_0, T_1, T_2, \ldots, T_k$, where $k = O(\log n)$, such that $T_k$ consists only of a single triangle (the exterior face of $T_0$), and each triangle in $T_{i+1}$ overlaps a constant number of triangles in $T_i$.

We will see how to use such a structure for point location queries later, but for now let us concentrate on how to build such a sequence of triangulations. Assuming that we have $T_i$, we wish to compute $T_{i+1}$. In order to guarantee that this process will terminate after $O(\log n)$ stages, we will want to make sure that the number of vertices in $T_{i+1}$ decreases by some constant factor from the number of vertices in $T_i$. In particular, this will be done by carefully selecting a subset of vertices of $T_i$ and deleting them (and along with them, all the edges attached to them). After these vertices have been deleted, we need retriangulate the resulting graph to form $T_{i+1}$. The question is: How do we select the vertices of $T_i$ to delete, so that each triangle of $T_{i+1}$ overlaps only a constant number of triangles in $T_i$?

There are two things that Kirkpatrick observed at this point, that make the whole scheme work.

**Constant degree:** We will make sure that each of the vertices that we delete have constant ($\leq d$) degree (that is, each is adjacent to at most $d$ edges). Note that the when we delete such a vertex, the resulting *hole* will consist of at most $d - 2$ triangles. When we retriangulate, each of the new triangles, can overlap at most $d$ triangles in the previous triangulation.

**Independent set:** We will make sure that no two of the vertices that are deleted are adjacent to each other, that is, the vertices to be deleted form an *independent set* in the current planar graph $T_i$. This will make retriangulation easier, because when we remove $m$ independent vertices (and their incident edges), we create $m$ independent *holes* (non triangular faces) in the subdivision, which we will have to retriangulate. However, each of these holes can be triangulated independently of one another. (Since each hole contains a constant number of vertices, we can use any triangulation algorithm, even brute force, since the running time will be $O(1)$ in any case.)

An important question to the success of this idea is whether we can always find a sufficiently large independent set of vertices with bounded degree. We want the size of this set to be at least a constant fraction of the current number of vertices. Fortunately, the answer is "yes," and in fact it is quite easy to find such a subset. Part of the trick is to pick the value of $d$ to be large enough (too small and there may not be enough of them). It turns out that $d = 8$ is good enough.

**Lemma:** Given a planar graph with $n$ vertices, there is an independent set consisting of vertices of degree at most 8, with at least $n/18$ vertices. This independent set can be constructed in $O(n)$ time.

We will present the proof of this lemma later. The number 18 seems rather large. The number is probably smaller in practice, but this is the best bound that this proof generates. However, the size of this constant is one of the reasons that Kirkpatrick's algorithm is not used in practice. But the construction is quite clever, nonetheless, and once a optimal solution is known to a problem it is often not long before a practical optimal solution follows.

**Kirkpatrick Structure:** Assuming the above lemma, let us give the description of how the point location data structure, the *Kirkpatrick structure*, is constructed. We start with $T_0$, and repeatedly select an independent set of vertices of degree at most 8. We never include the three vertices $a$, $b$, and $c$ (forming the outer face) in such an independent set. We delete the vertices from the independent set from the graph, and retriangulate the resulting *holes*. Observe that each triangle in the new triangulation can overlap at most 8 triangles in the previous triangulation. Since we can eliminate a constant fraction of vertices with each stage, after $O(\log n)$ stages, we will be down to the last 3 vertices.

The constant factors here are not so great. With each stage, the number of vertices falls by a factor of $17/18$. To reduce to the final three vertices, implies that $(18/17)^k = n$ or that

$$k = \log_{18/17} n \approx 12 \lg n.$$

It can be shown that by always selecting the vertex of smallest degree, this can be reduced to a more palatable $4.5 \lg n$.

The data structure is based on this decomposition. The root of the structure corresponds to the single triangle of $T_k$. The nodes at the next lower level are the triangles of $T_{k-1}$, followed by $T_{k-2}$, until we reach the leaves, which are the triangles of our initial triangulation, $T_0$. Each node for a triangle in triangulation $T_{i+1}$, stores pointers to all the triangles it overlaps in $T_i$ (there are at most 8 of these). Note that this structure is a directed acyclic graph (DAG) and not a tree, because one triangle may have many parents in the data structure. This is shown in the following figure.

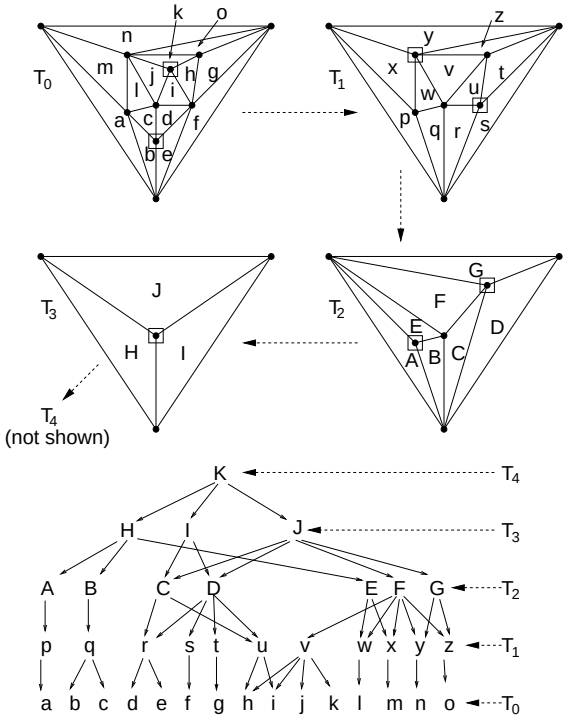



Fig. 117: Kirkpatrick's point location structure.

To locate a point, we start with the root, $T_k$. If the query point does not lie within this single triangle, then we are done (it lies in the exterior face). Otherwise, we search each of the (at most 8) triangles in $T_{k-1}$ that overlap this triangle. When we find the correct one, we search each of the triangles in $T_{k-2}$ that overlap this triangles, and so forth. Eventually we will find the triangle containing the query point in the last triangulation, $T_0$, and this is the desired output. See the figure below for an example.

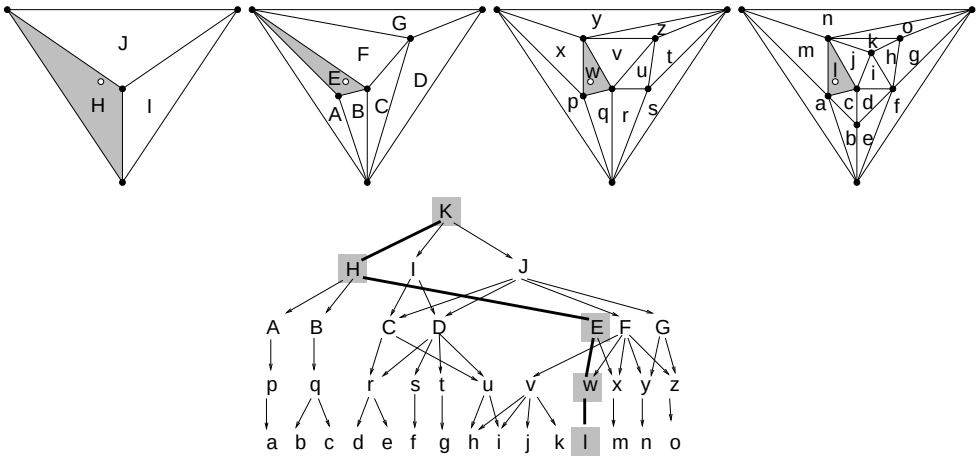



Fig. 118: Point location search.

**Construction and Analysis:** The structure has $O(\log n)$ levels (one for each triangulation), it takes a constant amount of time to move from one level to the next (at most 8 point-in-triangle tests), thus the total query time is $O(\log n)$. The size of the data structure is the sum of sizes of the triangulations. Since the number of triangles in a triangulation is proportional to the number of vertices, it follows that the size is proportional to

$$n(1 + 17/18 + (17/18)^2 + (17/18)^3 + \ldots) \leq 18n,$$

(using standard formulas for geometric series). Thus the data structure size is $O(n)$ (again, with a pretty hefty constant).

The last thing that remains is to show how to construct the independent set of the appropriate size. We first present the algorithm for finding the independent set, and then prove the bound on its size.

(1) Mark all nodes of degree $\geq 9$.

(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.

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 27: Divide-and-Conquer Algorithm for Voronoi Diagrams

**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.

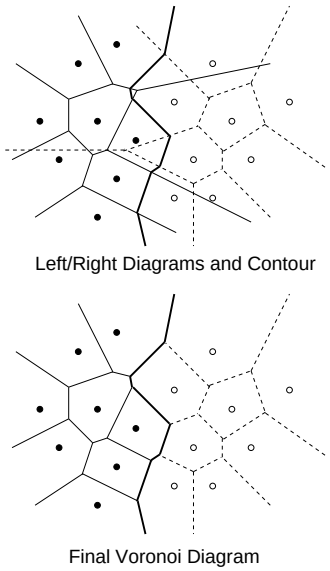



Fig. 119: Merging Voronoi diagrams.

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.

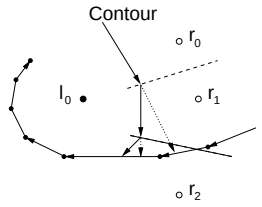



Fig. 120: Tracing the contour.

We have to avoid this repeated rescanning. However, there is a way to scan the boundary of each Voronoi polygon at most once. Observe that as we walk along the contour, each time we stay in the same polygon $V(l_0)$, we are adding another edge onto its Voronoi polygon. Because the Voronoi polygon is convex, we know that the edges we are creating turn consistently in the same direction (clockwise for points on the left, and counterclockwise for points on the right). To test for intersections between the contour and the current Voronoi polygon, we trace the boundary of the polygon clockwise for polygons on the left side, and counterclockwise for polygons on the right side. Whenever the contour changes direction, we continue the scan from the point that we left off. In this way, we know that we will never need to rescan the same edge of any Voronoi polygon more than once.

# Lecture 28: Delaunay Triangulations and Convex Hulls

**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.

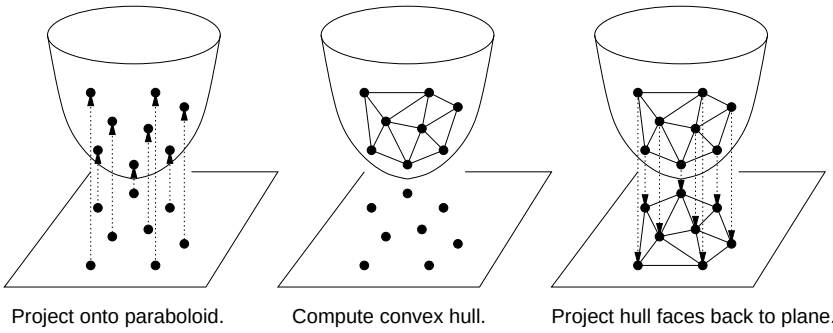



Fig. 121: Delaunay triangulations and convex hull.

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 $\gamma$ 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 $p, q, r$.

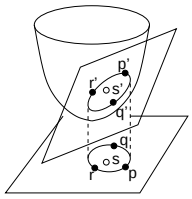



Fig. 122: Planes and circles.

Now we can prove the main result.

**Theorem:** Given a set of points $S$ in the plane (assume no 4 are cocircular), and given 3 points $p, q, r \in S$, the triangle $\triangle pqr$ is a triangle of the Delaunay triangulation of $S$ if and only if triangle $\triangle p'q'r'$ is a face of the lower convex hull of the projected set $S'$.

From the definition of Delaunay triangulations we know that $\triangle pqr$ is in the Delaunay triangulation if and only if there is no point $s \in S$ that lies within the circumcircle of $pqr$. From the previous lemma this is equivalent to saying that there is no point $s'$ that lies in the lower convex hull of $S'$, which is equivalent to saying that $p'q'r'$ is a face of the lower convex hull. This completes the proof.

In order to test whether a point $s$ lies within the circumcircle defined by $p, q, r$, it suffices to test whether $s'$ lies within the lower halfspace of the plane passing through $p', q', r'$. If we assume that $p, q, r$ are oriented counterclockwise in the plane this this reduces to determining whether the quadruple $p', q', r', s'$ is positively oriented, or equivalently whether $s$ lies to the left of the oriented circle passing through $p, q, r$.

This leads to the incircle test we presented last time.

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

**Voronoi Diagrams and Upper Envelopes:** We know that Voronoi diagrams and Delaunay triangulations are dual geometric structures. We have also seen (informally) that there is a dual relationship between points and lines in the plane, and in general, points and planes in 3-space. From this latter connection we argued that the problems of computing convex hulls of point sets and computing the intersection of halfspaces are somehow "dual" to one another. It turns out that these two notions of duality, are (not surprisingly) interrelated. In particular, in the same way that the Delaunay triangulation of points in the plane can be transformed to computing a convex hull in 3-space, it turns out that the Voronoi diagram of points in the plane can be transformed into computing the intersection of halfspaces in 3-space.

Here is how we do this. For each point $p = (a, b)$ in the plane, recall the tangent plane to the paraboloid above this point, given by the equation

$$z = 2ax + 2by - (a^2 + b^2).$$

Define $H^+(p)$ to be the set of points that are above this halfplane, that is, $H^+(p) = \{(x, y, z) \mid z \geq 2ax + 2by - (a^2 + b^2)\}$. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of points. Consider the intersection of the halfspaces $H^+(p_i)$. This is also called the *upper envelope* of these halfspaces. The upper envelope is an (unbounded) convex polyhedron. If you project the edges of this upper envelope down into the plane, it turns out that you get the Voronoi diagram of the points.

**Theorem:** Given a set of points $S$ in the plane (assume no 4 are cocircular), let $H$ denote the set of upper halfspaces defined by the previous transformation. Then the Voronoi diagram of $H$ is equal to the projection onto the $(x, y)$-plane of the 1-skeleton of the convex polyhedron which is formed from the intersection of halfspaces of $S'$.

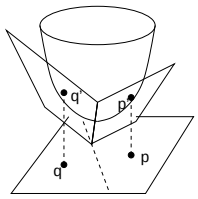



Fig. 123: Intersection of halfspaces.

It is hard to visualized this surface, but it is not hard to show why this is so. Suppose we have 2 points in the plane, $p = (a, b)$ and $q = (c, d)$. The corresponding planes are:

$$z = 2ax + 2by - (a^2 + b^2) \qquad \text{and} \qquad z = 2cx + 2dy - (c^2 + d^2).$$

If we determine the intersection of the corresponding planes and project onto the $(x, y)$-coordinate plane (by eliminating $z$ from these equations) we get

$$x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2).$$

We claim that this is the perpendicular bisector between $(a, b)$ and $(c, d)$. To see this, observe that it passes through the midpoint $((a + c)/2, (b + d)/2)$ between the two points since

$$\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2).$$

and, its slope is $-(a-c)/(b-d)$, which is the negative reciprocal of the line segment from $(a, b)$ to $(c, d)$. From this it can be shown that the intersection of the upper halfspaces defines a polyhedron whose edges project onto the Voronoi diagram edges.

# Lecture 29: Topological Plane Sweep

**Topological Plane Sweep:** In the last two lectures we have introduced arrangements of lines and geometric duality as important tools in solving geometric problems on lines and points. Today give an efficient algorithm for sweeping an arrangement of lines.

As we will see, many problems in computational geometry can be solved by applying line-sweep to an arrangement of lines. Since the arrangement has size $O(n^2)$, and since there are $O(n^2)$ events to be processed, each involving an $O(\log n)$ heap deletion, this typically leads to algorithms running in $O(n^2 \log n)$ time, using $O(n^2)$ space. It is natural to ask whether we can dispense with the additional $O(\log n)$ factor in running time, and whether we need all of $O(n^2)$ space (since in theory we only need access to the current $O(n)$ contents of the sweep line).

We discuss a variation of plane sweep called *topological plane sweep*. This method runs in $O(n^2)$ time, and uses only $O(n)$ space (by essentially constructing only the portion of the arrangement that we need at any point). Although it may appear to be somewhat sophisticated, it can be implemented quite efficiently, and is claimed to outperform conventional plane sweep on arrangements of any significant size (e.g. over 20 lines).

**Cuts and topological lines:** The algorithm is called *topological* plane sweep because we do not sweep a straight vertical line through the arrangement, but rather we sweep a curved *topological line* that has the essential properties of a vertical sweep line in the sense that this line intersects each line of the arrangement exactly once. The notion of a topological line is an intuitive one, but it can be made formal in the form of something called a *cut*. Recall that the faces of the arrangement are convex polygons (possibly unbounded). (Assuming no vertical lines) the edges incident to each face can naturally be partitioned into the edges that are *above* the face, and those that are *below* the face. Define a *cut* in an arrangement to be a sequence of edges $c_1, c_2, \ldots, c_n$, in the arrangement, one taken from each line of the arrangement, such that for $1 \leq i \leq n - 1$, $c_i$ and $c_{i+1}$ are incident to the same face of the arrangement, and $c_i$ is above the face and $c_{i+1}$ is below the face. An example of a topological line and the associated cut is shown below.

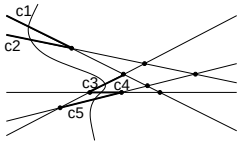



Fig. 124: Topological line and associated cut.

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.

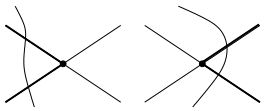


Fig. 125: Elementary step.

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.

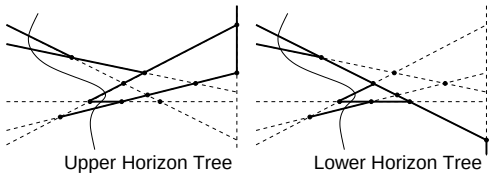



Fig. 126: Upper and lower horizon trees.

The important things about the UHT and LHT is that they give us an easy way to determine the right endpoints of the edges on the cut. Observe that for each edge in the cut, its right endpoint results from a line of smaller slope intersecting it from above (as we trace it from left to right) or from a line of larger slope intersecting it from below. It is easy to verify that the UHT and LHT determine the first such intersecting line of each type, respectively. It follows that if we intersect the two trees, then the segments they share in common correspond exactly to the edges of the cut. Thus, by knowing the UHT and LHT, we know where are the right endpoints are, and from this we can determine easily which pairs of consecutive edges share a common right endpoint, and from this we can determine all the elementary steps that are legal. We store all the legal steps in a stack (or queue, or any list is fine), and extract them one by one.

**The sweep algorithm:** Here is an overview of the topological plane sweep.

(1) Input the lines and sort by slope. Let $C$ be the initial (leftmost) cut, a list of lines in decreasing order of slope.

(2) Create the initial UHT incrementally by inserting lines in decreasing order of slope. Create the initial LHT incrementally by inserting line in increasing order of slope. (More on this later.)

(3) By consulting the LHT and UHT, determine the right endpoints of all the edges of the initial cut, and for all pairs of consecutive lines $(l_i, l_{i+1})$ sharing a common right endpoint, store this pair in stack $S$.

(4) Repeat the following elementary step until the stack is empty (implying that we have arrived at the rightmost cut).

   (a) Pop the pair $(l_i, l_{i+1})$ from the top of the stack $S$.

   (b) Swap these lines within $C$, the cut (we assume that each line keeps track of its position in the cut).

   (c) Update the horizon trees. (More on this later.)

   (d) Consulting the changed entries in the horizon tree, determine whether there are any new cut edges sharing right endpoints, and if so push them on the stack $S$.

The important unfinished business is to show that we can build the initial UHT and LHT in $O(n)$ time, and to show that, for each elementary step, we can update these trees and all other relevant information in $O(1)$ *amortized time*. By *amortized time* we mean that, even though a single elementary step can take more than $O(1)$ time, the total time needed to perform all $O(n^2)$ elementary steps is $O(n^2)$, and hence the average time for each step is $O(1)$.

This is done by an adaptation of the same incremental "face walking" technique we used in the incremental construction of line arrangements. Let's consider just the UHT, since the LHT is symmetric. To create the initial (leftmost) UHT we insert the lines one by one in decreasing order of slope. Observe that as each new line is inserted it will start above all of the current lines. The uppermost face of the current UHT consists of a convex polygonal chain, see the figure below left. As we trace the newly inserted line from left to right, there will be some point at which it first hits this upper chain of the current UHT. By walking along the chain from left to right, we can determine this intersection point. Each segment that is walked over is never visited again by this initialization process (because it is no longer part of the upper chain), and since the initial UHT has a total of $O(n)$ segments, this implies that the total time spent in walking is $O(n)$. Thus, after the $O(n \log n)$ time for sorting the segments, the initial UHT tree can be built in $O(n)$ additional time.

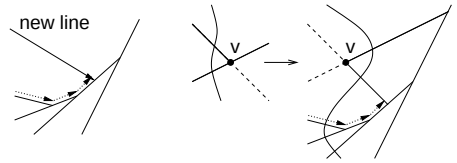



Fig. 127: Constructing and updating the UHT.

Next we show how to update the UHT after an elementary step. The process is quite similar, as shown in the figure right. Let $v$ be the vertex of the arrangement which is passed over in the sweep step. As we pass over $v$, the two edges swap positions along the sweep line. The new lower edge, call it $l$, which had been cut off of the UHT by the previous lower edge, now must be reentered into the tree. We extend $l$ to the left until it contacts an edge of the UHT. At its first contact, it will terminate (and this is the only change to be made to the UHT). In order to find this contact, we start with the edge immediately below $l$ the current cut. We traverse the face of the UHT in counterclockwise order, until finding the edge that this line intersects. Observe that we must eventually find such an edge because $l$ has a lower slope than the other edge intersecting at $v$, and this edge lies in the same face.

**Analysis:** A careful analysis of the running time can be performed using the same amortization proof (based on pebble counting) that was used in the analysis of the incremental algorithm. We will not give the proof in full detail. Observe that because we maintain the set of legal elementary steps in a stack (as opposed to a heap as would

be needed for standard plane sweep), we can advance to the next elementary step in $O(1)$ time. The only part of the elementary step that requires more than constant time is the update operations for the UHT and LHT. However, we claim that the total time spent updating these trees is $O(n^2)$. The argument is that when we are tracing the edges (as shown in the previous figure) we are "essentially" traversing the edges in the *zone* for $L$ in the arrangement. (This is not quite true, because there are edges above $l$ in the arrangement, which have been cut out of the upper tree, but the claim is that their absence cannot increase the complexity of this operation, only decrease it. However, a careful proof needs to take this into account.) Since the zone of each line in the arrangement has complexity $O(n)$, all $n$ zones have total complexity $O(n^2)$. Thus, the total time spent in updating the UHT and LHT trees is $O(n^2)$.

# Lecture 30: Ham-Sandwich Cuts

**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.

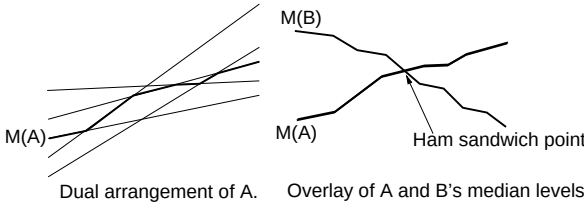



Fig. 128: Ham sandwich: Dual formulation.

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.

# Lecture 31: Shortest Paths and Visibility Graphs

**Shortest paths:** We are given a set of $n$ disjoint polygonal *obstacles* in the plane, and two points $s$ and $t$ that lie outside of the obstacles. The problem is to determine the shortest path from $s$ to $t$ that avoids the interiors of the obstacles. (It may travel along the edges or pass through the vertices of the obstacles.) The complement of the interior of the obstacles is called *free space*. We want to find the shortest path that is constrained to lie entirely in free space.

Today we consider a simple (but perhaps not the most efficient) way to solve this problem. We assume that we measure lengths in terms of Euclidean distances. How do we measure paths lengths for curved paths? Luckily, we do not have to, because we claim that the shortest path will always be a polygonal curve.

**Claim:** The shortest path between any two points that avoids a set of polygonal obstacles is a polygonal curve, whose vertices are either vertices of the obstacles or the points $s$ and $t$.

**Proof:** We show that any path $\pi$ 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, $\pi$ is not shortest, a contradiction.

Thus $\pi$ 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 $\pi$ is not a shortest path.

From this it follows that the edges that constitute the shortest path must travel between $s$ and $t$ and vertices of the obstacles. Each of these edges must have the property that it does not intersect the interior of any obstacle, implying that the endpoints must be visible to each other. More formally, we say that two points $p$ and $q$ are *mutually visible* if the open line segment joining them does not intersect the interior of any obstacle. By this definition, the two endpoints of an obstacle edge are not mutually visible, so we will explicitly allow for this case in the definition below.

**Definition:** The *visibility graph* of $s$ and $t$ and the obstacle set is a graph whose vertices are $s$ and $t$ the obstacle vertices, and vertices $v$ and $w$ are joined by an edge if $v$ and $w$ are either mutually visible or if $(v, w)$ is an edge of some obstacle.

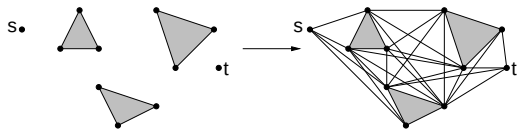



Fig. 129: Visibility graph.

It follows from the above claim that the shortest path can be computed by first computing the visibility graph and labeling each edge with its Euclidean length, and then computing the shortest path by, say, Dijkstra's algorithm (see CLR). Note that the visibility graph is not planar, and hence may consist of $\Omega(n^2)$ edges. Also note that, even if the input points have integer coordinates, in order to compute distances we need to compute square roots, and then sums of square roots. This can be approximated by floating point computations. (If exactness is important, this can really be a problem, because there is no known polynomial time procedure for performing arithmetic with arbitrary square roots of integers.)

**Computing the Visibility Graph:** We give an $O(n^2)$ procedure for constructing the visibility graph of $n$ line segments in the plane. The more general task of computing the visibility graph of an arbitrary set of polygonal obstacles is a very easy generalization. In this context, two vertices are visible if the line segment joining them does not intersect any of the obstacle line segments. However, we allow each line segment to contribute itself as an edge in the visibility graph. We will make the general position assumption that no three vertices are collinear, but this is not hard to handle with some care. The algorithm is *not* output sensitive. If $k$ denotes the number of edges in the visibility graph, then an $O(n \log n + k)$ algorithm does exist, but it is quite complicated.

The text gives an $O(n^2 \log n)$ time algorithm. We will give an $O(n^2)$ time algorithm. Both algorithms are based on the same concept, namely that of performing an angular sweep around each vertex. The text's algorithm operates by doing this sweep one vertex at a time. Our algorithm does the sweep for all vertices simultaneously. We use the fact (given in the lecture on arrangements) that this angular sort can be performed for all vertices in $O(n^2)$ time. If we build the entire arrangement, this sorting algorithm will involve $O(n^2)$ space. However it can be implemented in $O(n)$ space using an algorithm called *topological plane sweep*. Topological plane sweep provides a way to sweep an arrangement of lines using a "flexible" sweeping line. Because events do not need to sorted, we can avoid the $O(\log n)$ factor, which would otherwise be needed to maintain the priority queue.

Here is a high-level intuitive view of the algorithm. First, recall the algorithm for computing trapezoidal maps. We shoot a bullet up and down from every vertex until it hits its first line segment. This implicitly gives us the vertical visibility relationships between vertices and segments. Now, we imagine that angle $\theta$ 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.

Let us consider this "multiple angular sweep" in greater detail.

It is useful to view the problem both in its primal and dual form. For each of the $2n$ segment endpoints $v = (v_a, v_b)$, we consider its dual line $v^* : y = v_a x - v_b$. Observe that a significant event occurs whenever a bullet path in the primal plane jumps from one line segment to another. This occurs when $\theta$ reaches the slope of the line joining two visible endpoints $v$ and $w$. Unfortunately, it is somewhat complicated to keep track of which endpoints are visible and which are not (although if we could do so it would lead to a more efficient algorithm). Instead we will take events to be *all* angles $\theta$ between two endpoints, whether they are visible or not. By duality, the slope of such an event will correspond to the $a$-coordinate of the intersection of dual lines $v^*$ and $w^*$ in the

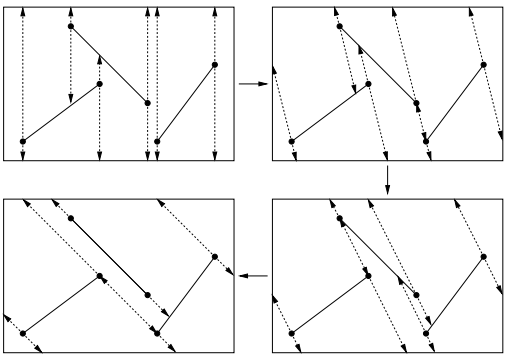


Fig. 130: Visibility graph by multiple angular sweep.

dual arrangement. (Convince yourself of this.) Thus, by sweeping the arrangement of the $2n$ dual lines from left-to-right, we will enumerate all the slope events in angular order.

Next let's consider what happens at each event point. Consider the state of the angular sweep algorithm for some slope $\theta$. For each vertex $v$, there are two bullet paths emanating from $v$ along the line with slope $\theta$. 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 $\theta$ 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 $\theta$ there are exactly two vertices $v$ and $w$ that are involved. Here are the possible scenarios:

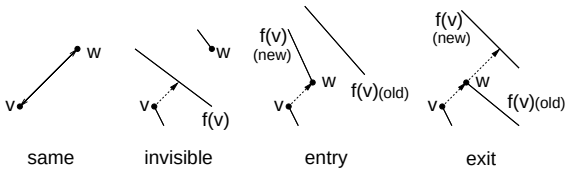



Fig. 131: Possible events.

**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).

Compute the contact point of the bullet path shot from $v$ in direction $\theta$ with segment $f(v)$. If this path hits $f(v)$ strictly before $w$, then we know that $w$ is not visible to $v$, and so this is a "non-event".

**Segment entry:** Consider the segment that is incident to $w$. Either the sweep is just about to enter this segment or is just leaving it. If we are entering the segment, then we set $f(v)$ to this segment.

**Segment exit:** If we are just leaving this segment, then the bullet path will need to shoot out and find the next segment that it hits. Normally this would require some searching. (In particular, this is one of the reasons that the text's algorithm has the extra $O(\log n)$ factor—to perform this search.) However, we claim that the answer is available to us in $O(1)$ time.

In particular, since we are sweeping over $w$ at the same time that we are sweeping over $v$. Thus we know that the bullet extension from $w$ hits $f(w)$. All we need to do is to set $f(v) = f(w)$.

This is a pretty simple algorithm (although there are a number of cases). The only information that we need to keep track of is (1) a priority queue for the events, and (2) the $f(v)$ and $b(v)$ pointers for each vertex $v$. The

priority queue is not stored explicitly. Instead it is available from the line arrangement of the duals of the line segment vertices. By performing a topological sweep of the arrangement, we can process all of these events in $O(n^2)$ time.

# Lecture 32: Motion Planning

**Motion planning:** Last time we considered the problem of computing the shortest path of a point in space around a set of obstacles. Today we will study a much more general approach to the more general problem of how to plan the motion of one or more robots, each with potentially many degrees of freedom in terms of its movement and perhaps having articulated joints.

**Work Space and Configuration Space:** The environment in which the robot operates is called its *work space*, which consists of a set of obstacles that the robot is not allowed to intersect. We assume that the work space is static, that is, the obstacles do not move. We also assume that a complete geometric description of the work space is available to us.

For our purposes, a *robot* will be modeled by two main elements. The first is a *configuration*, which is a finite sequence of values that fully specifies the position of the robot. The second element is the robot's geometric shape description. Combined these two element fully define the robot's exact position and shape in space. For example, suppose that the robot is a 2-dimensional polygon that can translate and rotate in the plane. Its configuration may be described by the $(x, y)$ coordinates of some reference point for the robot, and an angle $\theta$ that describes its orientation. Its geometric information would include its shape (say at some canonical position), given, say, as a simple polygon. Given its geometric description and a configuration $(x, y, \theta)$, 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*.

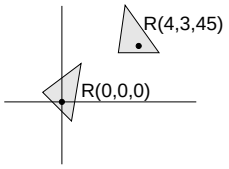



Fig. 132: Configurations of a translating and rotating robot.

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.

Because of limitations on the robot's physical structure and the obstacles, not every point in configuration space corresponds to a legal placement of the robot. Any configuration which is illegal in that it causes the robot to intersect one of the obstacles is called a *forbidden configuration*. The set of all forbidden configurations is denoted $C_{\text{forb}}(\mathcal{R}, S)$, and all other placements are called *free configurations*, and the set of these configurations is denoted $C_{\text{free}}(\mathcal{R}, S)$, or *free space*.

Now consider the *motion planning* problem in robotics. Given a robot $\mathcal{R}$, an work space $S$, and initial configuration $s$ and final configuration $t$ (both points in the robot's free configuration space), determine (if possible)

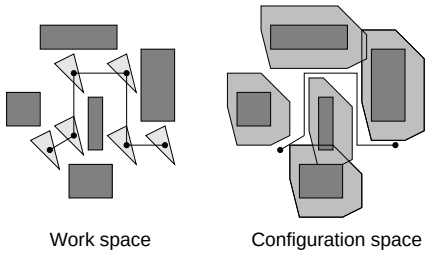



Fig. 133: Work space and configuration space.

a way to move the robot from one configuration to the other without intersecting any of the obstacles. This reduces to the problem of determining whether there is a path from $s$ to $t$ that lies entirely within the robot's free configuration space. Thus, we map the task of computing a robot's motion to the problem of finding a path for a single point through a collection of obstacles.

Configuration spaces are typically higher dimensional spaces, and can be bounded by curved surfaces (especially when rotational elements are involved). Perhaps the simplest case to visualize is that of translating a convex polygonal robot in the plane amidst a collection of polygonal obstacles. In this cased both the work space and configuration space are two dimensional. Consider a reference point placed in the center of the robot. As shown in the figure above, the process of mapping to configuration space involves replacing the robot with a single point (its reference point) and "growing" the obstacles by a compensating amount. These grown obstacles are called *configuration obstacles* or $C$-obstacles.

This approach while very general, ignores many important practical issues. It assumes that we have complete knowledge of the robot's environment and have perfect knowledge and control of its placement. As stated we place no requirements on the nature of the path, but in reality physical objects can not be brought to move and stop instantaneously. Nonetheless, this abstract view is very powerful, since it allows us to abstract the motion planning problem into a very general framework.

For the rest of the lecture we will consider a very simple case of a convex polygonal robot that is translating among a convex set of obstacles. Even this very simple problem has a number of interesting algorithmic issues.

**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.

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,

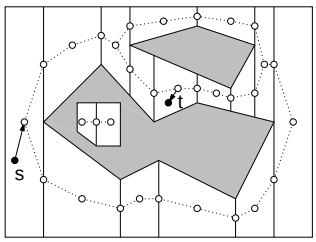



Fig. 134: Motion planning using road maps.

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:** Let us consider how to build a configuration space for a set of polygonal obstacles. We consider the simplest case of translating a convex polygonal robot amidst a collection of convex obstacles. If the obstacles are not convex, then we may subdivide them into convex pieces.

Consider a robot $\mathcal{R}$, whose placement is defined by a translation $\vec{p} = (x, y)$. Let $\mathcal{R}(x, y)$ (also denoted $\mathcal{R}(\vec{p})$) be the placement of the robot with its reference point at $\vec{p}$. Given an obstacle $P$, the *configuration obstacle* is defined as all the placements of $\mathcal{R}$ that intersect $P$, that is

$$\mathcal{CP} = \{\vec{p} \,|\, \mathcal{R}(\vec{p}) \cap P \neq \emptyset\}.$$

One way to visualize $\mathcal{CP}$ is to imagine "scraping" $\mathcal{R}$ along the boundary of $P$ and seeing the region traced out by $\mathcal{R}$'s reference point.

The problem we consider next is, given $\mathcal{R}$ and $P$, compute the configuration obstacle $\mathcal{CP}$. To do this, we first introduce the notion of a *Minkowski sum*. Let us violate our notions of affine geometry for a while, and think of points $(x, y)$ in the plane as vectors. Given any two sets $S_1$ and $S_2$ in the plane, define their *Minkowski sum* to be the set of all pairwise sums of points taken from each set:

$$S_1 \oplus S_2 \;=\; \{\vec{p} + \vec{q} \,|\, \vec{p} \in S_1,\;\; \vec{q} \in S_2\}.$$

Also, define $-S = \{-\vec{p} \,|\, \vec{p} \in S\}$. Observe that for the case of a translating robot, we can define $\mathcal{R}(\vec{p})$ as $\mathcal{R} \oplus \vec{p}$.

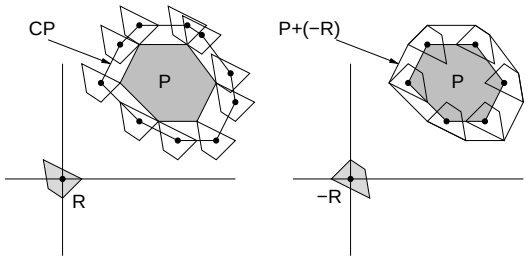



Fig. 135: Configuration obstacles and Minkowski sums.

**Claim:** Given a translating robot $\mathcal{R}$ and an obstacle $P$, $\mathcal{CP} = P \oplus (-\mathcal{R})$.

**Proof:** We show that $\mathcal{R}(\vec{q})$ intersects $P$ if and only if $\vec{q} \in P \oplus (-\mathcal{R})$. Note that $\vec{q} \in P \oplus (-\mathcal{R})$ if and only if there exist $\vec{p} \in P$ and $\vec{r} \in \mathcal{R}$ such that $\vec{q} = \vec{p} - \vec{r}$. Similarly, $\mathcal{R}(\vec{q})(= \mathcal{R} \oplus \vec{q})$ intersects $P$ if and only if there exists points $\vec{r} \in \mathcal{R}$ and $\vec{p} \in P$ such that $\vec{r} + \vec{q} = \vec{p}$. These two conditions are clearly equivalent.

Note that the proof made no use of the convexity of $\mathcal{R}$ or $P$. It works for any shapes and in any dimension. However, computation of the Minkowski sums is most efficient for convex polygons.

It is an easy matter to compute $-\mathcal{R}$ in linear time (by simply negating all of its vertices) the problem of computing the C-obstacle $\mathcal{CP}$ reduces to the problem of computing a Minkowski sum of two convex polygons. We claim that this can be done in $O(m+n)$ time, where $m$ is the number of vertices in $\mathcal{R}$ and $n$ is the number of vertices in $P$. We will leave the construction as an exercise.

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 algorithm for computing $P \oplus \mathcal{R}$, 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 $\mathcal{R}$ that is extreme in this direction. Whenever $\vec{d}$ is perpendicular to an edge of either $P$ or $\mathcal{R}$, 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. The technique of applying one or more angular sweeps to a convex polygon is called the method of *rotating calipers*.

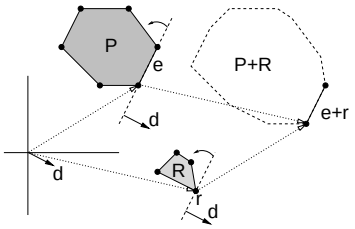



Fig. 136: Computing Minkowski sums.

Assuming $P$ and $\mathcal{R}$ are convex, observe that each edge of $P$ and each edge of $\mathcal{R}$ contributes exactly one edge to $P + \mathcal{R}$. (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 $\mathcal{R}$, with $n$ and $m$ edges respectively, their Minkowski sum $P \oplus \mathcal{R}$ can be computed in $O(n+m)$ time, and consist of at most $n+m$ edges.

**Complexity of Minkowski Sums:** We have shown that free space for a translating robot is the complement of a union of C-obstacles $\mathcal{CP}_i$, each of which is a Minkowski sum of the form $P_i \oplus \mathcal{R}$, where $P_i$ ranges over all the obstacles in the environment. If $P_i$ and $\mathcal{R}$ are polygons, then the resulting region will be a union of polygons. How complex might this union be, that is, how many edges and vertices might it have?

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

**Nonconvex Robot and Nonconvex Obstacles:** Suppose that both $\mathcal{R}$ and $P$ are nonconvex simple polygons. Let $m$ be the number of sides of $\mathcal{R}$ and $n$ be the number of sides of $P$. How many sides might there be in the Minkowski sum $P \oplus \mathcal{R}$ 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:

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

It follows that

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

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 $P \oplus -\mathcal{R}$ 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.)

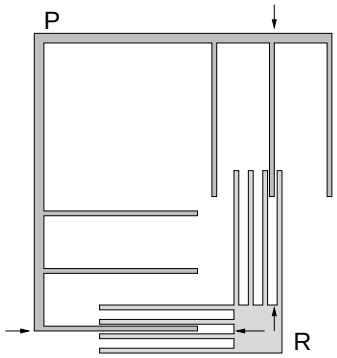



Fig. 137: Minkowski sum of $O(n^2m^2)$ complexity.

As a final observation, notice that the upper bound holds even if $P$ (and $\mathcal{R}$ 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 $P \oplus \mathcal{R}$ 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

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

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

Is $O(nm)$ an attainable bound? The idea is to go back to our analogy of "scraping" $\mathcal{R}$ around the boundary of $P$. Can we arrange $P$ such that most of the edges of $\mathcal{R}$ scrape over most of the $n$ vertices of $P$? Suppose that $\mathcal{R}$

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 $\mathcal{R}$. In this case $\mathcal{R}$ will have many sides scrape across each of the pointy ends of the teeth, implying that the final Minkowski sum will have total complexity $\Omega(nm)$.

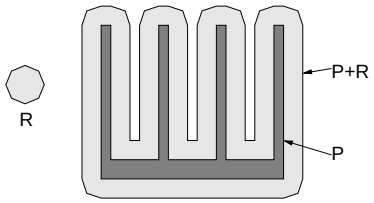



Fig. 138: Minkowski sum of $O(nm)$ complexity.

**The Union of Pseudodisks:** Consider a translating robot given as an $m$-sided convex polygon and a collection of polygonal obstacles having a total of $n$ vertices. We may assume that the polygonal obstacles have been triangulated into at most $n$ triangles, and so, without any loss of generality, let us consider an instance of an $m$-sided robot translating among a set of $n$ triangles. As argued earlier, each C-obstacle has $O(3+m) = O(m)$ sides, for a total of $O(nm)$ line segments. A naive analysis suggests that this many line segments might generate as many as $O(n^2m^2)$ intersections, and so the complexity of the free space can be no larger. However, we assert that the complexity of the space will be much smaller, in fact its complexity will be $O(nm)$.

To show that $O(nm)$ is an upper bound, we need some way of extracting the special geometric structure of the union of Minkowski sums. Recall that we are computing the union of $T_i \oplus \mathcal{R}$, where the $T_i$'s have disjoint interiors. A set of convex objects $\{o_1, o_2, \ldots, o_n\}$ is called a *collection of pseudodisks* if for any two distinct objects $o_i$ and $o_j$ both of the set-theoretic differences $o_i \backslash o_j$ and $o_j \backslash o_i$ are connected. That is, if the objects intersect then they do not "cross through" one another. Note that the pseudodisk property is not a property of a single object, but a property that holds among a set of objects.

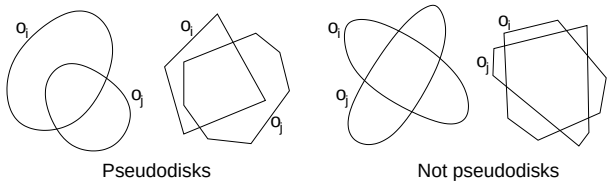



Fig. 139: Pseudodisks.

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

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

is a collection of pseudodisks.

**Proof:** Consider two polygons $T_1$ and $T_2$ with disjoint interiors. We want to show that $T_1 \oplus \mathcal{R}$ and $T_2 \oplus \mathcal{R}$ do not cross over one another.

Given any directional unit vector $\vec{d}$, the *most extreme* point of $\mathcal{R}$ in direction $\vec{d}$ is the point $r \in \mathcal{R}$ that maximizes the dot product $(\vec{d} \cdot r)$. (Recall that we treat the "points" of the polygons as if they were vectors.) The point of $T_1 \oplus \mathcal{R}$ that is most extreme in direction $d$ is the sum of the points $t$ and $r$ that are most extreme for $T_1$ and $\mathcal{R}$, respectively.

Given two convex polygons $T_1$ and $T_2$ with disjoint interiors, they define two outer tangents, as shown in the figure below. Let $\vec{d}_1$ and $\vec{d}_2$ be the outward pointing perpendicular vectors for these tangents. Because

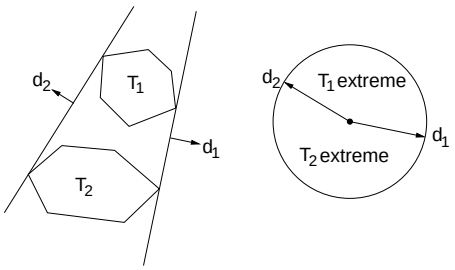



Fig. 140: Alternation of extremes.

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 $T_1 \oplus \mathcal{R}$ and $T_2 \oplus \mathcal{R}$ had a crossing intersection, then observe that we can find points $p_1$ $p_2$, $p_3$, and $p_4$, in cyclic order around the boundary of the convex hull of $(T_1 \oplus \mathcal{R}) \cup (T_2 \oplus \mathcal{R})$ such that $p_1, p_3 \in T_1 \oplus \mathcal{R}$ and $p_2, p_4 \in T_2 \oplus \mathcal{R}$. First consider $p_1$. Because it is on the convex hull, consider the direction $\vec{d_1}$ perpendicular to the supporting line here. Let $r$, $t_1$, and $t_2$ be the extreme points of $\mathcal{R}$, $T_1$ and $T_2$ in direction $\vec{d_1}$, respectively. From our basic fact about Minkowski sums we have

$$p_1 = r + t_1 \qquad p_2 = r + t_2.$$

Since $p_1$ is on the convex hull, it follows that $t_1$ is more extreme than $t_2$ in direction $\vec{d_1}$, that is, $T_1$ is more extreme than $T_2$ in direction $\vec{d_1}$. By applying this same argument, we find that $T_1$ is more extreme than $T_2$ in directions $\vec{d_1}$ and $\vec{d_3}$, but that $T_2$ is more extreme than $T_1$ in directions $\vec{d_2}$ and $\vec{d_4}$. But this is impossible, since from the observation above, there can be at most one alternation in extreme points for nonintersecting convex polygons. See the figure below.

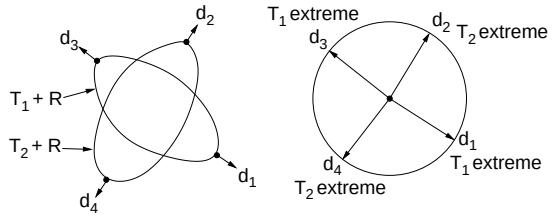



Fig. 141: Proof of Lemma 1.

**Lemma 2:** Given a collection of pseudodisks, with a total of $n$ vertices, the complexity of their union is $O(n)$.

**Proof:** This is a rather cute combinatorial lemma. We are given some collection of pseudodisks, and told that altogether they have $n$ vertices. We claim that their entire union has complexity $O(n)$. (Recall that in general the union of $n$ convex polygons can have complexity $O(n^2)$, by criss-crossing.) The proof is based on a clever charging scheme. Each vertex in the union will be charged to a vertex among the original pseudodisks, such that no vertex is charged more than twice. This will imply that the total complexity is at most $2n$.

There are two types of vertices that may appear on the boundary. The first are vertices from the original polygons that appear on the union. There can be at most $n$ such vertices, and each is charged to itself. The more troublesome vertices are those that arise when two edges of two pseudodisks intersect each other. Suppose that two edges $e_1$ and $e_2$ of pseudodisks $P_1$ and $P_2$ intersect along the union. Follow edge $e_1$ into the interior of the pseudodisk $e_2$. Two things might happen. First, we might hit the endpoint $v$ of this $e_1$ before leaving the interior $P_2$. In this case, charge the intersection to $v$. Note that $v$ can get at most

two such charges, one from either incident edge. If $e_1$ passes all the way through $P_2$ before coming to the endpoint, then try to do the same with edge $e_2$. Again, if it hits its endpoint before coming out of $P_1$, then charge to this endpoint. See the figure below.

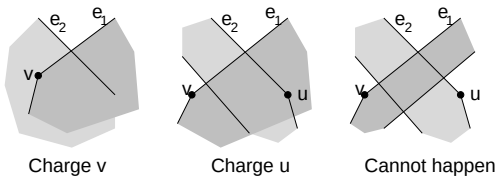



Fig. 142: Proof of Lemma 2.

But what do we do if both $e_1$ shoots straight through $P_2$ and $e_2$ shoots straight through $P_1$? Now we have no vertex to charge. This is okay, because the pseudodisk property implies that this cannot happen. If both edges shoot completely through, then the two polygons must cross over each other.

Recall that in our application of this lemma, we have $n$ C-obstacles, each of which has at most $m + 3$ vertices, for a total input complexity of $O(nm)$. Since they are all pseudodisks, it follows from Lemma 2 that the total complexity of the free space is $O(nm)$.

# Lecture 33: Fixed-Radius Near Neighbors

**Fixed-Radius Near Neighbor Problem:** As a warm-up exercise for the course, we begin by considering one of the oldest results in computational geometry. This problem was considered back in the mid 70's, and is a fundamental problem involving a set of points in dimension $d$. We will consider the problem in the plane, but the generalization to higher dimensions will be straightforward. The solution also illustrates a common class of algorithms in CG, which are based on grouping objects into buckets that are arranged in a square grid.

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

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

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

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

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

The approach described here seems to work only for the reporting case. There is a more efficient solution to the counting problem, but this requires more sophisticated methods.

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

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

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

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

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

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

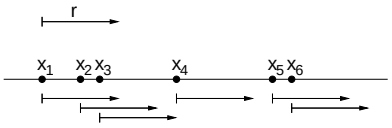



Fig. 143: Fixed radius nearest neighbor on the line.

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

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

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

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

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

We refer to these disjoint intervals as *buckets*. Given the interval $[br, (b+1)r)$, its *bucket index* is the integer $b$. Given any point $x$, it is easy to see that the index of the containing bucket is just $b(x) = \lfloor x/r \rfloor$. Thus, in $O(n)$ time we can associate the $n$ points of $P$ with a set of $n$ integer bucket indices, $b(x)$ for each $x \in P$. Although there are an infinite number of buckets, at most $n$ will be *occupied*, meaning that they contain at least one point of $P$.

There are a number of ways to organize the occupied buckets. They could be sorted, but then we are back to $O(n \log n)$ time. Since bucket indices are integers, a better approach is to store the occupied bucket indices in a *hash table*. Recall from basic data structures that a hash table is a data structure that supports the following operations in $O(1)$ expected time:

**insert**$(o, b)$ : Insert object $o$ with key value $b$. We allow multiple objects to be inserted with the same key.
$L \leftarrow$ **find**$(b)$ : Return a list $L$ of references to objects having key value $b$. This operation takes $O(1 + |L|)$ expected time. If no keys have this value, then an empty list is returned.
**remove**$(o, b)$ : Remove the object indicated by reference $o$, having key value $b$ from the table.

Each point is associated with the key value given by its bucket index $b = \lfloor x/r \rfloor$. Thus in $O(1)$ expected time, we can determine which bucket contains a given point and look this bucket up in the hash table.

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

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

---

Fixed-Radius Near Neighbor on the Line by Bucketing

(1) For each $p \in P$, insert $p$ in the hash table with the key value $b(p)$.

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

  (a) Let $b(p)$ be the bucket containing $p$.

  (b) Enumerate all the points of buckets $b(p)$ and $b(p) + 1$, and for each point $q \in b(p) \cup b(p) + 1$ such that $q \neq p$ and $|q - p| \leq r$, output the pair $(p, q)$.

---

Note that this will output duplicate pairs $(p, q)$ and $(q, p)$. If this bothers you, we could add the additional condition that $q > p$. The key question is determining the time complexity of this algorithm is how many distance computations are performed in step (2b). We compare each point in bucket $b$ with all the points in buckets $b$ and $b + 1$. However, not all of these distance computations will result in a pair of points whose distance is within $r$. Might it be that we waste a great deal of time in performing computations for which we receive no benefit? The lemma below shows that we perform no more than a constant factor times as many distance computations and pairs that are produced.

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

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

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

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

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

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

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

First we bound the total number of distance computations $D$ as a function of $S$. Each point in bucket $b$

computes the distance to every other point in bucket $b$ and every point in bucket $b+1$, and hence

$$\begin{aligned} D &= \sum_{b\in B} n_b(n_b + n_{b+1}) = \sum_{b\in B} n_b^2 + n_b n_{b+1} = \sum_{b\in B} n_b^2 + \sum_{b\in B} n_b n_{b+1} \\ &\leq \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2 + n_{b+1}^2}{2} \\ &= \sum_{b\in B} n_b^2 + \sum_{b\in B} \frac{n_b^2}{2} + \sum_{b\in B} \frac{n_{b+1}^2}{2} = S + \frac{S}{2} + \frac{S}{2} = 2S. \end{aligned}$$

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

Therefore we have

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

which completes the proof.

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

A worthwhile exercise to consider at this point is the issue of the bucket width $r$. How would changing the value of $r$ affect the implementation of the algorithm and its efficiency? For example, if we used buckets of size $r/2$ or $2r$, would the above algorithm (after suitable modifications) have the same asymptotic running time? Would buckets of size any constant times $r$ work?

**Generalization to $d$ dimensions:** This bucketing algorithm is easy to extend to multiple dimensions. For example, in dimension 2, we bucket points into a square grid in which each grid square is of side length $r$. (As before, you might consider the question of what values of bucket sizes lead to a correct and efficient algorithm.) The bucket index of a point $p : (p_x, p_y)$ is a pair $B(p) = (b(p_x), b(p_y)) = (\lfloor p_x/r \rfloor, \lfloor p_y/r \rfloor)$. We apply a hash function that accepts two arguments. To generalize the algorithm, for each point we consider the points in its surrounding $3 \times 3$ subgrid of buckets. The result is shown in the following code fragment.

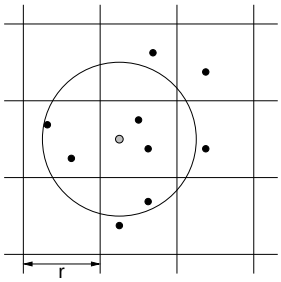



Fig. 144: Fixed radius nearest neighbor on the plane.

Fixed-Radius Near Neighbor in the Plane

(1) For each $p = (p_x, p_y)$ of $P$, insert $p$ in the hash table with the (2-dimensional) bucket index $B(p) = (b(p_x), b(p_y))$.

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

(a) Let $B(p) = (b(p_x), b(p_y))$ be the bucket index for $p$.

(b) Enumerate all the points of buckets $(b(p_x) + i, b(p_y) + j)$, for $i, j \in \{-1, 0, +1\}$. For each such point $q$, if $\|pq\| \leq r$, output the pair $(p, q)$.

By generalizing the analysis used in the 1-dimensional case, it can be shown that the algorithm's expected running time is $O(n + k)$. The details are left as an exercise (we just have more terms to consider, but each cell is involved with at most 9 other cells which is absorbed into the constant factor hidden by the O-notation).

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

# Lecture 34: Multidimensional Polytopes and Convex Hulls

**Polytopes:** Today we consider convex hulls in dimensions 3 and higher. Although dimensions greater than 3 may seem rather esoteric, we shall see that many geometric optimization problems can be stated as some search over a polytope in $d$-dimensional space, where $d$ may be greater than 3.

Before delving into this, let us first present some basic terms. We define a *polytope* (or more specifically a $d$-polytope) to be the convex hull of a finite set of points in $\mathbb{R}^d$. We say that a set of $k$ points is *affinely independent* if no one point can be expressed as an affine combination (that is, a linear combination whose coefficients sum to 1) of the others. For example, three points are affinely independent if they are not on the same line, four points are affinely independent if they are not on the same plane, and so on. The convex hull of $k + 1$ affinely independent points is called a *simplex* or *$k$-simplex*. For example, the line segment joining two points is a 1-simplex, the triangle defined by three points is a 2-simplex, and the tetrahedron defined by four points is a 3-simplex.

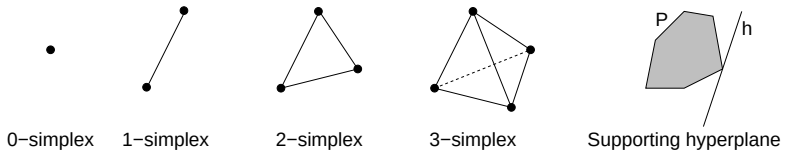



Fig. 145: Simplices and supporting hyperplane.

Any $(d-1)$-dimensional hyperplane $h$ in $d$-dimensional space divides the space into (open) halfspaces, denoted $h^-$ and $h^+$, so that $\mathbb{R}^d = h^- \cup h \cup h^+$. Let us define $\overline{h^-} = h^- \cup h$ and $\overline{h^+} = h^+ \cup h$ to be the closures of these halfspaces. We say that a hyperplane *supports* a polytope $P$ (and is called a *supporting hyperplane* of $P$) if $h \cap P$ is nonempty and $P$ is entirely contained within either $\overline{h^-}$ or $\overline{h^+}$. The intersection of the polytope and any supporting hyerplane is called a *face* of $P$. Faces are themselves convex polytopes of dimensions ranging from 0 to $d - 1$. The 0-dimensional faces are called *vertices*, the 1-dimensional faces are called *edges*, and the $(d-1)$-dimensional faces are called *facets*. (Note: When discussing polytopes in dimension 3, people often use the term "face" when they mean "facet". It is usually clear from context which meaning is intended.)

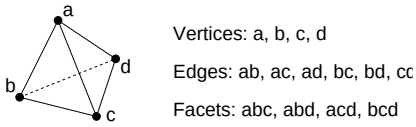



Fig. 146: A tetrahedron and its proper faces.

The faces of dimensions 0 to $d - 1$ are called *proper faces*. It will be convenient to define two additional faces.

The empty set is said to be a face of dimension $-1$ and the entire polytope is said to be a face of dimension $d$. We will refer to all the faces, including these two additional faces as the *improper faces* of the polytope.

There are a number of facts that follow from these definitions.

- The boundary of a polytope is the union of its proper faces.
- A polytope has a finite number of faces. Each face is a polytope.
- A polytope is the convex hull of its vertices.
- A polytope is the intersection of a finite number of closed halfspaces. (Note that the converse need not be true, since the intersection of halfspaces may generally be unbounded. Such an unbounded convex body is either called a *polyhedron* or a *unbounded polytope*.)

Observe that a $d$-simplex has a particularly regular face structure. If we let $v_0, v_1, v_2, \ldots, v_d$ denote the vertices of the simplex, then for each 2-element set $\{v_i, v_j\}$ there is an edge of the simplex joining these vertices, for each 3-element set $\{v_i, v_j, v_k\}$ there is a 3-face joining these these three vertices, and so.

**Fact:** The number of $j$-dimensional faces on a $d$-simplex is equal to the number $(j+1)$-element subsets of domain of size $d+1$, that is,

$$\binom{d+1}{j+1} = \frac{(d+1)!}{(j+1)!(d-j)!}.$$

**Incidence Graph:** How are polytopes represented? In addition to the geomtric properties of the polytope (e.g., the coordinates of its vertices or the equation of its faces) it is useful to store discrete connectivity information, which is often referred to as the *topology* of the polytope. There are many representations for polytopes. In dimension 2, a simple circular list of vertices suffices. In dimension 3, we need some sort of graph structure. There are many data structures that have been proposed. They are evaluated based on the ease with which the polytope can be traversed and the amount of storage needed. (Examples include the *winged-edge*, *quad-edge*, and *half-edge* data structures. We may discuss these later in the semester.)

A useful structure for polytopes in arbitrary dimensions is called the *incidence graph*. Each node of the incidence graph corresponds to an (improper) face of the polytope. We create an edge between two faces if their dimension differs by 1, and one (of lower dimension) is contained within the other (of higher dimension). An example is shown in Fig. 147 below for a simplex. Note the similarity between this graph and the lattice of subsets based on inclusion relation.

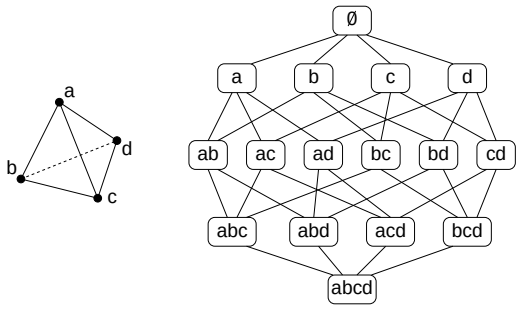



Fig. 147: The incidence graph for a tetrahedron.

**Polarity:** There are two natural ways to create polytopes. One is as the convex hull of a set of points and the other is as the intersection of a collection of closed halfspaces (assuming it is bounded). These two concepts are essentially identical, and this can be observed by the polar transformation, which maps points to hyperplanes and vice versa. Fix any point $O$ in $d$-dimensional space. We may think of $O$ as the origin, and therefore, any

point $p \in \mathbb{R}^d$ can be viewed as a $d$-element vector. (If $O$ is not the origin, then $p$ can be identified with the vector $p - O$.) The *polar hyperplane* of $p$, denoted $p^*$ is defined by

$$p^* = \{x \in \mathbb{R}^d \mid (p \cdot x) = 1\},$$

where the expression $(p \cdot x)$ is just the standard vector ***dot-product*** ($(p \cdot x) = p_1x_1 + p_2x_2 + \cdots + p_dx_d$). Observe that if $p$ is on the unit sphere centered about $O$, then $p^*$ is a hyperplane that passes through $p$ and is orthogonal to the vector $\overline{Op}$. As we move $p$ away from the origin along this vector, the dual hyperplane moves closer to the origin, and vice versa, so that the product of their distances from the origin is always 1.

Now, let $h$ be any hyperplane that does not contain $O$. The *pole* of $h$, denoted $h^*$ is the point that satisfies

$$(h^* \cdot x) = 1 \qquad \text{for all } x \in h.$$

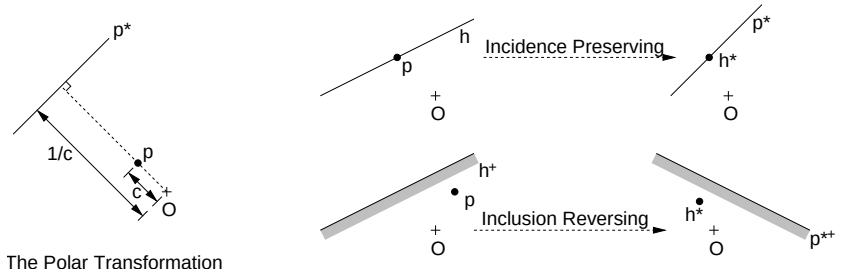



Fig. 148: The polar transformation and its properties.

Clearly this double transformation is an involution, that is, $(p^*)^* = p$ and $(h^*)^* = h$. The polar transformation preserves important geometric relationships. Given a hyperplane $h$, define

$$h^+ = \{x \in \mathbb{R}^d \mid (x \cdot h^*) < 1\} \qquad h^- = \{x \in \mathbb{R}^d \mid (x \cdot h^*) > 1\}.$$

That is, $h^+$ is the open halfspace containing the origin and $h^-$ is the other open halfspace for $h$.

**Claim:** Let $p$ be any point in $\mathbb{R}^d$ and let $h$ be any hyperplane in $\mathbb{R}^d$. The polar transformation satisfies the following two properties.

**Incidence preserving:** The polarity transformation preserves incidence relationships between points and hyerplanes. That is, $p$ belongs to $h$ if and only if $h^*$ belongs to $p^*$.

**Inclusion Reversing:** The polarity transformation reverses relative position relationships in the sense that $p$ belongs to $h^+$ if and only if $h^*$ belongs to $(p^*)^+$, and $p$ belongs to $h^-$ if and only if $h^*$ belongs to $(p^*)^-$.

In general, any bijective transformation that preserves incidence relations is called a *duality*. The above claim implies that polarity is a duality.

We can now formalize the aforementioned notion of polytope equivalence. The idea will be to transform a polytope defined as the convex hull of a finite set of points to a polytope defined as the intersection of a finite set of closed halfspaces. To do this, we need a way of mapping a point to a halfspace. Our approach will be to take the halfspace that contains the origin. For any point $p \in \mathbb{R}^d$ define the following closed halfspace based on its polar:

$$p^{\#} = \overline{p^{*+}} = \{x \in \mathbb{R}^d \mid (x \cdot p) \leq 1\}.$$

(The notation is ridiculous, but this is easy to parse. First consider the polar hyperplane of $p$, and take the closed halfspace containing the origin.) Observe that if a halfspace $h^+$ contains $p$, then by the inclusion-reversing property of polarity, the polar point $h^*$ is contained within $p^{\#}$.

Now, for any set of points $P \subseteq \mathbb{R}^d$, we define its *polar image* to be the intersection of these halfspaces

$$P^{\#} = \{x \in \mathbb{R}^d \mid (x \cdot p) \leq 1, \forall p \in P\}.$$

Thus $P^{\#}$ is the intersection of an (infinite) set of closed halfspaces, one for each point $p \in P$. A halfspace is convex and the intersect of convex sets is convex, so $P^{\#}$ is a convex set.

To see the connection with convex hulls, let $S = \{p_1, \ldots, p_n\}$ be a set of points and let $P = conv(S)$. Let us assume that the origin $O$ is contained within $P$. (We can guarantee this in a number of ways, e.g., by translating $P$ so its center of mass coincides with the origin.) By definition, the convex hull is the intersection of the set of all closed halfspaces that contain $S$. That is, $P$ is the intersect of an infinite set of closed halfspaces. What are these halfspaces? If $h^+$ is a halfspace that contains all the points of $S$, then by the inclusion-reversing property of polarity, the polar point $h^*$ is contained within all the hyperplanes $\overline{p_i^{*+}}$, which implies that $h^* \in P^{\#}$. This means that, through polarity, the halfspaces whose intersection is the convex hull of a set of points is essentially equivalent to the polar points that lie within the polar image of the convex hull.

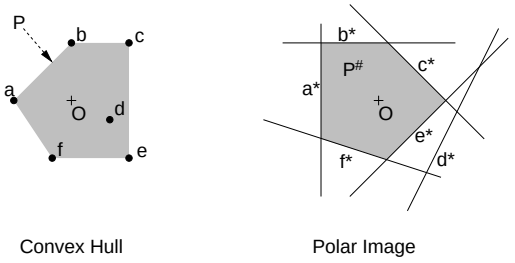



Fig. 149: The polar image of a convex hull.

**Lemma:** Let $S = \{p_1, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$ and let $P = conv(S)$. Then its polar image is the intersection of the corresponding polar halfspaces, that is,

$$P^{\#} = \bigcap_{i=1}^{n} \overline{p_i^{*+}}.$$

Furthermore:

(i) A point $a \in \mathbb{R}^d$ lies on the boundary of $P$ if and only if the polar hyperplane $a^*$ supports $P^{\#}$.

(ii) Each $k$-face of $P$ corresponds to a $(d-1-k)$-face of $P^{\#}$ and given faces $f_1$, $f_2$ of $P$ where $f_1 \subseteq f_2$, the corresponding faces $f_1^{\#}$, $f_2^{\#}$ of $P^{\#}$ satisfy $f_1^{\#} \supseteq f_2^{\#}$. (That is, inclusion relations are reversed.)

It is not hard to prove that the polar image of a polytope is an involution, that is $(P^{\#})^{\#} = P$. (See Boissonnat and Yvinec for proofs of all these facts.)

Thus, the polar image $P^{\#}$ of a polytope is structurally isomorphic to $P$ and all affine relations on $P$ map through polarity to $P^{\#}$. From a computational perspective, this means that we compute the polar of all the points of $P$, consider the halfspaces that contain the origin, and take the intersection of these halfspaces. Thus, the problems of computing convex hulls and computing the intersection of halfspaces are computationally equivalent. (In fact, once you have computed the incidence graph for one, you just flip it "upside-down" to get the other.)

For example, if you know your *Platonic solids* (tetrahedron, cube, octahedron, dodecahedron, and icosahedron), you may remember that the square and octahedron are polar duals, the dodecahedron and icosohedron are polar duals, and the tetrahedron is self-dual.

**Simple and Simplicial Polytopes:** Observe that if a polytope is the convex hull of a set of points in general position, then for $0 \leq j \leq d-1$, each $j$-face is a $j$-simplex. A polytope is *simplicial* if all its proper faces are simplices.

If we take a dual view, consider a polytope that is the intersection of a set of $n$ halfspaces in general position. Then each $j$-face is the intersection of exactly $(d-j)$ hyperplanes. A polytope is said to be *simple* if each $j$-face is the intersection of exactly $(d-j)$-hyperplanes. In particular, this implies that each vertex is incident to exactly $d$ facets. Further, each $j$-face can be uniquely identified with a subset of $d-j$ hyperplanes, whose intersection defines the face. Following the same logic as in the previous paragraph, it follows that the number of vertices in such a polytope is naively at most $O(n^d)$. (Again, we'll see later that the tight bound is $O(n^{\lfloor d/2 \rfloor})$.) It follows from the results on polarity that a polytope is simple if any only if its polar is simplicial.

An important observation about simple polytopes is that the local region around each vertex is equivalent to a vertex of a simplex. In particular, if we cut off a vertex of a simple polytope by a hyperplane that is arbitrarily close to the vertex, the piece that has been cut off is a $d$-simplex.

It easy to show that among all polytopes having a fixed number of vertices, simplicial polytopes maximize the number of faces of all higher degrees. (Observe that otherwise there must be degeneracy among the vertices. Perturbing the points breaks the degeneracy, and will generally split faces of higher degree into multiple faces of lower degree.) Dually, among all polytopes having a fixed number of facets, simple polytopes maximize the number of faces of all lower degrees.

Another observation allows us to provide crude bounds on the number of faces of various dimensions. Consider first a simplicial polytope having $n$ vertices. Each $(j-1)$-face can be uniquely identified with a subset of $j$ points whose convex hull gives this face. Of course, unless the polytope is a simplex, not all of these subsets will give rise to a face. Nonetheless this yields the following naive upper bound on the numbers of faces of various dimensions. By applying the polar transformation we in fact get two bounds, one for simplicial polytopes and one for simple polytopes.

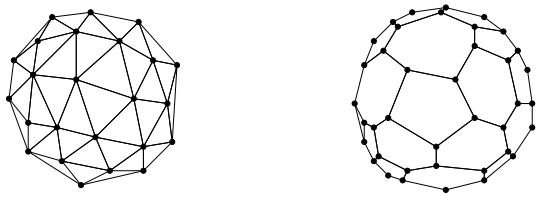



Fig. 150: Simplicial and simple polytopes.

**Lemma:** (Naive bounds)

(i) The number faces of dimension $j$ of a polytope with $n$ vertices is at most $\binom{n}{j+1}$.

(ii) The number of faces of dimension $j$ of a polytope with $n$ facets is at most $\binom{n}{d-j}$.

These naive bounds are not tight. Tight bounds can be derived using more sophisticated relations on the numbers of faces of various dimensions, called the *Dehn-Sommerville relations*. We will not cover these, but see the discussion below of the Upper Bound Theorem.

**The Combinatorics of Polytopes:** Let $P$ be a $d$-polytope. For $-1 \leq k \leq d$, let $n_k(P)$ denote the number of $k$-faces of $P$. Clearly $n_{-1}(P) = n_d(P) = 1$. The numbers of faces of other dimensions generally satisfy a number of combinatorial relationships. The simplest of these is called *Euler's relation*:

**Theorem:** (Euler's Relation) Given any $d$-polytope $P$ we have $\sum_{k=-1}^{d}(-1)^k n_k(P) = 0$.

This says that the alternating sum of the numbers of faces sums to 0. For example, a cube has 8 vertices, 12 edges, 6 facets, and together with the faces of dimension $-1$ and $d$ we have

$$-1 + 8 - 12 + 6 - 1 = 0.$$

Although the formal proof of Euler's relation is rather complex, there is a very easy way to see why its true. First, consider the simplest polytope, namely a $d$-simplex, as the base case. (This is easy to see if you recall that for a simplex $n_j = \binom{d+1}{j+1}$. If you take the expression $(1-1)^{d+1}$ and expand it symbolically (as you would for example for $(a+b)^2 = a^2 + 2ab + b^2$) you will get exactly the sum in Euler's formula. Clearly $(1-1)^{d+1} = 0$. The induction part of the proof comes by observing that in order making a complex polytope out of a simple one, essentially involves a series of splitting operation. Every time you split a face of dimension $j$, you do so by adding a face of dimension $j-1$. Thus, $n_{j-1}$ and $n_j$ each increase by one, and so the value of the alternating sum is unchanged.

Euler's relation can be used to prove that the convex hull of a set of $n$ points in 3-space has $O(n)$ edges and $O(n)$ faces. However, what happens as dimension increases? We will prove the following theorem. The remarkably simple proof is originally due to Raimund Seidel. We will state the theorem both in its original and dual form.

**The Upper Bound Theorem:** A polytope defined by the convex hull of $n$ points in $\mathbb{R}^d$ has $O(n^{\lfloor d/2 \rfloor})$ facets.

**Upper Bound Theorem (Polar Form):** A polytope defined by the intersection of $n$ halfspaces in $\mathbb{R}^d$ has $O(n^{\lfloor d/2 \rfloor})$ vertices.

**Proof:** It is not hard to show that among all polytopes, simplicial polytopes maximize the number of faces for a given set of vertices and simple polytopes maximize the number of vertices for a given set of faces. We will prove just the polar form of the theorem, and the other will follow by polar equivalence.

Consider a polytope defined by the intersection of $n$ halfspaces in general position. Let us suppose by convention that the $x_d$ axis is the vertical axis. Given a face, its highest vertex and lowest vertices are defined as those having the maximum and minimum $x_d$ coordinates, respectively. (There are no ties if we assume general position.)

The proof is based on a charging argument. We will place a charge at each vertex. We will then move the charge for each vertex to a specially chosen incident face, in such a way that no face receives more than two charges. Finally, we will show that the number of faces that receive charges is at most $O(n^{\lfloor d/2 \rfloor})$.

First, we claim that every vertex $v$ is either the highest or lowest vertex for a $j$-face, where $j \geq \lceil d/2 \rceil$. To see this, recall that the for a simple polytope, the neighborhood immediately surrounding any vertex is isomorphic to a simplex. Thus, $v$ is incident to exactly $d$ edges (1-faces). (See Fig. 151 for an example in dimension 5.) Consider a horizontal (that is, orthogonal to $x_d$) hyperplane passing through $v$. Since there are $d$ edges in all, at least $\lceil d/2 \rceil$ of these edges must lie on the same side of this hyperplane. (By general position we may assume that no edge lies exactly on the hyperplane.)

As we observed earlier in the lecture, the local neighborhood about each vertex of a simple polytope is isomorphic to a simplex, which imples that there is a face of dimension at least $\lceil d/2 \rceil$ that spans these edges and is incident to $v$. Therefore, $v$ is the lowest or highest vertex for this face. We charge this face for the charge on vertex $v$. Thus, we may charge every vertex of the polytope to face of dimension at least $\lceil d/2 \rceil$, and every such face will be charged at most twice (once by its lowest and once by its highest vertex).

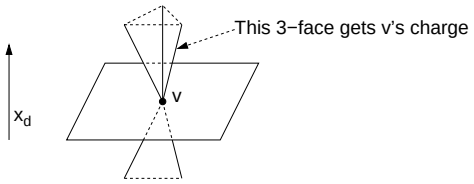



Fig. 151: Proof of the Upper Bound Theorem in dimension 5. In this case the three edges above $v$ span a 3-face whose lowest vertex is $v$.

All that remains is to count the number of faces that have been charged and multiply by 2. Recalling our earlier lemma on the naive bound on the number of $j$-faces of a simple polytope with $n$ facets is $\binom{n}{d-j}$.

(Each $j$-face is arises from the intersection of $d-j$ hyperplanes and this is number of $(d-j)$-element subsets of hyerplanes.) Summing this up over all the faces of dimension $\lceil d/2 \rceil$ and higher we find that the number of vertices is at most

$$2 \sum_{j=\lceil d/2 \rceil}^{d} \binom{n}{d-j}.$$

By changing the summation index to $k = d - j$ and making the observation that $\binom{n}{k}$ is $O(n^k)$, we have that the number of vertices is at most

$$2 \sum_{k=0}^{\lfloor d/2 \rfloor} \binom{n}{k} = \sum_{k=0}^{\lfloor d/2 \rfloor} O(n^k).$$

This is a geometric series, and so is dominated asymptotically by its largest term. Therefore it follows that the number of charges, that is, the number of vertices is at most

$$O\left(n^{\lfloor d/2 \rfloor}\right),$$

and this completes the proof.

Is this bound tight? Yes it is. There is a family of polytopes, called *cyclic polytopes*, which match this asymptotic bound. (See Boissonnat and Yvinec for a definition and proof.)

# Lecture 35: Planar Graphs, Polygons and Art Galleries

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

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

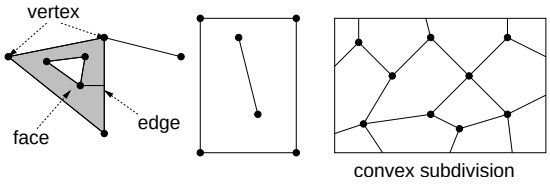



Fig. 152: Planar straight-line subdivision.

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

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

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

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

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

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

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

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

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

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

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

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

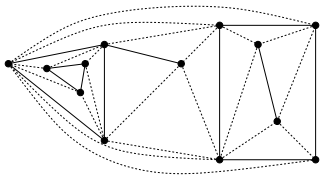


Fig. 153: Triangulating a planar graph.

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

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

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

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

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

This completes the proof.

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

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

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

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

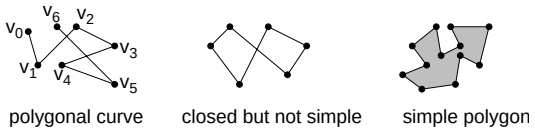



Fig. 154: Polygonal curves

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

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

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

Before getting into a solution, let's consider some basic facts. Could there be polygons for which no finite number of guards suffice? It turns out that the answer is no, but the proof is not immediately obvious. You

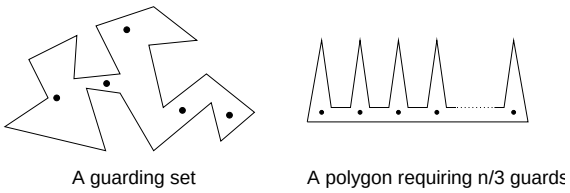



Fig. 155: Guarding sets.

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

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

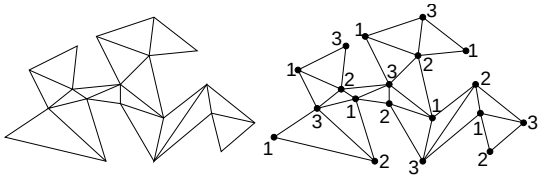


Fig. 156: Polygon triangulation and a 3-coloring.

**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.)

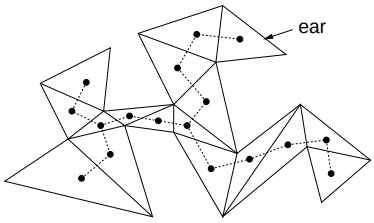



Fig. 157: Dual graph of triangulation.

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

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

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