CMSC 754
Computational Geometry[1]

David M. Mount
Department of Computer Science
University of Maryland
Spring 2026

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. It's 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 grew rapidly in the late 70's and through the 80's and 90's, and it is still a very active field of research. 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 often assumed that the dimension of the space is a constant, and running times may conceal exponential factors in the dimension. Recent work in this area has considered problems in high dimensional spaces, where running times are required to be polynomial both in the input size and in the dimension. In this course, our focus will be largely on problems in 2-dimensional space, with occasional forays into spaces of higher dimensions.

Discrete Geometry: Because the field was developed by researchers whose training was in discrete algorithms (as opposed to more continuous areas such as a numerical analysis, integral geometry, or differential geometry) the field has also focused principally on the *discrete* aspects of geometric problem solving. The mixture of discrete and geometric elements gives rise to an abundance of interesting questions. (For example, given a collection of n points in the plane, how many pairs of points can there be that are exactly one unit distance apart from each other?) Another distinctive feature is that computational geometry 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.

Computational geometry finds applications in numerous areas of science and engineering. 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 algorithms and *theoretical analytic tools* needed to analyze the performance of these algorithms. 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.

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

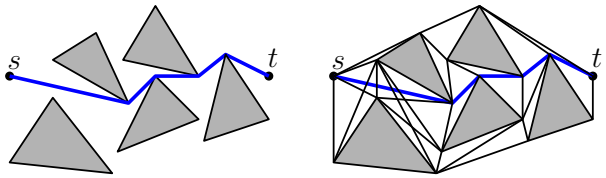


Fig. 1: Shortest path problem.

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.

Features of Computational Geometry: As is the case with any field of science, computational geometry owes its structure to the interests and expertise of its founders in combination with the gradual evolution over time from the members of this research community. Here are a number of features of the field.

Rigor: Prior to computational geometry, there were many *ad hoc* solutions to geometric computational problems, some efficient, some inefficient, and some simply incorrect. From its beginnings, computational geometry has focused on mathematical rigor in its approach to problems. This field has made great strides in establishing correct, provably efficient algorithmic solutions to many fundamental geometric problems.

Correctness/Robustness: Many classical geometric software systems have been 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 exactly due to floating point rounding errors.) Software that is based on discrete decisions, each of which is subject to some error, is inherently unreliable. Much effort has been devoted in computational geometry to the robust and correct computing of geometric primitives, thus forming a solid foundation for geometric computations.

Emphasis on Discreteness: Many applications of geometric computing involve quantities that are naturally continuous, such as the 3-dimensional scalar field representing the air temperature in a room. Since computers and computation is inherently "finite" in nature, solving such problems requires discretizing them. Fields like *computational fluid dynamics* involve computing approximate solutions to such continuous problems. In contrast, the field of computational geometry has focused on problems that are naturally discrete in nature (e.g., involving a finite set of points) as opposed to discretizations of continuous problems.

Combinatorial Geometry: The asymptotic analysis of algorithms in discrete geometry has inspired a great deal of new work in the topic of *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.

Flatness: Fields such as solid modeling in industrial design deal naturally with curved objects, like the body of an automobile. In contrast, computational geometry typically deals with flat objects, such as points, lines, planes, hyperplanes, and sets composed from these (such as line segments, polygons, and polytopes). Exceptions are usually fairly simple, such as Euclidean balls and algebraic surfaces of constant degree, such as ellipsoids.

Low Dimensionality: It is typically assumed in computational geometry that inputs are drawn from spaces of constant dimension. Many algorithms work only in 2-dimensional or 3-dimensional space. Many algorithms that work in spaces of higher dimension suffer from the so-called "curse of dimensionality," where there are factors that grow exponentially with dimension. While there are notable examples, the majority of work in the field has this limitation.

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).

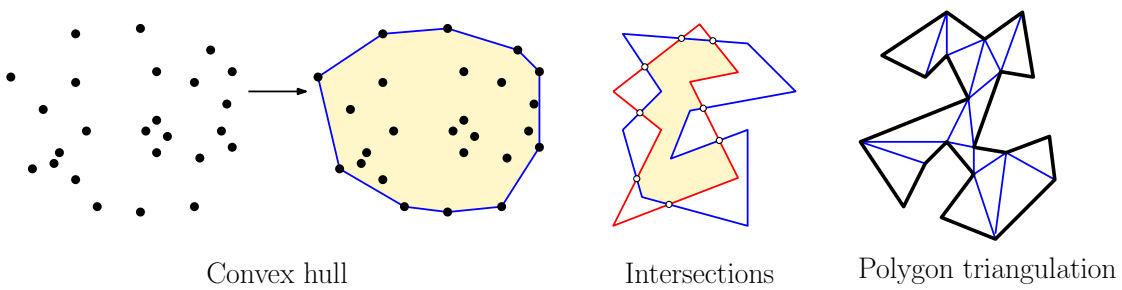


Fig. 2: Convex hulls, intersections, and polygon triangulation.

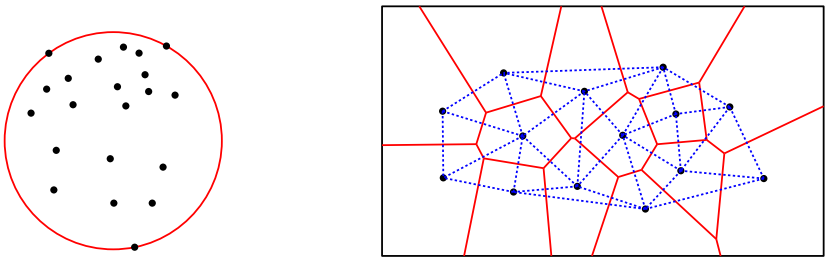
Intersections: One of the most basic geometric problems is that of determining when two sets of objects intersect one another. Determining whether complex objects intersect often reduces to determining which individual pairs of primitive entities (e.g., line segments) intersect (see Fig. 2). We will 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 (see Fig. 2). 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.

Optimization and Linear Programming: Many optimization problems in computational geometry can be stated in the form of *linear programming*, namely, finding 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 that can be posed as low dimensional linear programming problems or variants thereof. One example is computing the smallest

circular disk that encloses a set of points (see Fig. 3). 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.



Smallest enclosing disk Voronoi diagram and Delaunay triangulation

Fig. 3: Voronoi diagram and Delaunay triangulation.

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.

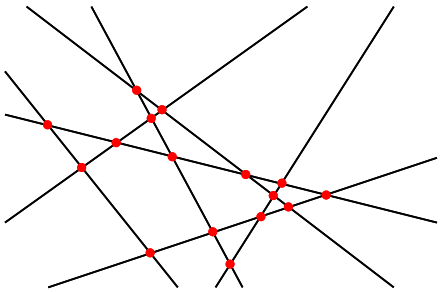


Fig. 4: An arrangement of lines in the plane.

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 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, disk, or polygonal shape, like a triangle.

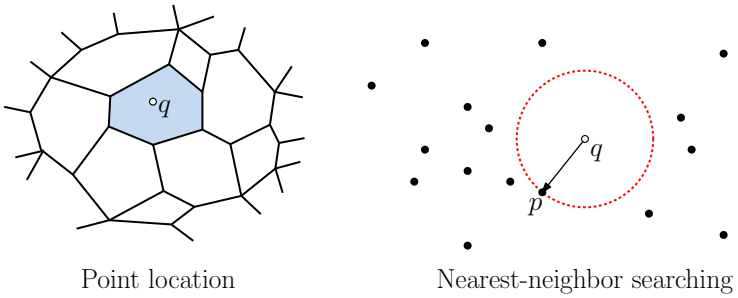


Fig. 5: Geometric search problems. The point-location query determines the triangle containing q. The nearest-neighbor query determines the point p that is closest to q.

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 Δ 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 ε is a fixed constant. As ε approaches zero, the running time increases.

...and more: The above examples are just a small number of numerous types of problems that are considered in computational geometry. Throughout the semester we will be exploring these and many others.

Computational Model: (Optional) We should say a few words about the model of computation that we will be using throughout in this course. It is called the *real RAM*. The "real" refers to real numbers (not the realism of the model!) and RAM is an acronym for "random-access machine", which distinguishes it from other computational models, like Turing machines, which assume memory is stored on tape. The real RAM is a mathematical model of computation in which it is possible to perform arithmetic (addition, subtraction, multiplication, and division, and comparisons) exactly with real numbers in constant time.

Why should we care? As an example, consider a geometric version of a famous optimization problem, known as the *Traveling Salesperson Problem* (TSP). We are given a set of n points in the plane, and we wish to compute the circuit of minimum total Euclidean distance that visits all these points. Ignoring the difficulty of this combinatorial problem (which is NP-Hard), consider the (apparently) much simpler question of whether one tour is shorter than another. (Consider, for example, the blue and red tours shown in Fig. 6.) Clearly, if we are going to solve the general problem, we must be able to at least compare the relative lengths of two tours.

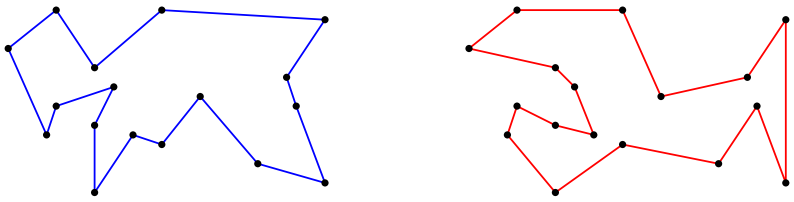


Fig. 6: Comparing the length of two TSP tours on the same point set.

While this problem can be solved approximately using floating point computations, it is quite difficult to solve the problem exactly. To see why, observe that the Euclidean distance between two points involves computing square roots. Two square roots can be compared by squaring both sides to eliminate the square root, thus reducing the computation to one involving simple arithmetic operations. Unfortunately, when you attempt this with a sequence of n numbers, each of which is a square root, many squarings upon squarings must be performed. The sizes of the numbers (in terms of the number of bits required) grows exponentially. In practice, these computations would be impossible. But in the Real RAM model, each such computation can be computed in constant time each.

In spite of the unusual power of this model, it is possible to simulate this model of computation for many common geometric computations. This is done through the use of so-called *floating-point filters*, which dynamically determine the degree of accuracy required in order to resolve comparisons exactly. The CGAL libarary supports exact geometric computations through this mechanism.

Lecture 2: Convex Hulls 1: Graham's Scan and Other Algorithms

Convex Hulls: In this lecture we will 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. 7).

The (planar) *convex hull problem* is, given a discrete set of n points P in the plane, output a representation of P's convex hull. We may assume the points are given as a list of (x, y) coordinates. The convex hull is a closed convex polygon, the simplest representation is a cyclic (say, counterclockwise) enumeration of the vertices of the convex hull. In higher dimensions, the convex hull will be a convex polytope. We will discuss the representation of polytopes in future lectures, but in 3-dimensional space, the representation would consist of a vertices, edges, and faces that constitute the boundary of the polytope.

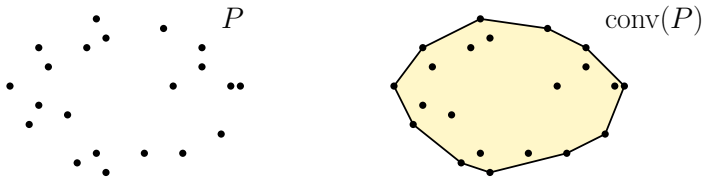


Fig. 7: A point set and its convex hull.

Applications: 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 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.

Basic Definitions: Before getting to the discussion of the various convex-hull algorithms, let's begin with a few standard definitions, which will be useful throughout the semester. For any $d \geq 1$, let $\mathbb{R}^d$ denote real d-dimensional space, that is, the set of d-dimensional vectors over the real numbers. We refer to elements of $\mathbb{R}^d$ either as "points" or "vectors", depending on what we intend them to represent (a location in space or a displacement, respectively). We refer to real numbers as *scalars*.

A point/vector $p \in \mathbb{R}^d$ is expressed as a d-vector $(p_1, \ldots, p_d)$, where $p_i \in \mathbb{R}$. Following standard terminology from linear algebra, given two vectors $u, v \in \mathbb{R}^d$ and a scalar $\alpha \in \mathbb{R}$, let "$u + v$" be the vector-valued sum, αv be scalar-vector product, and let "$u \cdot v$" denote the standard scalar-valued dot-product.

Affine and convex combinations: Given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ in $\mathbb{R}^d$, we can express any point on the (infinite) line $\overleftrightarrow{pq}$ as a linear combination of their coordinates, where the coefficient sum to 1:

$$(1-\alpha)p + \alpha q \;=\; ((1-\alpha)p_x + \alpha q_x, (1-\alpha)p_y + \alpha q_y).$$

This is called an *affine combination* of p and q (see Fig. 8(a)).

By adding the additional constraint that $0 \leq \alpha \leq 1$, the set of points generated lie on the *line segment* $\overline{pq}$ (see Fig. 8(b)). This is called a *convex combination*. Notice that this can be viewed as taking a weighted average of p and q. As α approaches 1, the point lies closer to p and when α approaches zero, the point lies closer to q.

It is easy to extend both types of combinations to more than two points. For example, given k points $\{p_1, \ldots, p_k\}$ an affine combination of these points is the linear combination

$$\sum_{i=1}^{k} \alpha_i p_i, \qquad \text{such that } \alpha_1 + \cdots + \alpha_k = 1.$$

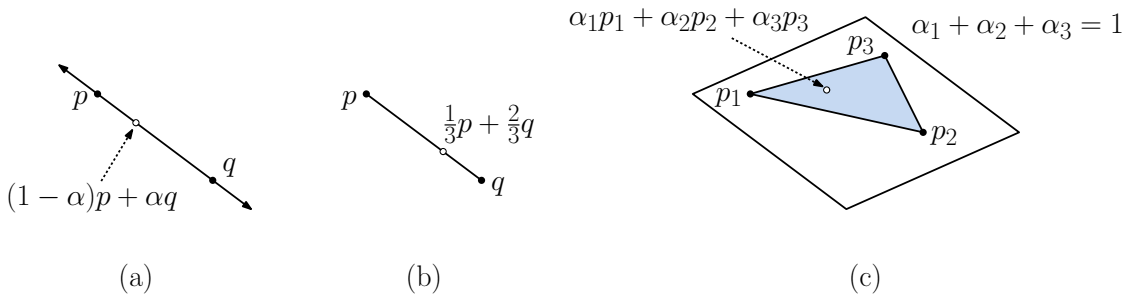


Fig. 8: Affine and convex combinations.

When $0 \leq \alpha_i \leq 1$ for all i, the result is called a *convex combination*.

The set of all affine combinations of three (non-collinear) points generates a plane, and generally, the resulting set is called the *affine span* or *affine closure* of the points. Given three noncollinear points in $\mathbb{R}^3$, their affine span generates the points on the plane passing through these points. The set of all convex combinations generates the triangle defined by the points. This generalizes in a natural way to all higher dimensions.

Hyperplanes/Halfspaces: Given a nonzero vector $v \in \mathbb{R}^d$ and a scalar $\alpha \in \mathbb{R}$, the set of points $\{p \mid p \cdot v = \alpha\}$ is a $(d-1)$-dimensional affine subspace, more often called a *hyperplane*. In the special cases $\mathbb{R}^2$ and $\mathbb{R}^3$, this defines a *line* and *plane*, respectively. If v is a unit vector and $\alpha \geq 0$, this hyperplane is orthogonal to v and lies at distance α from the origin (see Fig. 9(a)). If we change the equality to an inequality, we obtain a *halfspace* consisting of points lying on one side of the hyperplane, for example $\{p \mid p \cdot v \leq \alpha\}$.

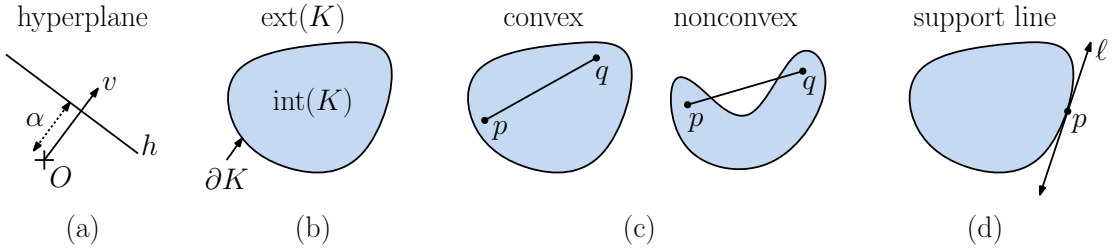


Fig. 9: Basic concepts.

Concepts from Topology: There are a number of useful concepts that arise from topology: open and closed sets, interior, exterior, and boundary, connectivity, etc. We will use these without definitions, since for the simple objects that we will be working with, these concepts are easily understood at an intuitive level.[2] We will use $\text{int}(K)$, $\text{ext}(K)$, and ∂K to denote the interior, exterior, and boundary of a set K, respectively (see Fig. 9(b))).

Convexity: A set $K \subseteq \mathbb{R}^d$ is *convex* if given any points $p, q \in K$, the line segment $\overline{pq}$ is entirely contained within K (see Fig. 9(c)). Otherwise, it is called *nonconvex*. This is equivalent to saying that K is "closed" under convex combinations. Examples of convex sets in the plane include circular disks (the set of points contained within a circle), the set of points lying within any regular n-sided polygon, lines (infinite), line segments (finite), rays, and halfspaces.

[2]See, for example https://en.wikipedia.org/wiki/Boundary_(topology) for definitions.

Support line/hyperplane: An important property of any convex set K in $\mathbb{R}^2$ is that at every point p on the boundary of K, there exists a (not necessarily unique) line ℓ that passes through p such that K lies entirely to one side of ℓ (see Fig. 9(d)). This is called a *supporting line* (or *support line*) for K. In higher dimensions $\mathbb{R}^d$ this will generally by a $(d-1)$-dimensional hyperplane, called a *supporting hyperplane*. Observe that there may generally be multiple (indeed an infinite number of) supporting lines passing through a given boundary point of K. This happens when p is a vertex of K.

Convex Hulls: Formally, the convex hull of a point set P is defined to be smallest convex set containing P. It can be characterized in two provably equivalent ways (one additive and one subtractive). First, it is equal to the set of all convex combinations of points in P and second, it is equal to the intersection of all halfspaces that contain P.

When computing convex hulls, we will usually take P to be a finite set of points. In such a case, $\text{conv}(P)$ 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.

General Position: As in many of our algorithms, it will simplify the presentation to avoid lots of special cases by assuming that the points are in *general position*. This effectively means that "degenerate" geometric configurations do not arise in the input.

What do we mean by "degenerate"? This can be defined formally (by giving a definition on the *measure* of point sets and excluding certain configurations that have measure zero), but for our purposes, we will give a more intuitive, but less rigorous, definition. A *degeneracy* is any geometric property that can be destroyed by some infinitesimal perturbation of the geometric objects (or more accurately, the real-valued parameters that define these objects).

Here are some concrete examples of degenerate configurations (see Fig. 10):

- three points in the plane that are collinear
- two points that share the same x-coordinate
- three lines that pass through the same point (coincident)
- four points that lie on the same circle

Note, by the way that *three* points lying on the same circle is not a degeneracy, since generally three points uniquely determine a circle.

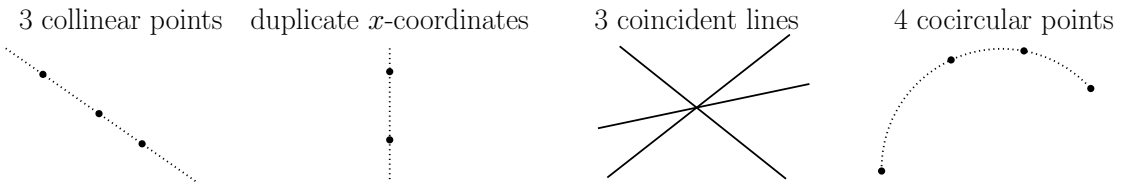


Fig. 10: Examples of degenerate configurations of geometric objects.

When we say that "we assume that our input is in general position", we mean that the input set is free of any degenerate configurations of points (e.g., no three points from the set are collinear). Whenever we do this, we should explain precisely which geometric configurations we are excluding. But, often we will be sloppy and just omit this, since it will usually be obvious by inspection of the algorithm.

Graham's Scan: We will begin with a presentation of a simple $O(n \log n)$ algorithm for the convex hull problem. It is a simple variation of a famous algorithm for convex hulls, called *Graham's Scan*, which dates back to the early 1970's (and named for its inventor Ronald Graham). 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.

Let us assume that the input consists of a set P of n points $\{p_1, \ldots, p_n\}$, where $p_i = (x_i, y_i)$. While we will not do so, it is easy to prove that the convex hull of a finite set of points is a convex polygon, whose vertices are a subset of P. A natural representation of such a polygon is a cyclic (for example, counterclockwise) listing of the vertices of this polygon.

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. Instead, we will insert points in increasing order of x-coordinates. 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 themselves ordered from left to right. To do this, we can break the boundary of the convex hull into two *chains*, an *upper chain* consisting of the vertices along the upper part of the hull and a *lower chain* consisting of the vertices along the lower part of the hull. Both chains will start with the leftmost point of P and will end with the rightmost point of P. We will make the general-position assumption that no two vertices have the same x-coordinates, so these two points are unique (see Fig. 11(a)).

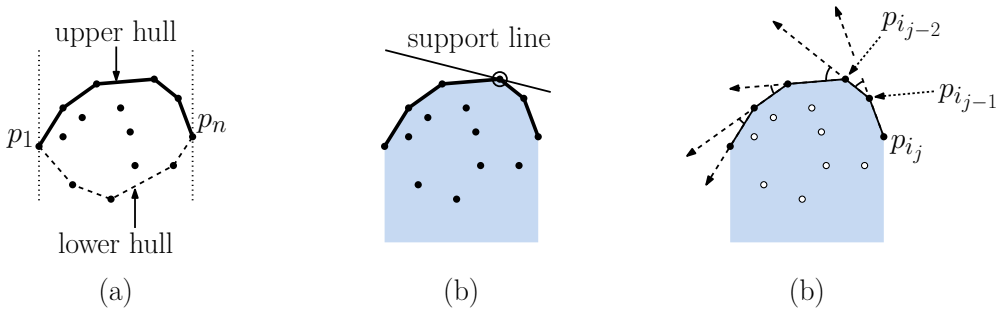


Fig. 11: (a) Upper and lower hulls, (b) supporting line, and (c) the left-hand turn property of points on the upper hull.

It suffices to show how to compute the upper hull, since the lower hull is symmetrical. (Just flip the picture upside down.) Once the two hulls have been computed, we can simply concatenate them with the reversal of the other to form the final hull.

Observe that a point $p \in P$ lies on the upper hull if and only if there is a supporting line passing through p such that all the points of P lie on or below this line (see Fig. 11(b)). Our algorithm will be based on the following lemma, which characterizes the upper hull of P. This is a simple consequence of the convexity. The first part says that the line passing through each edge of the hull is a supporting line, and the second part says that as we walk from right to left along the upper hull, we make successive left-hand turns (see Fig. 11(c)).

Lemma 1: Let $\langle p_{i_1}, \ldots, p_{i_m} \rangle$ denote the vertices of the upper hull of P, sorted from left to right. Then for $1 \leq j \leq m$, (1) all the points of P lie on or below the line $\overline{p_{i_j} p_{i_{j-1}}}$ joining consecutive vertices and (2) each consecutive triple $\langle p_{i_j} p_{i_{j-1}} p_{i_{j-2}} \rangle$ forms a left-hand turn.

Let $\langle p_1, \ldots, p_n \rangle$ denote the sequence of points sorted by increasing order of x-coordinates. For i ranging from 1 to n, let $P_i = \langle p_1, \ldots, p_i \rangle$. We will store the vertices of the upper hull of P_i on a stack S, where the top-to-bottom order of the stack corresponds to the right-to-left order of the vertices on the upper hull. Let $S[t]$ denote the stack's 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 form a (strict) left-hand turn (see Fig. 11(b)). As we push new points on the stack, we will enforce this property by popping points off of the stack that violate it.

Turning and orientations: Before proceeding with the presentation of the algorithm, we should first make a short digression to discuss the question of how to determine whether three points form a "left-hand turn." This can be done by a powerful primitive operation, called an *orientation test*, which is fundamental to many algorithms in computational geometry.

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.

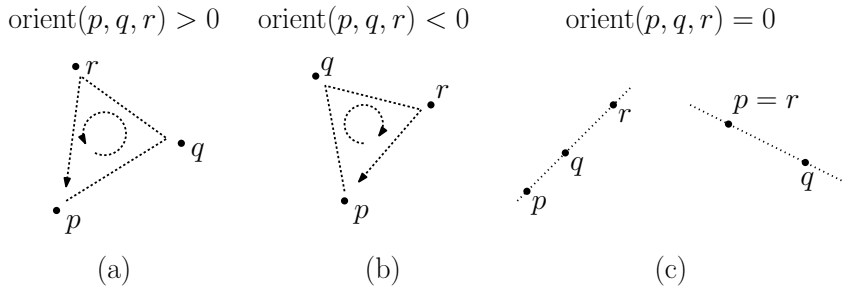


Fig. 12: Orientations of the ordered triple (p, q, r).

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

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

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 the familiar 1-dimensional binary relations $<, =, >$.

Also, observe 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 determinant of the matrix used in the transformation. (By the way, the notion of orientation can be generalized to $d + 1$ points in d-dimensional space, and is related to the notion of

chirality in Chemistry and Physics. For example, in 3-space the orientation is positive if the point sequence defines a right-handed screw.)

Why use the orientation rather than computing the actual angle? Due to its discrete nature, the orientation test has many uses in computational geometry. This means that if we implement this test once in a manner that is very efficient and numerically very stable, we do not need to worry about designing our own ad hoc geometric primitives.

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 $\text{orient}(p, q, r) > 0$.

Graham's Scan Details: We can now present the full algorithm. Recall that the input is a set P of n points in $\mathbb{R}^2$. We may assume that $n \geq 3$, since a hulls with fewer points are rather trivial.

Let us consider just the case of the upper hull, and let's see what happens when we process the insertion of the ith point, p_i (see Fig. 13(a)). First observe that p_i is on the upper hull of P_i (since it is the rightmost point seen so far). Let p_j be its predecessor on the upper hull of P_i. We know from Lemma 1 that all the points of P_i lie on or below the line $\overline{p_i p_j}$. Let p_{j-1} be the point immediately preceding p_j on the upper hull. We also know from this lemma that $\langle p_i p_j p_{j-1}\rangle$ forms a left-hand turn. Clearly then, if any triple $\langle p_i, S[t], S[t-1]\rangle$ does *not* form a left-hand turn (that is, $\text{orient}(p_i, S[t], S[t-1]) \leq 0$), we may infer that $S[t]$ is *not* on the upper hull, and hence it is safe to delete it by popping it off the stack. We repeat this until we find a left-turning triple (see Fig. 13(b)) or hitting the bottom of the stack. Once this happens, we push p_i on top of the stack, making it the rightmost vertex on the upper hull (see Fig. 13(c)). The algorithm is presented in the code block below.

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 S.

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

  (a) while ($|S| \geq 2$ and $\text{orient}(p_i, S[t], S[t-1]) \leq 0$) pop S.

  (b) push p_i onto S.

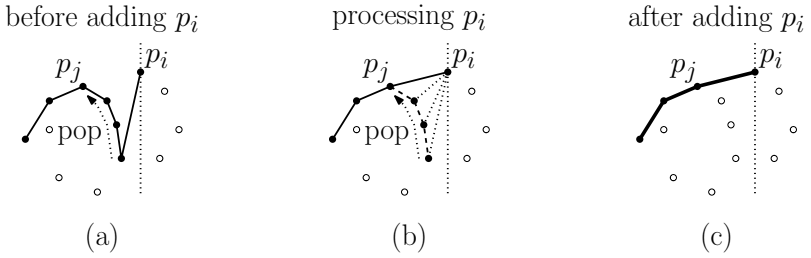


Fig. 13: Graham's scan.

Correctness: The correctness of Graham's scan follows from the following invariant. Let U_i denote the sequence of vertices forming the upper hull of $\{p_1, \ldots, p_i\}$, from left to right.

Lemma: Following any step $i \geq 2$, the stack contains (from bottom to top order) the vertices U_i of the upper convex hull.

Proof: The proof of this lemma naturally involves induction. The basis of induction is trivial, since clearly $U(2)$ just consists of p_1 and p_2.

Let us assume inductively that U_{i-1} has been correctly computed after stage $i-1$, and we will show that U_i is correctly computed after stage i. As we argued before, p_i must be the final vertex of U_i, and indeed the algorithm pushes it last. It remains to show that the algorithm will correctly pop all the points along U_{i-1} that are strictly between p_i and p_j from the stack, but leave p_j itself on the stack.

To see this, let H_{i-1} denote the convex body whose upper hull is U_{i-1} and no lower hull (that is, it extends down to $y = -\infty$) (see Fig. 14(a)). Clearly, all the vertices to be popped from the stack have x-coordinates that lie to the right of p_j and to the left of p_i. Since p_j is the predecessor of p_i on the final upper hull, all of the points of P lie below the supporting line $\ell = \overleftrightarrow{p_j p_i}$, and hence, all the points of U_{i-1} between p_j and p_i lie below the supporting line segment $\overline{p_j p_i}$.

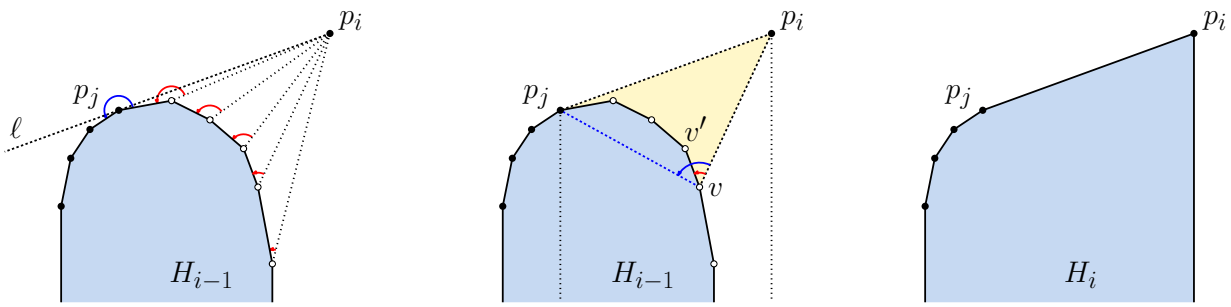


Fig. 14: Correctness of Graham's scan.

Let v denote the point on the stack currently under consideration and let v' denote its predecessor on the stack (see Fig. 14(b)). All the vertices along U_{i-1} between p_j and v lie below the segment $\overline{p_i p_j}$ and above the segment $\overline{p_j v}$. Therefore, they all lie within the triangle $\triangle p_i v p_j$. By basic geometry, the angle $\angle p_i v p_j$ is strictly smaller than 2π, which implies that $\text{orient}(p_i, v, p_j) < 0$. It follows that $\text{orient}(p_i, v, v') < 0$, and therefore v will be popped off the stack, as desired. By induction, all the vertices of U_{i-1} up to (but not including) p_j will be popped, and p_i will be pushed, as desired (see Fig. 14(c)).

How much detail? Wow, that was a lot of explaining! A question that often arises at this point of the semester is, "How much detail are you expecting in our geometrical proofs of correctness?" While geometric arguments are often aided by figures, you should not rely on your figures to do the "heavy lifting" in your proof. Your proof should reduce the task to a configuration involving a small number of points or lines. From there, you can appeal to easy geometric facts, without the need to spell them all out. There is a bit of art in doing this, and you will gain experience as the course progresses.

Don't be fooled by your drawings: There is a saying that "Geometry is reasoning well from badly drawn figures". Don't be fooled by coincidents that arise in your drawings. For example, if you were to draw Fig. 14 poorly, you might be tempted to assert that p_j is the point on U_{i-1} with the largest y-coordinate. While this might be true, it is clearly not always the case.

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 of the points. 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, $\sum_i d_i \le n$. 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 divide-and-conquer. It can be viewed as a generalization of the well-known MergeSort sorting algorithm (see any standard algorithms text). Here is an outline of the algorithm. As with Graham's scan, we will focus just on computing the upper hull, and the lower hull will be computed symmetrically.

The algorithm begins by sorting the points by their x-coordinate, in $O(n \log n)$ time. In splits the point set in half at its median x-coordinate, computes the upper hulls of the left and right sets recursively, and then merges the two upper hulls into a single upper hull. This latter process involves computing a common upper supporting line, called the *upper tangent*, for both hulls. The remainder of the algorithm is shown in the code section below.

Divide-and-Conquer (Upper) Convex Hull

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

(2) Otherwise, partition the point set P into two sets P' and P'' of roughly equal sizes by a vertical line.

(3) Recursively compute upper convex hulls of P' and P'', denoted H' and H'', respectively (see Fig. 15(a)).

(4) Compute the upper tangent $\ell = \overline{p'p''}$ (see Fig. 15(b)) by a process explained below.

(5) Merge the two hulls into a single upper hull by discarding all the vertices of H' to the right of p' and the vertices of H'' to the left of p'' (see Fig. 15(c)).

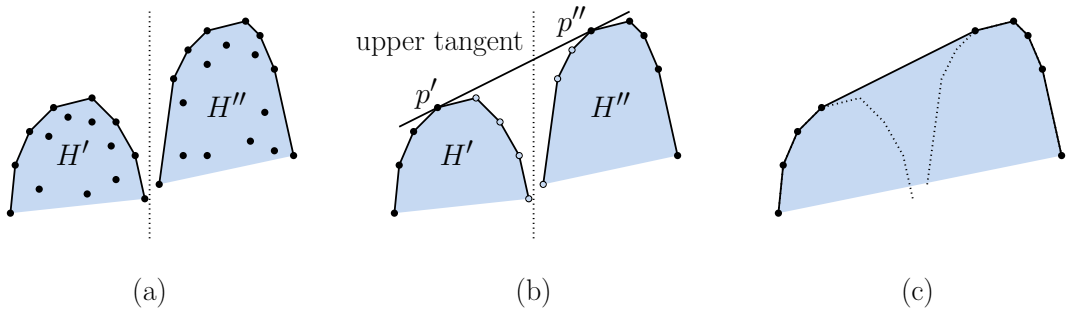


Fig. 15: Divide and conquer (upper) convex hull algorithm.

Computing the Upper Tangent: The only nontrival step is that of computing the common tangent line between the two upper hulls. Our algorithm will exploit the fact that the two

hulls are separated by a vertical line. The algorithm operates by a simple "walking procedure." We initialize p' to be the rightmost point of H' and p'' to be the leftmost point of H'' (see Fig. 16(a)). We will walk p' backwards along H' and walk p'' forwards along H'' until we hit the vertices that define the tangent line.

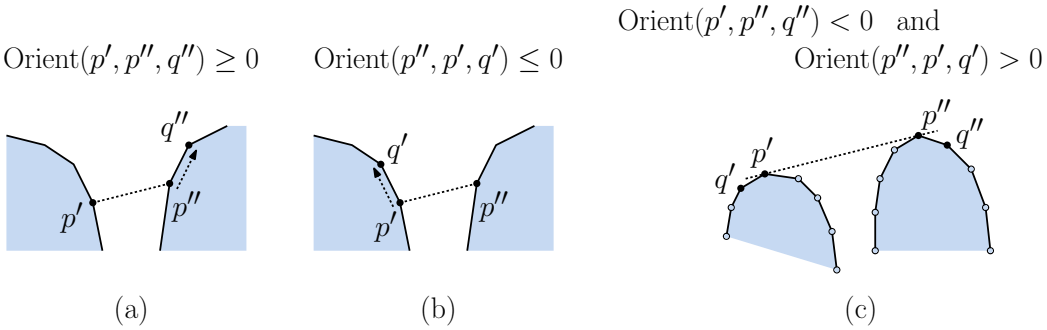


Fig. 16: Computing the upper tangent.

As in Graham's scan, it is possible to determine just how far to walk simply by applying orientation tests. In particular, let q' be the point immediately preceding p' on H', and let q'' be the point immediately following p'' on H''. Observe that if orient$(p', p'', q'') \geq 0$, then we can advance p'' to the next point along H'' (see Fig. 16(a)). Symmetrically, if orient$(p'', p', q') \leq 0$, then we can advance p' to its predecessor along H' (see Fig. 16(b)). When neither of these conditions applies, that is, orient$(p', p'', q'') < 0$ and orient$(p'', p', q') > 0$, we have arrived at the desired points of mutual tangency (see Fig. 16(c)).

There is one rather messy detail in implementing this algorithm. This arises if either q' or q'' does not exist because we have arrived at the leftmost vertex of H' or the rightmost vertex of H''. We can avoid having to check for these conditions by creating two *sentinel points*. We create a new leftmost vertex for H' that lies infinitely below its original leftmost vertex, and we create a new rightmost vertex for H'' that lies infinitely below its original rightmost vertex. The tangency computation will never arrive at these points, and so we do not need to add a special test for the case when q' and q'' do not exist. The algorithm is presented in the following code block.

Computing the Upper Tangent

UpperTangent(H', H'') :

(1) Let p' be the rightmost point of H', and let q' be its predecessor.
(2) Let p'' be the leftmost point of H'', and let q'' be its successor.
(3) Repeat the following until orient$(p', p'', q'') < 0$ and orient$(p'', p', q') > 0$:
  (a) while (orient$(p', p'', q'') \geq 0$) advance p'' and q'' to their successors on H''.
  (b) while (orient$(p'', p', q') \leq 0$) advance p' and q' to their predecessors on H'.
(4) return (p', p'').

Running-time analysis: 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 upper tangent line, and return the final result. Clearly, each of these can be performed in $O(n)$ time, assuming any standard list representation of the hull vertices. 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).

Jarvis's March: Our next convex hull algorithm, called *Jarvis's march*. This algorithm is *output sensitive*, meaning that its running time depends on the number of vertices on the final convex hull. Let n denote the number of input points, and let h denote the number of vertices on the final convex hull. We will see that Jarvis's march computes the convex hull in $O(nh)$ time by a process called "gift-wrapping." In the worst case, $h = n$, so this is inferior to Graham's algorithm for large h, it is superior if h is asymptotically smaller than $\log n$, that is, $h = o(\log n)$.

Jarvis's algorithm begins by identifying any one point of P that is guaranteed to be on the hull, say, the point with the smallest y-coordinate. (As usual, we assume general position, so this point is unique.) Call this v_1. It then repeatedly finds the next vertex on the hull in counterclockwise order.

Given a triple of distinct points $\langle p, q, r \rangle$, define the *turning angle* of r with respect to p and q to be the (CCW) angle between the directed line $\overline{pq}$ and the directed line $\overline{qr}$ (see Fig. 17(a)).

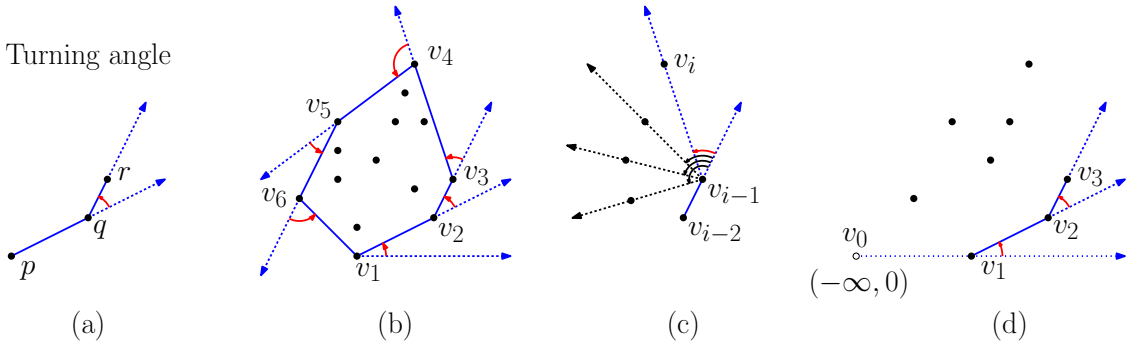


Fig. 17: Jarvis's march.

Jarvis's march works by repeatedly computing the next hull vertex v_i as the point of P that minimizes the turning angle with respect to the prior two, v_{i-2} and v_{i-1} (see Fig. 17(c)). Since we need two points, to get the ball rolling, it is convenient to define an imaginary "sentinel point" $v_0 = (-\infty, 0)$, which has the effect that the initial line $\overline{v_0 v_1}$ is directed horizontally to the right (see Fig. 17(d)).

Jarvis's March

(1) Given P, let $v_0 = (-\infty, 0)$ and let v_1 be the point of P with the smallest y-coordinate

(2) For $i \leftarrow 2, 3, \ldots$

  (a) $v_i \leftarrow$ the point of $P \setminus \{v_{i-1}, v_{i-2}\}$ that minimizes the turning angle with respect to v_{i-2} and v_{i-1}

  (b) If $v_i == v_1$, return $\langle v_1, \ldots, v_{i-1} \rangle$

The algorithm's correctness follows from the fact that (by induction) $\overline{v_{i-2} v_{i-1}}$ is a CCW-directed edge of the hull, and hence the next vertex of the hull is the one that minimizes the turning angle.

By basic trigonometry, turning angles can be computed in constant time. But it is interesting to note that it is possible to compare turning angles just using orientation tests. (Try this yourself.) This implies that if the input coordinates are integers, the vertices of the hull can be computed exactly (assuming double-precision integer computations).

To obtain the running time, observe that v_1 can be computed in $O(n)$ time, and each iteration can be implemented in $O(n)$ time. After h iterations, the algorithm terminates, so the total running time is $O(n + nh) = O(nh)$.

Lecture 3: Convex Hulls 2: Chan's Algorithm and Lower Bounds

Computational Complexity of the Hull: In the previous lecture we presented two planar convex hull algorithms, Graham's scan and the divide-and-conquer algorithm, both of which run in $O(n \log n)$ time. A natural question to consider is whether we can do better. Is the $O(\log n)$ factor necessary?

Recall that the output of the convex hull problem a convex polygon, that is, a cyclic enumeration of the vertices along its boundary. Thus, it would seem that in order to compute the convex hull, we would "need" to sort the vertices of the hull. It is well known that it is not generally possible to sort a set of n numbers faster than $\Omega(n \log n)$ time, assuming a model of computation based on binary comparisons. (There are faster algorithms for sorting small integers, but these are not generally applicable for geometric inputs.)

Can we turn this intuition into a formal lower bound? We will show that in $O(n)$ time it is possible to reduce the sorting problem to the convex hull problem. This implies that any $O(f(n))$-time algorithm for the convex hull problem implies an $O(n + f(n))$-time algorithm for sorting. Clearly, $f(n)$ cannot be smaller than $\Omega(n \log n)$ for otherwise we would obtain an immediate contradiction to the lower bound on sorting.

The reduction works by projecting the points onto a convex curve. In particular, let $X = \{x_1, \ldots, x_n\}$ be the n values that we wish to sort. We will map this into a 2-dimensional point set by projecting the points onto the boundary of a convex shape, so that the sorted order is preserved. For example, suppose that we project each point vertically onto the parabola $y = x^2$, by mapping x_i to the point $p_i = (x_i, x_i^2)$ (see Fig. 18(a)). Let P denote the resulting set of points. It is easy to see that all the points of P lie on its convex hull, and the sorted order of points along the lower hull is the same as the sorted order X (see Fig. 18(b)). Once we obtain the convex hull as a cycle sequence of vertices, in $O(n)$ additional time we can extract its lower hull from left to right, thus obtaining X in sorted order (see Fig. 18(c)).

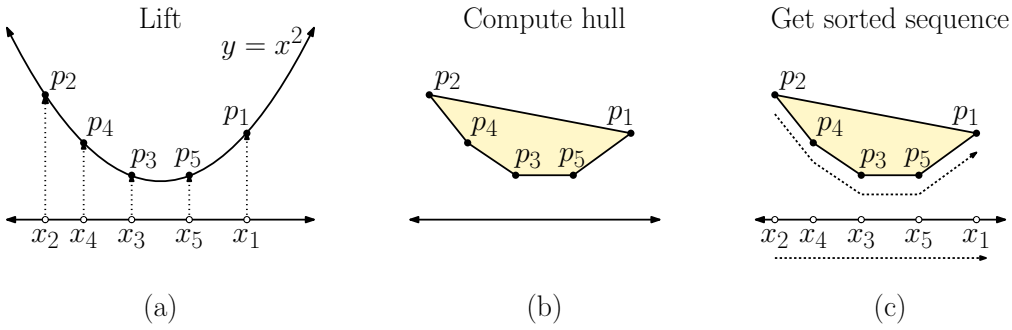


Fig. 18: Reduction from sorting to convex hull.

Theorem: Assuming computations based on comparisons (e.g., orientation tests) any algorithm for the convex hull problem requires $\Omega(n \log n)$ time in the worst case.

Is this the end of the story? Well, maybe not ...

- What if we don't require that the points be enumerated in cyclic order? For example, suppose we just wanted to count number of points on the convex hull. Can we do better?
- Rather than just using the input size n to measure performance, what if we also include the effect of the output size h? There are many instances, where very few of the input points are vertices of the convex hull. (For example, it is known that if n points are uniformly distributed in a unit square, the expected value of h is $\Theta(\log n)$.)

We will present the following results in this lecture:

- We will present Chan's algorithm, which computes the convex hull in $O(n \log h)$ time.
- We will present a lower bound argument that shows that, assuming a comparison-based algorithm, even answering the question "does the convex hull have h distinct vertices?" requires $\Omega(n \log h)$ time.

Together, these results imply that, from the perspective of these two parameters, the convex hull problem is essentially closed. An algorithm whose running time depends on the output size is called *output sensitive*. Both Jarvis's March and Chan's algorithm are output sensitive.

Chan's Algorithm: We have seen two algorithms (Graham Scan and the divide-and-conquer algorithm) that run in $O(n \log n)$ time and another (Jarvis's March) that runs in $O(nh)$ time. Depending on the value of h, Graham's scan may be faster or slower than Jarvis' march. This raises the intriguing question of whether there is an algorithm that *always* does as well or better than these algorithms. Next, we present a planar convex hull algorithm by Timothy Chan whose running time is $O(n \log h)$.

While this algorithm is too small an improvement over Graham's algorithm to be of significant practical value, it is quite interesting nonetheless from the perspective of the techniques that it uses:

- It cleverly combines two slower algorithms, Graham's and Jarvis's, to form a faster algorithm.
- It employs a clever guessing strategy to determine the value of a key unknown parameter, the number h of vertices on the hull.

To gain some intuition behind Chan's algorithm, let us first observe that in order to replace the $O(\log n)$ factor in Graham's algorithm with $O(\log h)$, we cannot afford to sort any set whose size is (significantly) larger than h. This would seem impossible at first glance, since we don't know the value of h until the algorithm terminates! We will get around this by playing a "guessing game" for the value of h. We'll start low, and work up to successively guesses for what h is. Throughout, let h denote the true number of vertices on the hull. The algorithm will maintain a variable h^*, which is our current "guess" on the value of h. As we shall see, if we guess wrong, we will discover our error, and we will need to increase our guess. For now, let us assume that a magical little bird has told us the value of h^* that works.

We will need to make use of a *utility function*, whose implementation we will leave as an exercise. Recall that a *supporting line* for a convex body is a line that contacts the boundary

of the body and the body lies entirely on one side of the line. Given a convex body Q and any point p external to Q, there are exactly two supporting lines of Q that pass through p. The next lemma shows that we can compute them logarithmic time.

Lemma: (Tangent Lemma) Given a convex polygon $Q = \langle q_1, \ldots, q_m \rangle$ (assuming the vertices are stored in an m-element array sorted in cyclic order around Q's boundary), and given any point p that is external to Q, in $O(\log m)$ time we can compute the two vertices q^- and q^+ of Q so that $\overline{pq^-}$ and $\overline{pq^+}$ are both supporting lines of Q (see Fig. 19).

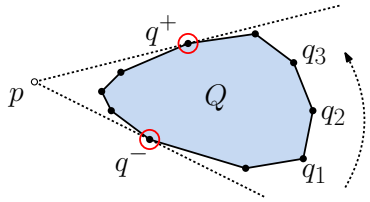


Fig. 19: Utility function - Computing tangent points.

We will leave the proof as an exercise. It involves performing binary search over the boundary of Q, using orientation tests to drive the search. We can now describe Chan's algorithm, conditioned on the fact that we have a guess h^* on the size of the hull. It will be correct as along as the actual hull size h is not greater than h^*.

Step 1: (Mini-hulls) Partition P (arbitrarily) into $k = \lceil n/h^* \rceil$ groups, each of size at most h^*. Call these $P_1, \ldots, P_k$ (see Fig. 20(b)). By Graham's algorithm, compute the convex hull of each subset. Let $H_1, \ldots, H_k$ denote the resulting *mini-hulls* (see Fig. 20(c)).

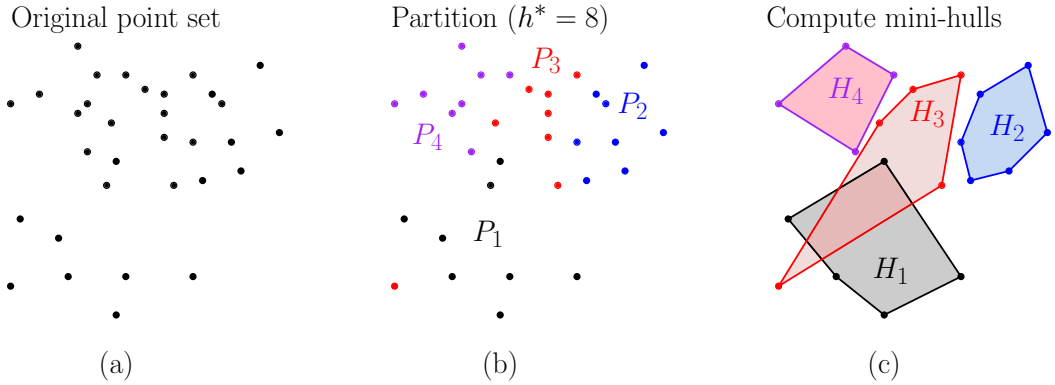


Fig. 20: Step 1: Partitioning and mini-hulls (for $k = 4$).

How long does this take? We can compute each mini-hull in time $O(h^* \log h^*)$. Applying this to each of the k groups, we have an overall time of $O(k(h^* \log h^*)) = O(n \log h^*)$. Note that if we guess the value of h correctly (that is, $h^* = h$) then this runs in time $O(n \log h)$, as desired.

Step 2: (Merging) The high-level idea is to run Jarvis's march on the mini-hulls (see Fig. 21(a)). We will treat each mini-hull as if it is a "fat point". In particular, we take the most recent vertex v_{i-1} and for each mini-hull H_j employ the utility function of the above lemma to compute the vertices q_j^- and q_j^+ for the supporting lines for this mini-hull (see Fig. 21(b)). As in Jarvis's algorithm, among all of these supporting points,

we take v_i to be the one that minimizes the turn angle with respect to v_{i-2} and v_{i-1} (see Fig. 21(c)). (By the nature of Jarvis's algorithm, we only need to compute the supporting line with the smaller turning angle, but computing both will not affect the asymptotic running time. Also note that we do this for the mini-hull containing v_{i-1} itself, but this is trivial since this is just the next vertex on the hull.)

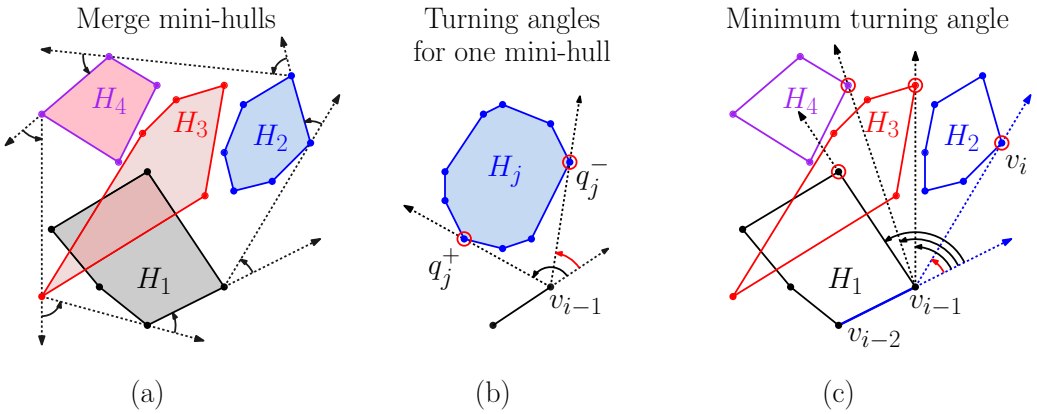


Fig. 21: Step 2: Merging the mini-hulls.

How long does this take? For each mini-hull, we can compute the two supporting lines in time $O(\log h^*)$ by the Tangent Lemma. The number of supporting lines is twice the number of mini-hulls, so for each step of Jarvis's algorithm, we can compute all the relevant turning angles in time $O(k \log h^*)$. Each iteration of Jarvis yields one more vertex of the final convex hull (of which there are h), so the overall running time is $O(h(k \log h^*))$. Observe that if we managed to guess the right value of h (that is, if $h^* = h$), then this takes time $O(h^*(k \log h^*)) = O(n \log h^*) = O(n \log h)$, as desired.

In summary, we have argued above that, if we were lucky enough to guess the correct hull size ($h^* = h$), then the method outlined above will yield the convex hull in time $O(n \log h)$.

The Conditional Algorithm: Let us present this idea a bit more formally. The algorithm is given a point set P and an estimate h^* of the number of vertices on P's convex hull. Letting h denote the actual number of vertices on the hull. We assume that $h^* \geq h$. If we discover this is incorrect, the algorithm returns "failure". For the sake of efficiency, we can tolerate h^* being somewhat larger than h. (We will make this precise below.) The algorithm is presented in the code block below.

Guessing the Hull's Size: The conditional algorithm assumes that we have a good estimate in h^* for h. What are the consequences of guessing wrong?

Too large? If we guess a value of $h^* > h$, then the Graham scans, which together run in $O(n \log h^*)$ time may be too slow. Notice, however, that we pay only a constant factor even if h^* is polynomially larger than h. For example, if $h < h^* \leq h^2$, then the running time of Graham's scan is $O(n \log(h^2)) = O(2n \log h) = O(n \log h)$, which is okay for us.

Too small? If we guess a value of $h^* < h$, then the merge phase will be too slow. It is easy to verify that (even if we didn't stop it because of the failure condition) it would run in time $O(n(h/h^*) \log h^*)$. If our estimate h^* is not within a constant factor of h, then we will not achieve our desired running time. This is why we chose to use the "failure"

ConditionalHull(P, h^*) :

(1) Let $k \leftarrow \lceil n/h^* \rceil$. Partition P (arbitrarily) into disjoint subsets $P_1, \ldots, P_k$, each of size at most h^*
(3) For $j \leftarrow 1$ to k, compute $H_j = \text{conv}(P_j)$ using Graham's scan, storing each in an ordered array
(4) Let $v_0 \leftarrow (-\infty, 0)$, and let v_1 be the bottommost point of P
(5) For $i \leftarrow 1, 2, \ldots, h^*$:
  (a) For $j \leftarrow 1$ to k, using the Tangent Lemma, compute the tangents points q_j^- and q_j^+ for H_j with respect to v_{i-1}.
  (b) Set v_i to be the tangent point that minimizes the turning angle with respect to v_{i-2} and v_{i-1}
  (c) If $v_i = v_1$ then return the pair (**success**, $V = \langle v_1, \ldots, v_{i-1} \rangle$)
(6) If we get here, we know that $h^* < h$, and we return (**failure**, $\emptyset$)

option. Since we never do more than h^* iterations, the running time of each failure phase is just $O(n \log h^*)$.

Here is what we'll do. We'll start with a low estimate for h^* (e.g., $h^* = 3$). If the algorithm returns "failure", we increase h^* until we succeed. The question is how quickly we should step up the value of h^*?

It is easy to show that increasing h^* in an arithmetic progression (e.g., $h^* = 3, 4, 5, \ldots$) will be way to slow. A smarter approach is to grow h^* through doubling (e.g. $h^* = 4, 8, 16, \ldots, 2^i$). We will leave it as an exercise to show that this is also too slow. (It will lead to a running time of $O(n \log^2 h)$.)

Recall that we are allowed to overshoot the actual value of h by any polynomial. Let's try *repeatedly squaring* the previous guess. In other words, let's try $h^* = 2, 4, 16, 256, \ldots, 2^{2^i}$. Clearly, as soon as we reach a value for which the restricted algorithm succeeds, we have $h < h^* \leq h^2$. Therefore, the running time for this last stage will be $O(n \log h)$, as desired. But what about the total time for all the previous stages?

To analyze the total time, consider the ith guess, $h_i^* = 2^{2^i}$. The ith trial takes time $O(n \log h_i^*) = O\big(n \log 2^{2^i}\big) = O(n 2^i)$. 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.[3]) Thus, the algorithm's total running time (up to constant factors) is

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

The summation is a geometric series. It is well known that a geometric series is asymptotically dominated by its largest term, which is $2^{\lg \lg h} = \lg h$. Thus, we obtain the final running time

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

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.

[3]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.

Hull(P) :

(1) $h^* \leftarrow 2$; status $\leftarrow$ failure
(2) while (status $\neq$ failure):
  (a) Let $h^* \leftarrow \min((h^*)^2, n)$
  (b) (status, V) $\leftarrow$ ConditionalHull(P, h^*)
(3) return V

Lower Bound (Optional): We 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 three 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 ternary (3-way splitting) tree. The height of the tree is a lower bound on the worst-case running time of the algorithm. There are at least $n!$ different possible inputs, each of which must be reordered differently, and so you have a ternary 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 branches based on the sign of some function of the input coordinates. For example, we could implement the conditional $z_i < z_j$ by checking whether the function $(z_j - z_i)$ is positive. More relevant to convex hull computations, we can express an orientation test as the sign of the determinant of a matrix whose entries are the six coordinates of the three points involved. The determinant of a matrix can be expressed as a polynomial function of the matrices entries. Such a function is called *algebraic*. We assume that each node of the decision tree branch three ways, depending on the sign of a given multivariate algebraic formula of degree at most d, where d is any fixed constant. For

example, we could express the orientation test involving points $p_1 = (z_1, z_2)$, $p_2 = (z_3, z_4)$, and $p_3 = (z_5, z_6)$ as an algebraic function of degree two as follows:

$$\det \begin{pmatrix} 1 & z_1 & z_2 \\ 1 & z_3 & z_4 \\ 1 & z_5 & z_6 \end{pmatrix} = (z_3 z_6 - z_5 z_4) - (z_1 z_6 - z_5 z_2) + (z_1 z_4 - z_3 z_2).$$

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.

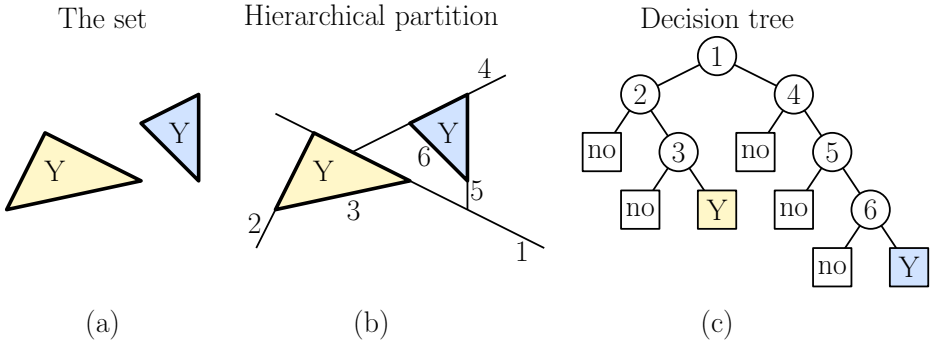


Fig. 22: The geometric interpretation of an algebraic decision tree.

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 4: Line Segment Intersection

Geometric intersections: One of the most basic problems in computational geometry is that of computing intersections. It has numerous applications.

- 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 a classical method for rendering scenes. The computationally most intensive part of ray shooting is determining the intersection of the ray with other objects.

In this lecture, we will focus the basic primitive of computing line segment intersections in the plane.

Line-Segment intersection: We are given a set $S = \{s_1, \ldots, s_n\}$ of n line segments in the plane. Let's assume that each is represented by its two endpoints $s_i = \overline{p_i q_i}$. We will consider the problem of reporting pairs of line segments that intersect (see Fig. 23(a)).

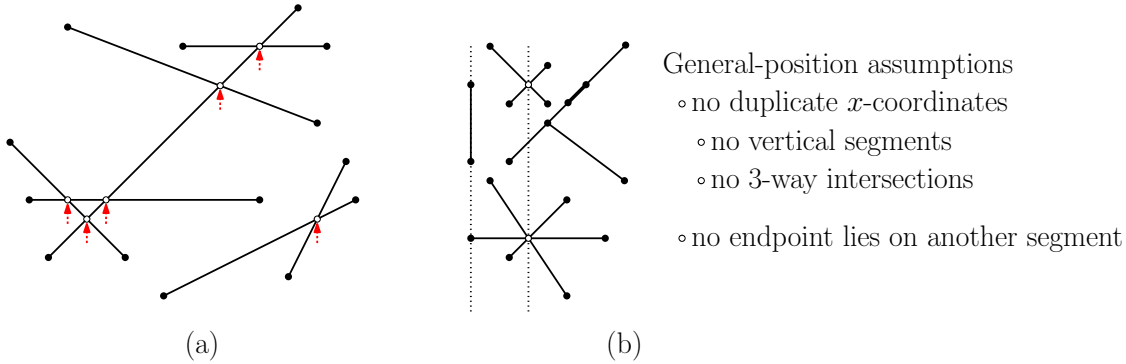


Fig. 23: Line segment intersection.

To simplify the presentation, we will make the usual *general-position assumptions*. In particular, we will rule out the following (see Fig. 23(b)):

- No duplicate x-coordinates among endpoints or intersection points (which implies no vertical line segments or multiple intersections)
- No segment endpoint lies on another segment

All these special cases are relatively easy to cope with, but the algorithms are more complex because they need to deal with various special cases. One common exception to the above is to allow multiple segments to share a common endpoint, since this occurs when dealing with polygons and spatial subdivision.

Output Sensitivity: 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, we will focus on *output sensitive algorithms*, meaning that the running time depends not only on the number of segments, but also the number of intersecting pairs.

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

Utility Data Structures: Our algorithm will make use of two standard dynamic data structures:

Ordered Dictionary: Stores a set of objects each associated with a key from an totally ordered domain. Supports the operations *insert*, *delete*, *find*, *predecessor*, *successor*,

get-kth (which returns the item at some given rank) and *swap* (which swaps two given adjacent entries) all in $O(\log n)$ time and $O(n)$ space, where n is the current number of objects. (This is typically implemented using balanced binary trees, such as red-black trees, or skip lists. Note that because of the predecessor, successor, and swap operations, we cannot used unordered dictionaries like hashing.)

Priority Queue: Stores a set of objects, where each object is associated with a priority value. Supports the operations *insert*, *delete*, and *extract-min* (which accesses and removes the object with the smallest priority value). We also assume that the insert operation returns a special *reference* to the inserted object, which allows us to efficiently delete this item. These operations are supported in $O(\log n)$ time and $O(n)$ space, where n is the current number of objects. (This is typically implemented as a binary heap.)

Plane Sweep: Let us now consider a natural approach for reporting the segment intersections. The method, called *plane sweep*, is a fundamental technique in planar 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.

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.

In any algorithm based on plane sweep there are three basic elements that need to be maintained (see Fig. 24). Let ℓ denote the current sweep line.

(1) the *partial solution* that has already been constructed to the left of ℓ (in our case, the discovered intersection points lying to the left of the sweep line),

(2) the *sweep-line status*, that is, the set of objects intersecting ℓ (in our case, the sorted segments intersecting the sweep line), and

(3) a subset of the *future events* lying to right of ℓ that are scheduled to be processed (in our case, a subset of the intersection points to the right of the sweep line).

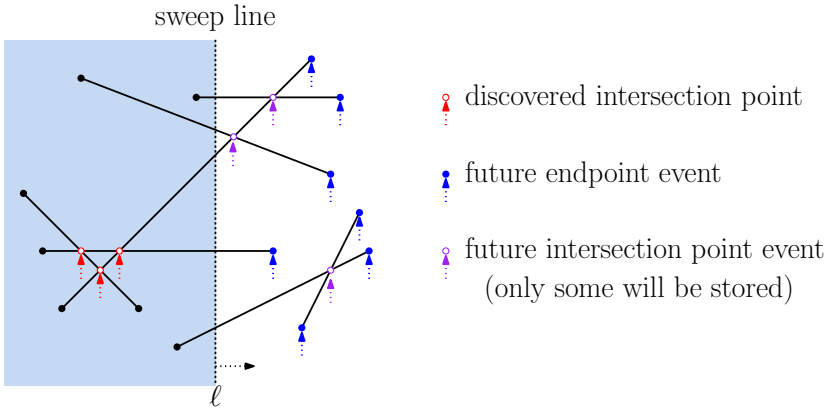


Fig. 24: Plane sweep.

The key to designing an efficient plane-sweep algorithm is determining how to efficiently 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 and above-below comparisons: We will simulate the sweeping of a vertical line ℓ from left to right. The sweep-line status consists 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 an *ordered dictionary*.

But hey! How can we possibly do this efficiently? Every time we move the sweep line even a tiny distance, all the y-coordinates of the intersection points change as well! Clearly, we cannot store the y-coordinates explicitly, for otherwise we would be doomed to $O(n)$ time per event, which would lead to an overall running time that is at least quadratic.

The key is that we do not need store the actual y-coordinates in the dictionary. Instead, we implement a function that compares two line segments at a given the x-coordinate. This function determines which segment intersects the sweep line above the other. Let's call this a *dynamic comparator*. (A more detailed description is provided at the end of the lecture notes.)

Observe that between consecutive event points (intersection points or segment endpoints) the relative vertical order of segments is constant (see Fig. 25(a)). For each segment, we can compute the associated line equation, and evaluate this function at x_0 to determine which segment lies on top. The ordered dictionary does not need actual numbers. It just needs a way of comparing objects (see Fig. 25(b)).

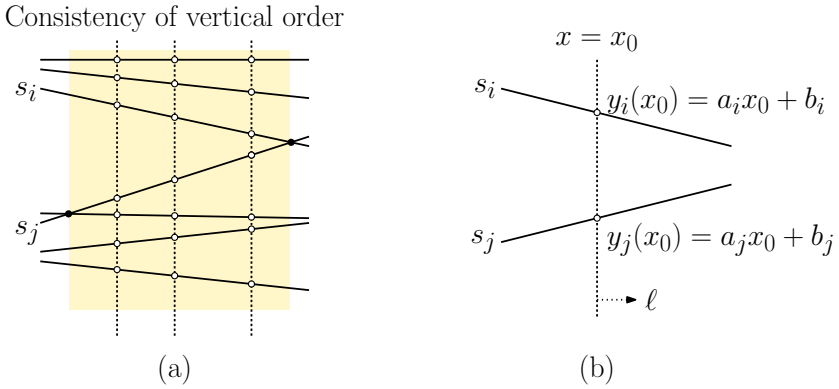


Fig. 25: The dictionary does not need to store absolute y-coordinates, just the ability to make above-below comparisons for any location of the sweep line.

Events and Detecting Intersections: It suffices to process events only when there is a change in the combinatorial structure of the sweep-line status. As mentioned above, 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 ((1) and (2)) can be *presorted* before the sweep runs. In contrast, intersection events (3) will be discovered *dynamically* as the sweep executes. It is important that each event be detected before the actual event occurs. Since each pair of segments along the sweep line might intersect, there are $O(n^2)$ potential intersection events to consider, which again would doom us to at least quadratic running time. How can we limit the number of potential intersection points to a manageable number?

Our strategy will be as follows. Whenever two line segments become *adjacent* along the sweep line (one immediately above the other), 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. We call this the *adjacent-segment rule*.

A natural question is whether this strategy of scheduling intersections between adjacent pairs is correct. In particular, might it be that two line segments intersect, but just prior to this intersection, they were not adjacent in the sweep-line status? If so, we would miss this event. Happily, this is not the case, but it requires a proof. (If you think it is trivial, note that it would fail to hold if the objects being intersected were general algebraic curves, not line segments.)

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. Then s_i and s_j are adjacent along the sweep line just after the event that immediately precedes p in the sweep.

Proof: By our general position assumptions, 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. 25), 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 and remain so just prior to processing p.

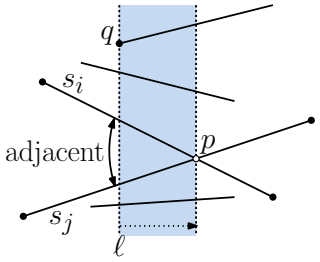


Fig. 26: Correctness of the adjacent-segment rule.

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 repeatedly and it guarantees that the total number of events can never exceed $O(n)$.

Data Structures: As mentioned above the segments that intersect the sweep line will be maintained in an ordered dictionary, sorted vertically from top to bottom. The future event points (segment endpoints and impending intersection points) will be stored in a *priority queue*, which will be ordered from left to right by x-coordinates.

Here are the operations assumed to be supported by the *ordered dictionary*, which stores the sweep-line status:

- $r \leftarrow \text{insert}(s)$: Insert s (represented symbolically) and return a reference r to its location in the data structure.

- delete(r): Delete the entry associated with reference r.
- $r' \leftarrow$ predecessor(r): Return a reference r' to the segment lying immediately above r (or null if r is the topmost segment).
- $r' \leftarrow$ successor(r): Return a reference r' to the segment lying immediately below r (or null if r is the bottommost segment).
- $r' \leftarrow$ swap(r): Swap r and its immediate successor, returning a reference to r's new location.

Next, here are the operations assumed to be supported by the *priority queue*, which stores the future events sorted by the x-coordinates:

- $r \leftarrow$ insert(e, x): Insert event e with "priority" x and return a reference r to its location in the data structure.
- delete(r): Delete the entry associated with reference r.
- $(e, x) \leftarrow$ extract-min(): Extract and return the event from the queue with the smallest priority x.

As mentioned earlier, all of these operations can be performed in $O(\log n')$ and $O(n')$ space, where n' is the current number of entries in the data structure.

The Final Algorithm: All that remains is explaining how to process the events. This is presented in the code block below, and the various cases are illustrated in Fig. 26. (Further details can be found in the 4M's.)

Correctness: The correctness of the algorithm essentially follows from our extensive derivation to the algorithm itself. Formally, the correctness proof is based on an induction proof showing that immediately after processing each event: (a) the sweep-line status contains the line segments intersecting the sweep line in sorted order and (b) the event queue contains exactly all the events demanded by the adjacent-segment rule.

Analysis: Altogether, there are $2n + m$ events processed (two endpoints per segment and m intersections). 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 $O((2n + m) \log n) = O(n \log n + m \log n)$. Note that if we output each intersection point without storing it, the total storage requirements never exceed $O(n)$. In summary, we have:

Theorem: Given a set of n line segments S in the plane (subject to our general-position assumptions), the above algorithm correctly reports all the m intersections between these segments in time $O((n + m) \log n)$ time and $O(n)$ space.

Lower Bound: Is this the best possible? No. There is a faster algorithm (which we may discuss later in the semester) that runs in time $O(n \log n + m)$. This latter algorithm is actually optimal. Clearly $\Omega(m)$ time is needed to output the intersections. The lower bound of $\Omega(n \log n)$ results from a reduction from a problem called *element uniqueness*. In this problem, we are give a list of n numbers $X = \langle x_1, \ldots, x_n \rangle$ and we are asked whether there are any duplicates (or all are 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 $O(n)$

(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: (see Fig. 27(a))

(a) Insert this line segment s into the sweep-line status, based on the y-coordinate of its left 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 an intersection between s and s^+, and if so, add it to the event queue. Do the same for s and s^-.

Right endpoint: (see Fig. 27(b))

(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 an intersection between s^+ and s^- to the right of the sweep line, and if so, add the corresponding event to the event queue.

Intersection: (see Fig. 27(c))

(a) Let s^+ and s^- be the two segments involved (with s^+ above just prior to the intersection). Report this intersection.

(b) Let s^{++} and s^{--} be the segments immediately above and below the intersection. Remove any event involving the pair (s^+, s^{++}) and the pair (s^-, s^{--}).

(c) Swap s^+ and s^- in the sweep-line status (they must be adjacent to each other).

(d) Test for an intersection between s^- and s^{++} to the right of the sweep line, and if so, add it to the event queue. Do the same for s^+ and s^{--}.

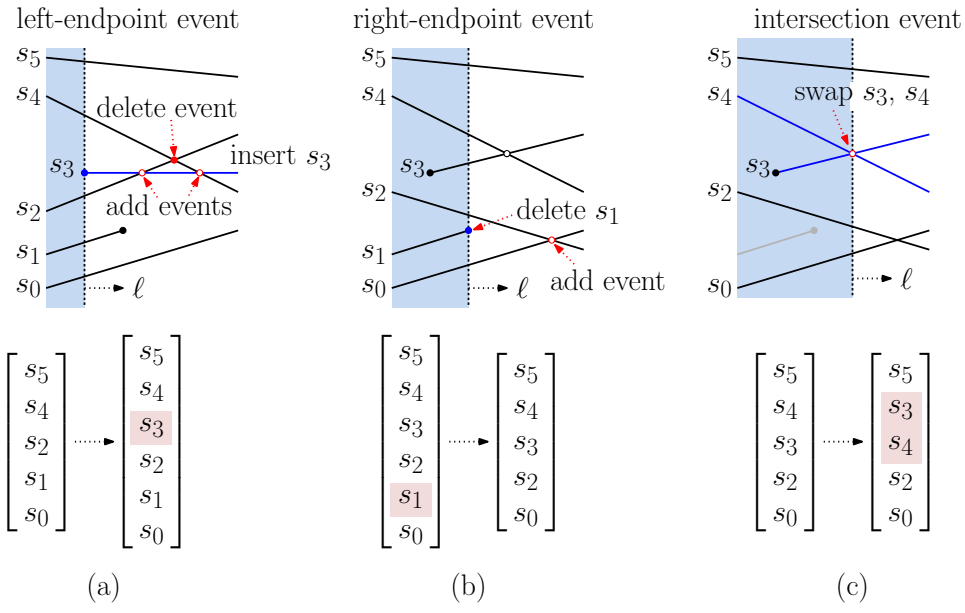


Fig. 27: Plane-sweep algorithm event processing.

time using hashing, but the algebraic decision-tree model does not allow integer division, which is needed by hashing.)

The reduction involves converting each x_i into a line segment s_i that passes through the point $(x_i, 0)$, but otherwise there are no other intersections. (A little cleverness is needed to guarantee that the general-position assumptions are satisfied.) Clearly, two segments s_i and s_j intersect if and only if two elements x_i and x_j of the list are identical. So, determining whether there is even a single intersection requires at least $\Omega(n \log n)$ time.

Dynamic Comparator: (Optional) Let's consider how to design a function that compares two line segments relative to the current sweep line. We are given the sweep line $x = x_0$ and two segments s_i and s_j. Assuming each segment is nonvertical and has endpoints $p_i = (p_{i,x}, p_{i,y})$ and $q_i = (q_{i,x}, q_{i,y})$, we can compute the associated line equations $\ell_i : y = a_i x + b_i$ and $\ell_j : y = a_j x + b_j$, by solving the simultaneous equations

$$p_{i,y} = a_i p_{i,x} + b_i \qquad q_{i,y} = a_i q_{i,x} + b_i,$$

which yields

$$a_i = \frac{p_{i,y} - q_{i,y}}{p_{i,x} - q_{i,x}} \qquad b_i = \frac{p_{i,x} q_{i,y} - p_{i,y} q_{i,x}}{p_{i,x} - q_{i,x}}.$$

By our general-position assumption, the segment is nonvertical, and the denominator is nonzero.

Given that the sweep line is at $x = x_0$, we can define our dynamic comparator to be:

$$\texttt{compare}(s_i, s_j; x_0) = \text{sign}((a_j x_0 + b_j) - (a_i x_0 + b_i)),$$

which returns $+1$ if s_j is above s_i, 0 if they coincide, and -1 if s_j is below s_i.

This is the sign of a rationally-valued function, but we can multiply out the denominator to obtain an algebraic function of degree-3 in the segment coordinates. Thus, if the coordinates are expressed as integers, we can determine the sign using at most triple-precision arithmetic.

Computing Intersection Points: (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 we might 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:

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

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

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

(see Fig. 28).

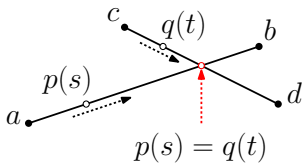


Fig. 28: Plane-sweep algorithm event processing.

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 as functions of the coordinates of a, b, c, and d. (A solution may generally not exist, but this means that the segments are parallel. By our assumption that no segments are collinear, this implies that the segments do not intersect.) Once s and t are known, it remains to just check with $0 \leq s, t \leq 1$, to confirm that the intersection point occurs within the line segment (and not in the extended infinite line).

As in our earlier example of determining the order of segments along the sweep line, if all the coordinates are integers, this yields formulas for s and t as rational numbers, and hence the coordinates of the intersection point are themselves rational numbers. If it is needed to perform exact computations on these coordinates, rather than converting them to floating point, it is possible to save the numerator and denominator of each coordinate as a pair of (multiple precision) integers.

Lecture 5: Polygon Triangulation

The Polygon Triangulation Problem: Triangulation is the general problem of subdividing a spatial domain into simplices, which in the plane means triangles. We will focus in this lecture on triangulating a *simple polygon* (see Fig. 29). Formal definitions will be given later. (We will assume that the polygon has no holes, but the algorithm that we will present can be generalized to handle such polygons.) Such a subdivision is not necessarily unique, and there may be other criteria to be optimized in computing the triangulation.

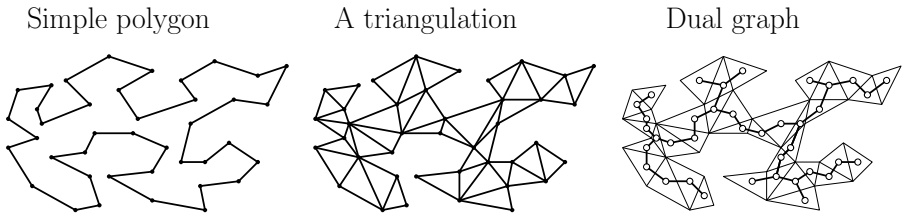


Fig. 29: Polygon triangulation.

Applications: 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 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. 29(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. (If the polygon has holes, then the dual graph will generally have cycles.)

Preliminaries: 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 longstanding 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 (while being a technical tour de force) is so complicated that it is not practical for implementation. 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.

A Triangulation in Two Movements: 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 (see Fig. 30(a)). 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 (see Fig. 30(c)). A closed simple polygonal curve decomposes the plane into two parts, its *interior* and *exterior*. Such a polygonal curve is called a *simple polygon* (see Fig. 30(c)). When we say "polygon" we mean simple polygon.

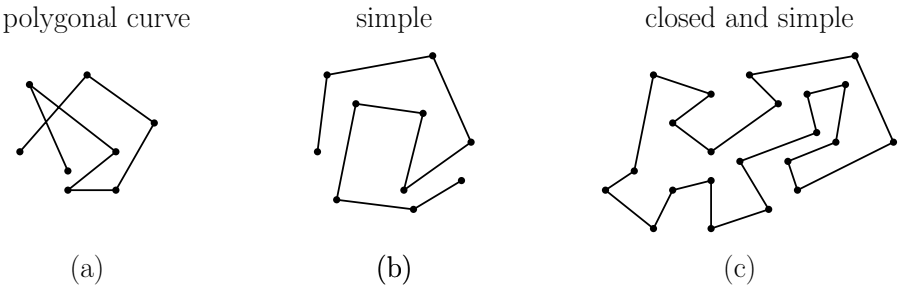


Fig. 30: Polygonal curves and simple polygons.

A polygonal curve C is *monotone* with respect to ℓ if each line that is orthogonal to ℓ 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 curve C is said to be *strictly monotone* with respect to a given line ℓ, if any line that is orthogonal to ℓ intersects C in at most one point. A simple polygon P is said to be *monotone* with respect to a line ℓ if its boundary, (sometimes denoted bnd(P) or ∂P), can be split into two curves, each of which is monotone with respect to ℓ (see Fig. 31(a)).

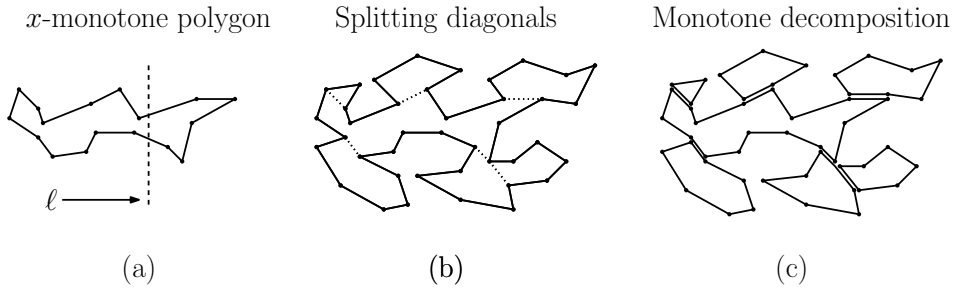


Fig. 31: Monotonicity.

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 curves, 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. Let's make the usual general position assumptions, that no two vertices have the same x-coordinates and no three consecutive vertices are collinear.

We define a *reflex vertex* to be a vertex of the polygon whose interior angle is at least π, and otherwise the vertex is *nonreflex*. We define a *reflex chain* to be a sequence of one or more consecutive reflex vertices along the polygon's boundary.

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. The trickiest aspect of implementing this idea is finding a clean invariant that characterizes the *untriangulated region* that lies to the left of the sweep line.

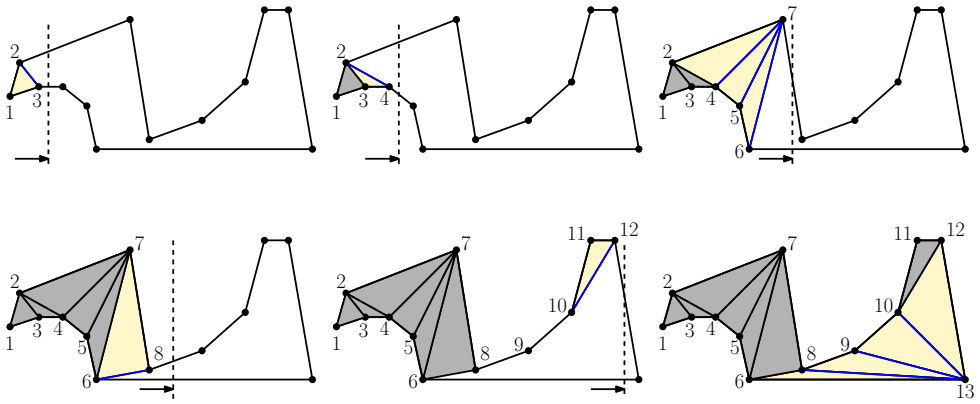


Fig. 32: Triangulating a monotone polygon.

To acquire some intuition, let's consider the example shown in Fig. 32. There is obviously nothing to do until we have at least three vertices. With vertex 3, it is possible to add the diagonal to vertex 2, and so we do this. In adding vertex 4, we can add the diagonal to vertex 2. However, vertices 5 and 6 are not visible to any other nonadjacent vertices so no new diagonals can be added. When we get to vertex 7, it can be connected to 4, 5, and 6. The process continues until reaching the final vertex.

Have we seen enough to conjecture what the untriangulated region to the left of the sweep line looks like? Ideally, this structure will be simple enough to allow 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. 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. 33(a)).

We will prove the invariant by induction, and in the process we will describe the triangulation algorithm. 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. 33(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}.

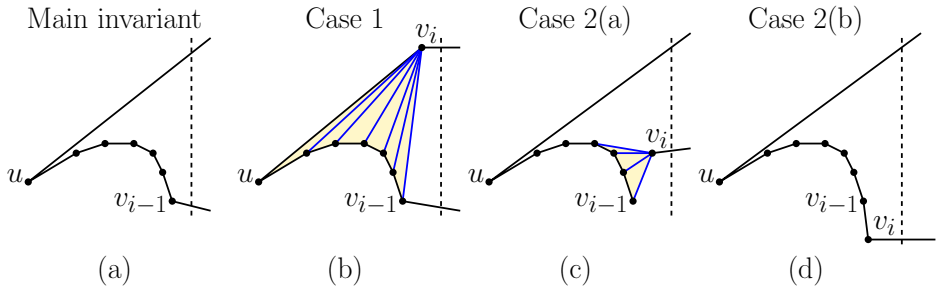


Fig. 33: Triangulation cases.

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: 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. 33(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 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. 33(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. 31).

Observe that the absence of x-monotonicity occurs only at vertices in which the interior angle is greater than 180° 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*.

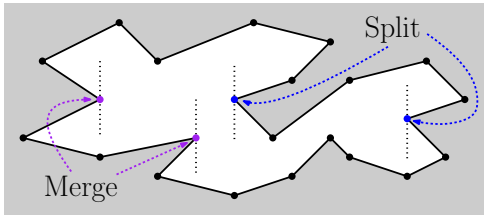


Fig. 34: Merge and split reflex vertices.

Our approach will be to apply a left-to-right plane sweep (see Fig. 35(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. 35(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 on or below e_a on the polygonal chain between e_a and e_b. This vertex may either be on e_a, e_b, or it may lie between them.

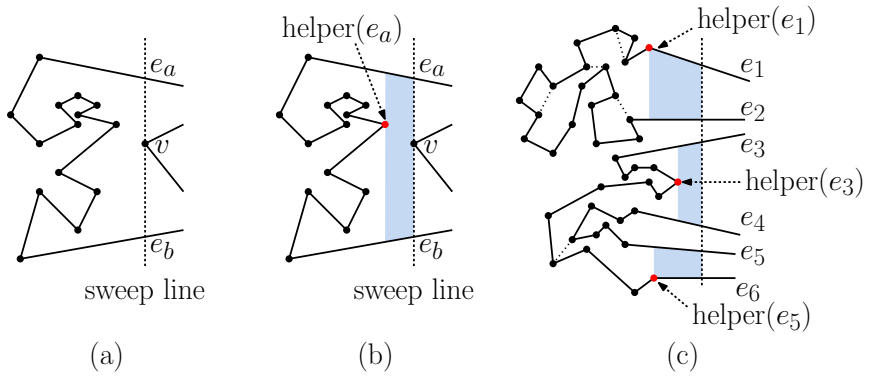


Fig. 35: Split vertices, merge vertices, and helpers.

Another way to visualize the helper is to imagine sweeping out a trapezoid to the left from the sweep line. The top side of the trapezoid lies on e_a, the bottom side lies on e_b, the right side lies on the sweep line, and the left side is sweeps as far as it can until hitting a vertex (see the shaded regions of Figs. 35(b) and (c)).

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. (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. 36). 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). The fix-up function 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'). Set the helper of e to v.

Start vertex(v)**:** (Both edges lie to the right of v, but the interior angle is smaller than π.) 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 larger than π.) 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.

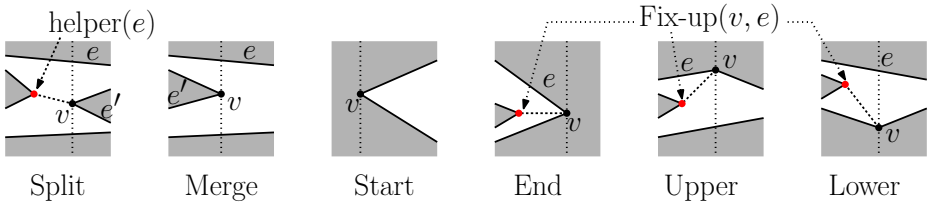


Fig. 36: Plane sweep cases, where v is the vertex being swept. The label e denotes the edge such that helper$(e) \leftarrow v$.

Correctness: Given the number of cases, establishing correctness is a bit of a pain. We will refer you to the 4M's book for a careful proof, but here are main points that need to be established in the proof.

Helpers are correctly updated: Immediately after processing each event, the helpers of all relevant edges have been properly updated.

Merge vertices are correctly resolved: Whenever we encounter a merge vertex, we add a diagonal to resolve this non-monotonicity.

Split vertices are correctly resolved: When a split vertex is visited, it becomes a helper of the edge e immediately above. We will resolve this non-monotonicity when e's helper changes by the invocation of fix-up.

Added diagonals do not intersect each other: Added diagonals lie within a single "helper trapezoid" which has no vertices except on its left and right vertical sides. If both of these vertices are scan reflex vertices (merger on the left and split on the right) we will add a single diagonal to resolve both (see the Split case of Fig. 36).

Analysis: Given a simple polygon with n vertices, there are n events, one for each vertex. Each event involves a constant amount of processing and a constant number of accesses to the

sweep-line dictionary. Thus, the time per event is $O(\log n)$, and hence the overall time is $O(n \log n)$. We have the following:

Theorem: Given an n-vertex simple polygon, in $O(n \log n)$ time, the above sweep-line algorithm correctly add diagonals to decompose it into monotone pieces.

By combining this with the $O(n)$ time algorithm for triangulating a monotone polygon, we obtain the following result.

Theorem: Given a simple polygon with n vertices, it is possible to triangulate it in $O(n \log n)$ time.

Lecture 6: Halfplane Intersection and Point-Line Duality

Halfplane Intersection: Today we begin studying another fundamental topic in geometric computing and convexity. Recall that 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*. We say that a halfplane is either *closed* or *open* depending, respectively, on whether or not it contains the line. Unless otherwise stated, we will assume that halfplanes are closed.

In the *halfplane intersection problem*, we are given a set of n closed halfplanes $H = \{h_1, \ldots, h_n\}$. We may assume that each halfplane is represented by a linear inequality, for example, $h_i = \{(x, y) \mid a_i x + b_i y \leq c_i\}$. The objective is to compute their intersection. It is easy to see that the intersection of halfspaces is a convex polygon (see Fig. 37(a)), but this polygon may be unbounded (see Fig. 37(b)) or even empty (see Fig. 37(c)).

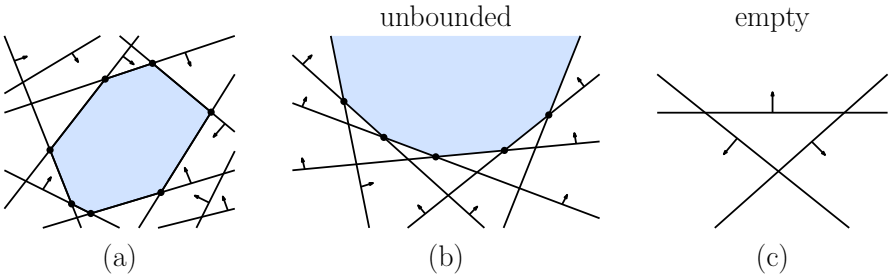


Fig. 37: Halfplane intersection.

Clearly, the number of sides of the resulting polygon is at most n, but may be smaller since some halfspaces may not contribute to the final shape. A natural choice for the output would be the (possibly empty) sequence of indices of the halfplanes that contribute to the final boundary, listed in cyclic order around the boundary. Note that we can easily compute the vertices as the points where adjacent bounding lines intersect.

Divide-and-Conquer Algorithm: Our first approach is based on applying divide and conquer.

(1) If $n = 1$, then just return this halfplane as the answer.
(2) Otherwise, partition H into subsets H_1 and H_2, each of size roughly $n/2$.
(3) Compute the intersections $K_1 = \bigcap_{h \in H_1} h$ and $K_2 = \bigcap_{h \in H_2} h$ recursively.
(4) If either either K_1 or K_2 is empty, return the empty set. Otherwise, compute the intersection of the convex polygons K_1 and K_2 (by the procedure described below).

Let's consider how to carry out step (4), where we intersect two convex polygons, K_1 and K_2 (see Fig. 38(a)). Note that these are somewhat special convex polygons because they may be empty or unbounded.

We can compute the intersection by a left-to-right plane sweep in $O(n)$ time (see Fig. 38(b)). 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, one for the upper chain and one for the lower chain. Hence, the sweep-line status contains at most four points. This implies that updates to the sweep-line status can be performed in $O(1)$ time. Also, we need keep track of a constant number of events at any time, namely the right endpoints of the current segments in the sweep-line status, and the intersections between consecutive pairs of segments. Thus, each step of the plane-sweep process can be performed in $O(1)$ time.

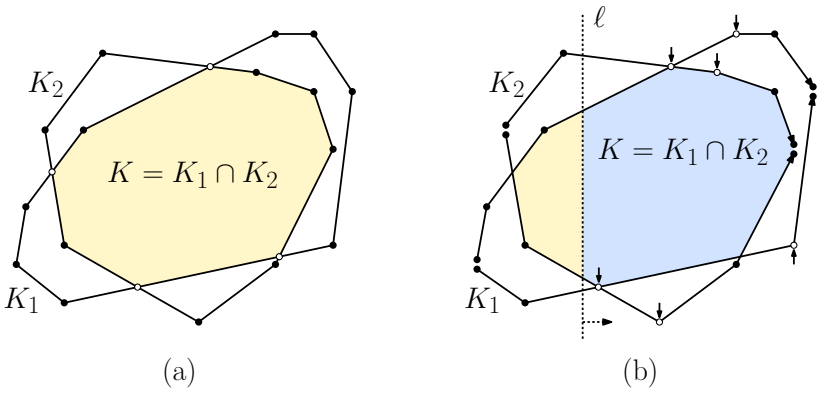


Fig. 38: Intersecting two convex polygons by plane sweep.

The total number of events is equal to the total number vertices, which is n, and the total number of intersection points. It is an easy exercise (which we leave to you) to prove that two convex polygons with a total of n sides can intersect at most $O(n)$ times. Thus, the overall running time is $O(n)$.

Given that we can compute the intersection of two polygons of size n in $O(n)$ time, we obtain the following recurrence (up to constant factors) for the overall running time of the divide-and-conquer algorithm.

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

It follows by standard results on recurrences (consult the *Master Theorem* in CLRS) that $T(n)$ is $O(n \log n)$.

Upper and Lower Envelopes of Lines: Let's next consider a variant of the halfplane intersection problem. Given any set of nonvertical lines $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ in the plane. Each line defines two natural halfplanes, and upper and lower halfplane. The intersection of all the lower halfplanes is called the *lower envelope* of L and the *upper envelope* is defined analogously (see Fig. 39). Let's assume that each line ℓ_i is given explicitly as $y = a_i x - b_i$.

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

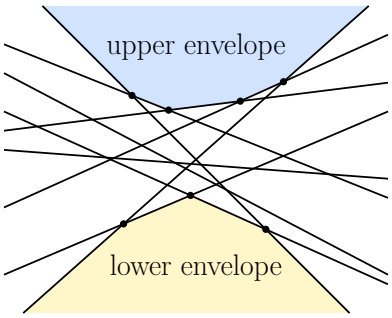


Fig. 39: Lower and upper envelopes.

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 (in fact, essentially the same as) solving the upper convex hull problem. Indeed, 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.

Motivating Duality: Consider any line $\ell : y = cx - d$ in the plane. Just as with point, it takes two real numbers (c and d) to define any line in $\mathbb{R}^2$. Thus, we can identify the lines in $\mathbb{R}^2$ with the set of points (c, d) in an alternative or "dual" space. For example, the line $\ell : y = 2x + 1$ corresponds to the point $(2, -1)$ in dual space, which we denote by ℓ^*. Conversely, each point $p = (a, b)$ is associated with a dual line, $p^* : y = ax - b$, which we denote by p^*.

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.

Primal Relation	**Dual Relation**
Two (nonparallel) lines meet in a point	Two points join to form a line
A point lies on a line	A line passes through a point
A point lies above/below a line	A line passes above/below a point
Three points may be collinear	Three lines may pass through the same point

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. 40(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. 40(b)). (As an exercise, prove this for two general lines.)

Dual Transformation: Let us explore this dual transformation more formally. Duality (or more specifically *point-line duality*) is a transformation that maps points in the plane to lines and

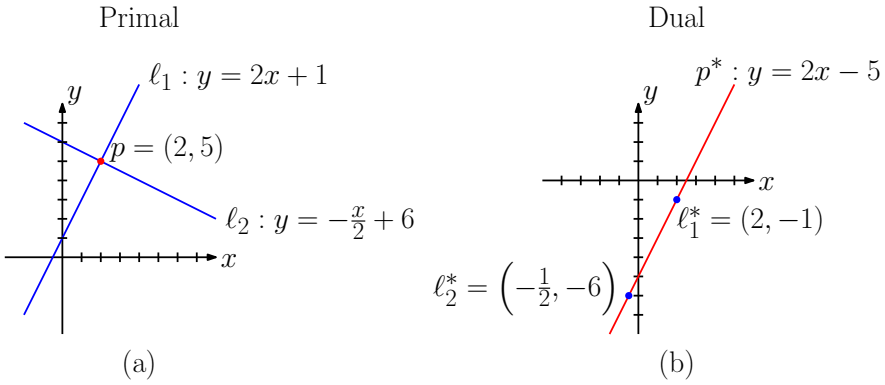


Fig. 40: The primal and dual planes.

lines to point. Given point $p = (a, b)$ and line $\ell : y = cx - d$, we define[4]

$$p^* : y = ax - b \qquad \text{and} \qquad \ell^* = (c, d).$$

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: $p^{**} = p$, $\ell^{**} = \ell$ (This is trivial.)

Incidence Preserving: Point p is on line ℓ if and only if point ℓ^* is on line p^*. (Proof: The first assertion is equivalent to $b = ac - d$ and the second is equivalent to $d = ca - b$. Clearly, one is satisfied if and only if the other is.)

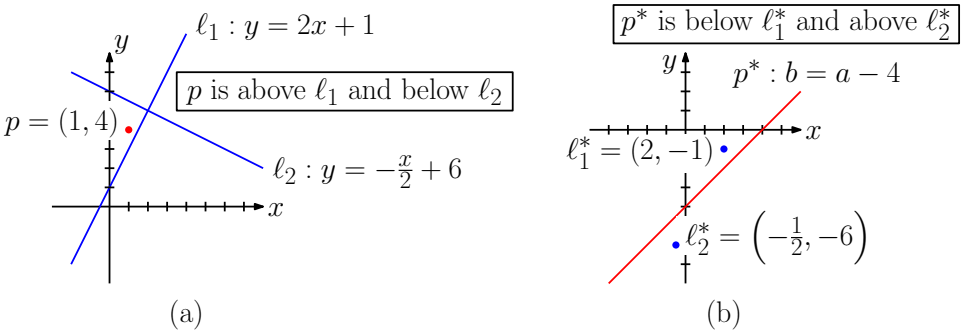


Fig. 41: Order reversal.

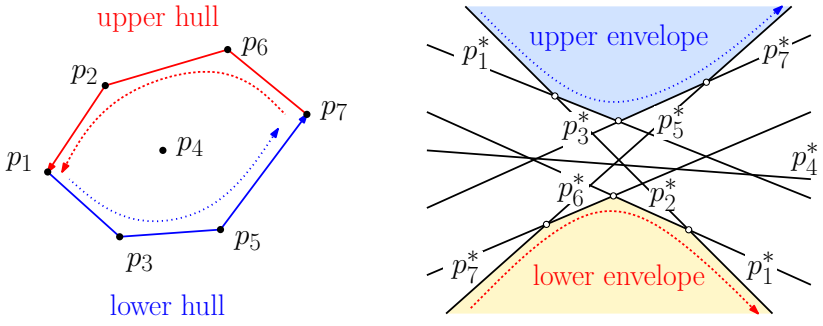
Order Reversal: Point p is above/below line ℓ if and only if line p^* is below/above point ℓ^*, respectively (see Fig. 41). (Proof: The first assertion is equivalent to $b > ac - d$ and the second to $d > ca - b$. Clearly, one is satisfied if and only if the other is.)

Intersection Preserving: Lines ℓ_1 and ℓ_2 intersect at point p if and only if the dual line p^* passes through points ℓ_1^* and ℓ_2^*. (Follows directly from incidence preservation.)

[4]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, where notions of above and below are based on the assumption that the y (or b) axis is "vertical."

Collinearity/Coincidence: Three points are collinear in the primal plane if and only if their dual lines intersect in a common point. (Follows directly from incidence preservation.)

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.



(a) (b)

Fig. 42: 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. 42(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. 42(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 ℓ_{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 ℓ_{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 ℓ_{ij}^*. This is equivalent to saying the ℓ_{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?)

Primal/Dual Equivalencies: There are a number of computational problems that are defined in terms of affine properties of point and line sets. These can be expressed either in primal or in dual form. In many instances, it is easier to visualize the solution in the dual form. We will discuss many of these later in the semester. For each of the following, can you determine what the dual equivalent is?

- Given a set of points P, find the narrowest slab (that is, a pair of parallel lines) that contains P. Define the width of the slab to be the vertical distance between its bounding lines (see Fig. 43(a)).

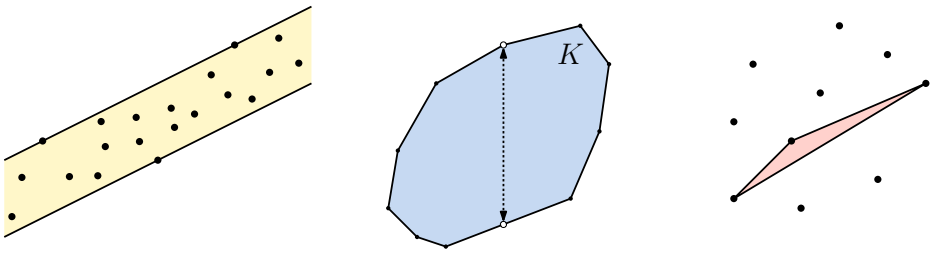


Fig. 43: Equivalence of hulls and envelopes.

- Given a convex polygon K, find the longest vertical line segment with one endpoint on K's upper hull and one on its lower hull (see Fig. 43(b)).
- Given a set of points P, find the triangle of smallest area determined by any three points of P (see Fig. 43(c)). (If three points are collinear, then they define a degenerate triangle of area 0.)

Polar Dual: The dual transform we have described (mapping the point $p = (a, b)$ to the line $y = ax - b$) is just one example of the general concept of a dual transformation. A very popular alternative, called *polar transformation*, maps any nonzero vector $v = (a, b)$ to the line $v^\circ = \{(x, y) : ax + by = 1\}$ and vice versa (see Fig. 44(a)).

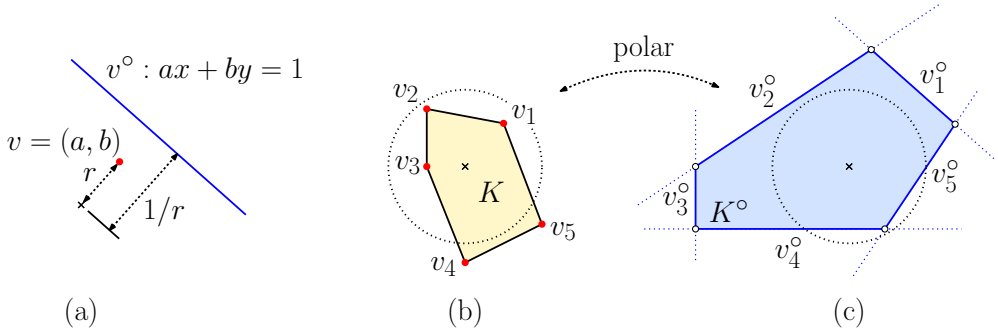


Fig. 44: Polar transformation.

This can be readily extended to any dimension. Given any nonzero vector $v = (a_1, \ldots, a_d)$ its polar is a $(d-1)$-dimensional hyperplane $v^\circ = \{x \in \mathbb{R}^d : \sum_i a_x x_i = 1\}$. This has the

following geometric interpretation. The vector v is transformed to the hyperplane that is perpendicular to v (and on the same side of the origin) and whose distance from the origin is the reciprocal of its length, $\|v\|$ (see Fig. 44(a)).

This transformation is often applied to convex bodies. Suppose that K is a body in $\mathbb{R}^d$ that contains the origin. Its *polar body* K° is formally defined

$$K^\circ \;=\; \{y \in \mathbb{R}^d : \langle y, x\rangle \leq 1, \forall x \in K\},$$

where $\langle y, x\rangle$ is the usual dot product. In the case of a polytope that is defined as the convex hull of verticess $K = \text{conv}(\{v_1, \ldots, v_n\})$, K° is the intersection of the halfspaces defined by v_i° that contain the origin.

There is an interesting reciprocal relationship between K and K°. As the vertices of K get closer to the origin, the nature of polarity causes the corresponding edges of K° be farther from the origin, by the reciprocal. Thus, one would expect that the volumes $\text{vol}(K)$ and $\text{vol}(K^\circ)$ vary inversely—as one gets larger the other gets smaller. Indeed, it is famous result from the theory of convex bodies that for any bounded convex body K that contains the origin in any dimension d, the product $\text{vol}(K) \cdot \text{vol}(K^\circ)$, called the *Mahler volume*, is both upper and lower bounded by a constant (depending on the dimension). A major open problem in mathematics involves determining which shapes achieve the extreme Mahler volume values. It is conjectured that it is minimizes for balls (and ellipses more generally) and it is maximized for simplices.

Higher Dimensions: In d-dimensional space the corresponding notion is a *halfspace*, which is the set of points lying to one side of a $(d-1)$-dimensional hyperplane. The intersection of halfspaces is a *convex polytope*. The resulting polytope will have at most n facets (at most one per halfspace).

The number of vertices, and more generally, the *combinatorial complexity* (total number of vertices, edges, and faces of all dimensions) can vary significantly. Assuming general position, it is reasonable to expect that the combinatorial complexity should be at least $\Omega(n)$, but it can be much higher. A famous result, called *McMullen's Upper-Bound Theorem* states that a polytope with n facets in dimension d can have up to $O(n^{\lfloor d/2\rfloor})$ vertices. (In dimensions 2 and 3, this is linear in the number of halfspaces, but even in dimension 4 the number of vertices can grow to $O(n^2)$.) Obtaining such a high number of vertices takes some care, but the bound is tight in the worst case. There is a famous class of polytopes, called the *cyclic polytopes*, that achieve this bound. Symmetrically, the convex hull of n points in dimension d defines a convex polytope that can have $O(n^{\lfloor d/2\rfloor})$ facets, and this bound is also tight.

Representing Lines and Hyperplanes: (Optional) While we will usually treat geometric objects rather abstractly, it may be useful to explore a bit regarding how lines, halfspaces, and their higher dimensional counterparts are represented. These topics would be covered in a more complete course on projective geometry or convexity.

Explicit Representation: If we think of a line as a linear function of the variable x, we can express any (nonvertical) line by the equation $y = ax + b$, where a is the slope and b is the y-intercept.

In dimension d, we can think of the dth coordinate as being special, and we will make the convention of referring to the d-th coordinate axis as pointing vertically upwards. We can express any "nonvertical" $(d-1)$-dimensional hyperplane by the set of points

$(x_1, \ldots, x_d)$, where $x_d = \sum_{i=1}^{d-1} a_i x_i + b$, thus x_d is expressed "explicitly" as a linear function of the first $d-1$ coordinates.

The associated halfspaces arise replacing "=" with an inequality, e.g., the *upper halfplane* is the set (x, y) such that $y \geq ax + b$, and the *lower halfplane* is defined analogously.

Implicit Representation: The above representation has the shortcoming that it cannot represent vertical objects. A more general approach (which works for both hyperplanes and curved surfaces) is to express the object implicitly as the zero-set of some function of the coordinates. In the case of a line in the plane, we can represent the line as the set of points (x, y) that satisfy the linear function $f(x, y) = 0$, where $f(x, y) = ax + by + c$, for scalars a, b, and c. The corresponding halfplanes are just the sets of points such that $f(x, y) \geq 0$ and $f(x, y) \leq 0$.

This has the advantage that it can represent any line in the Euclidean plane, but the representation is not unique. For example, the line described by $5x - 3y = 2$ is the same as the line described by $10x - 6y = 4$, or any scalar multiple thereof. We could apply some normalization to overcome this, for example, by requiring that $a^2 + b^2 = 1$. If this normalization is performed, then the line has the convenient geometric interpretation that it is orthogonal to the unit vector (a, b) and is at distance c from the origin.

Parametric Representation: One shortcoming of the explict and implicit representations is that it is not particularly easy to define lower dimensional objects. For example, suppose that you wanted to represent the points lying on a line segment in $\mathbb{R}^3$. A *parametric representation* defines a geometric object as the union of points, where an additional numeric parameter is used to specify these points.

To make this more concrete, suppose that we wanted to represent a line segment between two points p and q (in any dimension). We could do this by creating a real-valued parameter t, and defining the segment to be the set of convex combinations

$$\overline{pq} = \big\{(1-t)p + tq \;\big|\; t \in [0, 1]\big\}.$$

Another use is if we are given a point p and a directional vector u, we can represent the points on the ray in the direction of u as the set of points $\text{ray}(p, u) = \{p + tu \mid t \geq 0\}$.

Parametric forms are handy for representing curves. For example, we can describe a helix winding around the x-axis in $\mathbb{R}^3$ as $\{(t, \cos t, \sin t) \mid t \in \mathbb{R}\}$. We can also represent surfaces of any dimension by adding additional parameters. For example, by thinking of s and t as proxies for latitude and longitude, respectively, we can describe any point on a unit sphere in $\mathbb{R}^3$ as

$$\big\{(\sin(s)\cos(t), \sin(s)\sin(t), \cos(s)) \;\big|\; s \in [0, \pi], t \in [0, 2\pi]\big\}.$$

Homogeneous Coordinates: A popular variant to the implicit representation is to use *homogeneous coordinates* to represent points. This involves adding an additional coordinate. For example, we could represent a point (x, y) with the triple $(x, y, 1)$. This additional coordinate is usually set to 1, but there are other uses, depending on the context.

For example, in *projective geometry*, we use $(x, y, 1)$ to represent a "regular" point, and we represent a points "at infinity" using the notation $(x, y, 0)$. This refers to the point that is infinitely far away in the direction of the vector (x, y).

Another example is in *affine geometry*, where we distinguish between points and vectors as separate geometric entities. (Points specify locations and vectors specify direction

and magnitude.) We use $(x, y, 1)$ to represent the point with coordinates (x, y) and we use $(x, y, 0)$ to represent the vector from the origin to this point.

One advantage of homogeneous coordinates is that it simplifies dual representations. For example, we can describe any line in the plane as homogeneous 3-vector $\ell = (a, b, c)$. A point $p = (x, y, 1)$ lies on this line if and only if $\ell \cdot p = ax + by + c = 0$. (The name "homogeneous" derives from the fact that by eliminating the scalar term, we have a homogeneous equation.)

In dimension d, a point is represented by a $(d + 1)$-vector $p = (x_1, \ldots, x_d, 1)$ and a hyperplane by a $(d + 1)$-vector $h = (a_1, \ldots, a_{d+1})$, and a point lies on the associated hyperplane if $h \cdot p = \sum_{i=1}^{d+1} a_i x_i = 0$.

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 perhaps the simplest and best known example of multi-dimensional constrained optimization problems. In constrained optimization, the objective is to find a point in d-dimensional space that minimizes (or maximizes) a given *objective function*, subject to satisfying a set of *constraints* on the set of allowable solutions.

Linear programming is perhaps the simplest example of a constrained optimization problem, since both the constraints and objective function are linear functions. 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). There are, however, a number of useful (and even surprising) applications of linear programming in low-dimensional spaces.

A Bit of History: The problem of solving a system of linear inequalities dates back to at least the 1800's, where it was studied by Joseph Fourier (of "Fourier Transform" fame). Serious study of linear programming started in the 1940's where it was developed (independently) by Soviet mathematician and economist Leonid Kantorovich and George Dantzig and John von Neumann. Dantzig developed the *simplex algorithm*, which provided a practical method for solving large[5] instances of LP.

While simplex is fast in practice, it is known that it can take exponential time. In his original list of NP-hard problems, Richard Karp listed LP as one of the major open problems that are not known to be solvable in polynomial time or NP-hard. In 1979, Leonid Khachiyan had a major breakthrough by showing that LP could be solved in (weakly) polynomial time through a method called the *ellipsoid algorithm*. The term "weakly" means that the running time is polynomial not only in the input size, but also in the number of bits in the numbers involved. The ellipsoid algorithm was primarily of theoretical interest, but in 1984 Narendra Karmarkar developed a class of (also weakly polynomial) algorithms called *interior-point methods*. These are quite practical, and are widely used today.

The question of whether there is a (strongly) polynomial time algorithm for LP is among the most important open problems in computer science (right up there with $P = NP$). In

[5] Well, back in the 1940's, solving 100 inequalities was considered "large"! But today, this algorithm solves instances involving hundreds of thousands constraints.

this lecture, we will discuss a linear time algorithm for LP, but with the constraint that the dimension is a fixed constant.

Problem Definition: 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*[6] (see Fig. 45(a)).

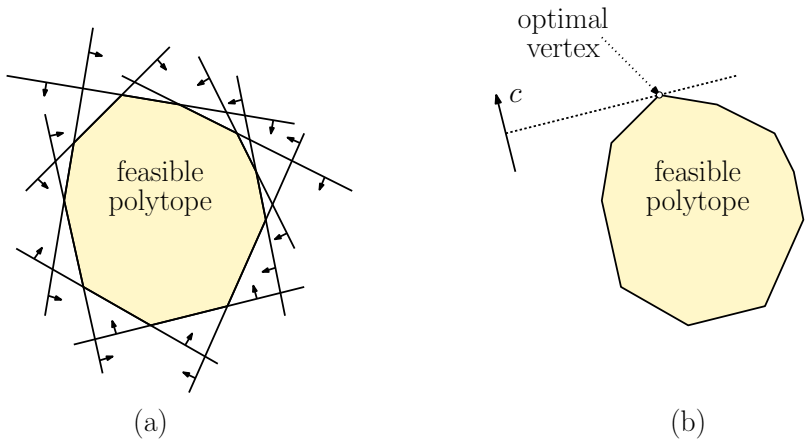


Fig. 45: 2-dimensional linear programming.

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 $c_1x_1 + \cdots + c_dx_d$, by specifying the coefficients c_i. (Again, there 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. 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* (see Fig. 45(b)).

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

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

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

$$\begin{array}{ll} \text{Maximize:} & c^{\mathsf{T}}x, \\ \text{Subject to:} & Ax \leq b. \end{array}$$

[6]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.

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. 46(a)).

Infeasible: The feasible polytope is empty, and there is no solution (see Fig. 46(b)).

Unbounded: The feasible polytope is unbounded in the direction of the objective function, and so no finite optimal solution exists (see Fig. 46(c)).

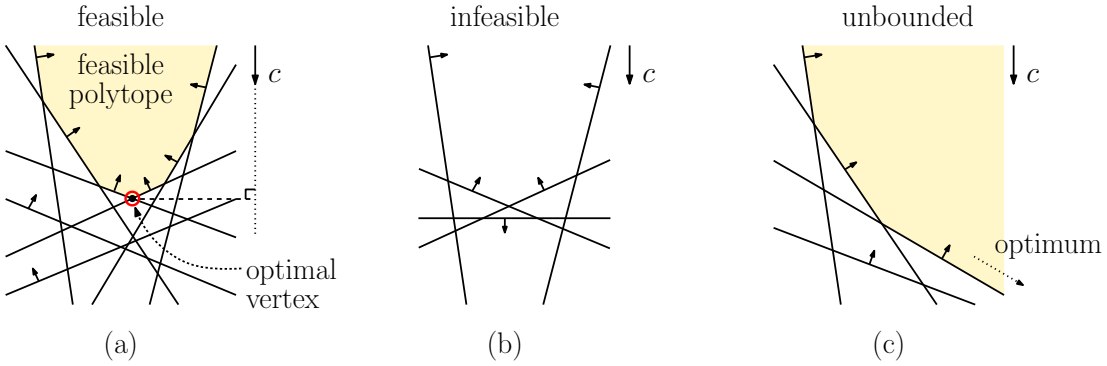


Fig. 46: Possible outcomes of linear programming.

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 downwards. (There is no loss of generality here, because we can always rotate space so that c is parallel any direction we like.) In this setting, the problem is just that of finding the lowest vertex (minimum y-coordinate) of the feasible polytope.

Example – Separating Sets in the Plane: As an example of how LP can be used to solve problems in computational geometry, consider the following question, which inspired from Support Vector Machines (SVMs) in machine learning. We are given two sets of points R (for red) and B (for blue) in $\mathbb{R}^2$ (see Fig. 47(a)). Let n denote the total number of points between both sets. Define a *corridor* to be the region bounded between two parallel lines in $\mathbb{R}^2$. Assuming the lines are not vertical, *vertical width* of the corridor is the vertical distance between the two lines, or equivalently, the difference between their y-intercepts in the graphs of the two lines. The problem is to compute the corridor of maximum vertical width that separates the two points. There are two cases, B above and R below and vice versa. We can explain how to solve just the first case, since the other case is symmetrical.

We will show that this problem can be reduced to LP in $\mathbb{R}^3$. First, let us denote the lines bounding the corridor as $\ell^+ : y = ex + f^+$ and $\ell^- : y = ex + f^-$. We wish to the determine the values of e, f^+, and f^- to maximize the difference in the y-intercepts, $f^+ - f^-$, such that the points of B lie above ℓ^+ and the point of R lie below ℓ^- (see Fig. 47(b)). Let $P = \{p_1, \ldots, p_n\}$ be the points, where $p_i = (p_{i,x}, p_{i,y})$. Let $B = \{p_1, \ldots, p_m\}$ and let $R = \{p_{m+1}, \ldots, p_n\}$. The constraint that the points of B lie above ℓ^+ can be expressed as

$$p_{i,y} \;\geq\; ep_{i,x} + f^+, \quad \text{for } 1 \leq i \leq m,$$

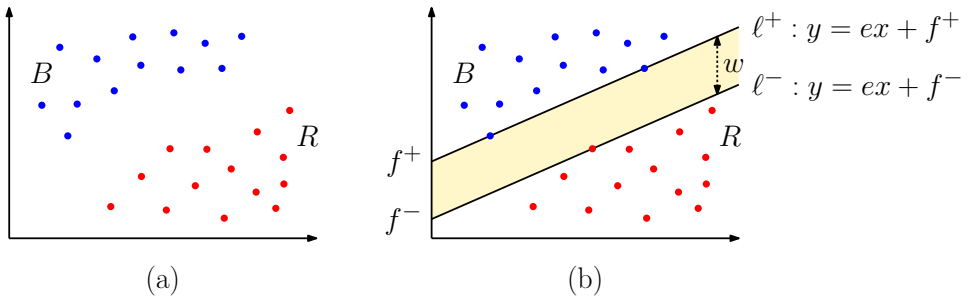


Fig. 47: Maximum vertical-width separating corridor.

and the constraint that the points of R lie below ℓ^- can be expressed as

$$p_{j,y} \leq ep_{j,x} + f^-, \quad \text{for } m+1 \leq j \leq n.$$

The unknown values in the problem are e, f^+, and f^-, which can be expressed as a (symbolic) column vector $x = (e, f^+, f^-)^\mathsf{T} \in \mathbb{R}^3$. The objective function is $f^+ - f^- = 0 \cdot e + 1 \cdot f^+ - 1 \cdot f^- = c^\mathsf{T} x$, where $c^\mathsf{T} = (0, +1, -1)$. We can rewrite the above constraints in the form expected by Eq. (1) as follows

$$\begin{aligned} p_{i,x} \cdot e &+ 1 \cdot f^+ + 0 \cdot f^- \leq p_{i,y}, && \text{for } 1 \leq i \leq m, \\ -p_{j,x} \cdot e &+ 0 \cdot f^+ - 1 \cdot f^- \leq -p_{j,y}, && \text{for } m+1 \leq j \leq n. \end{aligned}$$

This can be expressed as an LP in standard form as follows. Find $x = (e, f^+, f^-)^\mathsf{T}$ that maximizes $c^\mathsf{T} x$, where $c^\mathsf{T} = (0, +1, -1)$ subject to the following constraints:

$$\begin{bmatrix} p_{1,x} & 1 & 0 \\ \vdots & \vdots & \vdots \\ p_{m,x} & 1 & 0 \\ -p_{m+1,x} & 0 & -1 \\ \vdots & \vdots & \vdots \\ -p_{n,x} & 0 & -1 \end{bmatrix} \begin{bmatrix} e \\ f^+ \\ f^- \end{bmatrix} \leq \begin{bmatrix} p_{1,y} \\ \vdots \\ p_{m,y} \\ -p_{m+1,y} \\ \vdots \\ -p_{n,y} \end{bmatrix}$$

Along with the objective vector $c^\mathsf{T} = (0, +1, -1)$, this is just an instance of LP in $\mathbb{R}^3$.

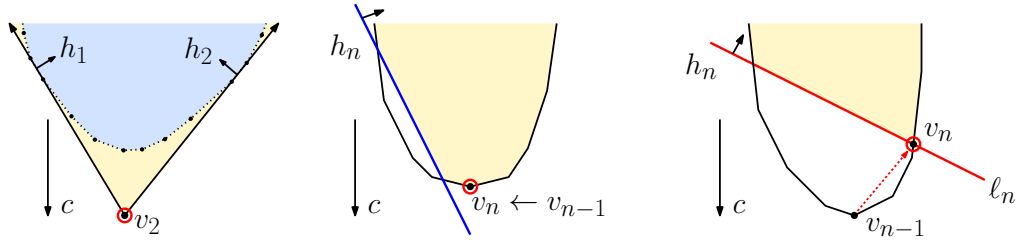
Our formulation contains a minor (subtle) error. The problem description implies that the corridor has a nonnegative vertical width, that is, $b^+ \geq b^-$. But we have done nothing to enforce this. We could either add one additional constraint that $b^+ \geq b^-$, or we could run the LP, and test that this holds in the final answer. (Since we are maximizing $b^+ - b^-$, if there is a positive-width solution, the LP will return it.) The original formulation always returns a feasible answer. If we add the addition constraint that the width is nonnegative, the LP might return "infeasible". This happens if it is not possible to separate the two sets by any line.

Solving LP in Spaces of Constant Dimension: While most instances of LP in practice involve both large numbers of points and high dimensionality, 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 $H = \{h_1, \ldots, h_n\}$ 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.[7] 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.[8] We denote this vertex as v_d (see Fig. 48(a)). We will then add halfspaces one by one, $h_{d+1}, h_{d+2}, \ldots, h_n$, and update the current optimal vertex after each insertion. For $1 \leq i \leq n$, let $H_i = \{h_1, \ldots, h_i\}$ denote the first i constraints, and let v_i denote the associated optimum vertex. Clearly, v_n will be the final solution to the LP. (In our figures, c points vertically downwards, so successive v_i's will move upwards as more constraints are considered.)



(a) (b) (c)

Fig. 48: (a) Starting the incremental construction, (b) v_{n-1} is feasible for h_n, and (c) v_{n-1} is not feasible, (d) the importance ofthe proof that the new optimum lies on ℓ_n.

Let's do this by induction. Assume that we have already correctly computed v_{n-1}, based on the first $n-1$ constraints H_{n-1}, and all that remains is to add the final constraint h_n. (The recursion bottoms out with v_d.) There are two cases that can arise:

Feasible: ($v_{n-1} \in h_n$) The optimum vertex does not change (see Fig. 48(b)). Set $v_n \leftarrow v_{n-1}$.

Conflict: ($v_{n-1} \notin h_n$) We need to update v_n as described below (see Fig. 48(c))).

The key observation for updating the optimum is presented in the following claim, which states that the new optimum vertex lies on the boundary of the new constraint.

[7] In practice, LP designers know that their problem is feasible, and it is usually easy to create such a set of constraints based on knowledge of the problem being solved. However, generating this starting set automatically given just the inputs A, b, and c is an interesting exercise. Our textbook explains how to overcome these assumptions in $O(n)$ additional time.

[8] In typical CG style, we will not explain how this is done, but it reduces to solving a system of d linear equations in $\mathbb{R}^d$, which can be done in $O(d^3)$ time through Gaussian elimination.

Lemma: If after the addition of constraint h_n the LP is still feasible but the optimum vertex changes (conflict case above), then the new optimum vertex lies on the hyperplane bounding h_n.

Proof: Let ℓ_n denote the bounding hyperplane for h_n. Let v_{n-1} denote the old optimum vertex. Suppose towards contradiction that the new optimum vertex v_n does *not* lie on ℓ_n (see Fig. 49(a)). Since $v_{n-1} \notin h_n$ and $v_n \in h_n$, the line segment $\overline{v_{n-1}v_n}$ must cross ℓ_n. Let $p = \overline{v_{n-1}v_n} \cap \ell_n$ denote the crossing point. By convexity, the line segment $\overline{v_{n-1}v_n}$ lies entirely within the feasible region after stage $n-1$, and therefore p itself is feasible. By linearity, the objective function decreases monotonically (gets better) as we walk from v_n to v_{n-1}. Therefore p is a better solution than v_n, a contradiction.

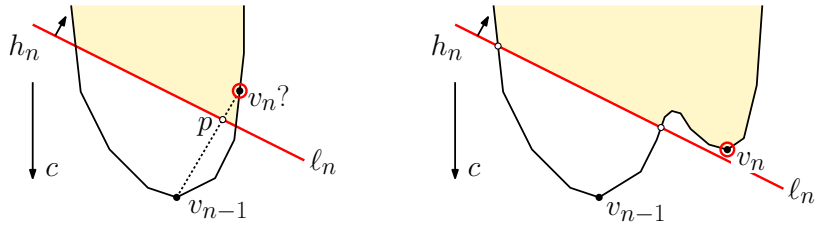


Fig. 49: (a) Proof that the new optimum vertex lies on the newly added constraint, and (b) on the importance of convexity.

Our algorithm will make critical use of the fact that the new optimum lies on ℓ_n. Convexity is important since this may fail if the feasible region is not convex (see, e.g., Fig. 49(a)).

Recursively Updating the Optimum Vertex: Using this observation, we can reduce the problem of finding the new optimum vertex to an LP problem in one lower dimension. Let us consider an instance where the old optimum vertex v_{n-1} does not lie within h_n (see Fig. 50(a)). Let ℓ_n denote the bounding hyperplane for the halfspace h_n. We intersect each of the halfspaces $\{h_1, \ldots, h_{n-1}\}$ with ℓ_n and project the results down one dimension to $\mathbb{R}^{d-1}$. (For an explanation of how this is done, see the remarks at the end of the lecture notes.) We do the same for the objective function (see Fig. 50(b)). This yields an LP problem involving $n-1$ halfspaces in dimension $d-1$, which we solve recursively. Finally, we "unproject" the solution back onto ℓ_n to obtain the final solution v_n (see Fig. 50(c).

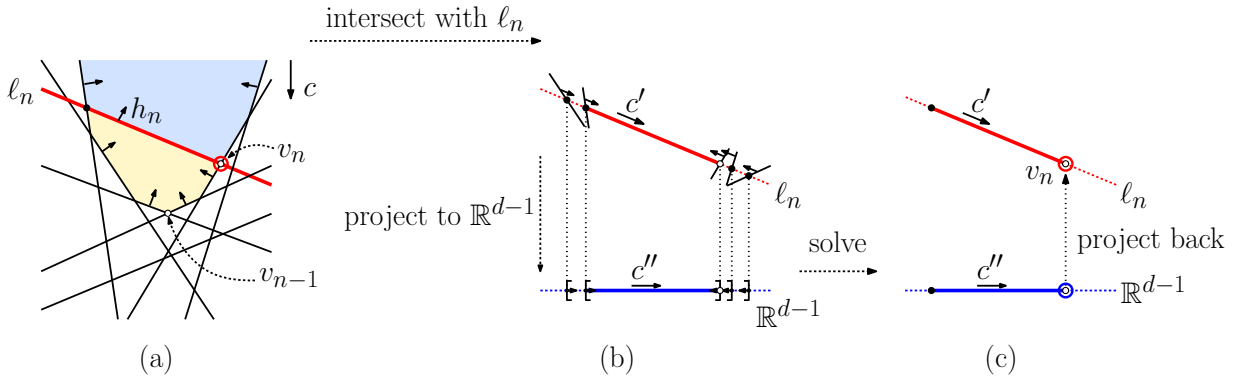


Fig. 50: Reducing the problem one dimension lower.

The recursion "bottoms out" when we reach 1-dimensional space (see the bottom of Fig. 50(b)). Each constraint is a simple inequality (either $v \leq a_i$ or $v \geq a_i$), and the objective function degenerates to simply "find the largest" or "find the smallest" feasible point. This is trivially solvable in $O(n)$ time. Since the original LP was bounded, the resulting 1-dimensional LP will also be bounded, but it might be infeasible.

Worst-Case Analysis: What is the running time of this algorithm? It turns out that the running time is sensitive to the order in which the halfspaces are inserted. (The complete algorithm is described below.) Let's consider what happens assuming the worst possible insertion order.

Since the algorithm is recursive, it is natural to express the running time as a recurrence. Let $W_d(n)$ denote the worst-case running time of our LP algorithm for n halfspaces in $\mathbb{R}^d$. Recall that we needed to begin by assuming that we have a bounded solution involving d halfspaces. This fact will simply complicate the analysis, and so for the sake of setting the basis cases, let's assume that $W_d(1) = 1$ and $W_1(n) = n$.

For general n and d, we begin by recursively computing the optimum LP solution for the first $n-1$ halfspaces, which takes $W_d(n-1)$ time. Next, we check whether the optimum vertex v_{n-1} is feasible for the final halfspace h_n. This can be tested in just $O(d)$ time. If it is feasible, we are done. Otherwise, we need to apply the projection process and invoke a $(d-1)$-dimensional LP on $n-1$ halfspaces. The projection/unprojection process can be carried out in time $O(d(n-1)) = O(dn)$. (See the remarks at the end of the notes.) By induction, it takes $W_{d-1}(n-1)$ time to compute the lower-dimensional LP. In the worst case, the "conflict scenario" always occurs, meaning that each step takes time $[dn + W_{d-1}(n-1)]$. Ignoring constant factors, this yields the following recurrence:

$$W_d(n) = \begin{cases} 1 & \text{if } n = 1, \\ n & \text{if } d = 1, \\ W_d(n-1) + d + \big[dn + W_{d-1}(n-1)\big] & \text{otherwise.} \end{cases}$$

The term inside the square brackets of the last case above represents the cost for processing the conflict scenario. This algorithm is in fact not very efficient. A detailed ananlysis show that its running time is $W_d(n) = O(n^d)$.[9]

Considering that n may be very large, a running time of $O(n^d)$ is unacceptably slow even in $\mathbb{R}^2$. This worst-case analysis is based on the *very pessimistic* assumption that ever insertion leads to the conflict scenario, where current optimum is not feasible for the newly added constraint. But is this worst-case realistic? Next, we'll consider a better strategy based on simply randomizing the insertion order.

Randomized Algorithm: Suppose that we apply the above algorithm, but we insert the halfspaces in *random order*. (As mentioned above, we need to start with d halfspaces to obtain our initial solution, but in our analysis, we will ignore this messy detail.) This is an example of a general class of algorithms called *randomized incremental algorithms*. A description is given in the code block below.

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. In particular, make

[9] Here is a quick-and-dirty analysis, which will hopefully convince you that this claim is believable. Suppose we were to ignore the "$d + dn$" term and replace the $W_{d-1}(n-1)$ with $W_{d-1}(n)$. Then, we would have the recurrence $W_d(n) = W_d(n-1) + W_{d-1}(n)$. Does this look familiar? It is essentially the same as Pascal's famous recurrence for the binomial coefficients, $\binom{n}{d} = \binom{n-1}{d} + \binom{n}{d-1}$. It is well known that $\binom{n}{d} = O(n^d)$, and so it is not surprising that we essentially get the same asymptotic bound for our recurrence.

Input: A set $H = \{h_1, \ldots, h_n\}$ of halfspaces in $\mathbb{R}^d$, such that the first d define an initial feasible vertex v_d, and the objective vector c.

Output: The optimum vertex v or an error status indicating that the LP is infeasible.

(1) If $d = 1$, solve the LP by brute force in $O(n)$ time

(2) Find an initial subset of d halfspaces $\{h_1, \ldots, h_d\}$ that provide a bounded solution v_d. (If no such set exists, report that the LP is unbounded.)

(3) Randomly select a halfspace from the remaining set $\{h_{d+1}, \ldots, h_n\}$. Call this h_n. Recursively solve the LP on the remaining $n-1$ halfspaces, letting v_{n-1} denote the result. (If the LP is infeasible, then return this.)

(4) If $(v_{n-1} \in h_n)$ return v_{n-1} as the final answer

(5) Otherwise, intersect $\{h_1, \ldots, h_{n-1}\}$ with the $(d-1)$-dimensional hyperplane ℓ_n that bounds h_n and project onto $\mathbb{R}^{d-1}$. Let c' be the projection of c onto ℓ_n and then onto $\mathbb{R}^{d-1}$. Recursively solve the resulting $(d-1)$-dimensional LP with $n-1$ halfspaces. (If the LP is infeasible, then return this.) Project the optimal vertex back onto ℓ_n, and return this point.

---

no assumptions about the distribution of the input. (This is an important characteristic of randomized algorithms. The performance might be affected by the random number generator, but not the input the user gives us.)

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.

Computing the Minimum (Optional): To motivate how backwards analysis works, let us consider a much simpler example, namely the problem of computing the minimum of a list of numbers. Suppose that we are given a sequence S of n distinct numbers. We permute the sequence and inspect them one-by-one. We maintain a variable that holds the smallest value seen so far. If we see a value that is smaller than the current minimum, then we *update* the current smallest. Of course, this takes $O(n)$ time, but the question we will consider is, in expectation *how many times does the current smallest value change*?

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.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
5	9	4	11	2	6	8	14	0	3	13	12	1	7	10

Let p_i denote the probability that the minimum value changes on inspecting the ith number of the random permutation. Thus, with probability p_i the minimum changes (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 ith value from the remaining $n-i+1$ elements of S. However, this leads to a complicated analysis involving conditional probabilities. (For example, if the minimum is among the first $i-1$ elements, $p_i = 0$, but if not then it is surely positive.) Let us instead consider an alternative approach, in which we work *backwards*. In particular, let us fix the first i values, and then 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 the $n!$ permutations that could have resulted in S_i, each of the $i!$ permutations of these first i elements are equally likely to occur. For how many of these permutations does the minimum change in the transition from S_{i-1} to S_i? Clearly, the minimum changes only for those sequences in which the smallest element of S_i is the ith element itself. 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 $\sum_i \frac{1}{i}$ is the *Harmonic series*, and it is a well-known fact that it is nearly equal to $\ln n$. (See, e.g., the Wikipedia entry for Harmonic Series.)

Note that by fixing S_i, and considering the possible (random) transitions that lead from S_{i-1} to S_i, we avoided the need to consider any conditional probabilities. 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 "backwards" 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 $T_d(n) = O(n)$, where the constant factor grows exponentially with the dimension. More precisely, we will show that $T_d(n) \leq \gamma\, d!\, n$, where γ is some constant that does not depend on dimension. It will make the proof simpler if we start by proving that $T_d(n) \leq \gamma_d\, d!\, n$, where γ_d does depend on dimension, and later we will eliminate this dependence.

Recall that our algorithm was complicated by the need to start with a solution involving d halfspaces. It will simplify the analysis to ignore this technical detail. As we did in our worst-case analysis, we will start with the basis cases $T_d(1) = 1$ and $T_1(n) = n$. Our randomized analysis will depend on the random event of whether we execute the inexpensive step (4) or the expensive step (5). Given $n \geq 1$, let p_n denote the probability that the insertion of the nth hyperplane in the random order results in a change in the optimum vertex. Before we derive the value of p_n, let's see how it affects our execution time.

Case 1: With probability $(1-p_n)$ there is no change in the optimum. It takes us $O(d)$ time to determine that this is the case (but we pay this cost irrespective of whether $v_{n-1} \in h_n$)

Case 2: With probability p_n, there is a change to the optimum. This means that we need to apply the projection process (which we saw earlier can be done in time $O(dn)$) and then invoke a $(d-1)$-dimensional LP involving $n-1$ halfspaces (which takes $T_{d-1}(n-1)$ time).

In either case, we start by solving an LP involving $n-1$ halfspaces, which requires $T_d(n-1)$ expected time. This suggests the following recurrence for the expected execution time:

$$T_d(n) = \begin{cases} 1 & \text{if } n = 1, \\ n & \text{if } d = 1, \\ T_d(n-1) + d + p_n(dn + T_{d-1}(n-1)) & \text{otherwise.} \end{cases}$$

The last case is the interesting one for us, since we'll be deriving an upper bound, we can simplify the last recursive term by replacing $n-1$ with n, yielding $T_d(n) \le T_d(n-1) + d + p_n(dn + T_{d-1}(n))$.

It remains is to determine what p_n is. Assuming general position, the final optimal vertex v_n is determined by the intersection of d halfspaces. This vertex is feasible with respect to all the other $n-d$ halfspaces. Since the final halfspace h_n was chosen randomly, the probability that it is among the d halfspaces that determine the final optimum is d/n. Therefore,

- With probability $p_n = d/n$, inserting h_n causes the optimum to change (see Fig. 51(b))
- With probability $1-p_n$, inserting h_n leaves the optimum unchanged (see Fig. 51(c))

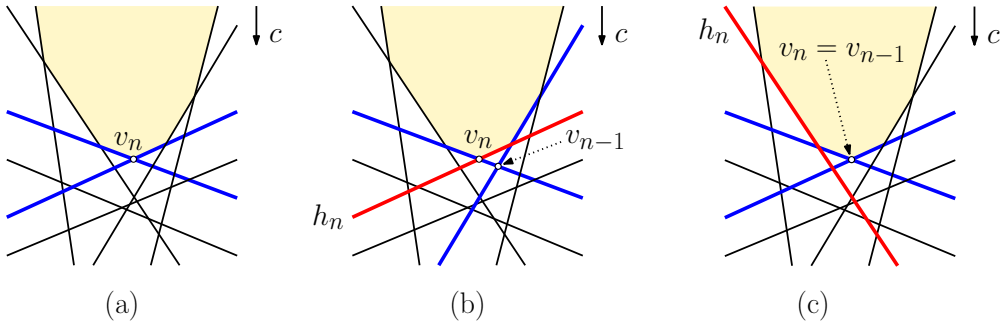


Fig. 51: Backwards analysis for the randomized LP algorithm.

Returning to our analysis, setting $p_n = d/n$ and applying our induction hypothesis (that $T_d(n) \le \gamma\, d!\, n$) we have

$$\begin{aligned} T_d(n) &\le T_d(n-1) + d + p_n\big(dn + T_{d-1}(n)\big) \\ &\le \gamma_d d!(n-1) + d + \frac{d}{n}\big(dn + (\gamma_{d-1}(d-1)!\, n)\big) \\ &= \gamma_d d!(n-1) + (d + d^2 + \gamma_{d-1} d!) \\ &= \gamma_d d!\, n + \big[d + d^2 + \gamma_{d-1} d! - \gamma_d d!\big]. \end{aligned}$$

To complete the induction proof, it suffices to show that the final term in brackets is less than zero. That is, we want to select γ_d such that

$$d + d^2 + \gamma_{d-1} d! - \gamma_d d! \le 0 \qquad \text{or equivalently} \qquad \gamma_d d! \ge d + d^2 + \gamma_{d-1} d!$$

To satisfy this we can set $\gamma_1 = 1$ and then for $d = 2, 3, \ldots$ we define

$$\gamma_d \leftarrow \frac{d + d^2}{d!} + \gamma_{d-1},$$

Recalling that d is a constant, it follows that γ_d is a constant, and therefore the final running time is $O(n)$, where the constant factor is dominated by $d!$.

Eliminating the Dependence on Dimension: As mentioned above, we don't like the fact that the "constant" γ_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 γ (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.

Why Randomization is Okay: 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? Can't we find an equally fast deterministic algorithm?

The answer to both questions is "yes". Unfortunately, the simplest deterministic algorithm is much more complex than the randomized algorithm. Also, in his original paper, Seidel proves that the probability that the algorithm exceeds its running time by a factor b is $O((1/c)^{bd!})$, 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!

Summary: We have presented a simple and elegant randomized incremental algorithm for solving linear programming in spaces of constant dimension. 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. In future lectures, we will see that there are numerous geometric problems that can be efficiently solved by reduction to LP (or something that is similar to LP).

Projecting Constraints: (Optional) Earlier in the lecture, we omitted discussion of the technical details on how to intersect constraint the various halfspaces h_j with the final hyperplane ℓ_n, and project the result down to $\mathbb{R}^{d-1}$ (recall Fig. 50). We mentioned there that this is essentially performing one step of Gauss elimination. Here are the technical details.

Let ℓ_n denote the hyperplane that bounds the final halfspace. We may assume it is given by the equation:

$$\ell_n : a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,d}x_d = b_n.$$

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

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

The idea is to apply one step of Gauss elimination using the equation of ℓ_n to eliminate a variable from all the other inequalities. By general position, we may assume that $a_{n,1} \neq 0$. Consider an arbitrary constraint h_j that we wish to intersect with ℓ_n:

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

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

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

To see that this works, consider an arbitrary point x on the hyperplane ℓ_n, which is equivalent to saying that $A_n x = b_n$. Suppose as well that that x satisfies constraint h_j, that is, $A_j x \leq b_j$. Then we have

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

Thus, every point that satisfies the original constraint also satisfies the modified constraint. The converse holds by a symmetrical argument. A similar elimination can be performed to the objective vector $\vec{c}$. Since the first term of each equation vanishes (is now zero), we are left with a $d-1$ dimensional problem. Reversing the process allows us to project the $d-1$ dimensional solution back into d-space.

Lecture 8: Trapezoidal Maps

Trapezoidal Map: Many techniques in computational geometry are based on generating decomposing a complex arrangement of objects into a collection of simple objects. We have seen triangulations as one example, where the interior of an n-vertex simple polygon is subdivided into a collection of $n-2$ triangles. Today, we will consider a considerably more general technique for subdividing space in the midst of a collection of disjoint line segments.

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 perhaps that two segments can intersect at their endpoints. (We allow segments to share common endpoints so that our results can be generalized to planar graphs and planar subdivisions.) Let us make the general-position assumptions that no two endpoints have the same x-coordinate, and (hence) 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. 52(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 (see Fig. 52(b)). 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. 52(c)), denoted $\mathcal{T}(S)$.

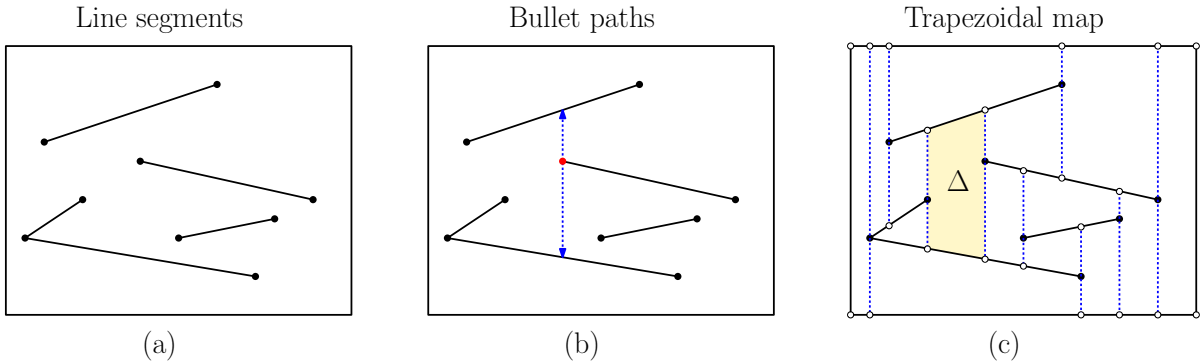


Fig. 52: A set of segments and the associated trapezoidal map.

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 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. This was not the case for the triangulations that we discussed earlier, where adjacent triangles met only along complete edges. (In the terminology of topology, a trapezoidal map is *not* a *cell complex*, while a triangulation is.) Trapezoidal maps are useful data structures, because the provide a way to convert a possibly disconnected collection of segments into a structure that covers the plane.

We begin by showing 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.

Lemma: Given set S of n line segments in the plane, the resulting trapezoidal map $\mathcal{T}(S)$ has at most $6n + 4$ vertices and $3n + 1$ trapezoids.

Proof: Each segment generates six vertices in the map, two for the segment's endpoints, two from the upward vertical rays, and two from the downward vertical rays (see Fig. 53(a)). In addition, there are four vertices for the bounding rectangle, resulting in a total of $6n + 4$ vertices.

Let us charge each trapezoid to the vertex along its left edge. Each segment is charged by three trapezoids, since the left endpoint of each segment is charged twice, one from the trapezoid above-right and one from the trapezoid below-right, and the right endpoint is charged once by the trapezoid to its right (see Fig. 53(b)). This yields a total of $3n$ trapezoids. There is one more trapezoid, namely the leftmost one, for a total of $3n + 1$.

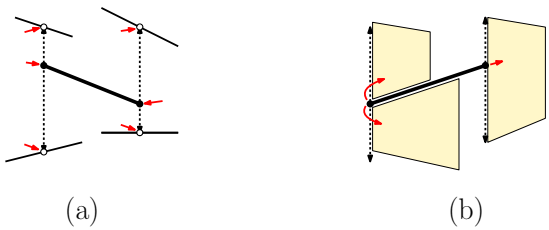


Fig. 53: (a) The six vertices associated with each segment and (b) the three left-bounding trapezoids associated with each segment.

An important fact to observe about each trapezoid is that 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 by a straightforward application of plane sweep. (By now, this should be an easy exercise for you. 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.*[10]

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 (randomly permuted) segments, and let $\mathcal{T}_i$ denote the resulting trapezoidal map.

To perform this update, we need to know which trapezoid of the current map contains the left endpoint of the newly added segment. 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. 54). 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. 54, where we insert a new segment (red) into the trapezoidal map from Fig. 52.

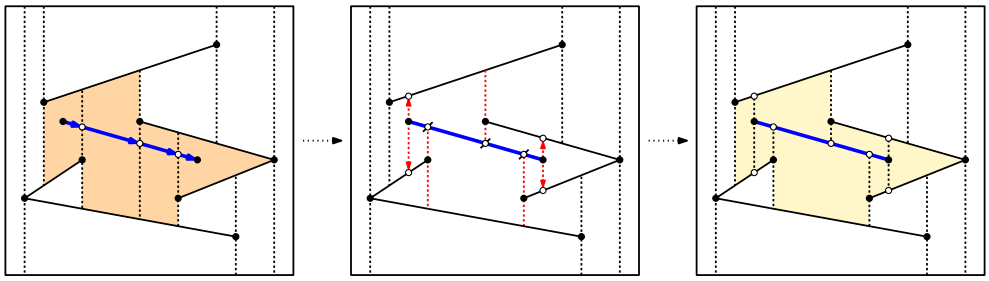


Fig. 54: Inserting a segment into the trapezoidal map: (a) Locate the left endpoint and trace the segment, (b) shoot bullet paths from endpoints and trim walls that have been crossed, (c) four original trapezoids (shaded red) have been replaced by seven new trapezoids (shaded yellow).

Observe that the structure of the trapezoidal decomposition does *not* depend on the order in which the segments are added. (This fact will be exploited later in the running time analysis,

[10]Historically, the randomized incremental algorithm that we will discuss arose as a method for solving a more general problem, namely 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. The original algorithm is due to Ketan Mulmuley.

and it is one of the reasons that trimming back the walls is so important.) Ignoring the time needed to determine the trapezoid that contains a segment's left endpoint (which will be discussed later), the time needed to insert a segment is proportional to the number of bullet paths it crosses. For the sake of our later analysis, it will be useful to express the insertion time in terms of the number of trapezoids created.

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

Proof: Consider the insertion of the ith segment, and let w_i 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 w_i walls, for a total of $w_i + 4$ 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 w_i walls we cross, we add one more to the number of newly created trapezoids, for a total of $w_i + 4$. Thus, letting $k_i = w_i + 4$ 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).

Analysis: We will analyze the expected time to build the trapezoidal map, assuming that segments are inserted in random order. (Note that we make no assumptions about the spatial distribution of the segments, other than the fact they do not intersect.) 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. (Too slow!)

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 is that the expected number of newly created trapezoids is a constant, no matter how many segments have already been inserted.

Lemma: Consider the randomized incremental construction of a trapezoidal map, and let k_i denote the number of new trapezoids created when the ith segment is added. Then for $1 \leq i \leq n$, $E[k_i] = O(1)$. (The expectation is taken over all $n!$ possible insertion orders of the segments.)

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 ith segment. Because we are averaging over all permutations, among the i segments that are present in $\mathcal{T}_i$, each one has an equal probability $1/i$ of being the last one to have been added.[11] 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 Δ of the existing map *depends* on an segment s, if s would have caused Δ to be created had s been the *last* segment to be inserted. (For example, in Fig. 55(a), the shaded trapezoids depend on s, and none of the others do.)

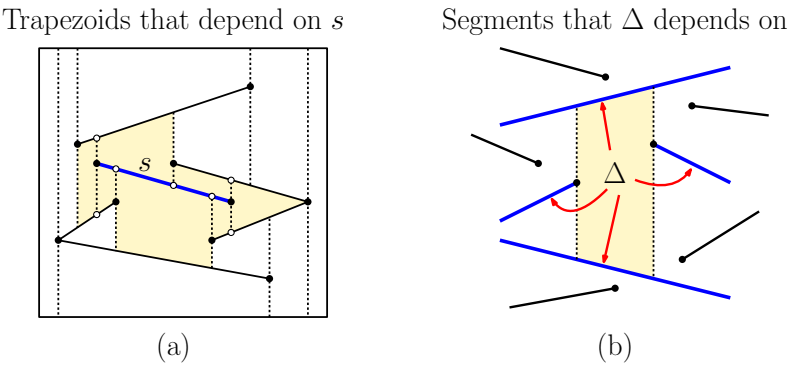


Fig. 55: Trapezoid-segment dependencies.

We want to count the number of trapezoids that depend on each segment, and then compute the average over all segments. Let $\delta(\Delta, s) = 1$ be an *indicator variable*, which is defined to be 1 if segment Δ depends on s, and 0 otherwise. The number of trapezoids that depend on segment s in $\mathcal{T}_i$ is $\sum_{\Delta \in \mathcal{T}_i} \delta(\Delta, s)$. Thus, the expected value is

$$E[k_i] = \sum_{s \in S_i} (\text{prob. that } s \text{ is last})(\text{num. of trapezoids that depend on } s)$$

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

Some segments might have resulted in the creation of lots of trapezoids and others would have resulted in very few. How can we analyze such an unruly quantity? The trick is, rather than counting the number of trapezoids that depend on each segment, we swap the order of the two summations and instead count the number segments upon which each trapezoid depends. 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

[11] An important feature of trapezoidal maps is that the structure of the map does not depend on the insertion order. So, any segment in the current diagram could have been the last.

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.[12]

In summary, each of the decomposition trapezoid is dependent on at most four segments, which implies that $\sum_{s \in S_i} \delta(\Delta, s) \leq 4$. Since $\mathcal{T}_i$ consists of at most $3i+1$ trapezoids we have

$$E[k_i] \;\leq\; \frac{1}{i} \sum_{\Delta \in \mathcal{T}_i} 4 \;=\; \frac{4}{i} |\mathcal{T}_i| \;\leq\; \frac{4}{i}(3i+1) \;=\; O(1).$$

We know that the total number of trapezoids in the end is at most $3n+1 = O(n)$. 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 (and perhaps destroyed) throughout the entire process is $O(n)$. This fact is important in bounding the total time needed for the randomized incremental algorithm.

Unfinished Business: 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 efficiently determine the trapezoid that contains a given point in our next lecture. There, we will see that this can be done in expected time $O(\log n)$ per segment, which leads to an overall running time of $O(n \log n)$.

Line-Segment Intersection Revisited: We have assumed that the line segments do not intersect, but you might wonder whether we can generalize the algorithm to handle the case where they do intersect. The answer is that we can readily adapt the algorithm to handle this.

The trapezoidal map definition is modified so that whenever two segments intersect, we shoot a bullet path upwards and downwards from this intersection point (see Fig. 56(a)). We can think of the original n segments as being split by the intersection points into a total of $O(n+m)$ nonintersecting subsegments. We can then think of the process as that of constructing a trapezoidal maps for these subsegments.

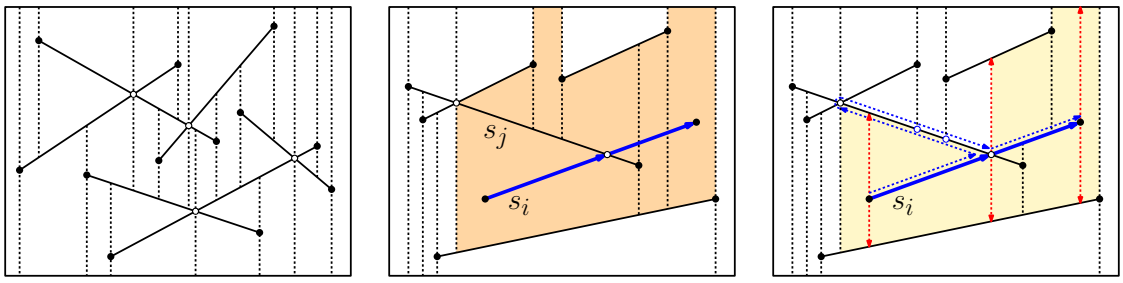


Fig. 56: Trapezoidal map of intersecting segments.

[12]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.

The principal new element of the algorithm involves the case when we are tracing a new segment s_i that intersects an existing subsegment s_j (see Fig. 56(b)). Observe that when this happens, the intersection point lies in two trapezoids, one above s_j and one below. We know the trapezoid that we arrived through, but we need to determine the trapezoid on the other side where the tracing continues. For concreteness, let's assume that the tracing of s_i hits the lower side of s_j. We walk along s_j, first tracing the bottom side and then wrapping around to the top side, until we reach the trapezoid containing the intersection point on the top side (see Fig. 56(c)). When we arrive at the intersection point on the top side of s_j, we shoot bullet paths up and down from the intersection point and continue with the tracing process.

How does the analysis change? The new element is the effort needed to process intersection events. The processing time is proportional to the number of new trapezoids created (same as the nonintersecting case) plus the number of bullet paths incident on the subsegments that are visited by the two-sided tracing process. While we will not show it here, it can be shown that in expectation, each subsegment contributes only a constant to this sum, which yields a total expected complexity of $O(n+m)$.[13] Thus, ignoring the time needed for computing the initial trapezoid containing the segment's left endpoints, the expected running time of the algorithm is $O(n+m)$.

Next time, we will show that this can be done in expected time $O(\log n)$ per segment endpoint, which leads to a total expected running time of $O(n \log n + (n+m)) = O((n \log n) + m)$. This beats the $O((n+m) \log n)$ running time of the plane sweep algorithm.

Lecture 9: Planar Point Location (using Trapezoidal Maps)

Point Location: In *planar point location* we are given a polygonal subdivision of the plane, and the objective is to preprocess this subdivision into a data structure so that given a query point q, it is possible to efficiently determine which face of the subdivision contains q (see Fig. 57(a)). For example, the subdivision might represent government subdivisions, such as countries, states, or counties, and we wish to identify the country, state, or county of a point given its GPS coordinates.

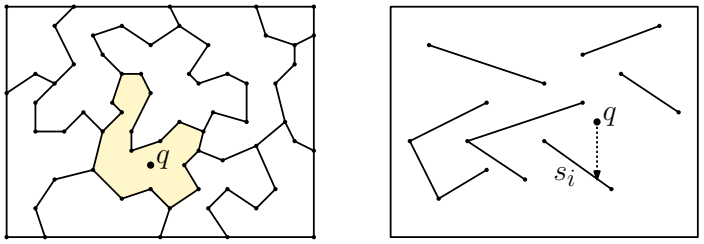


Fig. 57: (a) point location and (b) vertical ray-shooting queries.

It will be useful to generalize the above problem. Rather than assuming that the input is a subdivision of space into cells (what is commonly referred to as a *cell complex*), we will assume that the input is merely a set of n line segments $S = \{s_1, \ldots, s_n\}$. The objective is to

[13] The original analysis was done by Ketan Mulmuley, who observed that the trapezoidal map can be viewed as a planar graph with $O(n+m)$ vertices, and then exploiting the fact that the average degree of a face in any planar graph is $O(1)$.

answer *vertical ray-shooting queries*, which means, given a query point q, what line segment s_i (if any) lies immediately below the query point (see Fig. 57(b)). Observe that the ability to answer vertical ray-shooting queries implies that point-location queries can be answered. We simply label each segment with the identity of the subdivision cell that lies immediately above it.

We will make the usual general-position assumptions. In particular, except when two segments share a common endpoint, no two segment endpoints share the same x-coordinate. Also, we will assume no vertical segments. Let us also assume that the query point does not have the same x-coordinate of any segment endpoint.

Our objective is to obtain a data structure with $O(n)$ space and $O(\log n)$ query time, which we can build in $O(n \log n)$ time. This is asymptotically optimal (assuming a data structure based on comparisons).

History: When the problem was first considered, a number of solutions had been proposed, but they were all suboptimal by a log factor either in the query time or the space. The first optimal solution was proposed by David Kirkpatrick in the early 1980's. The algorithm involved a very clever incremental approach based on iteratively decimating a triangulation. The approach described here is both simpler and more efficient, but many of the ideas were inspired by Kirkpatrick's data structure. The method we will described is based on the randomized construction of trapezoidal maps, and so the construction time holds in the expected case. (In contrast, Kirkpatrick's solution was deterministic.)

Recap of Trapezoidal Maps: Our point-location data structure will be based on the randomized trapezoidal map construction from the previous lecture. In that lecture we showed that a trapezoidal map of $O(n)$ space could be constructed in (randomized) $O(n \log n)$ expected time. In this lecture we show how to modify the construction so that, as a by product, we obtain a data structure for answering vertical ray-shooting queries. The preprocessing time for the data structure will also be $O(n \log n)$ in the expected case, the space required for the data structure will be $O(n)$, and the query time will be $O(\log n)$. The latter two bounds will hold unconditionally.

Let us recap some of the concepts from the previous lecture. Recall that the input as a set of segments in the plane $S = \{s_1, \ldots, s_n\}$, which are assumed to have been randomly permuted, and which are enclosed in an axis-aligned bounding rectangle. Recall that the trapezoidal map, denoted $\mathcal{T}(S)$, is a subdivision generated by shooting bullet paths both upwards and downwards from each segment endpoint until first striking another segment (or hitting the bounding box of the input). Let $S_i = \{s_1, \ldots, s_i\}$, and let $\mathcal{T}_i = \mathcal{T}(S_i)$ denote the trapezoidal map of S_i, and $\mathcal{T} = \mathcal{T}_n$.

Recall from the previous lecture that each time we add a new line segment, it may result in the creation of the collection of new trapezoids, which are said to *depend* on this line segment. We showed that (under the assumption of the random insertion order) the expected number of new trapezoids that are created with each stage is $O(1)$. This fact will be used later in this lecture.

Point Location Data Structure: Typical search structures (red-black trees, AVL trees, etc.) are rooted binary trees. Our point location data structure will be a rooted binary tree with *shared* subtrees, or equivalently, 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. (Note that subtree sharing is important for our space efficiency.)

There are two types of internal nodes, *x-nodes* and *y-nodes*:

x-Nodes: Each x-node contains a reference to an endpoint of some segment. Its two children correspond to the points lying to the left and to the right of the vertical line passing through p (see Fig. 58(a)).

y-Nodes: Each y-node contains a reference 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. 58(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.

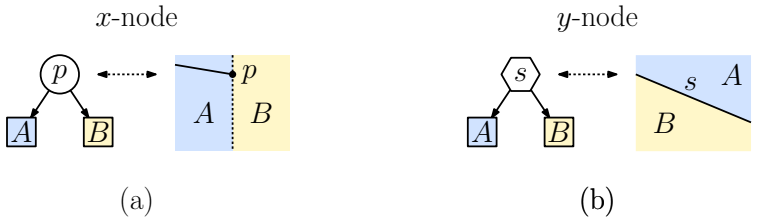


Fig. 58: (a) x-node and (b) y-node.

Our construction of the point location data structure mirrors the incremental construction of the trapezoidal map, as given in the previous lecture. 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. 59 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 endpoint p_1 (right child), then left of t_1 (left child), then below s_1 (right child), then right of p_2 (right child), then above s_2 (left child).

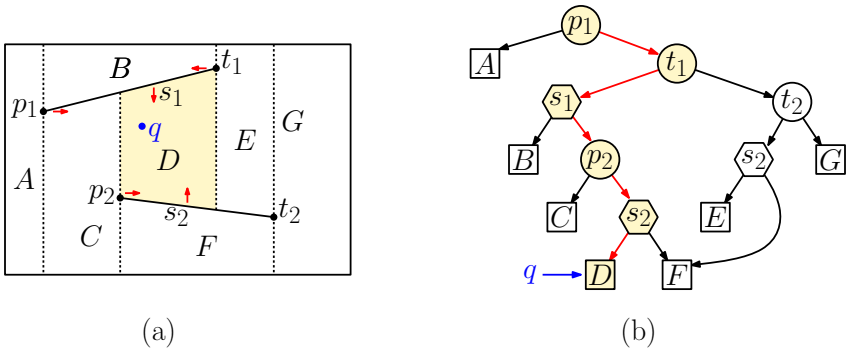


Fig. 59: Trapezoidal map point location data structure.

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 B, C, and D. Letting p denote the endpoint, we create an x-node for p, and one child is a leaf node for the trapezoid B that lies outside vertical projection of the segment. For the other child, we create a y-node whose children are the trapezoids C and D lying above and below the segment, respectively (see Fig. 60(a)).

Two segment endpoints: This happens when the segment lies entirely inside an existing trapezoid A, which is replaced by four trapezoids, E, B, C, and D. Letting p and t denote the left and right endpoints of the segment, we create two x-nodes, one for p and the other for t. We create a y-node for the line segment, and join everything together (see Fig. 60(b)).

No segment endpoints: This happens when the segment cuts completely through a trapezoid A. This trapezoid is replaced by two trapezoids, one above and one below the segment, denoted C and D. We replace the leaf node for the original trapezoid with a y-node whose children are leaf nodes associated with C and D (see Fig. 60(c)).

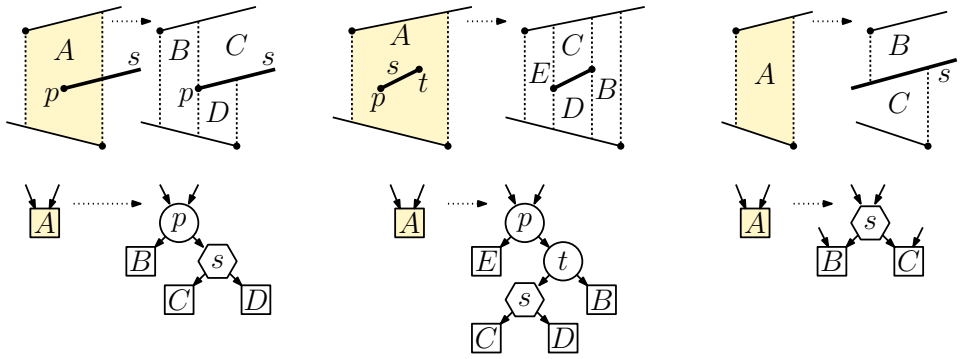


Fig. 60: Line segment insertion and updates to the point location structure. The single-endpoint case (left) and the two-endpoint case (right). The no-endpoint case is not shown.

It is important to notice that (through sharing) each trapezoid appears exactly once as a leaf in the resulting structure. How does this sharing occur? Whenever we add a segment, the wall trimming that results can result in two distinct trapezoids being merged into one (see trapezoid C in Fig. 61(a) and B and C in Fig. 61(b)). When this happens, the various paths leading into merged trapezoid are joined to a common node. An example showing the complete transformation to the data structure after adding a single segment is shown in Fig. 61 below.

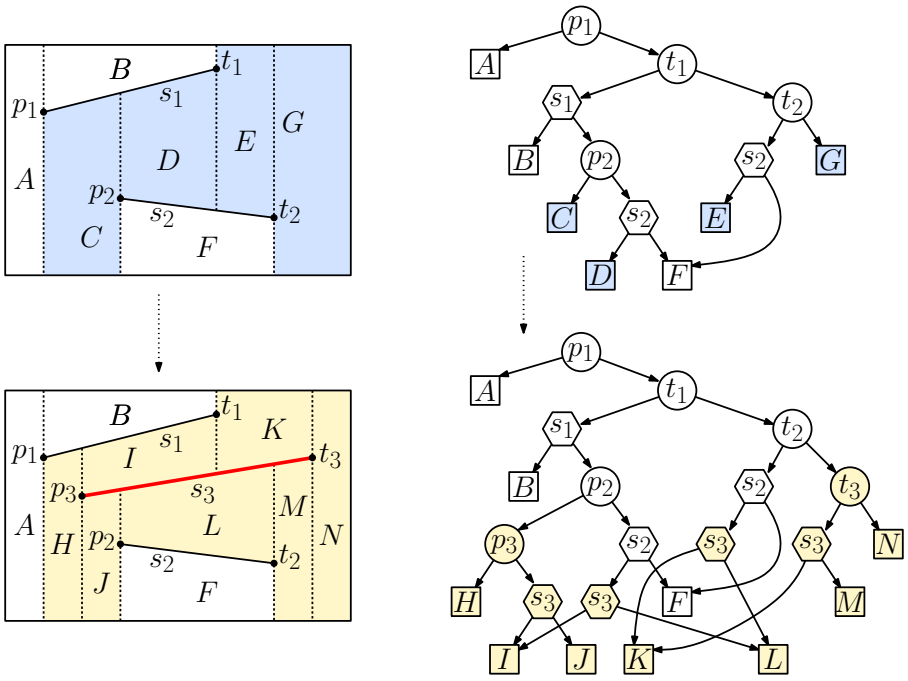


Fig. 61: Line segment insertion.

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

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

Analyzing the query time is a little subtler. In a normal probabilistic analysis of data structures we think of the data structure as being fixed, and then compute expectations over random queries. Here the approach will be to imagine that we have exactly one query point to handle. The query point can be chosen arbitrarily (imagine an adversary that tries to select the worst-possible query point) but this choice is made without knowledge of the random choices the algorithm makes. We will show that, 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 Δ_i denote the trapezoid of the map that q lies in after the insertion of the first i segments. Observe that if $\Delta_{i-1} = \Delta_i$, then insertion of the ith segment did not affect the trapezoid that q was in, and therefore q will stay where it is relative to the current search structure. (For example, if q was in trapezoid B prior to adding s_3 in Fig. 61 above, then the addition of s_3 does not incur any additional cost to locating q.)

However, if $\Delta_{i-1} \neq \Delta_i$, then the insertion of the ith segment caused q's trapezoid to be

replaced by a different one. As a result, q must now perform some additional comparisons to locate itself with respect to the newly created trapezoids that overlap Δ_{i-1}. Since there are a constant number of such trapezoids (at most four), there will be $O(1)$ work needed to locate q with respect to these. In particular, q may 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. 60(b)).

Since a point can descend at most three levels with each change of its containing trapezoid, the expected length of the search path to q is at most three times the number of times that q changes its trapezoid as a result of each insertion. For $1 \leq i \leq n$, let $X_i(q) = 1$ if q changes its trapezoid after the ith insertion 0 otherwise. Let $E(X_i(q))$ denote its expected value, which is equivalent to $\text{Prob}(\Delta_i(q) \neq \Delta_{i-1}(q))$. Letting $D(q)$ denote the average depth of q in the final search tree, we have

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

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 $\text{Prob}(X_i(q)) \leq 4/i$. We do this through a backwards analysis. Consider the trapezoid Δ_i that contained q *after* the ith 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, Δ_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 Δ_i's dependent segments was the last to be inserted is at most $4/i$. Therefore, $\text{Prob}(X_i(q)) \leq 4/i$.

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

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

Guarantees on Search Time: (Optional) 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.

It is far from trivial, but it can be shown that by repeated application of the randomized incremental construction, it is possible to achieve worst-case search time of $O(\log n)$, worst-case size of $O(n)$, and expected-case construction time is $O(n \log n)$.[14] The idea is to engineer the constants so that the probability of failure along any search path is extremely small (say $1/n^c$, for some constant $c \geq 1$). It follows that all the possible search paths will have the

[14] M. Hemmer, M. Kleinbort, and D. Halperin. Optimal randomized incremental construction for guaranteed logarithmic planar point location. *Comput. Geom.*, 58:110–123, 2016.

desired $O(\log n)$ depth with at least a constant probability. While we might be unlucky on any given execution of the algorithm, after a constant number of attempts, we expect one of them to succeed.

Summary: In the previous lecture, we showed that a randomized incremental algorithm allows to construct the trapezoidal map of any set of nonintersecting lines segments in expected time $O(n)$. Further, if the line segments have m intersections, this can be generalized to $O(n+m)$. However, we ignored the question of how to locate the trapezoid containing the left endpoint of each segment. In this lecture, we have shown that this can be done in $O(\log n)$ time per endpoint, for a total of $O(n \log n)$ time. (Note that when the segments intersect, the time is actually $O(\log(n+m))$, but since $m = O(n^2)$, this is the same as $O(\log n)$.)

This data structure applies more generally to vertical ray-shooting queries for any set of line segments. As observed earlier, we can solve the point-location problem in *any* planar subdivision of line segments with $O(n)$ space, $O(\log n)$ query times, with expected-case preprocessing of $O(n \log n)$.

Note that the randomization is not a necessity. It can be eliminated, but at the expense of higher constant factors and a more complicated algorithm. Also note that we can generalize this to curved objects, as long as they are x-monotone, or can be subdivided into a relatively small number of x-monotone pieces.

Finally, note that this data structure does not generalize to higher dimensions. Indeed, there is no known data structure for point location in subdivisions in dimensions 3 and higher. There are data structures that achieve $O(\log n)$ worst-case query time, but the space generally grows as $O(n^{O(d)})$. Conversely, if the space is constrained to $O(n)$, then the worst-case query time grows to a polynomial function of n. There are good heuristic data structure (such as quadtrees) that may achieve good performance in "typical" instances, but they cannot guarantee good performance for all inputs.

Lecture 10: Voronoi Diagrams and Fortune's Algorithm

Metric Spaces: In this lecture, we will make a significant transition in the subject of computational geometry by introducing the notion of distances. Distances in geometry are modeled formally by through the concept of a *metric*. This is a function f on pairs of points that satisfies the following properties for any points p, q, and r:

Symmetry: $f(p,q) = f(q,p)$

Positivity: $f(p,q) \geq 0$, and further $f(p,q) = 0$ if and only if $p = q$

Triangle inequality: $f(p,q) \leq f(p,r) + f(r,q)$.

There are many natural metrics over $\mathbb{R}^d$, but the most common by far is *Euclidean distance*, which for any $p, q \in \mathbb{R}^d$ is defined as

$$\|p - q\| \;=\; \left(\sum_{i=1}^{d} (p_i - q_i)^2 \right)^{1/2}.$$

Given any set of points P in $\mathbb{R}^d$, there are a number of natural structures whose definition involves the distances between points. Today, we will study one of the most important of these, called the Voronoi diagram.

Voronoi Diagrams: Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$, which we call *sites*. We want to subdivide $\mathbb{R}^d$ into regions according to the which site is most "influential". To make this formal, for any site p_i, define its *Voronoi cell*, denoted $\mathcal{V}_P(p_i)$, to be the set of points $q \in \mathbb{R}^d$ that are closer to p_i than to any other site, that is,

$$\mathcal{V}_P(p_i) \;=\; \{q \in \mathbb{R}^d : \|p_i - q\| < \|p_j - q\|, \forall j \neq i\},$$

When P is clear from context, we will omit it and refer to this simply as $\mathcal{V}(p_i)$. Clearly, the Voronoi cells of two distinct sites of P are disjoint. The union of the closure of the Voronoi cells defines a cell complex, which is called the *Voronoi diagram* of P, and is denoted $\mathrm{Vor}(P)$ (see Fig. 62(a)).

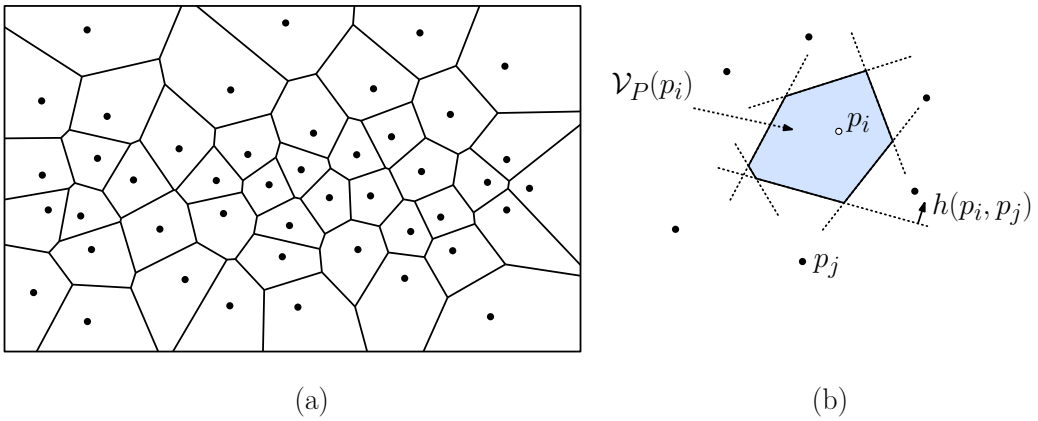


Fig. 62: Voronoi diagram $\mathrm{Vor}(P)$ of a set of sites.

The cells of the Voronoi diagram are (possibly unbounded) convex polyhedra. To see this, observe that the set of points that are strictly closer to one site p_i than to another site p_j is equal to the *open halfspace* whose bounding hyperplane is the perpendicular bisector between p_i and p_j. Denote this halfspace $h(p_i, p_j)$. 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,

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

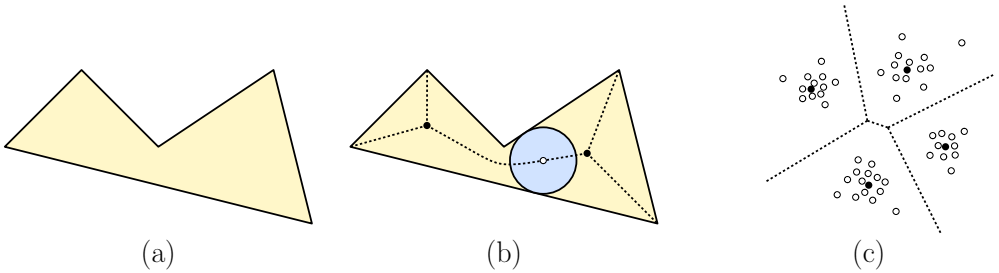
(see Fig. 62(b)). Since the intersection of convex objects is convex, $\mathcal{V}(p_i)$ is a (possibly unbounded) convex polyhedron.

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.

Nearest neighbor queries: Given a set P of point sites, we wish to preprocess P so that, given a query point q, it is possible to quickly determine the closest site of P to q. This can be answered by first computing a Voronoi diagram and then locating the cell of the diagram that contains q. (In the plane, this can be done by building the trapezoidal map of the edges of the Voronoi diagram. Each trapezoid lies within a single Voronoi cell, and can be labeled with the generating site.)

Computational morphology and shape analysis: A useful structure in shape analysis is called the medial axis. The *medial axis* of a shape (e.g., a simple polygon) is defined to

be the union of the center points of all locally maximal disks that are contained within the shape (see Fig. 63). If we generalize the notion of Voronoi diagram to allow sites that are both points and line segments, then the medial axis of a simple polygon can be extracted easily from the Voronoi diagram of these generalized sites.



(a) (b) (c)

Fig. 63: (a) A simple polygon, (b) its medial axis and a sample maximal disk, and (c) center-based clustering (with cluster centers shown as black points).

Center-based Clustering: Given a set P of points, it is often desirable to represent the union of a significantly smaller set of clusters. In center-based clustering, the clusters are defined by a set C of *cluster centers* (which may or may not be required to be chosen from P). The *cluster* associated with a given center point $q \in C$ is just the subset of points of P that are closer to q than any other center, that is, the subset of P that lies within q's Voronoi cell (see Fig. 63(c)). (How the center points are selected is another question.)

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

Properties of the Voronoi diagram: Here are some properties of the Voronoi diagrams in the plane. These all have natural generalizations to higher dimensions.

Empty circle properties: 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 any such point where p_i and p_j lie on this circle, and no other site is interior to the circle (see Fig. 64(a)).

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. 64(b)). Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior. (In $\mathbb{R}^d$, the vertex is defined by $d+1$ sites and the hypersphere centered at the vertex passing through these sites is empty.)

Degree: Generally three points in the plane define a unique circle (generally, $d+1$ points in $\mathbb{R}^d$). If we make the general position assumption that no four sites are cocircular, then each vertex of the Voronoi diagram is incident to three edges (generally, $d+1$ facets).

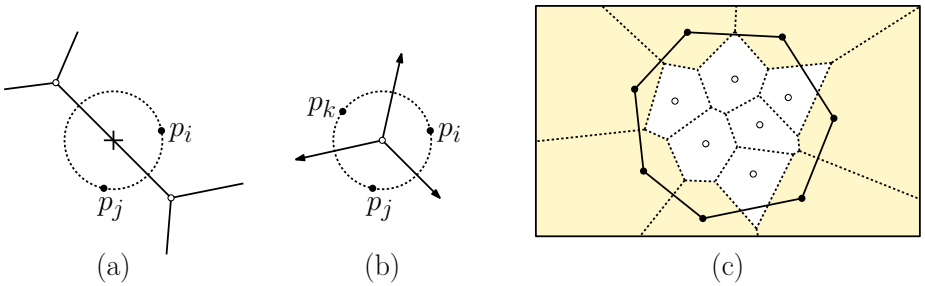


Fig. 64: Properties of the Voronoi diagram.

Convex hull: A cell of the Voronoi diagram is unbounded if and only if the corresponding site lies on the convex hull (see Fig. 64(c)). (Observe that a site is on the convex hull if and only if it is the closest site from some point at infinity, namely the point infinitely far along a vector orthogonal to the supporting line through this vertex.) Thus, given a Voronoi diagram, it is easy to extract the vertices of the convex hull in linear time.

Size: Letting n denote the number of sites, the Voronoi diagram with exactly n faces. It follows from *Euler's formula*[15] 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.

Sweep-Line Approach: 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.)

[15]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 (including a vertex at infinity that joins all the unbounded edge) is of degree at least three, we have $3v \leq 2e$, from which we infer that $v - (3/2)v + n \geq 2$, implying that $v \leq 2n - 4$. A similar argument can be used to bound the number of edges.

The challenge posed by Voronoi diagrams is that of predicting when and where the upcoming events will occur.

To better understand the difficulty, suppose that you are designing a plane sweep algorithm. Behind the sweep line you have constructed the Voronoi diagram based on the sites that have been encountered so far in the sweep. Now a site that lies *beyond* the sweep line (that is, in the "future") may generate a Voronoi vertex that lies *behind* the sweep line (in the past). How can any reasonable sweep-line algorithm know of the existence events generated by sites it has yet to encounter? It is these *unanticipated events* that make the design of a plane sweep algorithm challenging (see Fig. 65).

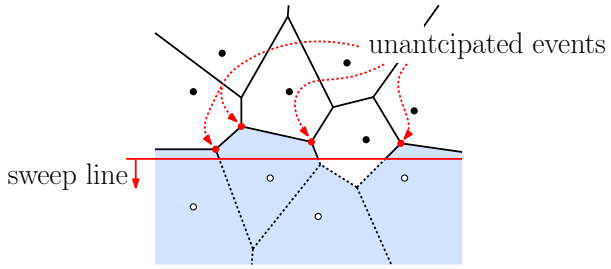


Fig. 65: Unanticipated events generated by sites that lie beyond the current sweep line.

The Beach Line: To deal with the difficulties of unanticipated events, our sweeping process will involve maintaining 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 lie beyond the current sweep line. Thus, there can be no unanticipated events. 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 this intuition more concrete. Given a point $q \in \mathbb{R}^2$ and a nonempty set R of points (which may be finite or infinite) define $\text{dist}(q, R)$ to be the closest Euclidean distance from q to any point in R, that is, $\text{dist}(q, R) = \min_{p \in R} \|q - p\|$. Given the current location of the sweep ℓ (which we assume to be horizontal and sweeping downwards) let $P^+(\ell) \subseteq P$ denote the subset of sites of P that lie strictly above ℓ. Let us assume that $P^+(\ell)$ contains at least one site. Define the *beach line* (for ℓ) to be the set of points $q \in \mathbb{R}^2$ (all above the sweep line) whose distance to its nearest site in P' is equal to its distance from the sweep line (see the blue curve in Fig. 66(a)). That is

$$\text{beach}(\ell) \;=\; \{q \in \mathbb{R}^2 \mid \text{dist}(q, P^+(\ell)) = \text{dist}(q, \ell)\}.$$

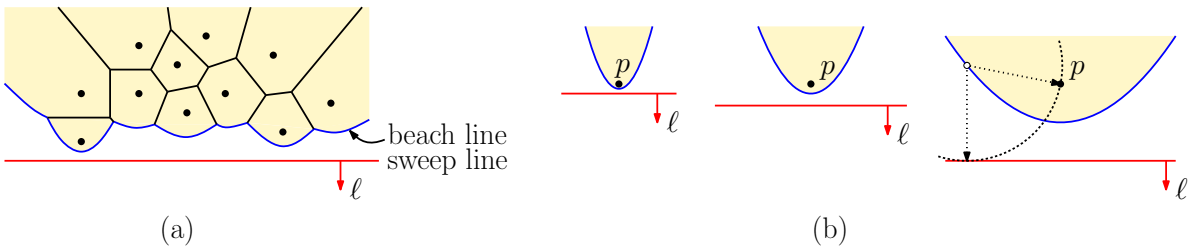


Fig. 66: (a) The beach line and (b) the evolution of a parabolic arc as the sweep line moves.

It is easy to see that for all points lying on or above the beach line, the closest site in P must lie above the sweep line, that is, it must lie within $P^+(\ell)$. This means that it cannot be affected by any site below the sweep line. This further implies that no part of any Voronoi cell of a point lying below the sweep line can affect a point on or above the beach line. Thus, the region above the beach line is "safe" in the sense that there can be no unanticipated events.

What does the beach line look like? Recall from your high-school geometry that the set of points that are equidistant from a point (in this case a site p) and a line (in this case the sweep line ℓ) is a *parabola* separating p from ℓ. The parabola's shape depends on the relative distance between p and ℓ, and in particular, ℓ moves further away from p, the parabola's shape becomes progressively "flatter" (see Fig. 66(b)). Note that in the special case when p lies on ℓ, the parabola degenerates into a vertical ray shooting up from p.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site (see Fig. 66(c)). Note that the parabola associated with some sites may be *redundant* in the sense that they will not contribute to the beach line. Also, some sites can contribute multiple arcs, but the total complexity of the beach line cannot exceed $O(|P^+(\ell)|) = O(n)$. Because the parabolas are x-monotone, so is the beach line. The beach-line point q where two beach-line arcs intersect is called a *breakpoint*. If these arcs are associated with sites p_i and p_j, then q is equidistant to these sites and they are the closest sites to q. Therefore, q lies on an edge of the final Voronoi diagram.

Fortune's Algorithm: Fortune's algorithm (at least, the variant described in our textbook) 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. (For help visualizing this, run the applet described at the top of the lecture.) 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. 67).

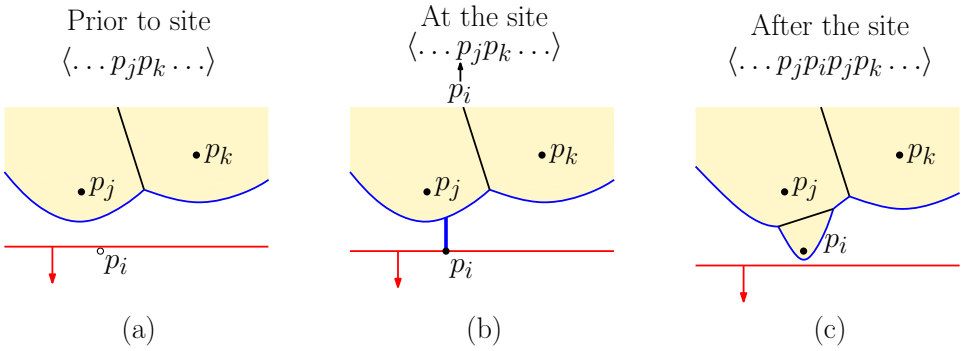


Fig. 67: Site event.

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 site can result in creating one new arc, and splitting an existing arc, for a net increase of two arcs per site (except the first). Note that a site 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. 68(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 sites lying below the sweep line (meaning that no future site 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. 68).

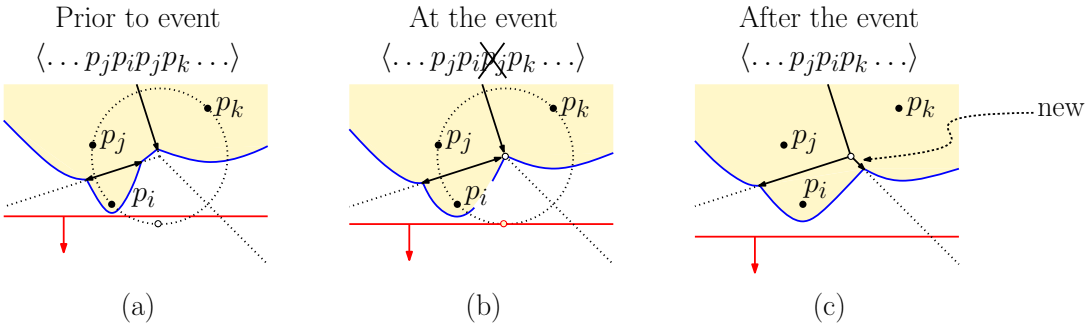


Fig. 68: Voronoi vertex event.

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 sites. (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. 67 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. 68 above).

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

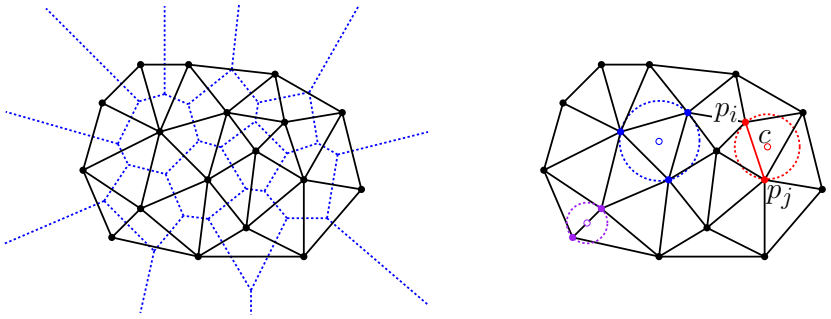
(2) Create a new vertex in the Voronoi diagram (at the circumcenter of $\{p_i, p_j, p_k\}$) and join the two Voronoi edges for the bisectors (p_i, p_j), (p_j, p_k) to this vertex.
(3) Create a new (dangling) edge for the bisector between p_i and p_k.
(4) Delete any events that arose from triples involving 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 11: Delaunay Triangulations: General Properties

Delaunay Triangulations: We have discussed the topic of Voronoi diagrams. In this lecture, we consider a closely related structure, called the *Delaunay triangulation* (DT). The Voronoi diagram of a set of sites in the plane is a planar subdivision, in fact, a cell complex. The *dual* of such subdivision is a cell complex 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. Each vertex of the Voronoi diagram corresponds to a face of the dual complex.

Recall that, under the assumption of general position (no four sites are collinear), the vertices of the Voronoi diagram all have degree three. It follows that the faces of the resulting dual complex (excluding the exterior face) are triangles. Thus, the resulting dual graph is a *triangulation* of the sites. This is called the *Delaunay triangulation* (see Fig. 69(a)).



(a) (b)

Fig. 69: (a) The Voronoi diagram of a set of sits (broken lines) and the corresponding Delaunay triangulation (solid lines) and (b) circle-related properties.

Delaunay triangulations have a number of interesting properties, that are immediate 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," that is, the interior of the associated circular disk contains no sites of P (see the blue circle in Fig. 69(b)).

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 (see the red circle in Fig. 69(b)).

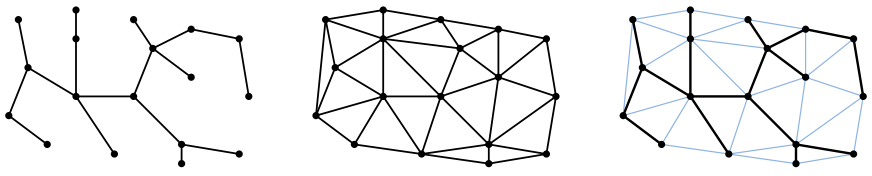
Proof: If two sites p_i and p_j are neighbors in the Delaunay triangulation, then their cells are neighbors in the Voronoi diagram, and so for any point on the Voronoi edge between these sites, a circle centered at this point passing through p_i and p_j cannot contain any other point (since they must 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 (see Fig. 69(b)). Thus the Voronoi cells of two sites are adjacent in the Voronoi diagram, implying that this edge is in the Delaunay triangulation.

Closest pair property: The closest pair of sites in P are neighbors in the Delaunay triangulation (see the green circle in Fig. 69(b)).

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.

Combinatorial Complexity: 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. Generally, in $\mathbb{R}^d$, the number of simplices (the d-dimensional generalization of a triangle) can range from $O(n)$ up to $O(n^{\lceil d/2 \rceil})$. For example, in $\mathbb{R}^3$ the Delaunay triangulation of n sites may have as many as $O(n^2)$ tetrahedra. (If you want a challenging exercise, try to create such a point set.)

Euclidean Minimum Spanning Tree: The Delaunay triangulation possesses a number of interesting properties that are not obviously 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. Given a graph, the minimum spanning tree (MST) is a set of $n - 1$ edges that connect the points (into a free tree) such that the total weight of edges is minimized. The MST of the Euclidean graph is called the *Euclidean minimum spanning tree* (EMST), see Fig. 70(c).



(a) (b) (c)

Fig. 70: (a) A point set and its EMST, (b) the Delaunay triangulation, and (c) the overlay of the two.

We could compute the EMST by brute force by constructing the Euclidean graph and then invoking Kruskal's algorithm to compute its MST. This would lead to a total running time of $O(n^2 \log n)$. However there is a much faster method for planar point sets 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, say, 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 P of point sites (in any dimension) is a subgraph of the Delaunay triangulation (see Fig. 70(c)).

Proof: Let T be the EMST 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 another site, call it c (see Fig. 71.)

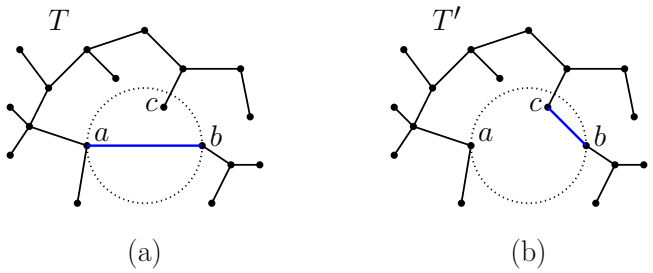


Fig. 71: The Delaunay triangulation and EMST.

The removal of $\overline{ab}$ from the EMST 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 EMST and add the edge $\overline{bc}$ in its place. The result will be a spanning tree T' whose weight is

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

Since ab is the diameter of the circle, any other segment lying within the circle is shorter. Thus, $\|bc\| < \|ab\|$. Therefore, we have $w(T') < w(T)$, and this contradicts the hypothesis that T is the EMST, completing the proof.

Minimum Weight Triangulation (Not!): Given the above result about EMSTs, it is tempting to conjecture that among all triangulations, the Delaunay triangulation minimizes the total (Euclidean) edge length. Unfortunately, this is not true and, indeed, there is a simple four-point counterexample. (This assertion was erroneously claimed in a highly cited paper on Delaunay triangulations, and you may still hear it quoted from time to time.)

The triangulation that minimizes total Euclidean edge weight is called the *minimum weight triangulation* (MWT). The computational complexity of MWT was open for many years, and many special cases were shown to be solvable in polynomial time. Finally, in 2008 it was proved by Mulzer and Rote[16] that this problem is NP-hard. The hardness proof is quite complex, and computer assistance was needed to verify the correctness of some of the constructions used in the proof.

[16] W. Mulzer and G. Rote, "Minimum-weight triangulation is NP-hard", *J. ACM*, 55 (2), 2008.

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 significantly longer than the straight-line distance. 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 $O(1)$-spanners having $O(n)$ edges.

It can be proved that the edges of the Delaunay triangulation form a spanner (see Fig. 72). We will not prove the following result, which is due to Keil and Gutwin.[17]

Theorem: Given a set of points P in the plane, the Delaunay triangulation of P is a t-spanner for $t = 4\pi\sqrt{3}/9 \approx 2.418$.

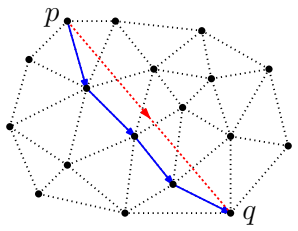


Fig. 72: Spanner property of the Delaunay Triangulation.

It had been conjectured for many years that the Delaunay triangulation is a $(\pi/2)$-spanner ($\pi/2 \approx 1.5708$). This was disproved in 2009, and the lower bound now stands at roughly 1.5846. Closing the gap between the upper and lower bound is an important open 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 stronger statement holds. Among all triangulations that maximize the smallest angle, the Delaunay triangulation maximizes the second smallest angle. Among all triangulations that maximize both the two smallest angles, the Delaunay triangulation maximizes the third smallest angle. That is, the Delaunay triangulation maximizes the sequence of angles lexicographically.

[17] J. M. Keil and C. A. Gutwin, "Classes of graphs which approximate the complete Euclidean graph", *Discrete & Computational Geometry*, 7, 1992.

To make this more formal, consider any triangulation of a given set P of n sites. Let t denote the number of triangles, so that the number of angles is $m = 3t$. We can associate this triangulation with a sorted *angle sequence*, by which we mean a non-decreasing sequence of angles $(\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_m)$ appearing in all 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 of triangles depends only on the number of sites n and the number of points on the convex hull h.)

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

Proof: (Optional) 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. 73(a) below, we have $\theta_1 > \theta_2 > \theta_3$.

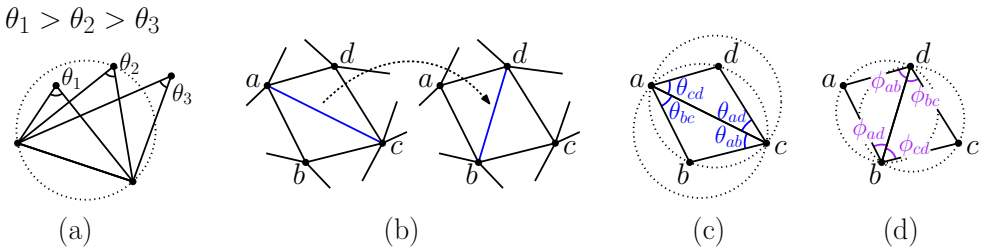


Fig. 73: Angles and edge flips.

We will not give a formal proof of the theorem. (A full proof 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 (see Fig. 73(b)). 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. 73(c) and (d), we have

$$\phi_{ab} > \theta_{ab}, \quad \phi_{bc} > \theta_{bc}, \quad \phi_{cd} > \theta_{cd}, \quad \text{and} \quad \phi_{da} > \theta_{da}.$$

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

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

Polytopes and Spatial Subdivisions: 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). On the other hand, if you look at the surface of the convex hull of a set of points in 3-dimensional space, the boundary structure looks much like a triangulation. (If the points are in general position, no four points are coplanar, so each face of the convex hull will be bounded by three vertices.)

Similarly, consider a boundary structure of a polytope defined by the intersection of a collection of halfplanes in 3-dimensional space. Assuming general position (no four planes intersecting at a common point), each vertex will be incident to exactly three faces, and hence to exactly three edges. Therefore, the boundary structure of this polytope will look very much like a Voronoi diagram.

We will show that there is a remarkably close relationship between these structures. In particular, we will show that:

- The Delaunay triangulation of a set of points in the plane is topologically equivalent to the boundary complex of the (lower) convex hull of an appropriate set of points in 3-space. It follows that it is possible to reduce the problem of computing Delaunay triangulations in dimension d to that of computing convex hulls in dimension $d+1$.
- The Voronoi diagram of a set of points in the plane is topologically equivalent to the boundary complex of the intersect of a set of halfspaces in 3-space. In general, it is possible to reduce the problem of computing Voronoi diagrams in dimension d to computing the upper envelope of a set of hyperplanes in dimension $d+1$.

We will demonstrate these results in 2-dimensional space, but the generalizations to higher dimensions are straightforward.

Delaunay Triangulations and Convex Hulls: Let us begin by considering the *paraboloid* Ψ defined by the equation $z = x^2 + y^2$. Observe that the vertical cross sections (constant x or constant y) are parabolas, and whose horizontal cross sections (constant z) are circles. For each point in $\mathbb{R}^2$, $p = (p_x, p_y)$, the *vertical projection* (also called the *lifted image*) of this point onto this Ψ is $p^\uparrow = (p_x, p_y, p_x^2 + p_y^2)$ in $\mathbb{R}^3$.

Given a set of points P in the plane, let $P^\uparrow$ denote the projection of every point in P onto Ψ. Consider the *lower convex hull* of $P^\uparrow$. This is the portion of the convex hull of $P^\uparrow$ which is visible to a viewer standing at $z = -\infty$. We claim that if we take the lower convex hull of $P^\uparrow$, and project it back onto the plane, then we get the Delaunay triangulation of P (see Fig. 74). In particular, let $p, q, r \in P$, and let $p^\uparrow$, $q^\uparrow$, $r^\uparrow$ denote the projections of these points onto Ψ. Then $\triangle p^\uparrow q^\uparrow r^\uparrow$ defines a *face* of the lower convex hull of $P^\uparrow$ if and only if $\triangle pqr$ is a triangle of the Delaunay triangulation of P.

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 $p, q, r \in P$ form a Delaunay triangle if and only no other point of P lies *within the circumcircle* of the triangle defined by these points.

Convex hull condition: Three points $p^\uparrow, q^\uparrow, r^\uparrow \in P^\uparrow$ form a face of the convex hull of $P^\uparrow$ if and only if no other point of P lies *below the plane* passing through $p^\uparrow$, $q^\uparrow$, and $r^\uparrow$.

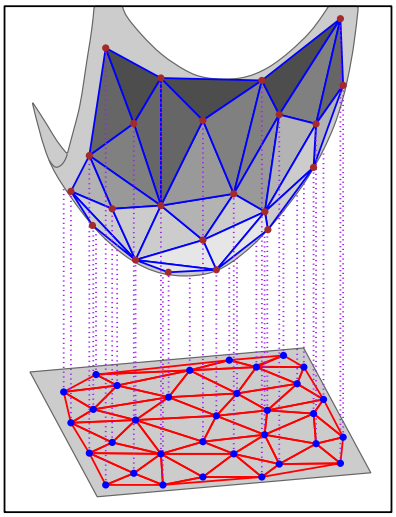


Fig. 74: The Delaunay triangulation and convex hull.

Clearly, the connection we need to establish is between the emptiness of circumcircles in the plane and the emptiness of lower halfspaces in 3-space. To do this, we will prove the following.

Lemma: Consider four distinct points p, q, r, and s in the plane, and let $p^\uparrow$, $q^\uparrow$, $r^\uparrow$, and $s^\uparrow$ denote their respective vertical projections onto Ψ, $z = x^2 + y^2$. The point s lies within the circumcircle of $\triangle pqr$ if and only if $s^\uparrow$ lies beneath the plane passing through $p^\uparrow$, $q^\uparrow$, and $r^\uparrow$.

To prove the lemma, first consider an arbitrary (nonvertical) plane in 3-space, which we assume is tangent to Ψ 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

$$z = 2ax + 2by + \gamma, \quad \text{for some real } \gamma.$$

To solve for γ we know that the plane passes through $(a, b, a^2 + b^2)$ so we solve giving

$$a^2 + b^2 = 2a \cdot a + 2b \cdot b + \gamma,$$

Implying that $\gamma = -(a^2 + b^2)$. Thus the plane equation is

$$z = 2ax + 2by - (a^2 + b^2). \tag{2}$$

If we shift the plane upwards by some positive amount r^2 we obtain the plane

$$z = 2ax + 2by - (a^2 + b^2) + r^2.$$

How does this plane intersect Ψ? Since Ψ is defined by $z = x^2 + y^2$ we can eliminate z, yielding

$$x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + r^2,$$

which after some simple rearrangements is equal to

$$(x-a)^2 + (y-b)^2 = r^2.$$

Hey! This is just a circle centered at the point (a, b). Thus, we have shown that the intersection of a plane with Ψ 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) whose radius equals the square root of the vertical distance by which the plane has been translated.

Thus, we conclude that the intersection of an arbitrary lower halfspace with Ψ, 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 Ψ, the projected points $p^\uparrow$, $q^\uparrow$ and $r^\uparrow$ define a plane. Since $p^\uparrow$, $q^\uparrow$, and $r^\uparrow$, lie at the intersection of the plane and Ψ, 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 $s^\uparrow$ onto Ψ lies within the lower halfspace of the plane passing through p, q, r (see Fig. 75).

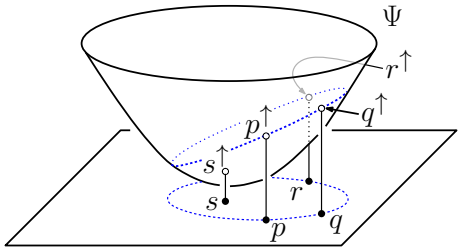


Fig. 75: Planes and circles.

Now we can prove the main result.

Theorem: Given a set of points P in the plane (assuming no four are cocircular), and given three points $p, q, r \in P$, the triangle $\triangle pqr$ is a triangle of the Delaunay triangulation of P if and only if triangle $\triangle p^\uparrow q^\uparrow r^\uparrow$ is a face of the lower convex hull of the lifted set $P^\uparrow$.

From the definition of Delaunay triangulations we know that $\triangle pqr$ is in the Delaunay triangulation if and only if there is no point $s \in P$ that lies within the circumcircle of pqr. From the previous lemma this is equivalent to saying that there is no point $s^\uparrow$ that lies in the lower convex hull of $P^\uparrow$, which is equivalent to saying that $\triangle p^\uparrow q^\uparrow r^\uparrow$ is a face of the lower convex hull. This completes the proof.

Voronoi Diagrams and Upper Envelopes: (Optional) Next, let us consider the relationship between Voronoi diagrams and 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, between 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, the Voronoi diagram of points in the plane can be transformed into computing the upper envelope of a set of planes in 3-space.

Here is how we do this. For each point $p = (a, b)$ in the plane, recall from Eq. (2) that the tangent plane to Ψ passing through the lifted point $p^\uparrow$ is

$$z = 2ax + 2by - (a^2 + b^2).$$

Define $h(p)$ to be this plane. Consider an arbitrary point $q = (q_x, q_y)$ in the plane. Its vertical projection onto Ψ is (q_x, q_y, q_z), where $q_z = q_x^2 + q_y^2$. Because Ψ is convex, $h(p)$ passes below Ψ (except at its contact point $p^\uparrow$). The vertical distance from $q^\uparrow$ to the plane $h(p)$ is

$$\begin{aligned} q_z - (2aq_x + 2bq_y - (a^2 + b^2)) &= (q_x^2 + q_y^2) - (2aq_x + 2bq_y - (a^2 + b^2)) \\ &= (q_x^2 - 2aq_x + a^2) + (q_y^2 - 2bq_y + b^2) \;=\; \|qp\|^2. \end{aligned}$$

In summary, the vertical distance between $q^\uparrow$ and $h(p)$ is just the squared distance[18] from q to p (see Fig. 76(a)).

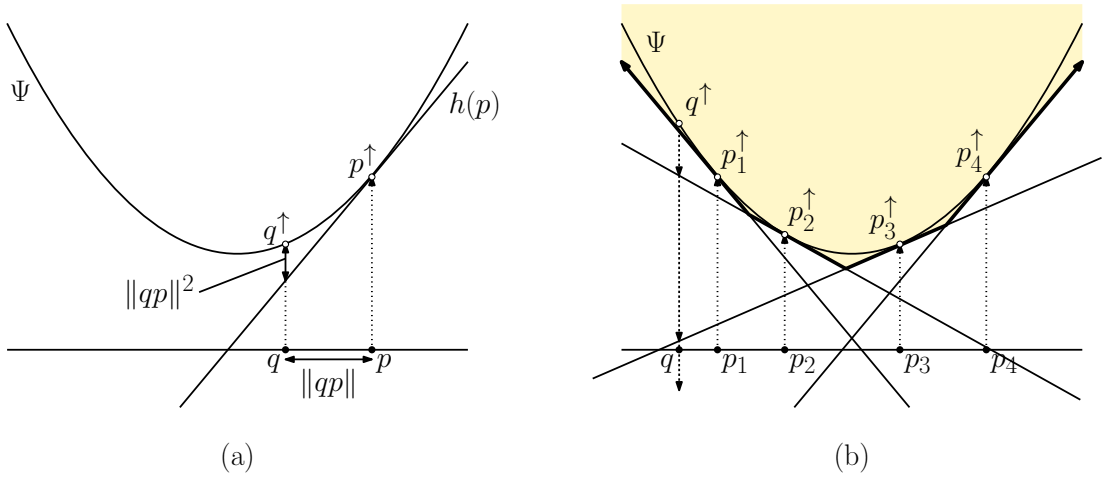


Fig. 76: The Voronoi diagram and the upper hull of tangent planes.

Now, consider a point set $P = \{p_1, \ldots, p_n\}$ and an arbitrary point q in the plane. From the above observation, we have the following lemma.

Lemma: Given a set of points P in the plane, let $H(P) = \{h(p) : p \in P\}$. For any point q in the plane, a vertical ray directed downwards from $q^\uparrow$ intersects the planes of $H(P)$ in the same order as the distances of the points of P from q (see Fig. 76(b)).

Consider the upper envelope $U(P)$ of $H(P)$. This is an unbounded convex polytope (whose vertical projection covers the entire x, y-plane). If we label every point of this polytope with the associated point of p whose plane $h(p)$ defines this face, it follows from the above lemma that p is the closest point of P to every point in the vertical projection of this face onto the plane. As a consequence, we have the following equivalence between the Voronoi diagram of P and $U(P)$ (see Fig. 77).

Theorem: Given a set of points P in the plane, let $U(P)$ denote the upper envelope of the tangent hyperplanes passing through each point $p^\uparrow$ for $p \in P$. Then the Voronoi diagram

[18] If you have studied machine learning, you may have learned about the concept of a Bregman divergence. We have essentially proved that the squared Euclidean distance is the Bregman divergence induced by the function $F(x, y) = x^2 + y^2$.

of P is equal to the vertical projection onto the (x, y)-plane of the boundary complex of $U(P)$ (see Fig. 77).

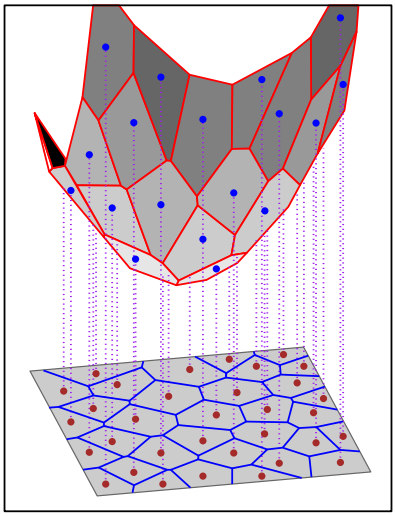


Fig. 77: The Voronoi diagram and an upper envelope.

Lecture 12: Delaunay Triangulations: Incremental Construction

Constructing the Delaunay Triangulation: We will present a simple randomized incremental algorithm for constructing the Delaunay triangulation of a set of n sites in the plane. Its expected running time is $O(n \log n)$ (which holds in the worst-case over all point sets, but in expectation over all random insertion orders). This simple algorithm had been known for many years as a practical solution, but it was dismissed by theoreticians as being inefficient because its worst case running time is $O(n^2)$. When the randomized analysis was discovered, the algorithm was viewed much more positively (albeit with the proviso to randomized the input order).

The algorithm is remarkably similar in spirit to the randomized algorithm for trapezoidal map algorithm in that it not only builds the triangulation, but it also provides a point-location data structure for the final triangulation as well. (Rather then building a point-location data structure, we will adopt an alternative method based on *bucketing* future sites.)

The input consists of a set $P = \{p_1, \ldots, p_n\}$ of point sites in $\mathbb{R}^2$. 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 the incremental algorithm for trapezoidal maps, we need some way of keeping track of where newly inserted sites are to be placed in the diagram. We will store each of the uninserted sites in a *bucket* according to the triangle in the current triangulation that contains it. We will show that the expected number of times that a site is rebucketed throughout the course of the algorithm is $O(\log n)$, which when summed over all the sites leads to a total time of $O(n \log n)$.

Incremental update: It will be convenient to assume that each newly added site lies within some triangle of the triangulation. This will not be true when sites are added that lie outside the convex hull of the current point set. To satisfy this, we well start by adding three bogus *sentinel sites* that will form an infinitely large triangle that contains all the sites. After the final triangulation is completed, we will remove these sentinel sites and their incident triangles. (In our trapezoidal map algorithm, this is analogous to putting all the segments in an enclosing rectangle.)[19] We won't show this triangle in our figures, but imagine that it is there nonetheless.

We permute the sites in random order and insert one by one. 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 into this triangle, and join the site to the three surrounding vertices (see Fig. 78(a)). This creates three new triangles incident to p, $\triangle pab$, $\triangle pbc$, and $\triangle pca$. For each triangle (say, $\triangle pab$), we check whether vertex of the triangle that lies on the *opposite side* of the edge ab lies within the circumcircle of $\triangle pab$. This is called a *local Delaunay condition*. (If there is no such vertex, because this edge is on the convex hull, then we are done.) If this vertex, call it d, fails the local Delaunay condition, we swap the edge ab out and replace it with pd. We repeat the same test process recursively with these triangles (see Fig. 78(b)).

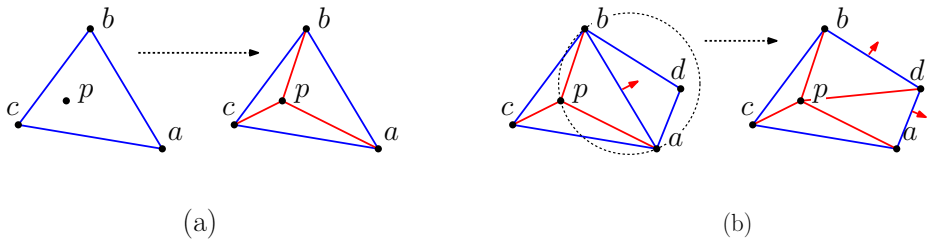


Fig. 78: Delaunay site insertion.

Local vs. Global Delaunay: An obvious issue with the above process is that we do not test all the sites for violation of the circumcircle condition, just for adjacent triangles. Why this works is related to an important issue in Delaunay triangulations. We know from the empty circumcircle condition that in a Delaunay triangulation, the circumcircle of every triangle is empty of other sites. This suggests two different criteria for testing whether a triangulation is Delaunay:

Global Delaunay: The circumcircle of each triangle $\triangle abc$ contains no other site d. (Fig. 79(a) shows a violation.)

Local Delaunay: For each pair of neighboring triangles $\triangle abc$ and $\triangle acd$, d lies outside the circumcircle of $\triangle abc$. (Fig. 79(b) shows a violation.)

Clearly, if a triangulation is globally Delaunay it is locally Delaunay. Our incremental algorithm only checks the local-Delaunay condition, however. Could it be that a triangulation might satisfy the condition locally, but fail to satisfy it globally (see Fig. 79(c))? Delaunay proved, however, that the two conditions are in fact equivalent:

Delaunay's Theorem: A triangulation is globally Delaunay iff it is locally Delaunay.

[19] Some care must be taken in the construction of this enclosing triangle. It is not sufficient that it simply contains all the sites. 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. Our book suggests a symbolic alternative, which is more reliable.

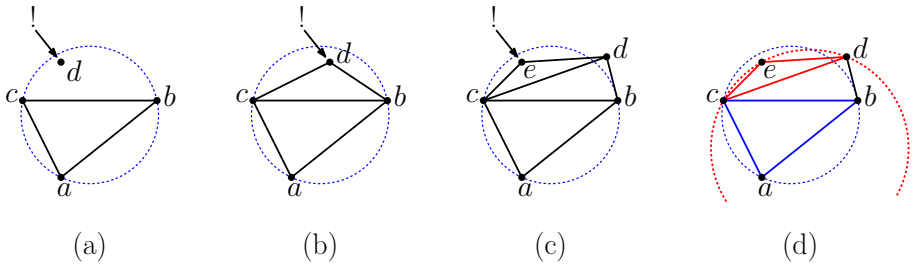


Fig. 79: Global- and local-Delaunay conditions.

Proof: (Sketch) The global to local implication is trivial, so it suffices to prove that local implies global. Consider any triangle $\triangle abc$ of a locally Delaunay triangulation, and let d be the remaining vertex of neighboring triangle that lies on the opposite side of edge bc. We assert that if d lies outside the circumcircle of $\triangle abc$, then no other site can lie within this circumcircle.

A formal justification will take too much work, so we'll just consider a limited scenario, which illustrates the key idea. Suppose that d is outside the circumcircle of $\triangle abc$ (the blue circle Fig. 79(d)) but (to the contrary) the vertex e opposite the edge cd lies within this circumcircle (see Fig. 79(d)). Consider the circumcircle of $\triangle cde$ (the red circle Fig. 79(d)). By an elementary (but somewhat tedious) analysis of the configuration of these sites, it follows that b lies within this circumcircle. Since b is in the neighboring triangle to $\triangle cde$, this implies that the triangulation is *not* locally Delaunay, which yields the contradiction.

Because the algorithm checks that all the newly created triangles are locally Delaunay, the algorithm's correctness follows as a direct consequence.

Incircle Test: Before presenting the algorithm, we shall introduce the geometric primitives involved in testing whether triangles satisfy the Delaunay condition. 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 $\triangle abc$? Let's assume that the vertices of the triangle $\triangle abc$ are given in counterclockwise order. We claim that d lies in the circumcircle determined by the $\triangle abc$ if and only if the following determinant is positive (see Fig. 80(a)).

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

This is called the *incircle test* in $\mathbb{R}^2$. (The incircle test can be generalized to any dimension.) It is notable that the incircle test in $\mathbb{R}^2$ can be viewed as an orientation test in 3-D, where we have effectively lifted the sites onto a paraboloid Ψ in $\mathbb{R}^3$ by creating an addition z-coordinate whose value is x^2+y^2. (As d moves from within the circle to outside, the lifted point $d^\uparrow$ moves from below to above the hyperplane through the lifted points $a^\uparrow$, $b^\uparrow$, and $c^\uparrow$ (see Fig. 80(b)).

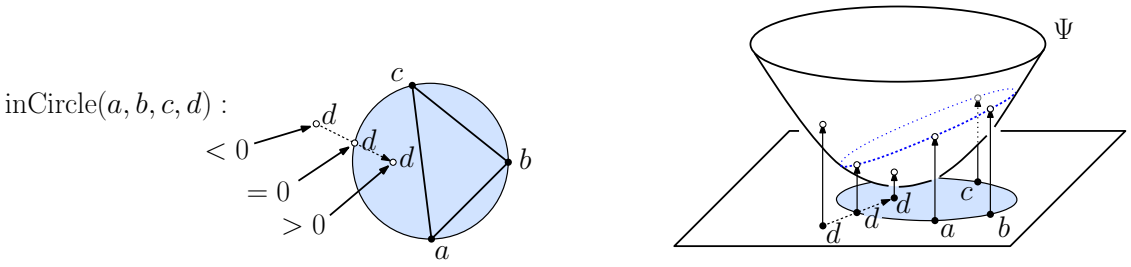


Fig. 80: Incircle test.

It is also noteworthy that, while we have defined the incircle test as testing d with respect to $\triangle abc$, the basic laws of determinants imply that (subject to a possible sign change) this can be used to test any of the four sites with respect to the triangle defined by the other three.

Deriving the Incircle Test (Optional): We will not prove the correctness of this test, but we will show a somewhat simpler assertion, namely that if the four points are cocircular then the above determinant is equal to zero. (It follows from continuity that as d moves from inside the circle to the outside, the sign of the determinant changes as well.)

Suppose that a, b, c, and d are all cocircular then 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. (We won't compute q and r, but merely assume their existence for now.) Expanding this and collecting common terms we have

$$\begin{aligned} 0 &= (a_x^2 + a_y^2) - 2q_x a_x - 2q_y a_y + (q_x^2 + q_y^2 - r^2) \\ &= (-2q_x)a_x + (-2q_y)a_y + 1 \cdot (a_x^2 + a_y^2) + (q_x^2 + q_y^2 - r^2) \cdot 1. \end{aligned}$$

If we do the same for the other three points, b, c, and d, and express this in the form of a matrix, we have

$$\begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} \begin{pmatrix} -2q_x \\ -2q_y \\ 1 \\ q_x^2 + q_y^2 - r^2 \end{pmatrix} = 0.$$

In other words, there exists a linear combination of the columns of the 4×4 matrix that is equal to the zero vector. We know from linear algebra that this is true if and only if the determinant of the matrix is zero.

Randomized Incremental Construction: We can now present the complete algorithm. Given the set $P = \{p_1, \ldots, p_n\}$ of sites, we first compute the sentinel triangle containing them all. We then permute the sites randomly and insert them into the triangulation one by one.

The algorithm for the incremental algorithm is shown in the code block below, and an example is presented in Fig. 81. The current triangulation is kept in a global data structure, implemented as a doubly-connected edge list (DCEL). All of the basic operations described below (finding edges and vertices and flipping edges) can be done in $O(1)$ time.

```
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                      // replace ab with pd
        SwaptTest(ad)                     // check/fix the new suspect edges
        SwaptTest(db)
    }
}
```

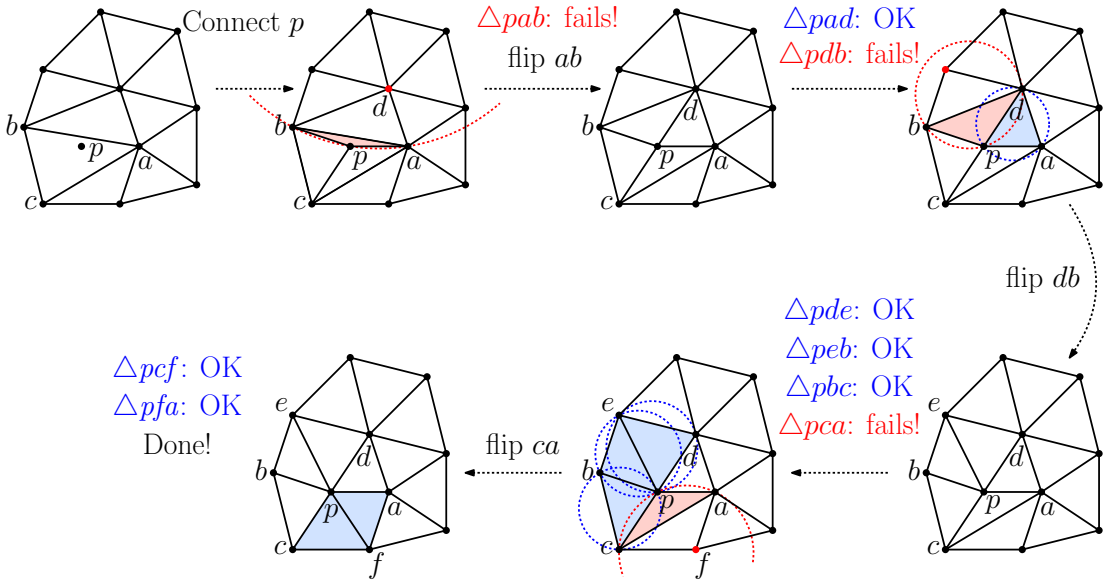


Fig. 81: Incremental site insertion.

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 (a DCEL, for example). The other issue is locating the triangle that contains p. We will discuss this below.

Running-Time Analysis: To analyze the expected running time of algorithm we will establish two bounds, each averaged over all possible insertion orders. With the addition of each site:

(1) $O(1)$ structural changes are made to the triangulation (in expectation), and

(2) $O(\log n)$ time is spent determining which triangle contains each newly inserted site (in expectation).

These bounds depend *only* on the insertion order, not the distribution of the sites.

Bounding the 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 (see Fig. 82). Although any one vertex may have a very high degree, we will exploit the fact that in a planar graph, the average vertex degree is just a constant.

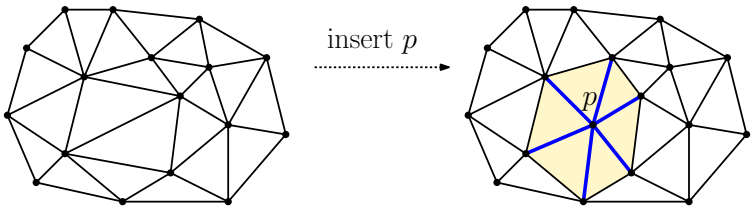


Fig. 82: Number of structural changes is equal to p's degree after insertion (three initial edges and three edge flips).

To perform the backwards analysis, we consider the situation after the insertion of the ith site. Let d_i be a random variable that indicates the degree of the newly inserted site in our randomized algorithm. Let $P_i = \{p_1, \ldots, p_i\}$ denote the first i sites to be inserted. Although the diagram depends on which particular i sites are in this subset, our analysis will not. For $1 \leq j \leq i$, let $\deg(p_j)$ denote the degree of site p_j in triangulation $DT(P_i)$ just after the ith insertion.

Because the diagram does not depend on the insertion order, each of the sites of P_i has an equal probability of $\frac{1}{i}$ of being the last site to be inserted. Recall that (by Euler's formula), the triangulation has at most $3i$ edges. It is easy to see that the sum of vertex degrees is equal to twice the total number of edges (since each edge is counted twice), that is, $6i$. We

conclude that expected value of d_i, denoted $E[d_i]$, satisfies:

$$E[d_i] \;=\; \frac{1}{i}\sum_{j=1}^{i} \deg(p_i) \;\leq\; \frac{6i}{i} \;=\; 6.$$

Therefore, by the magic of backwards analysis, the expected number of structural changes following the insertion of the ith site is, in expectation, just 6.

Bounding the Location Cost: The second aspect of the expected-case running time is the cost of determining which triangle contains each newly created site. As mentioned earlier, we will employ a bucketing approach, as we did with the trapezoidal-map algorithm. Think of each triangle of the current triangulation as a *bucket* that holds the sites that lie within this triangle and have yet to be inserted (see Fig. 83(a)). When a new site p is inserted, a number of old triangles are deleted (shaded red in Fig. 83(a)) and a number of new triangles are created (shaded blue in Fig. 83(b)). All the sites in the buckets of the old triangles need to be moved into the associated new triangle. This process is called *rebucketing*.

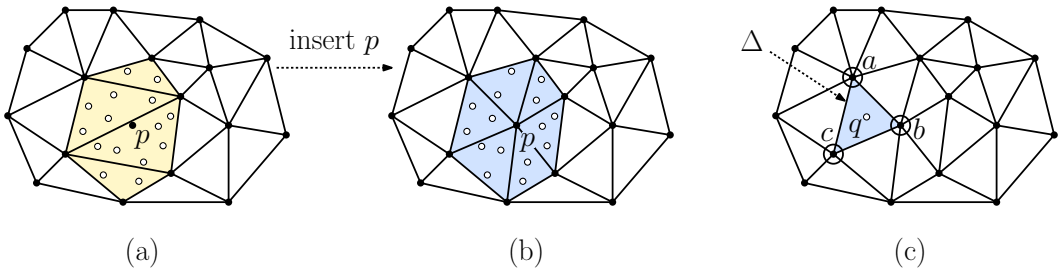


Fig. 83: Rebucketing sites after inserting site p.

For the sake of simplifying the analysis, let us assume that the cost of rebucketing a single site during a single insertion is $O(1)$. (The issue is that the cost of rebucketing depends on the degree of the newly inserted site. In the previous section we showed that the average degree is a constant, so this assumption is not unreasonable.) We will show through a backwards analysis that, in expectation, any fixed site is rebucketed $O(\log n)$ times.

Let us fix a site $q \in P$. Consider the situation just after the insertion of the ith site. We may assume that q has not yet been inserted, since otherwise its rebucketing cost is zero after the ith insertion. For $1 \leq i \leq n$, let $X_i(q)$ denote the random event that q is moved to a new triangle after the ith insertion, and let $\text{Prob}(X_i(q))$ denote the probability of this event. Letting $B(q)$ denote the average number of times that q is rebucketed throughout the algorithm, we have

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

To bound $\text{Prob}(X_i(q))$, let Δ be the triangle containing q after the ith insertion. As observed above, after we insert the ith site, all the newly created triangles are incident to this new site. Thus, Δ would have come into existence as a result of the last insertion if and only if one of its three incident vertices happened to be the last to be inserted (see Fig. 83(c)). Since Δ is incident to exactly three sites, and every site is equally likely to be the last inserted, it follows that the probability that Δ came into existence is $\frac{3}{i}$. (We are cheating a bit here by ignoring the three initial sites at infinity.) Therefore, $\text{Prob}(X_i(q)) \leq \frac{3}{i}$.

From this, it follows that the expected number of times that the site q is rebucketed is

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

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

$$B(q) \;\leq\; 3 \cdot \ln n \;=\; O(\log n).$$

Although the diagram depends on the order in which the sites have been added, this bound does not. Summing over all the n sites, it follows that the total time spent rebucketing all the sites is $\sum_{i=1}^{n} B(p_i) = O(n \log n)$.

Point Location Data Structure: The above analysis is reminiscent of the analysis we performed for point-location for the incremental algorithm for trapezoidal maps. Just as we did there, rather than just rebucketing points, we could build a directed acyclic graph (DAG) recording the history of rebucketing operations on a triangle-by-triangle basis. The result is a data structure that can answer point-location queries. The query time is the same as that of the trapezoidal map (modulo constant factors).

Lecture 13: Line Arrangements and Applications

Line Arrangements: We have studied a number of the most fundamental structures in computational geometry: convex hulls, Voronoi diagrams and Delaunay triangulations. These are all defined over a finite set of points. As we saw earlier, points and lines in the plane are related to each other through the *dual transformation*. In this lecture, we will studey a fundamental structure defined for a finite set of lines, called a *line arrangement*.

Consider a finite set $L = \{\ell_1, \ldots, \ell_n\}$ of lines in the plane. 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. 84(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, also called *cells*. 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 (see Fig. 84(b)). An arrangement can be represented using any standard data structure for cell complexes, a DCEL for example.

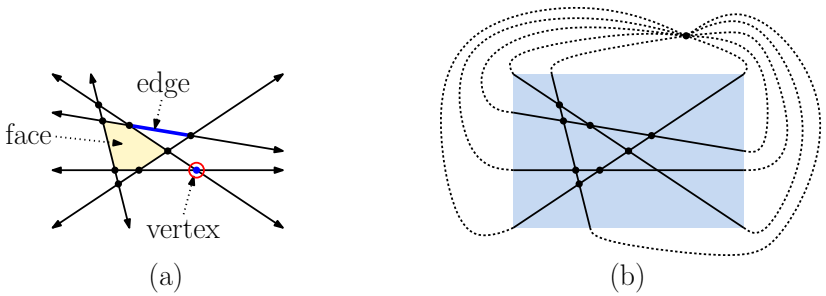


Fig. 84: Arrangement of lines; (a) the basic elements of an arrangement and (b) adding a vertex at infinity to form a proper planar graph.

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. Although we will not discuss it, line arrangements in $\mathbb{R}^2$ can be generalized to hyperplane arrangements in $\mathbb{R}^d$. In such a case the arrangement is a polyhedral cell complex.

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 we will be interested only in simple arrangements. We will also assume that no two lines are parallel. 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 $\binom{n}{2} = \frac{1}{2}(n^2-n)$.

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

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

Proof: The fact that the number of vertices is $\binom{n}{2}$ is clear from the fact that (since no two are parallel) each pair of lines intersects in a single point.

The number of edges follows from the fact that each line contains n lines. This is because each line is cut by each of the other $n-1$ lines (assuming no two parallel lines), which splits the line into n edges.

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.

Note that all these quantities are $\Theta(n^2)$. 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.[20] 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.

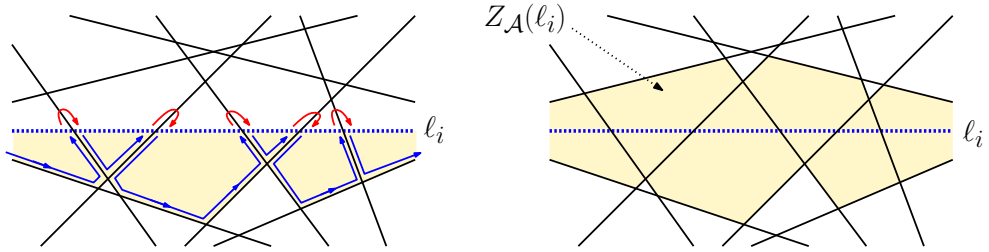
[20] This is not quite accurate. For some applications, it suffices to perform a plane-sweep of the arrangement. There is a plane-sweep algorithm specially tailored for line arrangements, called topological plane sweep.

Let $L = \{\ell_1, \ldots, \ell_n\}$ denote the set of lines. We will add lines one by one and update the arrangement after each insertion. We will show that the ith line can be inserted in $O(i)$ time (irrespective of the insertion order). Summing over i, this yields a total running time proportional to $\sum_{i=1}^{n} i = O(n^2)$.

Suppose that the first $i-1$ lines have already been inserted. Consider the insertion of ℓ_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 (irrespective of their y-intercepts). In time $O(i)$ we can determine where the slope of ℓ_i falls relative to the slopes of the prior $i-1$ lines, and this determines the leftmost face of the arrangement that contains this line.[21]

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 ℓ_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 ℓ_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 ℓ_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 ℓ_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. 85(a). The DCEL data structure supports such local traversals in time linear in the number of edges traversed. (You might wonder why we don't generalize the trapezoidal map algorithm. We could build a trapezoidal map of the arrangement and walk the new segment through a sequence of trapezoids. It turns out that this would be just as efficient.)



(a) (b)

Fig. 85: Adding the line ℓ_i to the arrangement; (a) traversing the arrangement and (b) the zone of a line ℓ_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 $O(i^2)$, which is much higher than the $O(i)$ time that we promised earlier. Why is this wrong? It is based on bound of the total complexity of the faces traversed. To improve this, we need to delve more deeply into an important combinatorial structure related to arrangements.

[21] In fact, we could do this in $O(\log i)$ time by storing the slopes in an ordered dictionary, but this would not improve our overall running time.

Zones and the Zone Theorem: Given an arrangement $\mathcal{A}$ of a set L of n lines, and given a line ℓ that is not in L, the *zone* of ℓ in $\mathcal{A}(L)$, denoted $Z_{\mathcal{A}}(\ell)$, is the set of faces of the arrangement that are intersected by ℓ (shaded in Fig. 85(b)). The running time of the incremental construction clearly grows linearly with the total number of edges of $Z_{\mathcal{A}}(\ell)$. (Actually, we only really care about the edges that lie below ℓ, but this won't affect the asymptotics.)

As argued above, the combinatorial complexity of a zone is clearly at most $O(n^2)$. However, we will show next that (remarkably!) the total complexity of any zone is at most $O(n)$.

Zone Theorem: Given an arrangement $\mathcal{A}(L)$ of n lines in the plane, and given any line ℓ in the plane not in L, the total number of edges in all the cells of the zone $Z_{\mathcal{A}}(\ell)$ is at most $6n$.

As with many combinatorial proofs, the key is to organize matter so that the counting can be done in an easy way. This is not trivial. We cannot count cell-by-cell, since some cells have high complexity and some low. We also cannot count line-by-line, because some lines contribute many edges to the zone and others just a few. The key in the proof is finding a (clever!) way to add up the edges so that each line appears to induce only a constant number of edges into the zone. (Note that our text counts zone edges a bit differently.)

Proof: The proof is based on a simple inductive argument. For the sake of illustration, let us rotate the plane so that ℓ is horizontal. By general position, we may assume that none of the lines of L are parallel to ℓ. 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. An edge of a face is said to be *left bounding* if the face lies in the right halfplane of the line defining this edge, and a face is *right bounding* if the face lies in the left halfplane of the line defining this edge (see Fig. 86). We will show that there are at most $3n$ left-bounding edges in the zone, and by a symmetrical argument there are at most $3n$ right-bounding edges. This will yield the desired total of $6n$ edges.

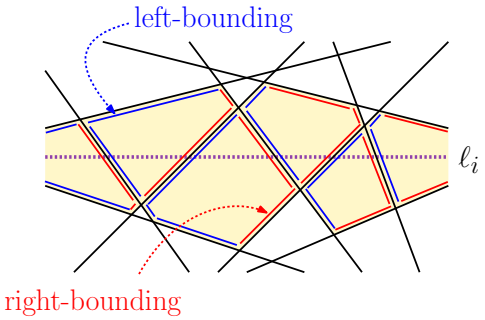


Fig. 86: Left- and right-bounding edges in the zone.

The proof is by induction on n. For the basis case, when $n = 1$, then there is exactly one left-bounding edge in ℓ's zone, and $1 \leq 3 = 3n$. For the induction step, let us assume the induction hypothesis is true for any set of $n-1$ lines, and we will show that it holds for an arrangement of n lines. Consider the rightmost line of the arrangement to intersect ℓ. Call this ℓ_1 (see Fig. 87(b)). Prior to its existence, the induction hypothesis implies that there are at most $3(n-1)$ left-bounding edges in the zone of the remaining $n-1$ lines.

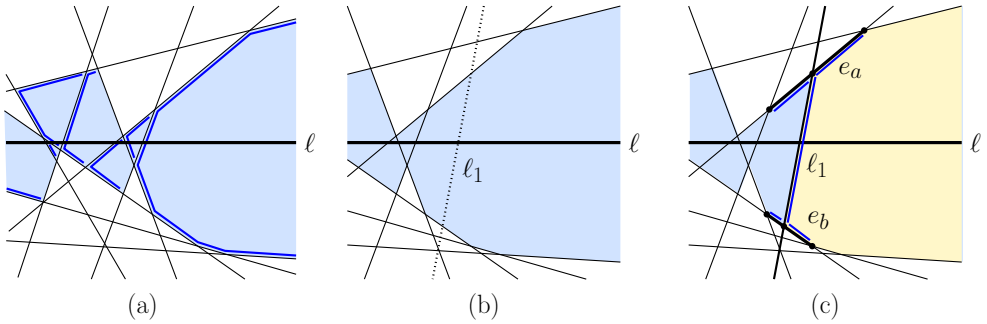


Fig. 87: Proof of the Zone Theorem.

Now, let us add ℓ_1 and see how many more left-bounding edges are generated. Because ℓ_1 is leftmost, it intersects the rightmost face of the zone. Observe that all of the edges of this face are left-bounding edges. By convexity, ℓ_1 intersects the boundary of this face in two edges, denoted e_a and e_b, where e_a is above ℓ, and e_b is below (see Fig. 87(c)). Its insertion creates a new left-bounding edge running along ℓ_1 between e_a and e_b, and it splits each of the edges e_a and e_b into two new left-bounding edges. Thus, there is a net increase by three edges, for a total of $3(n-1)+3 = 3n$ edges.

We assert that ℓ_1 cannot contribute any other left-bounding edges to the zone. This is because the lines containing e_a and e_b block any possibility of this. Therefore, there are at most $3n$ left bounding edges, as desired.

Applications of Arrangements and Duality: When combined with point-line duality, line arrangements 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. 88(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. 88(b)).

Maximum stabbing line: Given a set of n line segments in the plane, compute the line ℓ that stabs (intersects) the maximum number of these line segments (see Fig. 88(c)).

Ham Sandwich Cut: Given n red points and m blue points, find a single line ℓ 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 sepa-

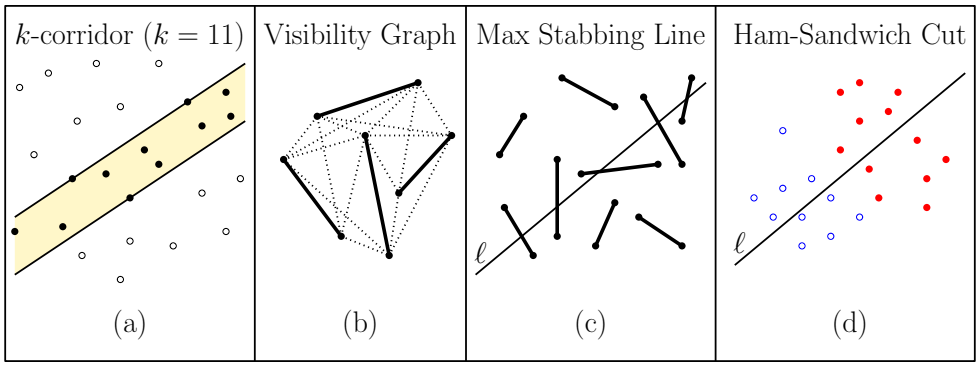


Fig. 88: Applications of arrangements.

rable 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. 88(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, we can update the sweep-line status by swapping two adjacent entries. Thus, instead of an ordered dictionary, it suffices to store the lines in a simple n-element array, sorted, say, from top to bottom. This means the **sweep-line updates can be performed in $O(1)$ time**, rather than $O(\log n)$(see Fig. 89(a)). We still need to maintain the priority queue, and these operations take $O(\log n)$ time each.

Sweeping an arrangement 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 arrangement.

Topological Plane Sweep: (Optional) As mentioned above, the priority queue is the slowest part of plane sweeping an arrangement. Remarkably, there is a way to save this $O(\log n)$ factor, but we must abandon hope of sweeping events in purely left-to-right order. There is a somewhat more "relaxed" version of plane sweep, which works for line arrangements in the plane. The method is called *topological plane sweep*. It runs in $O(n^2)$ time (thus, eliminating an $O(\log n)$ factor from the running time) and uses $O(n)$ space.

It achieves efficiency relaxing the requirement that vertices be swept in strict left-to-right order. Rather, it uses a more "local" approach for deciding which vertex of the arrangement

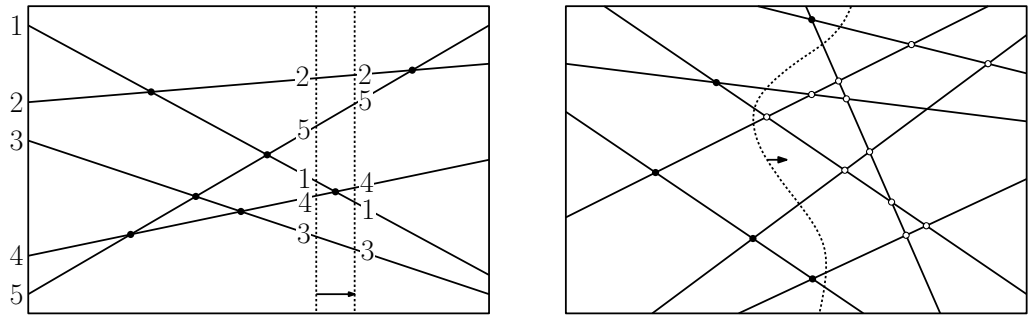


Fig. 89: Sweeping a line arrangement.

to sweep next.[22] This local approach guarantees that the vertices along each line are swept in their proper order, even though vertices lying on different lines are not necessarily swept in their proper left-to-right order. Intuitively, we can think of the sweep line as a sort of *pseudoline*, that intersects each line of the arrangement exactly once (see Fig. 89(b)). Although we will not present any justification, it method applicable to all the problems we will discuss in today's lecture. *Note that topological plane sweep applies only to infinite lines, not line segments.*

Duality: Many of our applications will involve the dual transformation, which we introduced earlier in the semester. Recall that the dual of a point $p = (a, b)$ is the line $p^* : y = ax - b$, and the dual of a line $\ell : y = ax - b$ is the point $\ell^* = (a, b)$. Also recall the *order-reversing property* that the point p lies above line ℓ (at vertical distance h) if and only if the dual line p^* lies below dual point ℓ^* (also at vertical distance h).

Narrowest k-corridor: 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. (We call this a *slab* or *corridor*.) We define the *width* of the corridor to be the vertical distance between these. Our objective is to compute the corridor of minimum width that encloses k points (which may lie on the corridor's boundary.) It is straightforward to adapt the algorithm to minimize the perpendicular distance between the lines. 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.

Consider any corridor defined by parallel lines ℓ_a (above) and ℓ_b (below) (see Fig. 90(a)). Since the lines are parallel, these points share the same a-coordinate, which implies that the line segment $\overline{\ell_a^* \ell_b^*}$ is vertical (see Fig. 90(b)). The vertical distance between the lines (that is, the difference in their y-intercepts) is the same as the vertical distance between the dual points.

By the order-reversing property, points that lie above/below/within the corridor (shown in blue, red, and black in Fig. 90(a), respectively) are mapped to dual lines that pass below/above/through this segment (see Fig. 90(b)). Thus, we have the following equivalent dual formulation of this problem:

[22]For details, see "Topologically sweeping an arrangement" by H. Edelsbrunner and L. J. Guibas, *J. Comput. Syst. Sci.*, 38 (1989), 165–194.

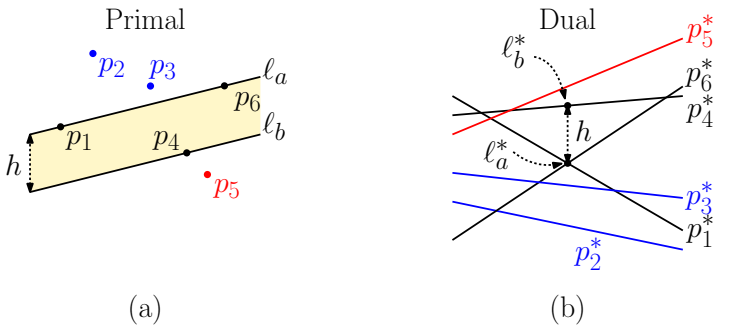


Fig. 90: A 3-corridor in primal and dual forms. (Note that the corridor is not as narrow as possible.)

Shortest vertical k-stabber: Given an arrangement of n lines, determine the shortest vertical segment that stabs (intersects) k lines of the arrangement.

It is easy to show that the shortest vertical k-stabber may be assumed to have one of its endpoints on a vertex of the arrangement. (If the vertical line has endpoints in the interior of two edges, we can slide it left or right and decrease its length.)

3-stabber: The 3-stabber is the simplest to describe. A perform a simple plane sweep of the arrangement (using a vertical sweep line). Whenever we encounter a vertex of the arrangement, we consider the distance from this vertex to the edge of the arrangement lying immediately above this vertex and the edge lying immediately below. (These are illustrated by the blue broken lines in Fig. 91(a).) 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 in the running time can be removed.)

Note that we can use this to test whether points are in general position. It is easy to prove that a set of n points has three or more collinear points if and only if the dual arrangement's minimum 3-stabber is of length zero.

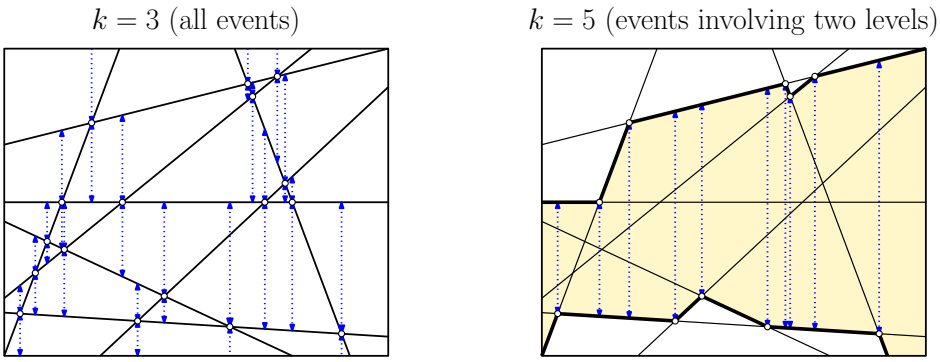


Fig. 91: The critical events in computing the shortest vertical 3-stabber (a) and 5-stabber (b).

k-stabber: Whenever we encounter a vertex in the plane sweep, we determine the distance to the lines of the arrangement that lie $k-2$ above and $k-2$ below (see the blue broken lines of Fig.91(b)). The reason for the "-2" is to account for two lines that pass through

the vertex itself. Recalling that the sweep-line status can be stored in a simple n-element array, it is possible to access these entries in $O(1)$ time for any value of k.

Halfplane Discrepancy: Next we consider a problem derived geometric 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 possess is that for any halfplane h, we the fraction of points of P that lie within h should be roughly equal to the area of intersection of h with U. More precisely, define $\mu(h)$ to be the area of $h \cap U$, and $\mu_P(h) = |P \cap h|/|P|$. A sample is good if $\mu(h) \approx \mu_P(h)$, for any choice of h.

To make this more formal, we define the *discrepancy* (or more accurately, the *halfplane discrepancy*) of a finite point set P with respect to a halfplane h to be

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

For example, in Fig. 92(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:

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

Let's consider the problem of computing the discrepancy of a given point set P.

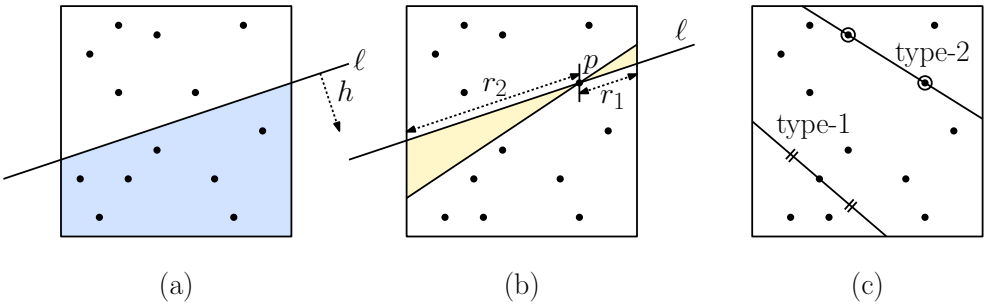


Fig. 92: Discrepancy of a point set.

Since there are an uncountably infinite number of halfplanes that intersect the unit square, we should first 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 ℓ denote the line that bounds h. Then either:

(i) ℓ passes through one point of P, and this point is the midpoint of the line segment $\ell \cap U$, or

(ii) ℓ passes through two points of P.

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 produces 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: We will show that any line can be moved until it satisfies either (i) or (ii) in such a manner that the discrepancy never decreases. First, if ℓ 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, until it does pass through a point of P. Next, if ℓ 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. 92(b). Suppose that the line ℓ passes through point p and let $r_1 < r_2$ denote the two lengths along ℓ from p to the sides of the square. Observe that if we rotate ℓ through a small angle θ, then to a first order approximation, the 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 θ. 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 θ, the discrepancy cannot be achieved as long as such a rotation is possible.

We say that a line is *type-1* if it satisfies condition (i) and it is *type-2* if it satisfies condition (2) (see Fig. 92(c)). We will show that the discrepancy for each types of lines can be computed in $O(n^2)$ time.

Type-1: For each point $p \in P$, there are only a constant number of lines ℓ (at most two, I believe) 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. We can compute the discrepancy of each such line in $O(n)$ time, which leads to a total running time of $O(n^2)$.

Type-2: Consider a type-2 line ℓ that passes through two points $p_i, p_j \in P$. This line defines two halfplanes, one above and one below. We'll explain how to compute the discrepancy of the lower halfplane, h^-, and the upper halfplane, h^+, is symmetrical. First, observe that we can compute $\mu(h^-)$ in constant time, so all that remains is computing $\mu_P(h^-)$, that is, the number of points lying on or below ℓ.

If we apply our standard dual transformation, ℓ is mapped in the dual plane to a point ℓ^* at which the dual lines p_i^* and p_j^* intersect. This is just a vertex in the line arrangement $\mathcal{A}(P^*)$. By the order-reversing property of the dual transformation, the points lying on or below ℓ coincide with the dual lines that lie on or above this vertex.

We can compute this quantity in constant time for each vertex of the line arrangement. Recall that the sweep-line status is stored in a simple n-element array, sorted from top to bottom. The vertex in the arrangement corresponds to two consecutive entries in the sweep-line status, say at positions $k-1$ and k. The number of dual lines lying on or above the vertex is therefore just k (assuming that we index the array from 1 to n).

For example, consider the vertex being swept in Fig. 89(a). These intersecting lines are at indices $k-1 = 3$ and $k = 4$ in the sweep-line status, and hence there are 4 lines on or above this vertex, which implies that there are 4 points lying on or below the corresponding type-2 line $\overline{p_1p_4}$ in the primal configuration. Since we can compute the discrepancy for each type-2 line in $O(1)$ time, the overall time to compute the discrepancies of all type-2 lines is $O(n^2)$.

Levels: The analysis that was done above for type-2 lines suggests another useful structure within a line arrangement. We can classify each element of the arrangement according to the number of lines lying above and below it. We say that a point is at *level* k, denoted $\mathcal{L}_k$, in an arrangement if there are at most $k-1$ lines (strictly) above this point and at most $n-k$ lines (strictly) below it. It is easy to see that $\mathcal{L}_k$ is an x-monotone polygonal curve (see Fig. 93(a)). For example, $\mathcal{L}_1$ is the upper envelope of the lines, and $\mathcal{L}_n$ is the lower envelope. Assuming general position, each vertex of the arrangement is on two levels, which meet at this vertex in a "knocked-knee" manner. Given the arrangement $\mathcal{A}(L)$ of a set of n lines, it is an easy matter to label every edge of the arrangement with its level number, by tracking its index in the sweep-line status.

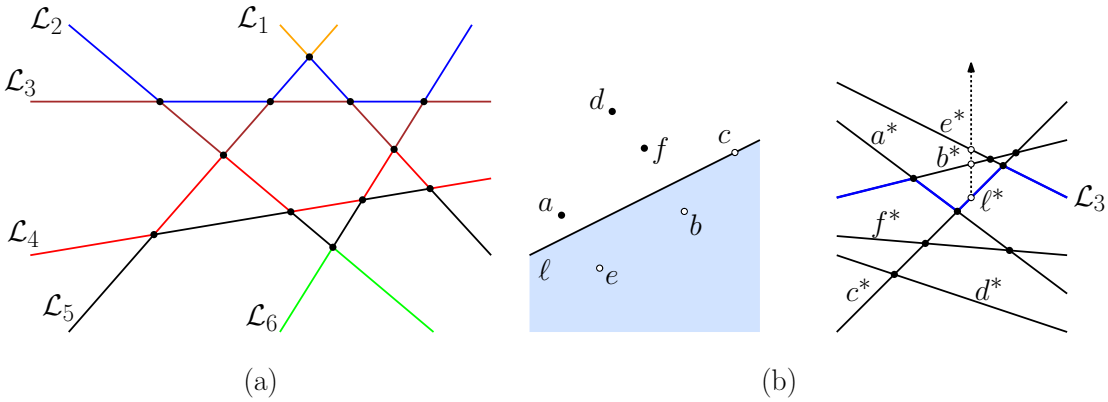


Fig. 93: Levels in an arrangement and k-sets.

There is a dual equivalence between a level in an arrangement and a concept called k-sets. Given an n-element point set P and integer $1 \le k \le n$, a k-element subset of $P' \subseteq P$ is called a *k-set* of P if there exists a halfplane h such that $P' = P \cap h$. For example, if (p_i, p_j) is an edge of the convex hull of P, then $P' = \{p_i, p_j\}$ is a 2-set of P. A classical question in combinatorial geometry is, as a function of n and k, what is the maximum number of possible k-sets that any n-element set can have. (The current best bounds range between $O(n \log k)$ and $O(nk^{1/3})$.)

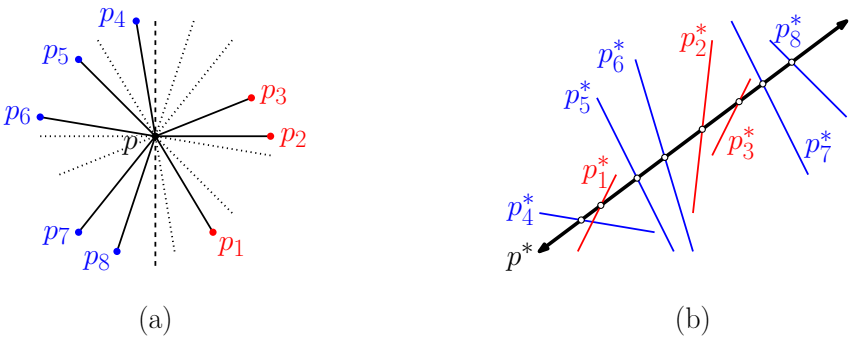
There is a close relationship between the k-sets of P and level k of the dual arrangement $\mathcal{A}(P^*)$. To see this, let us first distinguish between two types of k-sets. We say that a k-set is a *lower k-set* if it lies in the halfplane beneath a line ℓ and otherwise it is an *upper k-set*. Let's just consider lower k-sets, since upper k-sets are symmetrical (by reflecting the points about the x-axis).

Consider any lower k-set defined by some line ℓ. We may assume that ℓ passes through a point of P, and hence there are $k-1$ points strictly below ℓ. The associated dual point ℓ^* lies on an edge of the dual arrangement, and by the order-reversing property of the dual transformation, there are $k-1$ lines of $\mathcal{A}(P^*)$ that pass strictly above ℓ^*. (For example, in Fig. 93(b), the lower 3-set $\{c, b, e\}$ is defined by a line ℓ, which passes through c. In the dual setting, the point ℓ^* lies on the dual line c^* and lies on level $\mathcal{L}_3$ because lines b^* and e^* lie above it.) The upper k-sets can be identified with level $\mathcal{L}_{n-k+1}$, because each point on this level has k lines passing on or below it, and hence $n-k+1$ lines on or above.

Sorting all angular sequences: Earlier, we introduced the problem of computing visibility graphs. We will not explicitly discuss the solution of that problem here, but we will discuss a fun-

damental subroutine in this algorithm. Consider a set of n points in the plane. For each point p in this set we want to sort the remaining $n-1$ point in cyclic order. Clearly, we can compute the cyclically sorted order about any given point in $O(n \log n)$ time, and this leads to an overall running time of $O(n^2 \log n)$. We will show that we can do this $O(n^2)$ time. (This is rather surprising. Lower bounds on sorting imply that we cannot sort n sets of $n-1$ numbers faster than $\Omega(n^2 \log n)$ time. But here we are exploiting the special structure of the cyclically ordered point sets.)

Here is how we do it. 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. 94(a)). Consider the arrangement defined by the dual lines p_i^*. How does this order manifest itself in the arrangement?



(a) (b)

Fig. 94: 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, which is not quite the same as the *angular ordering*.

However, given this slope ordering, we can simply test which primal points lie to the left of p (shown in blue in Fig. 94(a)), and separate them from the points that lie to the right of p (shown in red in Fig. 94(a)). 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. For example, in Fig. 94, we partition the slope-sorted sequence $\langle 4, 1, 5, 6, 2, 3, 7, 8 \rangle$ into the left sequence $\langle 4, 5, 6, 7, 8 \rangle$ and the right sequence $\langle 1, 2, 3 \rangle$, and then we concanate them to obtain the final angle-sorted sequence $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$.

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. (Topological plane sweep can also be used, but since the output size is $\Omega(n^2)$, there no real benefit to be achieved by using topological plane sweep.)

Lecture 14: 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. This is an NP-hard problem, but there exists a *polynomial time approximation scheme* (or PTAS), which is an algorithm that achieves an approximation factor of $1+\varepsilon$, for any $\varepsilon > 0$.

Another source arises for algorithms in higher dimensions. For example, in an earlier lecture we showed that the *Euclidean minimum spanning tree* (EMST) could be solved in $O(n \log n)$ time in $\mathbb{R}^2$, but in dimensions 3 and higher, the running time can be considerably higher (growing close to $O(n^2)$, a running time achievable without any consideration of geometry). However, it is possible to achieve an approximation in time $O(n \log n)$ in any space of constant dimension (see Fig. 95(a)). Another example is computing the *convex hull* of a point set. We saw in an earlier lecture that the complexity can be as high as $\Omega(n^{\lfloor d/2 \rfloor})$, which is $\Omega(n^2)$ in $\mathbb{R}^3$. We will show that it is possible to compute an approximation to the convex hull in time $O(n \log n)$ in any constant dimension (see Fig. 95(b)).

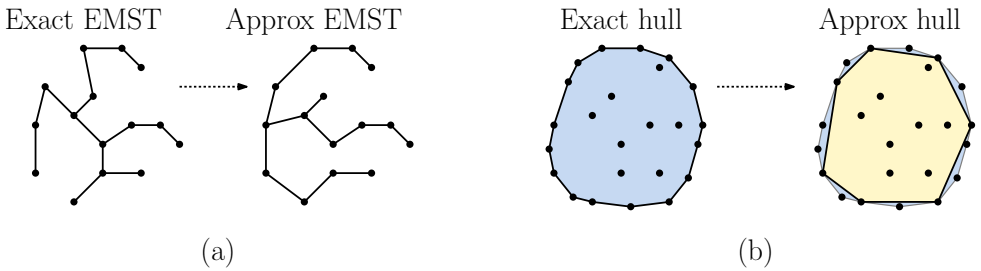


Fig. 95: Geometric approximations: (a) Euclidean traveling salesman, (b) approximate convex hull.

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.

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*[23] (or FMM for

[23] As an indication of how important this algorithm is, it was listed among the top-10 algorithms of the 20th century,

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 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. 96).

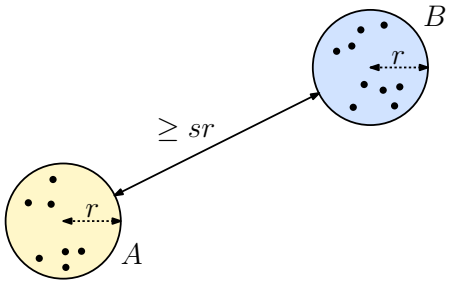


Fig. 96: A well separated pair with separation factor s.

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) 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 $1 \leq i \leq m$

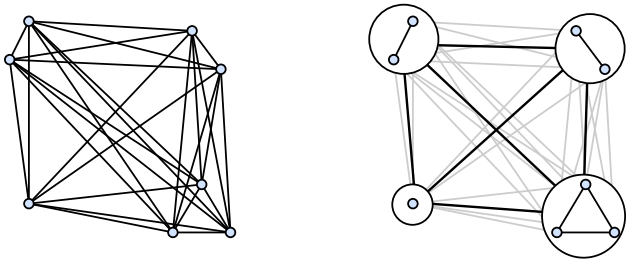
along with quicksort, the fast Fourier transform, and the simplex algorithm for linear programming.

(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 $A_i \otimes B_i$ (that is, the cover of $P \otimes P$ is actually a partition), our construction will have this further property. An example is shown in Fig. 97. (Although there appears to be some sort of hierarchical structure here, note that the pairs are not properly nested within one another.)



28 pairs 11 well-separated pairs

Fig. 97: 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. 98. The points are shown in Fig. 98(a), the node structure in Fig. 98(b), and the final tree in Fig. 98(c).) Quadtrees can be used to store various types of data. Formally, the structure we have just described in called a *PR-quadtree* (for "point-region quadtree").

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. 99(a)).

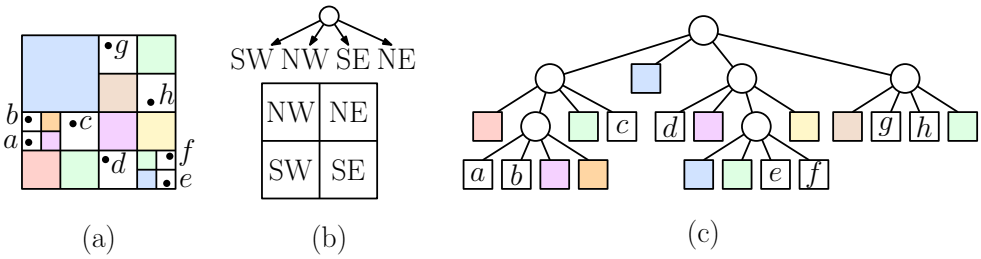


Fig. 98: The quadtree for a set of eight points.

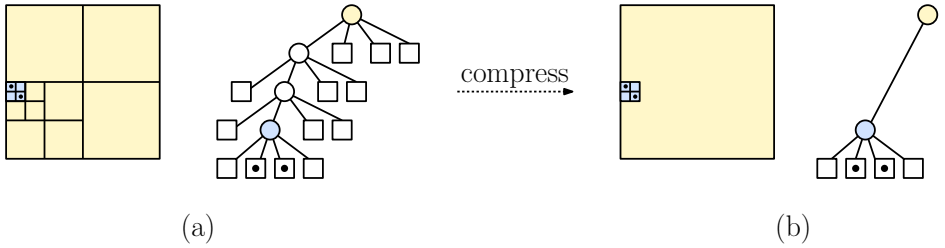


Fig. 99: Compressed quadtree: (a) The original quadtree, (b) after path compression.

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. 99(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.)

Summary and Augmentation: Here are the key properties that we will use about quadtrees:

(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 children* (at most 2^d).

(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 *shrinks 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 a *packing property* of quadtrees, which limits the number of quadtree cells of a given side length that can overlap a Euclidean ball. We will return to this later.

For the construction of the WSPD, we need to make a few augmentations to the quadtree structure.

Associated set: Given any node u, let $P(u)$ denote the set of points stored in u's subtree. (We will represent each well separated pairs concisely as a pair of nodes $\{u, v\}$, but the actual WSP is implicitly understood to be $\{P(u), P(v)\}$.)

Representative: Each node u (other than empty leaves) is associated with an arbitrary point from $P(u)$, called its *representative*, and denoted rep(u). We can compute the representatives by a simple bottom-up traversal of the quadtree.

Level: We label each node u with its *level* in the tree, denoted level(u). It is defined to be the depth of the node in the tree, where the root is at level 0, the root's children at at depth 1, its grandchildren at depth 2, and so on. Observe that if the points are scaled so that the bounding box has side length 1, then a node of depth i is associated with a cell of side length $1/2^i$. This, the higher the level, the smaller the cell.

For technical reasons, it will be convenient to treat leaf nodes differently. For each leaf node u, set level$(u) = +\infty$. The algorithm always splits nodes of lower level first, and this will prevent us from ever splitting a leaf node.

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.

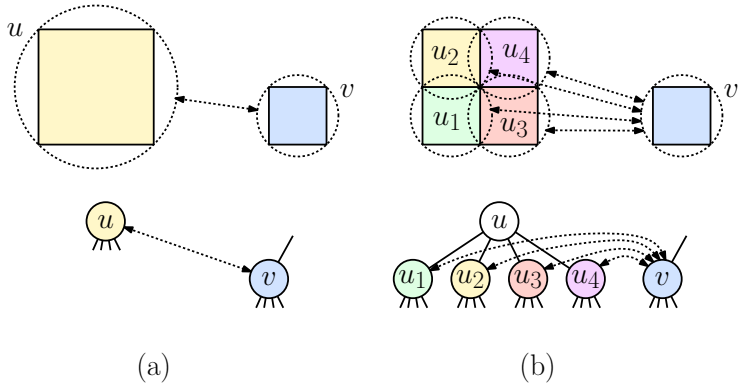


Fig. 100: WSPD recursive decomposition step.

The algorithm is based on a recursive subdivision process. Consider a pair of nodes $\{u, v\}$ that arise in the decomposition process. If either of the nodes is an empty leaf, then we may ignore this pair. If both of the nodes are leaves, then they are clearly well-separated (irrespective of the value of s), and we may output this pair. Otherwise, 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. (Recall that a leaf node is treated as an infinitesimally small quadtree cell that contains the node's point, and its level is defined to be $+\infty$. So if an internal node and a leaf node are compared, the internal node is always deemed to have the larger cell.) Consider the two smallest Euclidean balls of equal radius that enclose u's cell and v's cell (see Fig. 100(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. 100(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 (u and v are leaves and u = v) return;
    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_k denote the children of u;
        return ⋃_{i=1}^{k} 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 cells. 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.

Remark: Due to its symmetry, this procedure may produce duplicate pairs $\{P(u), P(v)\}$ and $\{P(v), P(u)\}$. A simple disambiguation rule can be applied to eliminate this issue.

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 the procedure 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 $\max(s, 1)$). The analysis involves a number of steps.

Terminal/Non-Terminal Calls: To evaluate the number of well separated pairs, we will count calls to the procedure 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.

Charging: 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.

Who gets charged? 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. 101(a)).

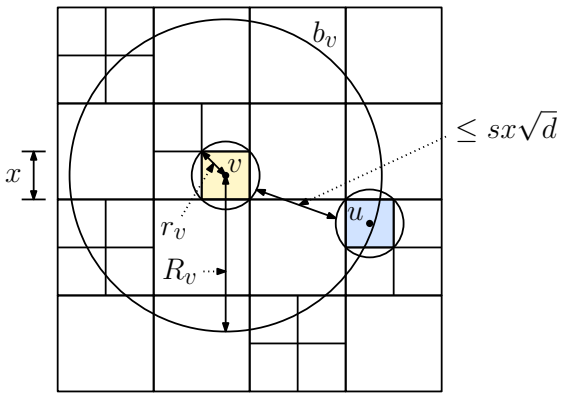


Fig. 101: WSPD analysis.

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.

Packing Lemma: To bound the number of charges assessed, we prove a lemma that 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.

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. 102(b)).

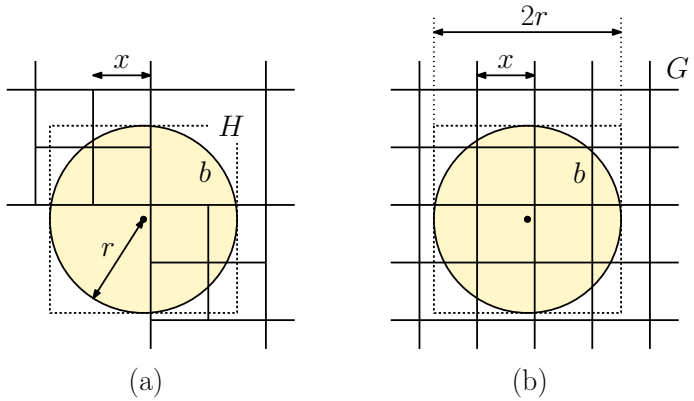


Fig. 102: Proof of the Packing Lemma.

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. 102). 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 $O((r/x)^d)$.

Putting it all together: Thus, by the Packing Lemma, the total number of nodes that can assess a charge to node v is at most C, where

$$\begin{aligned} C \;&\leq\; \left(1 + \left\lceil \frac{2R_v}{x} \right\rceil\right)^d + \left(1 + \left\lceil \frac{2R_v}{2x} \right\rceil\right)^d \;\leq\; 2\left(1 + \left\lceil \frac{2R_v}{x} \right\rceil\right)^d \\ &\leq\; 2\left(1 + \left\lceil \frac{6sx\sqrt{d}}{x} \right\rceil\right)^d \;\leq\; 2(2 + 6s\sqrt{d})^d. \end{aligned}$$

(In the last inequality, we used the fact that $\lceil z \rceil \leq 1 + z$.) Since the dimension d is assumed to be a constant and $s \geq 1$, this is $O(s^d)$.

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.

In summary, we have the following:

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 $\max(s, 1)$. In the next lecture we will consider applications of WSPDs to solving a number of geometric approximation problems.

Lecture 15: Applications of WSPDs

Review of WSPDs: 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 Euclidean balls of radius r such that the closest distance between these spheres is at least sr (see Fig. 103(a)). Given an n-element point set P in $\mathbb{R}^d$ 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 $1 \leq i \leq m$ (disjoint)
(3) $\bigcup_{i=1}^{m} A_i \otimes B_i = P \otimes P$ (cover all pairs)
(4) A_i and B_i are s-well separated, for $1 \leq i \leq m$, (well separated)

where $A \otimes B$ denotes the set of all unordered pairs from A and B. We refer to the pairs $\{A_i, B_i\}$ as *well-separated pairs* or *WSPs*.

In the previous lecture we showed that, given P and $s \geq 1$, there exists an s-WSPD of size $O(s^d n)$, which can be constructed in time $O(n \log n + s^d n)$. Because the construction concealed exponential factors depending on the dimension, we assumed that d is a constant, but we do not assume that s is a constant. (The WSPD definition allows for any $s > 0$, but we can simply apply this construction by with a modified separation factor $s' \leftarrow \max(s, 1)$.)

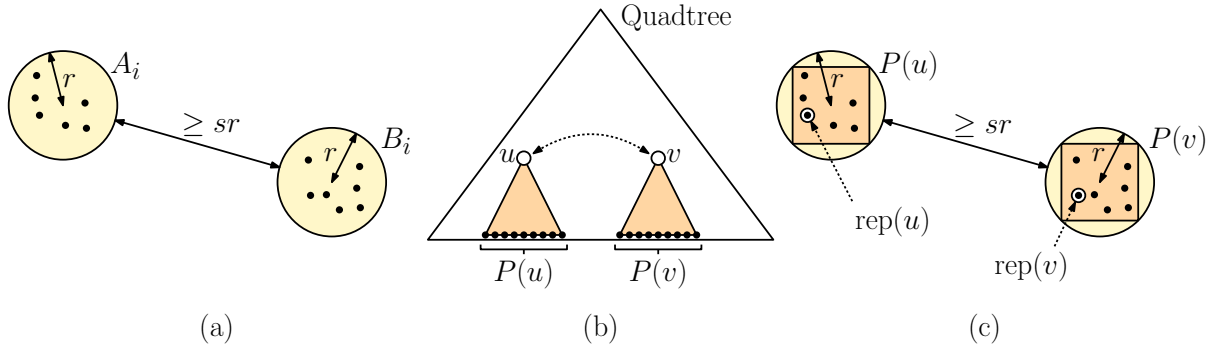


Fig. 103: WSPD and quadtree-based representation.

Quadtree-based Representation: Recall that our approach to constructing the WSPD was to first construct a *compressed quadtree* of size $O(n)$ storing the points of P in its leaves (see Fig. 103(b)). Each node u is implicitly associated with the set of points $P(u)$ stored within its

subtree, and the WSPD is represented as a set of unordered pairs of nodes the quadtree. The pair $\{u, v\}$ implicitly refers to the WSP $\{P(u), P(v)\}$. Thus, the m pairs of the WSPD can be represented in $O(m)$ space. Also recall that each (nonempty) node u of the compressed quadtree is assumed to store a *representative point*, denoted rep(u). This can be chosen arbitrarily from among u's descendants (or its choice may be based on the application we have in mind).

Utility Lemma: If two sets are s-well separated, we can infer some bounds on the relative distances between point. This is encapsulated in the following useful lemma (see Fig. 104). (When reading this, it is useful to imagine that s is a large number, so quantities like $2/s$ are small.)

Lemma: (WSPD Utility Lemma) If the pair $\{P(u), P(v)\}$ is s-well separated and $x, x' \in P(u)$ and $y, y' \in P(v)$ then:

(i) $\|x - x'\| \leq \frac{2}{s} \cdot \|x - y\|$ (Pairs on the same side are close compared to opposite sides)

(ii) $\|x' - y'\| \leq \left(1 + \frac{4}{s}\right) \|x - y\|$ (Pairs on opposite sides have similar distances)

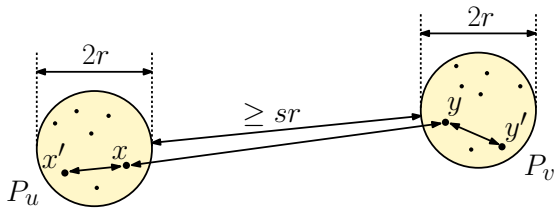


Fig. 104: WSPD Utility Lemma.

Proof: Since the pair is s-well separated, we can enclose each of $P(u)$ and $P(v)$ in a ball of radius r such that the minimum separation between these two balls is at least sr. It follows that $\max(\|x - x'\|, \|y - y'\|) \leq 2r$, and any pair from $\{x, x'\} \times \{y, y'\}$ is separated by a distance of at least sr. Thus, we have

$$\|x - x'\| \;\leq\; 2r \;=\; \frac{2r}{sr} sr \;\leq\; \frac{2r}{sr}\|x - y\| \;=\; \frac{2}{s}\|x - y\|,$$

which proves (i). Also, through an application of the triangle inequality ($\|a - c\| \leq \|a - b\| + \|b - c\|$) and the fact that $2r \leq \frac{2}{s}\|x - y\|$ we have

$$\begin{aligned} \|x' - y'\| &\leq \|x' - x\| + \|x - y\| + \|y - y'\| \;\leq\; 2r + \|x - y\| + 2r \\ &\leq \frac{2}{s}\|x - y\| + \|x - y\| + \frac{2}{s}\|x - y\| \;=\; \left(1 + \frac{4}{s}\right)\|x - y\|, \end{aligned}$$

which proves (ii).

Approximating the Diameter: The *diameter* of a point set P is defined to be the maximum distance between any pair of points of the set (see Fig. 105(a)). That is,

$$\text{diam}(P) \;=\; \max_{x,y \in P} \|x - y\|.$$

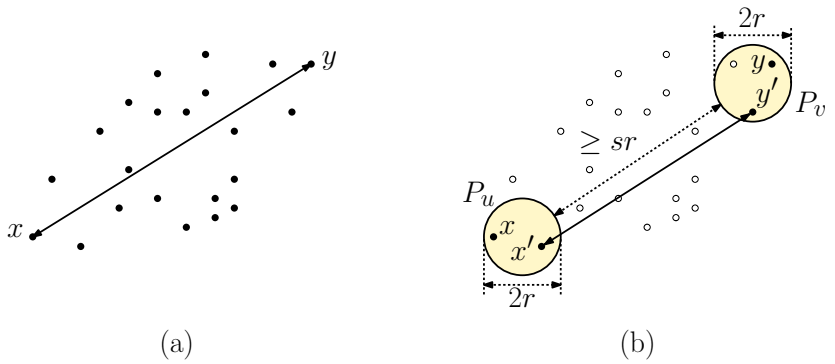


Fig. 105: Approximating the diameter.

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[24] in $O(n \log n)$ time. Unfortunately, generalizing this idea 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 ε-approximation to the diameter of a point set P in linear time. Given ε, 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. Let $x' = \text{rep}(u)$ and $y' = \text{rep}(v)$ denote the representative points associated with u and v, respectively. For every well separated pair $\{P(u), P(v)\}$, we compute the distance $\|x' - y'\|$ between their representative, and output the pair achieving 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 x' and y' denote their respective representatives. By the WSPD Utility Lemma we have

$$\|x - y\| \;\leq\; \left(1 + \frac{4}{s}\right)\|x' - y'\| \;=\; (1+\varepsilon)\|x' - y'\|.$$

Since $\{x, y\}$ is the diametrical pair, we have

$$\frac{\|x - y\|}{1+\varepsilon} \;\leq\; \|x' - y'\| \;\leq\; \|x - y\|,$$

which implies that the output pair $\{x', y'\}$ is an ε-approximation to the diameter.[25] The running time is dominated by the time to construct the WSPD, which is $O(n \log n + s^d n) = O(n \log n + n/\varepsilon^d)$. If we treat ε as a constant, this is $O(n \log n)$.

Closest Pair (Exact!): The same sort of approach we used for the diameter (farthest pair) could be applied to produce an ε-approximation to the closest pair as well. For example, we could first compute the s-WSPD where $s \approx 1/\varepsilon$ and compute the distances between all representatives and take the pair with the smallest value. However, we can in fact do much better and obtain an exact solution. The reason is given by the following lemma.

[24]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.

[25]You might wonder, why we didn't select the representatives more carefully, so that x' and y' are the diametrical pair for the WSPD. Remember that there is only one representative for each node u, and there are many WSPs involving u. So, no one representative will work for all WSPs.

Lemma: Let P be any point set in $\mathbb{R}^d$ and let x and y be the closest pair of points in P. For any separation factor $s > 2$ an s-WSPD for P contains a pair $\{\{x\}, \{y\}\}$. (That is, x and y show up as singleton sets in some pair.)

Assuming this lemma for now, this yields the following simple solution. First, take any $s > 2$ (e.g., $s = 2.000001$) and compute an s-WSPD for P. Then for each WSP $\{u, v\}$, compute $\|\text{rep}(u) - \text{rep}(v)\|$ and return the minimum of all of these. By the above lemma, there must exist a pair where $P(u) = \{x\}$ and $P(v) = \{y\}$, and therefore, the associated representatives must be the closest pair x and y. Therefore, the algorithm will correctly find the closest pair (exactly!). Since s is a constant, the running time of the WSPD construction is $O(n \log n + s^d n) = O(n \log n)$. This is essentially optimal (in the algebraic decision tree model of computing).

To prove the above lemma, let u and v be the WSP that separated x and y. Such a pair must exist by definition of WSPDs. We assert that $P(u) = \{x\}$ and $P(v) = \{y\}$ (that is, u and v are leaves containing x and y, respectively.) Suppose to the contrary that either $P(u)$ or $P(v)$ contains another point. Without loss of generality, suppose that $P(u)$ has another point x'.

By the utility lemma and the fact that $s > 2$, we have

$$\|x - x'\| \;\leq\; \frac{2}{s} \cdot \|x - y\| \;<\; \|x - y\|,$$

but this contradicts the hypothesis that x and y are the closest pair in P. Therefore, the lemma holds.

Low-Stretch Spanners: In an earlier lecture we defined the notion of a spanner graph. Recall that any 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. Intuitively, a spanner is a *sparse graph* (having only $O(n)$ edges) in which shortest paths are not significantly longer than the Euclidean distance between points. Such a graph is called a (Euclidean) *spanner*.

More formally, suppose that we are given a set P in $\mathbb{R}^d$ and a parameter $t \geq 1$, called the *stretch factor*. A *t-spanner* is any weighted graph G whose vertex set is P and, given any pair of points $x, y \in P$ we have

$$\|x - y\| \;\leq\; \delta_G(x, y) \;\leq\; t \cdot \|x - y\|,$$

where $\delta_G(x, y)$ denotes the length of the shortest path between x and y in G.

In an earlier lecture, we showed that the Delaunay triangulation of P is an $O(1)$-spanner. This was only really useful in the plane, since in dimension 3 and higher, the Delaunay triangulation can have a quadratic number of edges. Here we consider the question of how to produce a spanner in any space of constant dimension that achieves any desired stretch factor $t > 1$. There are many different ways of building spanners. Here we will discuss a straightforward method based on a WSPD of the point set.

WSPD-based Spanner Construction: Given the point set P and a (constant) stretch factor t, the idea is to build an s-WSPD for P, where s is an appropriately chosen separation factor

(which will depend on t). We will then create one edge in the spanner from each well-separated pair.

Given t, we set $s = 4(t+1)/(t-1)$. (Later we will justify the mysterious choice.) 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. 106). G will be the desired spanner. Clearly the number of edges of G is equal to the number of well-separated pairs, which is $O(s^d n) = O(n)$, and it can be built in the same $O(n \log n + s^d n) = O(n \log n)$ running time as the WSPD construction.

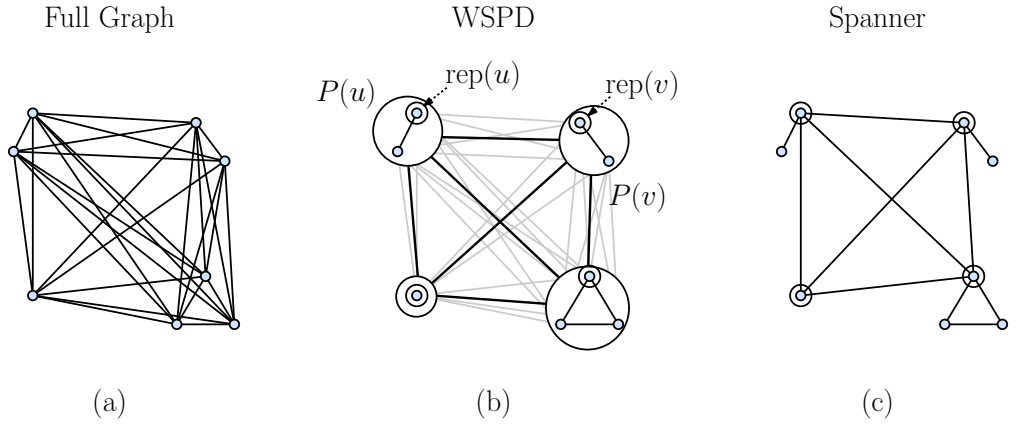


Fig. 106: WSPD-based spanner construction.

Correctness: To establish the correctness of our spanner construction algorithm, it suffices to show that for all pairs $x, y \in P$, we have

$$\|x - y\| \;\leq\; \delta_G(x, y) \;\leq\; t \cdot \|x - y\|.$$

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 $\delta_G(x, y) = \|x - y\| \leq t \cdot \|x - y\|$ for all $t \geq 1$.

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 $x' = \text{rep}(u)$ and $y' = \text{rep}(v)$ be the respective representative points. (It might be that $x' = x$ or $y' = y$, but not both.) Let us consider the length of the path from x to x' to y' to y. Since the edge $\{x', y'\}$ is in the graph, we have

$$\begin{aligned} \delta_G(x, y) &\leq \delta_G(x, x') + \delta_G(x', y') + \delta_G(y', y) \\ &\leq \delta_G(x, x') + \|x' - y'\| + \delta_G(y', y). \end{aligned}$$

(See Fig. 107.)

The paths from x to x' and y' to y are subpaths of the full spanner path from x to y, and hence they use fewer edges. Thus, we may apply the induction hypothesis, which yields

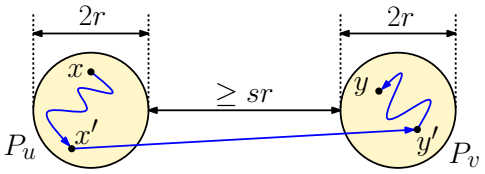


Fig. 107: Proof of the spanner bound.

$\delta_G(x, x') \leq t\|x - x'\|$ and $\delta_G(y', y) \leq t\|y' - y\|$, yielding

$$\delta_G(x, y) \;\leq\; t(\|x - x'\| + \|y' - y\|) + \|x' - y'\|. \tag{3}$$

By the WSPD Utility Lemma (with $\{x, x'\}$ from one pair and $\{y, y'\}$ from the other) we have

$$\max(\|x - x'\|, \|y' - y\|) \;\leq\; \frac{2}{s} \cdot \|x - y\| \qquad \text{and} \qquad \|x' - y'\| \;\leq\; \left(1 + \frac{4}{s}\right)\|x - y\|.$$

Combining these observations with Eq. (3) we obtain

$$\delta_G(x, y) \;\leq\; t\left(2 \cdot \frac{2}{s} \cdot \|x - y\|\right) + \left(1 + \frac{4}{s}\right)\|x - y\| \;=\; \left(1 + \frac{4(t+1)}{s}\right)\|x - y\|.$$

To complete the proof, observe that it suffices to select s so that $1 + 4(t+1)/s \leq t$. Towards this end, let us set

$$s \;=\; 4\left(\frac{t+1}{t-1}\right).$$

This is well defined for any $t > 1$. By substituting in this value of s, we have

$$\delta_G(x, y) \;\leq\; \left(1 + \frac{4(t+1)}{4(t+1)/(t-1)}\right)\|x - y\| \;=\; (1 + (t-1))\|x - y\| \;=\; t \cdot \|x - y\|,$$

which completes the correctness proof.

Because we have one spanner edge for each well-separated pair, 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: The Euclidean Minimum Spanning Tree (EMST) of a point set P is the minimum spanning tree of the complete Euclidean graph on P. In an earlier lecture, we showed that the EMST is a subgraph of the Delaunay triangulation of P. This provided an $O(n \log n)$ time algorithm in the plane. Unfortunately, the generalization to higher dimensions was not interesting because the worst-case number of edges in the Delaunay triangulation is quadratic in dimensions 3 and higher.

We will now that for any constant approximation factor ε, it is possible to compute an ε-approximation to the minimum spanning tree in any constant dimension d. Given a graph G with v vertices and e edges, it is well known that the MST of G can be computed in time $O(e + v \log v)$. It follows that we can compute the EMST 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 EMST, we first construct a $(1+\varepsilon)$-spanner, call it G, and then compute and return the MST of G (see Fig. 108). This approach has an overall running time of $O(n \log n + s^d n)$.

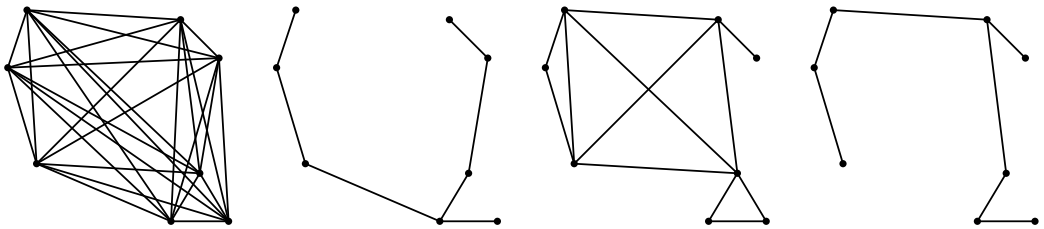


Fig. 108: Approximating the Euclidean MST.

Let EMST(P) denote P's EMST, and let emst(P) denote its total weight. Similarly, let MST(G) denote G's (exact) MST, and let mst(G) be its weight. We define a spanning subgraph of G, call it H, as follows. For each edge (x, y) in EMST(P), find the shortest path between x and y in G and add all these edges to H. Let $\delta_G(x, y)$ be the total weight of these edges, and let $w(H)$ denote the total weight of all of H's edges. By definition of the spanner, $\delta_G(x, y) \leq (1+\varepsilon)\|x - y\|$. (Note that some of these paths may share common edges, and so the total weight of H may be smaller than the sum of the weights of these paths). Since the edges of the EMST span all the points of P, union of paths from G also spans these points. Since the MST is the spanning structure of minimum weight, we have $\text{mst}(G) \leq w(H)$. Putting this all together, we obtain

$$\begin{aligned} \text{mst}(G) \;&\leq\; w(H) \;\leq\; \sum\nolimits_{(x,y)\in \text{EMST}(P)} \delta_G(x, y) \\ &=\; (1+\varepsilon) \sum\nolimits_{(x,y)\in \text{EMST}(P)} \|x - y\| \\ &=\; (1+\varepsilon) \cdot \text{emst}(P), \end{aligned}$$

as desired.

Lecture 16: Coresets and Kernels

Approximation by Sampling: 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? The simplest and most natural approach is to select a *random subset*. While this may work for many problems, there are some instances where it may fail miserably. For example, suppose that you wanted to approximate the minimum enclosing ball (MEB) for a point set P (see Fig. 109(a)). A random subset may result in a ball that is much smaller than the MEB. The problem is that the MEB is determined by points that are "peripheral", whereas random sampling may sample an excessive number of points near the "center" of the point set (see Fig. 109(b)). We would like a sampling method that is tailored to selecting points that will be most valuable to the MEB construction (see Fig. 109(c)).

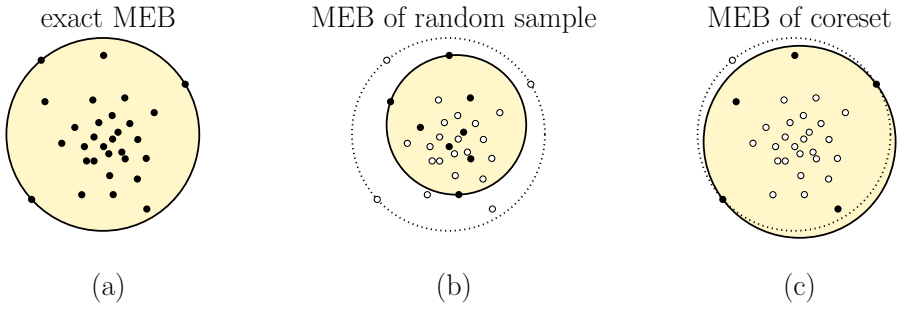


Fig. 109: (a) exact MEB, (b) MEB of a random sample, (c) MEB of a coreset.

Coresets: Abstractly, consider any optimization problem on point sets. For a point set P, let $f^*(P)$ denote the value of the objective function for the optimal solution. (For example, in the minimum enclosing ball problem, $f^*(P)$ is the radius of the MEB.) Given $\varepsilon > 0$, we say that subset $Q \subseteq P$ is an *ε-coreset* for this problem if, the relative error committed by solving the problem on Q is at most ε, 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 and it should be computable much faster than solving the original optimization problem. For many optimization problems, the coreset size is actually independent of n (but does depend on ε).

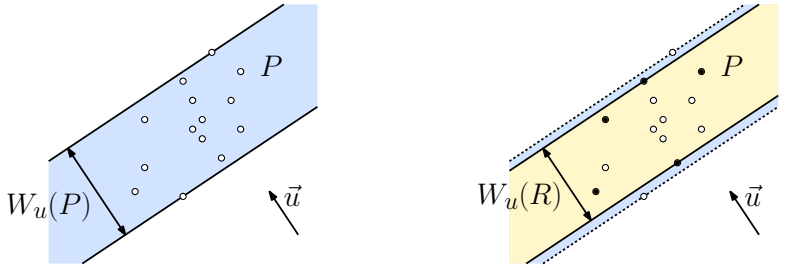
In this lecture, we will present algorithms for computing coresets for a problem called the *directional width*. This problem can be viewed as a way of approximating the convex hull of a point set.

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

$$W_u(P) \;=\; \max_{p \in P}(\vec{p} \cdot \vec{u}) - \min_{p \in P}(\vec{p} \cdot \vec{u})$$

(see Fig. 110(a)). Note that this is a signed quantity, but we are typically interested only in its magnitude.



(a) (b)

Fig. 110: Directional width and kernels. In (b) the points of R 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.

Suppose we want to answer *width queries*, where we are given a vector $\vec{u}$ and we want to efficiently compute the width in this direction. We want a solution that is substantially faster than the $O(n)$ time brute force solution. We saw earlier in the semester that if P is a planar point set, then by dualizing the point set into a set P^* of lines, the vertical distance between two parallel lines that enclose P is the same as the vertical distance between two points, one on the upper hull of P^* and one on the lower hull. This observation holds in any dimension. Given the vertical width for any slope, it is possible to apply simple trigonometry to obtain the orthogonal width. The problem, however, with this approach is that the complexity of the envelopes grows as $O(n^{\lfloor d/2 \rfloor})$. Thus, a solution based on this approach would be quite inefficient (either with regard to space or query time).

Given $0 < \varepsilon < 1$, we say that a subset $R \subseteq P$ is an *ε-coreset for directional width*, also called an *ε-kernel*, if for any unit vector $\vec{u}$,

$$(1 - \varepsilon)W_u(P) \leq W_u(R) \;\leq\; W_u(P).$$

Since $R \subseteq P$, the second inequality is trivially satisfied, so it is the first inequality that important. Intuitively, this says that for any unit vector $\vec{u}$, the directional width of R is smaller than that of P by a factor of only $(1 - \varepsilon)$ (see Fig. 110(b)). We will show that, given an n-element point set P in $\mathbb{R}^d$, it is possible to compute an ε-kernel whose size depends only on ε and d (not on n).

Note that kernels combine nicely. In particular, it is easy to prove the following:

Chain Property: If X is an ε-kernel of Y and Y is an ε'-kernel of Z, then X is an $(\varepsilon + \varepsilon')$ kernel of Z.

Union Property: If X is an ε-kernel of P and X' is an ε-kernel of P', then $X \cup X'$ is an ε-kernel of $P \cup P'$.

Canonical Position: Let's begin by considering a very simple, but not very efficient, ε-kernel. Before giving the algorithm, it will be useful to precondition the point set by making it "fat". Let's start with some definitions. Given $\alpha \leq 1$, we say that a convex body K in $\mathbb{R}^d$ is α-*fat* if there exist two positive scalars $\lambda_- \leq \lambda_+$, such that K is sandwiched between two concentric Euclidean balls of radii λ_- and λ_+ such that $\lambda_-/\lambda_+ = \alpha$ (see Fig. 111(a)).

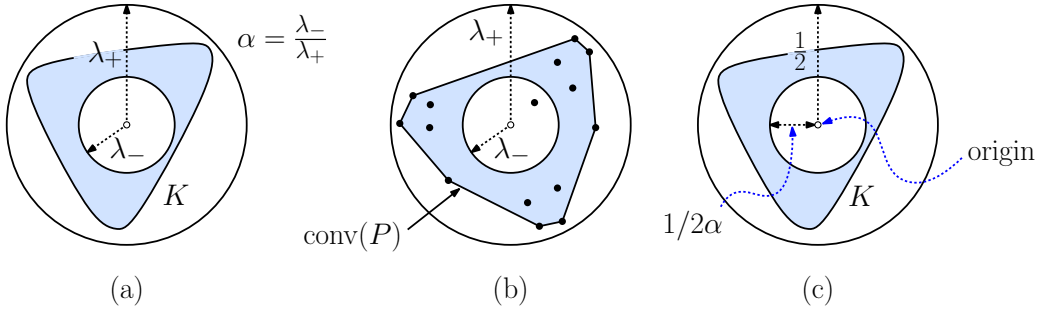


Fig. 111: The definition of α-fatness for: (a) a convex body K and (b) for a point set P.

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 α-fat if its convex hull, conv(P), is α-fat (see Fig. 111(b)).

Once we have fattened the body, it will also be nice to scale it to a convenient size and center it about the origin. We say that a convex body K is in α-*canonical position* if it is α-fat and the two sandwiching balls are of radii $\lambda_+ = \frac{1}{2}$ and $\lambda_- = \frac{1}{2\alpha}$, respectively. The reason for chosing the outer radius to be $\frac{1}{2}$ is so that body has diameter at most 1. The following lemma makes reference to an "affine transformation". Such a transformation consists of a linear transformation (as you learned in linear algebra) combined with an arbitrary translation.

Lemma 1: Given an n-element point set $P \subset \mathbb{R}^d$, there exists an affine transformation T such that $T(P)$ is in $\frac{1}{d}$-canonical position. Further, a subset $R \subseteq P$ is an ε-kernel for P if and only if $T(R)$ is an ε-kernel for $T(P)$. 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 (and remarkable) fact from the theory of convexity. *John's Theorem*, states that if E is a maximum volume ellipsoid contained within K (called K's *John ellipsoid*), then K is contained within dE, where dE denotes a concentrically scaled copy of E by a factor of d (see Fig. 112(a)).

There is an $O(n)$ time algorithm for computing the maximum volume ellipsoid of a convex body given as the intersection of k halfspaces in any fixed dimensional space (by Chazelle and Matoušek), and there is an efficient algorithm which computes a good enough approximation to the John ellipsoid (by Barequet and Har-Peled).

To obtain the transformation T, we first translate E so that its center coincides with the origin. It is a well-known fact from linear algebra that for any origin-centered ellipsoid E, there exists an affine transformation that maps it to a unit ball. (Intuitively, this involves scaling each of the principal axes of the ellipsoid to unit length.) To complete the transformation to canonical position, we apply an additional uniform scaling by $1/2d$. Take T to be the resulting affine transformation (see Fig. 112(b)).

In the scaling process, the inner ball has radius $\frac{1}{2d}$ and the outer ball has radius $\frac{1}{2}$, and so the resulting body is in $\frac{1}{d}$-canonical position, as desired.

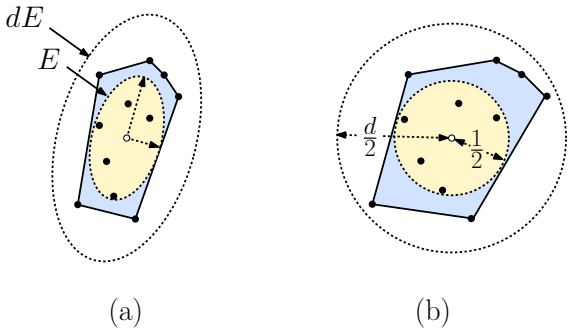


Fig. 112: The John ellipsoid and transformation to canonical position.

Generally, affine transformations do not preserve lengths, but the do preserve the ratio of lengths of vectors that the parallel to each other. The proof that ratios of directional widths are preserved will involve some technical details of linear algebra, which are beyond the scope of this course.

Quick and Dirty Kernel: Armed with the above lemma, we can compute an ε-kernel by the following "quick-and-dirty" approach. First, we assume that P has been mapped to $\frac{1}{d}$-canonical position by the above lemma. Since P is sandwiched between two balls of diameters 1 and $\frac{1}{d}$, it follows that the width of P along any direction is at most 1 and at least $\frac{1}{d}$. In order to achieve a relative error of ε, it suffices to approximate any width to an *absolute error* of at most $\frac{\varepsilon}{d}$, which we will denote by ε'.

Let $H = \left[-\frac{1}{2}, +\frac{1}{2}\right]^d$ be a hypercube of unit side length centered at the origin. Clearly, $P \subseteq H$. Subdivide H into a uniform square grid into cells of side length at most $\varepsilon'/2\sqrt{d}$, or equivalently of diameter at most $\varepsilon'/2$ (see Fig. 113(a)).

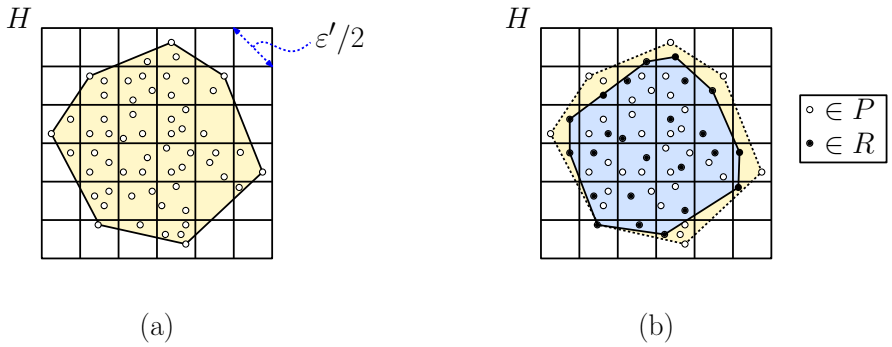


Fig. 113: (a) The grid and (b) the quick-and-dirty kernel construction.

Each side of H is subdivided into $O(1/\varepsilon') = O(1/\varepsilon)$ intervals. Thus, the total number of cells in the grid is $O(1/\varepsilon^d)$. Next, we determine which cell of the grid each point of P belongs to. (This involves a straightforward application of integer division.) Once we know the indices of the grid square that contains a point, we can apply hashing to bucket all the points that lie within the same grid square. Since hashing takes $O(1)$ time in expectation, we can hash all the points into their grid squares in $O(n)$ expected time.

To obtain the final kernel, we select one *representative point* from each nonempty grid cell. (These are indicated using black dots in Fig. 113(b).) We output the resulting set R as the

ε-kernel. Since we have at most one representative per grid square, we have $|R| = O(1/\varepsilon^d)$. The entire contstruction takes $O(n)$ time.

To establish the correctness consider any direction $\vec{u}$, and let $p_1, p_2 \in P$ be the two points of P that determine $W_u(P)$. Let $q_1, q_2 \in R$ be the corresponding representative points from these respective cells. (It may be that $q_1 = p_1$ and/or $q_2 = p_2$.) Since each cell is of diameter at most $\varepsilon'/2$, the directional width of the pair $\{q_1, q_2\}$ is smaller than the directional width of p_1 and p_2 by at most $\varepsilon'/2 + \varepsilon'/2 = \varepsilon'$. Since q_1 and q_2 are arbitrary elements of R, $W_u(R)$ is at least as large as the directional width of these two points. Because P is in canonical position, we know that $W_u(P) \geq \frac{1}{d}$. Putting this together, we have

$$\begin{aligned} W_u(P) &= W_u(\{p_1, p_2\}) \\ &\leq \varepsilon' + W_u(\{q_1, q_2\}) \leq \varepsilon' + W_u(R) = \frac{\varepsilon}{d} + W_u(R) \\ &\leq \varepsilon W_u(P) + w_u(R), \end{aligned}$$

which implies that $(1-\varepsilon)W_u(P) \leq w_u(R)$. Since $R \subseteq P$, we have $W_u(R) \leq w_u(P)$. Therefore, for any unit vector u, we have

$$(1-\varepsilon)W_u(P) \leq w_u(R) \leq w_u(P),$$

which implies that R is an ε-kernel of size $O(1/\varepsilon^d)$.

Improved Construction: It is possible make a small improvement in the size of the "quick-and-dirty" kernel. Observe from Fig. 113(b) that we may select many points from the interior of $\text{conv}(P)$. Clearly, many of these play no useful role for the purposes of a kernel. The fix is that, rather than partition H into small hypercubes, we can instead partition it into thin columns of unit height. In particular, we create a square grid on just the upper $(d-1)$-dimensional facet of H into hypercubes of diameter at most $\varepsilon'/2$ (where ε' is the same as the previous construction). For each grid, we "extrude" it vertically into a square column of height 1. Finally, we select two representatives from each column, the topmost and bottommost points (see Fig. 114(a)). We output these representatives as the ε-kernel.

Since the top facet is of dimension $d-1$, the total number of columns in our construction is $O(1/\varepsilon^{d-1})$, and hence the total number of representatives R is also $O(1/\varepsilon^{d-1})$. (This is better than the previous construction by a factor of $1/\varepsilon$.)

To show correctness, as above we can show that for any unit vector $\vec{u}$, if p is the extreme point of P along this direction, then we assert that there exists a representative $q \in R$ from p's column that is within distance $\varepsilon'/2$ with respect to $\vec{u}$ (see Fig. 114(b)). We will not give a formal proof, but this follows because the diameter of the column is at most $\varepsilon'/2$. It is a relatively simple geometric exercise to prove that the highest (or lowest) point of the column has a directional distance with respect to u that is within $\varepsilon'/2$ from the true extreme point. In summary, we have the following.

Theorem: Given a set P of points in $\mathbb{R}^d$ and $\varepsilon > 0$, in $O(n)$ time it is possible to construct an ε-kernel of size $O(1/\varepsilon^{d-1})$.

Can we do even better than $O(1/\varepsilon^{d-1})$? In the next construction, we will show that we can cut the exponent in half with a more clever construction.

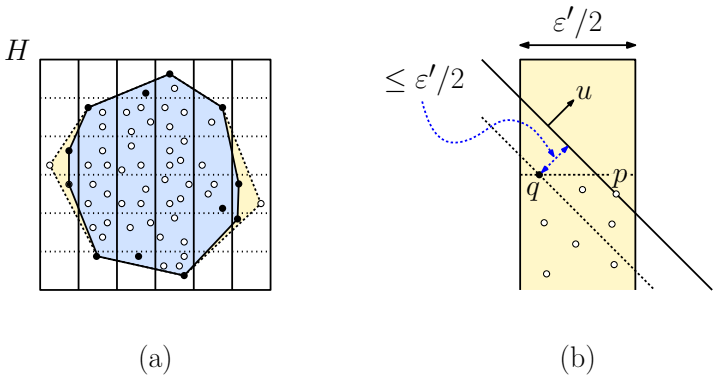


Fig. 114: Correctness of the improved construction.

Optimal Kernel Construction: The above kernel construction has the advantage of simplicity, but, it is possible to do much better, reducing the kernel size from $O(1/\varepsilon^{d-1})$ to $O(1/\varepsilon^{(d-1)/2})$. It can be shown that this is asymptotically optimal (as a function of ε and d, assuming d is a constant).

Our general approach will be similar to the one taken above. First, we map P into canonical position, so it is sandwiched between balls of diameter 1 and $\frac{1}{d}$. After this, the width of P in any direction is between $\frac{1}{d}$ and 1. Thus, in order to achieve a relative error of ε, it suffices to compute a kernel whose absolute difference in width along any direction is at most $\varepsilon' = \frac{\varepsilon}{d}$.

A natural approach to solving this problem would involve uniformly sampling a large number (depending on ε) 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 R. 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 ε, 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 $\delta = \sqrt{\varepsilon/4d}$. (The source of this "magic number" will become apparent later.) On the sphere S, construct a δ-dense set of points, denoted Q (see Fig. 115). This means that, for every point on S, there is a point of Q within distance δ. 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.[26] Let R denote the resulting subset of P. We will show that R is the desired kernel.

In the figure we have connected each point of Q to its closest point on $\text{conv}(P)$. It is a bit easier to conceptualize the construction as sampling points from $\text{conv}(P)$. (Recall that the kernel definition requires that the kernel 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

[26] 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.

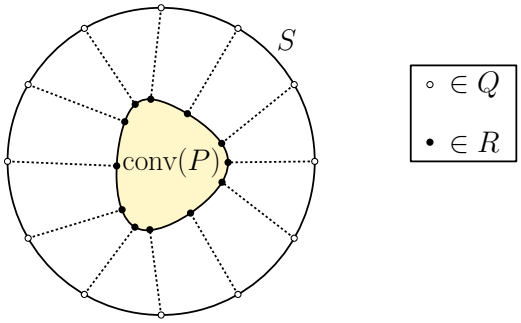


Fig. 115: Smarter kernel construction. (Technically, the points of Q are connected to the closest point of P, not conv(P).)

points of P that lie close to regions where the curvature of P's convex hull is higher (see Fig. 115). This is useful, because areas of high curvature need more points to approximate them well. Also, because the points on S are chosen to be δ-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. 116). 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 $-\delta^2$.

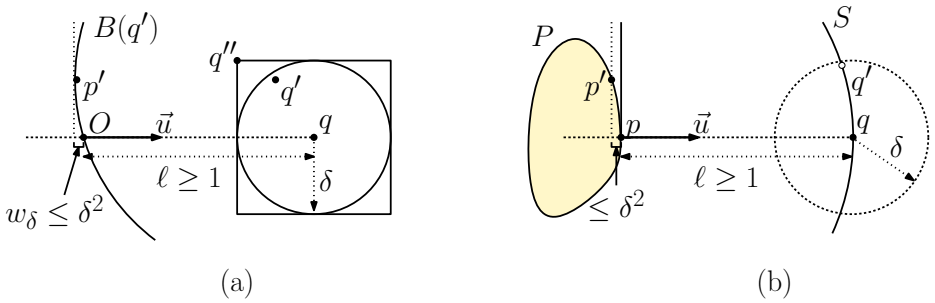


Fig. 116: Analysis of the kernel construction.

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 2δ centered at q. This suffices because this square contains all points that lie within distance δ 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. 116(a)). Call this corner point q''.

The distance between q'' and the origin is $\sqrt{(\ell-\delta)^2+\delta^2}$, which we denote by ℓ''. This is the same as the horizontal distance from q'' to the leftmost point of $B(q'')$. The horizontal distance between q'' and the origin is $\ell-\delta$. Therefore, the amount by which the ball of radius $\|q''\|$ centered at $\|q''\|$ may protrude over the $(-x)$-axis is at most w_q, which we define to be

$$w_\delta \;=\; \ell'' - (\ell-\delta) \;=\; \sqrt{(\ell-\delta)^2+\delta^2} - (\ell-\delta)$$

Since p lies in this ball, to complete the proof it suffices to show that $w_\delta \le \delta^2$.

To simplify this, let us multiply it by a fraction whose numerator and denominator are both $\sqrt{(\ell-\delta)^2+\delta^2}+(\ell-\delta)$. Clearly, $\sqrt{(\ell-\delta)^2+\delta^2} \ge \ell-\delta$. Also, since $\ell \ge 1$ and $\delta \le \frac{1}{2}$, we have $\ell-\delta \ge \frac{1}{2}$. Therefore,

$$\begin{aligned} w_\delta \;&=\; \frac{((\ell-\delta)^2+\delta^2)-(\ell-\delta)^2}{\sqrt{(\ell-\delta)^2+\delta^2}+(\ell-\delta)} \;=\; \frac{\delta^2}{\sqrt{(\ell-\delta)^2+\delta^2}+(\ell-\delta)} \\ &\le\; \frac{\delta^2}{(\ell-\delta)+(\ell-\delta)} \;\le\; \frac{\delta^2}{(1/2)+(1/2)} \;=\; \delta^2, \end{aligned}$$

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 $p' \in R$ such that $(p\cdot\vec{u}) - (p'\cdot\vec{u}) \le \varepsilon'/2$. 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. 116(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 δ of q, that is $\|q'-q\| \le \delta$. Let p' be the nearest neighbor of P to q'. Again, by our construction p' is in the kernel. Since q lies on a sphere of radius 2 and P is contained within the unit ball, it follows that $\|q\| \ge 1$. Thus, we satisfy the conditions of Lemma 2. Therefore,

$$(p'\cdot\vec{u}) \;\ge\; -\delta^2 \;=\; -\frac{\varepsilon}{4d} \;\ge\; -\frac{\varepsilon'}{2}.$$

Thus, the absolute error in the directional distance to p' compared to p is at most $\varepsilon'/2$. By combining the errors from both sides, the total absolute error is at most ε'. By the remarks made earlier (see also the analysis of the "quick and dirty" construction), this implies that the total relative error is ε, as desired.

Theorem: Given a set P of points in $\mathbb{R}^d$ and $\varepsilon > 0$, it is possible to construct an ε-kernel of size $O(1/\varepsilon^{(d-1)/2})$.

Lecture 17: Geometric Sampling, VC-Dimension, and Applications

Geometric Set Systems: Many problems in computational geometry involve an interaction between points and subsets of points defined by geometric objects. For example, suppose that a point set P represents a set of n locations on campus where students tend to congregate (see Fig. 117(a)). An internet wireless service provider wants to place a set of towers around the campus equipt with wireless routers to provide high-capacity data service to these locations. Due to power considerations, each wireless user needs to be within a certain distance δ of one of these towers in order to benefit from the special service. The service provider would like to

determine the smallest number of locations such that each of the congregation points is within distance δ of one of these towers (see Fig. 117(b)). This is equivalent to a set-cover problem, where we want to cover a set of n points with set of circular disks of radius δ. In general, set cover is a hard problem, but the constraint of having geometric sets can help ameliorate the situation. We begin with a discussion of the concept of geometric range spaces.

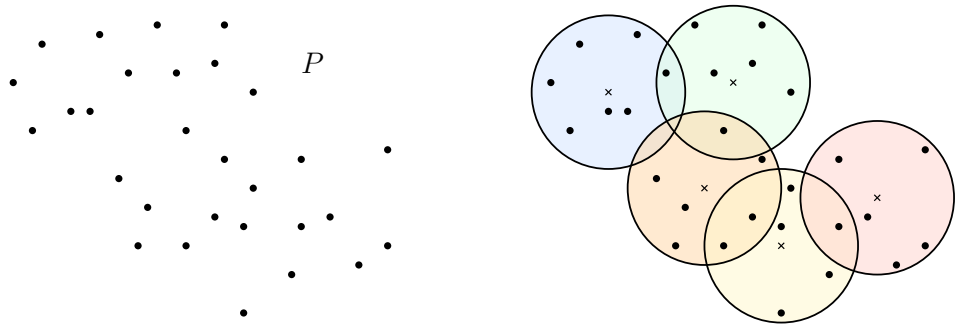


Fig. 117: Set cover by circular disks.

Range Spaces: Given a set P of n points in $\mathbb{R}^d$, its *power set*, denoted 2^P, is the set of all subsets of P, including P and the empty set. The power set has 2^n elements. If we constrain ourselves to subsets formed by some geometric property (e.g., the subsets of P lying within a circular disk, a halfplane, or a rectangle), this severely limits the types of subsets can can be formed.

We can characterize such geometric set systems abstractly as follows. A *range space* is defined to be a pair $(X, \mathcal{R})$, where X is an arbitrary set (which might be finite or infinite) and $\mathcal{R}$ is a subset of the power set of X. We will usually apply range spaces to finite point sets. Given a set $P \subseteq X$, define the *restriction* (sometimes called the *projection*) of $\mathcal{R}$ to P as

$$\mathcal{R}_{|P} \;=\; \{P \cap Q \mid Q \in \mathcal{R}\}.$$

For example, if $X = \mathbb{R}^d$, P is a set of n points in $\mathbb{R}^d$, and $\mathcal{R}$ consists of the subsets of real space contained within axis-parallel rectangles, then $\mathcal{R}_{|P}$ consists of the subsets of P contained within axis-parallel rectangles (see Fig. 118). Note that not all subsets of P may be in $\mathcal{R}_{|P}$. For example, the sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be formed by intersecting P with axis-parallel rectangular ranges.

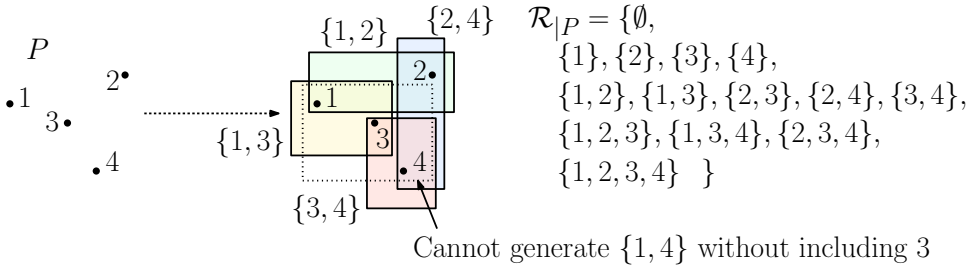


Fig. 118: A 4-point set and the range space of axis-parallel rectangles. Note that sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be generated.

Measures, Samples, and Nets: When dealing with range spaces over very large point sets, it is often desirable to approximate the set with a much smaller sample of the set that does a good job of representing the set. What does it mean for a sample to be "good"? (We have already seen one example in our coverage of coresets.) The concept of a range space provides one way of making this precise.

Given a range space $(P, \mathcal{R})$, where P is finite, and given a range $Q \in \mathcal{R}$, we define Q's *measure* to be the fraction of points of P that it contains, that is

$$\mu(Q) \;=\; \frac{|Q \cap P|}{|P|}.$$

Given a subset $S \subseteq P$ (which we want to think of as being our sample, so that $|S| \ll |P|$) it provides an *estimate* on the measure of a range. Define

$$\widehat{\mu}_S(Q) \;=\; \frac{|Q \cap S|}{|S|}.$$

(When S is clear, we will omit it from the subscript and just write $\widehat{\mu}(Q)$.) A set S is a good sample of P if the estimate is close to the actual measure. That is, we would like to select S so that for all $Q \in \mathcal{R}$, $\widehat{\mu}(Q) \approx \mu(Q)$.

There are two common ways of characterizing good sample sets: ε-samples and ε-nets.

ε-sample: Given a range space $(P, \mathcal{R})$ and any $\varepsilon > 0$, a subset $S \subseteq P$ is an *ε-sample* if for any range $Q \in \mathcal{R}$ we have

$$|\mu(Q) - \widehat{\mu}(Q)| \;\leq\; \varepsilon.$$

For example, suppose that $\varepsilon = 0.1$ and Q encloses 60% of the points of P ($\mu(Q) = 0.6$) then Q should enclose a fraction of 60±10% (50–70%) of the points of S (see Fig. 119(b)). If this is true for every possible choice of Q, then S is a 0.1-sample for P.

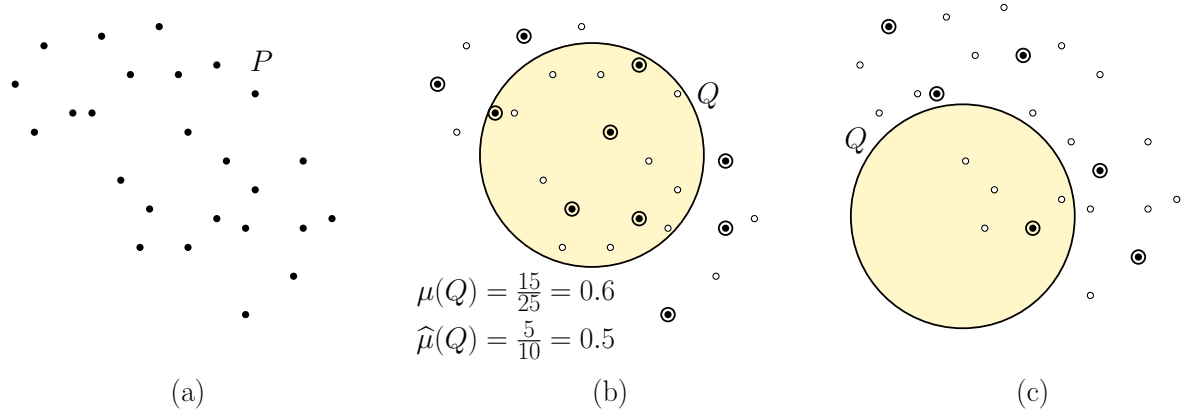


Fig. 119: ε-samples and ε-nets.

ε-net: While ε-samples intuitively correspond a desirable standard for good samples, it is often the case that we are can be satisfied with something weaker, where we just require the sample to provide us with a single point. Given a range space $(P, \mathcal{R})$ and any $\varepsilon > 0$, a subset $S \subseteq P$ is an *ε-net* if for any range $Q \in \mathcal{R}$, if $\mu(Q) \geq \varepsilon$ then Q contains at least one point of S. For example, if $\varepsilon = 0.2$ and $|P| = 25$, then any range Q that contains at least $0.2 \cdot 25 = 5$ points of P must contain at least one point of the ε-net (see Fig. 119(c)).

Observe that if S is an ε-sample, then it is surely an ε-net. The reason that ε-nets are of interest is that they are usually much smaller than ε-samples, and so it is more economical to use ε-nets whenever they are applicable.

VC Dimension: In looking at the definitions, it is clear that any sufficiently large random sample is likely to be an ε-sample (or ε-net) for an arbitrary range Q. But we want it to be the case that the sampling/netting condition applies to *all* ranges. Generally, this cannot be achieved unless we impose additional constraints on the range space, but these constraints are usually satisfied by the sorts of ranges that arise in geometric contexts.

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 $\mathcal{R}_{|P}$ 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 generally determined by at most four points of P, that is, the points that lie on the rectangle's top, bottom, left, and right sides (see Fig. 120(b)). (It might be fewer if a point lies in the corner of the range.) Since there are essentially at most $\binom{n}{4} = O(n^4)$ distinct ranges, we have $|\mathcal{R}_{|P}| = O(n^4)$.

How can we generalize this to arbitrary range spaces? (Including those where ranges are not defined by geometric shapes.) Remarkably, the definition makes no mention of geometry at all! This is the notion of *VC-dimension*, which is short for *Vapnik-Chervonenkis dimension*.[27]

Given an arbitrary range space $(X, \mathcal{R})$ and finite point set P, we say that $\mathcal{R}$ *shatters* P if $\mathcal{R}_{|P}$ 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. 118 is *not* shattered by the range space of axis-parallel rectangles. However, the four-element point set P shown in Fig. 120 is shattered by this range space, because we can form all $2^4 = 16$ subsets of this set.

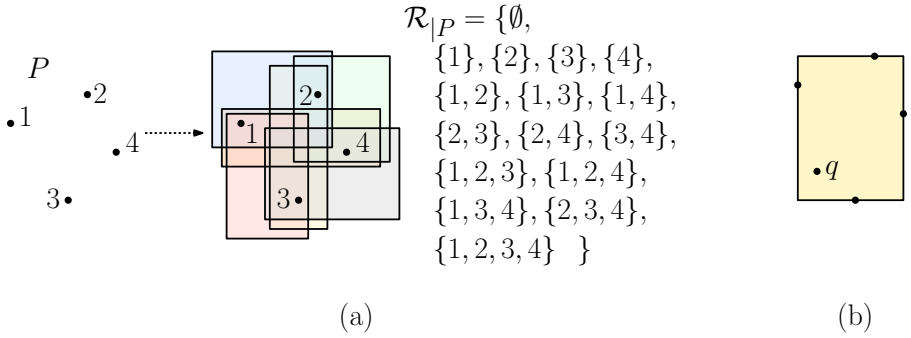


Fig. 120: (a) a 4-element point set that is shattered by the range space of axis-parallel rectangles (showing only the 2-element subsets in the drawing), and (b) the proof that no 5-element point set is shattered.

Definition: 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.

We will denote the VC-dimension as $\dim_{\text{VC}}(X, \mathcal{R})$, or simply $\dim_{\text{VC}}(\mathcal{R})$ when X is clear. Here are a couple of examples:

[27] 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.

Axis-parallel rectangles: Axis-parallel rectangles have VC-dimension four. In Fig. 120(a) we gave a 4-element point set that can be shattered. We assert that no five points can be shattered. Consider any set P of five points in the plane, and assume the points are in general position. Because of general position, at least one of the points of P, call it q, does not lie on the boundary of P's smallest enclosing axis-parallel rectangle (see Fig. 120(b)). It is easy to see that it is not possible to form the subset $P \setminus \{q\}$, since any axis-parallel rectangle containing the points that define the minimum bounding rectangle must contain all the points of P.

Euclidean disks in the plane: Planar Euclidean disks have VC-dimension three. A 3-element point set that is shattered is shown Fig. 121(a).

To see that no four-element set can be shattered, consider any set of four points $\{a, b, c, d\}$ in general position (see Fig. 121(b)). If any point lies in the convex hull of the other three, then clearly it is not possible to form the subset that excludes this one point and contains all the others. Otherwise, all the points are on the convex hull. Consider their Delaunay triangulation. Let a and b denote the two points of the group that are *not* connected by an edge of the triangulation (see Fig. 121(b)). Because $\overline{ab}$ is *not* and edge of the Delaunay triangulation, by the empty-circle property, any circle that contains a and b, must contain at least one other point of the set. Therefore, the subset $\{a, b\}$ cannot be generated.

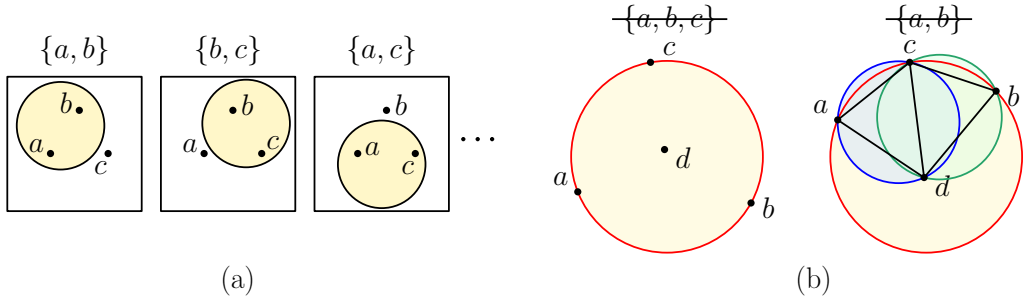


Fig. 121: (a) a 3-element point set that is shattered by the range space of Euclidean disks (showing just the 2-element subsets), and (b) the proof that no 4-element point set is shattered.

Sauer's Lemma: We have seen (1) that the range space of axis-parallel rectangles over an n element point set contains $O(n^4)$ ranges and (2) that such a range space has VC-dimension four. This raises the interesting conjecture that the size of *any* range space is related to its VC-dimension. Indeed, this is the case, and it is proved by a useful result called *Sauer's Lemma* (also called the Sauer-Shelah Lemma).

Before giving this lemma, let us first define a useful function. Given $0 \leq d \leq n$, define $\Phi_d(n)$ to be the number of subsets of size at most d over a ground set of size n, that is,

$$\Phi_d(n) \;=\; \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{d} \;=\; \sum_{i=0}^{d} \binom{n}{i}.$$

An important fact about this function is that it satisfies the following recurrence

$$\Phi_d(n) \;=\; \Phi_d(n-1) + \Phi_{d-1}(n-1).$$

An intuitive way to justify the recurrence is to fix one element x_0 of the n-element set. The number of sets of size at most d that do not contain x_0 is $\Phi_d(n-1)$ (since the element itself is not available from the n elements) and the number of sets that do contain x_0 is $\Phi_{d-1}(n-1)$ (since once x_0 is removed from each of these sets, we have $d-1$ remaining elements to pick from).

Sauer's Lemma: If $(X, \mathcal{R})$ is a range space with VC-dimension d and $|X| = n$, then $|\mathcal{R}| \leq \Phi_d(n)$.

Proof: The proof is by induction on d and n. It is trivially true if $d = 0$ or $n = 0$. (Note that $\Phi_0(n) = \Phi_d(0) = 1$, and the range space consists of just the empty set.) Assuming $n \geq 1$, fix any one element $x \in X$. Consider the following two range sets:

$$\begin{aligned} \mathcal{R}_x &= \{Q \setminus \{x\} \ : \ Q \cup \{x\} \in \mathcal{R} \text{ and } Q \setminus \{x\} \in \mathcal{R}\} \\ \mathcal{R} \setminus \{x\} &= \{Q \setminus \{x\} \ : \ Q \in \mathcal{R}\} \end{aligned}$$

Intuitively, $\mathcal{R}_x$ is formed from pairs of ranges from $\mathcal{R}$ that are identical except that one contains x and the other does not. (For example, if x is along the side of some axis-parallel rectangle, then there is a range that includes x and a slightly smaller one that does not. We put the range that does not contain x into $\mathcal{R}_x$.) The set $\mathcal{R} \setminus \{x\}$ is the result of throwing x entirely out of the point set and considering the remaining ranges.

We assert that $|\mathcal{R}| = |\mathcal{R}_x| + |\mathcal{R} \setminus \{x\}|$. To see why, suppose that we charge each range of $\mathcal{R}$ to its corresponding range in $\mathcal{R} \setminus \{x\}$. Every range of $\mathcal{R} \setminus \{x\}$ receives at least one charge, but it receives two charges if there exist two ranges that are identical except that one contains x and one doesn't. The elements of $\mathcal{R}_x$ account for these extra charges.

Now, let us apply induction. Observe that the range space $(X \setminus \{x\}, \mathcal{R}_x)$ has VC-dimension $d-1$. In particular, we claim that no set P' of size d can be shattered. To see why, suppose that we were to throw x back into the mix. The pairs of sets of $\mathcal{R}$ that gave rise to the ranges of $\mathcal{R}_x$ would then shatter the $d+1$ element set $P' \cup \{x\}$. (This is the critical step of the proof, so you should take a little time to convince yourself of it!) Clearly, the VC-dimension of $\mathcal{R} \setminus \{x\}$ cannot be larger than the original, so its VC-dimension is at most d. Since both range sets involve one fewer element $(n-1)$, by applying the induction hypothesis and our earlier recurrence for $\Phi_d(n)$, we have

$$|\mathcal{R}| \ = \ |\mathcal{R}_x| + |\mathcal{R} \setminus \{x\}| \ \leq \ \Phi_{d-1}(n-1) + \Phi_d(n-1) \ = \ \Phi_d(n).$$

And this completes the proof.

Clearly, $\Phi_d(n) = \Theta(n^d)$, so Sauer's Lemma implies that an range space of VC-dimension d over a point set of size n contains at most $O(n^d)$ ranges. It can be shown that this bound is tight.

On the Sizes of ε-nets and ε-samples: One of the important features of range spaces of low VC-dimension is that there exist good samples of small size. Intuitively, by restricting ourselves to simple geometric ranges, we do not have the power to construct arbitrarily complicated sets. Observe that if sets of arbitrary complexity are allowed, then it would be hopeless to try to construct ε-samples or ε-nets, because given any sample, we could find some nasty range Q that manages to exclude every point of the sample and include all the remaining points of P (see Fig. 122).

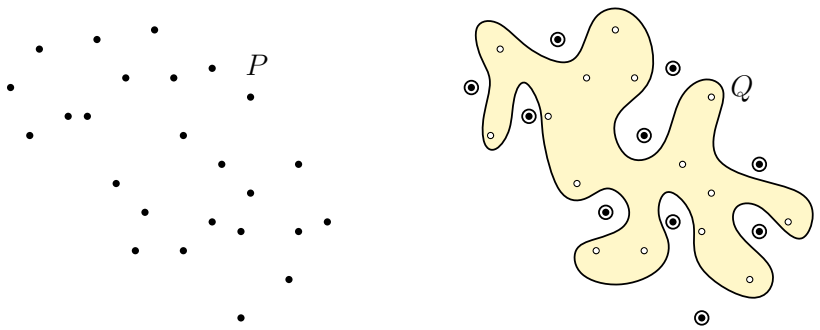


Fig. 122: Why VC-dimension matters.

If a range space has VC-dimension d, we will show that there exist ε-samples and ε-nets whose sizes depend on ε and d alone, independent of the original point set n! This is very important in geometric approximation algorithms, because it allows us to extract a tiny set from a huge one, with the knowledge that the tiny set is guaranteed to do a good job of representing the huge one.

Theorem: (ε-Sample Theorem) Let $(X, \mathcal{R})$ be a range space of VC-dimension d, and let P be any finite subset of X. There exists a positive constant c (independent of the range space) such that with probability at least $1 - \varphi$ any random sample S of P of size at least

$$\frac{c}{\varepsilon^2}\left(d \log \frac{d}{\varepsilon} + \log \frac{1}{\varphi}\right)$$

is an ε-sample for $(P, \mathcal{R})$. Assuming that d and φ are constants, this is $O((1/\varepsilon^2)\log(1/\varepsilon))$.

Theorem: (ε-Net Theorem) Let $(X, \mathcal{R})$ be a range space of VC-dimension d, and let P be any finite subset of X. There exists a positive constant c (independent of the range space) such that with probability at least $1 - \varphi$ any random sample S of P of size at least

$$\frac{c}{\varepsilon}\left(d \log \frac{1}{\varepsilon} + \log \frac{1}{\varphi}\right)$$

is an ε-sample for $(P, \mathcal{R})$. Assuming that d and φ are constants, this is $O((1/\varepsilon)\log(1/\varepsilon))$.

We will not prove these theorems. Both involve fairly standard applications of techniques from probability theory (particularly the Chernoff bounds), but there are quite a few non-trivial technical details involved.

Application — Geometric Set Cover: Nets and samples have applications in many areas of computational geometry. We will discuss one such application involving geometric set cover. Given an n-element ground set X and a collection of subsets $\mathcal{R}$ over X, the *set cover problem* problem is that of computing a subset of $\mathcal{R}$ of minimum size whose union contains all the elements of X. It is well known that this problem is NP-hard, and assuming that P $\neq$ NP, it is hard to approximate to within a factor of $\Omega(\log n)$.

There is a well-known greedy approximation algorithm for set cover that achieves an approximation ratio of $\ln n$. This algorithm repeatedly selects the set of $\mathcal{R}$ that contains the largest number of elements of X that have not yet been covered. This algorithm can be

applied to arbitrary set systems, but we will show that if the range space $(X, \mathcal{R})$ has constant VC-dimension then there exists an approximation algorithm that achieves an approximation ratio of $O(\log k^*)$, where k^* is the number of sets in the optimal solution. If $k^* \ll n$, then this algorithm provides a significant theoretical improvement over the greedy algorithm. (In practice, the greedy heuristic is very good.)

For the sake of simplicity, we will present this algorithm in a slightly simpler context, but it readily generalizes to any range space of constant VC-dimension. Rather than solve the set cover problem, we will present an approximation for the dual problem, called *hitting set*. Here, we are given a set system $(X, \mathcal{R})$, and the problem is to compute a subset of $H \subseteq X$ (as small as possible) so that every set Q in $\mathcal{R}$ contains at least one element (that is, it is "hit") by H.

The set cover and hitting set problems are actually dual equivalents. Given a set system $(X, \mathcal{R})$ for the set cover problem (see Fig. 123(a)), we can form an equivalent instance of hitting set as follows. We create a new set system $(X', \mathcal{R}')$, where each element of X' corresponds to a set in $\mathcal{R}$, and each set in $\mathcal{R}'$ corresponds to an element in X. For each $x \in X$, let $R_{i_1}, R_{i_2}, \ldots, R_{i_j}$ denote the sets of $\mathcal{R}$ that contain x. We create the set $\{i_1, \ldots, i_j\}$ and add it to $\mathcal{R}'$. (See Fig. 123(c), where the point x is mapped to the disk x^* and ranges R_1, R_2, R_3 are mapped to points R_1^*, R_2^*, R_3^*. Observe that $x \in R_i$ and $R_i^* \in x^*$.) It is easy to see that any set cover for $(X, \mathcal{R})$ (see Fig. 123(b)) corresponds to an equivalent hitting set for $(X', \mathcal{R}')$ (see Fig. 123(d)). (We'll leave the formal proof as an exercise.)

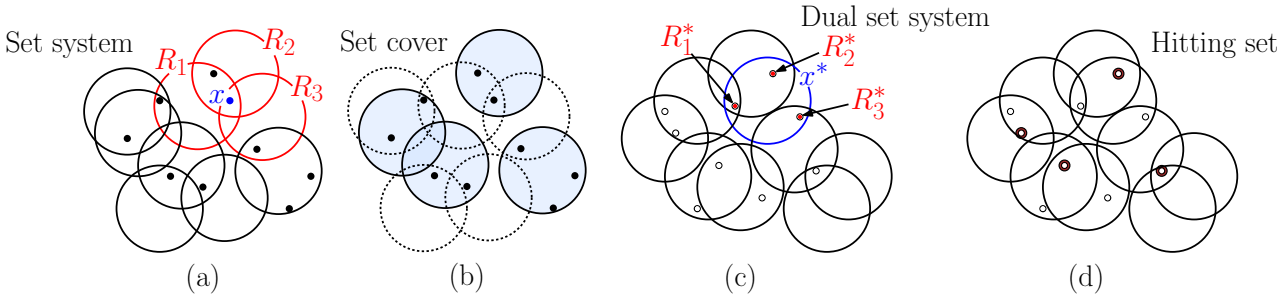


Fig. 123: (a) Set system $(X, \mathcal{R})$, (b) a set cover, (c) the dual set system $(X', \mathcal{R}')$ (where points become disk centers and disk centers become points), and (d) the equivalent hitting set

Iterative Reweighting Algorithm: Given $(X\mathcal{R})$ and k, we present an algorithm which determine either that there exists a hitting set of size at most $k' = ck \log k$ (where c is a suitably chosen constant), or it will report failure, indicating that there is no hitting set of size k. To approximate the hitting set problem, we simply run this algorithm with $k = 1, 2, 4, \ldots, 2^j$, until we achieve success. Letting k^* denote the size of the optimal hitting set, it follows that when the algorithm first reports success, the hitting set can have size at most $O(k^* \log k^*)$, and therefore it is an $O(\log k^*)$ factor approximation.

To start, we associate each point of X with a positive integer *weight*, which initially is 1. When computing measures, a point $x \in X$ with weight $w(x)$ will contribute $w(x)$ to the weight. For a suitable value of ε (which will depend on k) we compute a weighted ε-net S of size k' for X. This means that any disk of radius δ whose weight is at least ε times the total weight of X must contain at least one point of S. If S is a hitting set, we output S and we are done. If not, we must have failed to hit some disk. We double the weights of the points within this disk (thus making them more likely to be sampled in the future). If we

don't succeed after a sufficient number of iterations, we declare that no hitting set of size k exists. Here is a detailed description:

(1) Let $\varepsilon \leftarrow 1/(4k)$. For a suitable value of c (depending on ε) set $k' \leftarrow ck \log k$.

(2) Repeat the following either $2k \log(n/k)$ times or until returns "success"

  (a) Compute a weighted ε-net S of X of size k' (see Fig. 124(a)). (By the ε-Net Theorem, this can be done by computing a random sample of X of size k', where the probability that a point is sampled is proportional to its weight.)

  (b) Enumerate the disks of $\mathcal{R}$, and determine whether there exists any disk that is *not* hit by any of the points of S.

    (i) If there is no such disk, then S is a hitting set, and we return "success" (and output S as the answer).

    (ii) If there is some disk Q that is not hit, we double the weight of all the points lying within Q (see Fig. 124(b)).

(3) If we exit the loop without success, we return "failure". (This means that there is no hitting set of size k or smaller.)

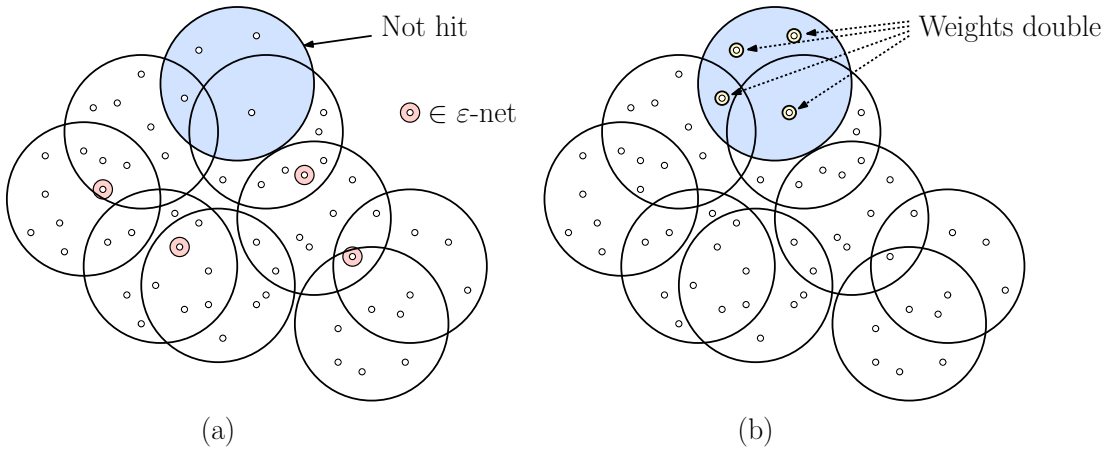


Fig. 124: The hitting-set approximation algorithm

Analysis: Before delving into the analysis, let's see intuitively what the algorithm is doing. Clearly, if this algorithm terminates, then it has computed a hitting set of size k'. We want to argue that if such a hitting set exists, the algorithm will find it within $2k \log(n/k)$ iterations. Observe that if an iteration is not successful, then some disk was not hit by our random sample. Because (by our assumption) the random sample is an ε-net, such a disk cannot contain more than an ε fraction of the total weight. All the points within this disk have their weights doubled. It follows that the total weight of the entire point set does not increase very much as a result, basically by a factor of at most $(1 + \varepsilon)$. Since the optimal hitting set must hit all disks, at least one of these doubled points is in the optimal hitting set. It follows that the total weight of the points in the optimal hitting set are increasing rapidly. Thus, the overall weight is growing slowly and the weight of the optimum set is growing rapidly. But since the optimum hitting set is a subset of the overall set, its weight can never be larger. Therefore, this process cannot go on forever. The analysis provides a formal bound on when it must end.

Lemma: If $(X, \mathcal{R})$ has constant VC-dimension, and there exists a hitting set of size k, then after at most $2k \log(n/k)$ iterations, the above algorithm will terminate successfully with a hitting set of size $O(k \log k)$. If not, it terminates with failure.

Let us assume that there exists a hitting set H of size k (which we'll call the optimal hitting set). We will show that the algorithm terminates within $2k \log(n/k)$ iterations. Let $W_i(X)$ denote the total weight of all the points of X after the ith iteration. When the algorithm starts, each of the n points of X has weight 1, so $W_0(X) = n$.

Let's consider the trnasition from the $i-1$st to the ith iteration in detail. The set S is an ε-net, which means that any disk that is not hit by S must have a weight smaller than $\varepsilon W_{i-1}(X)$. If the iteration is not successful, then there is a disk Q that was not hit. All the points within Q have their weights doubled, which implies that the total weight has increased by at most $W_{i-1}(Q) \leq \varepsilon W_{i-1}(X)$. Therefore, after an unsuccessful iteration, we have

$$W_i(X) \;=\; W_{i-1}(X) + W_{i-1}(Q) \;\leq\; W_{i-1}(X) + \varepsilon W_{i-1}(X) \;=\; (1+\varepsilon) W_{i-1}.$$

Since $W_0(X) = n$, we have $W_i(X) \leq (1+\varepsilon)^i n$. Using the standard inequality $1 + x \leq e^x$, we have $W_i(X) \leq n \cdot e^{\varepsilon i}$.

Because any hitting set (including the optimal) must hit all the disks, we know that there is at least one point of the optimal hitting set that lies within the "unhit" disk, meaning that at least one of the k optimal points will have its weight doubled. For $1 \leq j \leq k$, let $t_i(j)$ denote the number of times that the jth optimal point has been doubled during stage i. (It's either once or not at all.) Since each of these points started with a weight of 1, the total weight of the optimal hitting set after i iterations, which we will denote by $W_i(H)$ satisfies

$$W_i(H) \;=\; \sum_{j=1}^{k} 2^{t_i(j)}.$$

Because the function $f(x) = 2^x$ is a convex function, it follows from standard combinatorics (in particular, Jensen's inequality) that this sum is minimized when all the $t_i(j)$'s are as nearly equal as possible. We know that at least one point must be doubled with each iteration, and therefore the minimum occurs when $t_i(j) = i/k$, for all j. (We'll ignore the minor inconvenience that $t_i(j)$ is an integer. It won't affect the asymptotics.) Therefore:

$$W_i(H) \;\geq\; k 2^{i/k}.$$

Because $H \subseteq X$, we know that $W_i(H) \leq W_i(X)$. Therefore, we know that the number of iterations i must satisfy

$$k 2^{i/k} \;\leq\; n \cdot e^{\varepsilon i}.$$

Simplifying and recalling that $\varepsilon = 1/(4k)$, we obtain

$$\lg k + \frac{i}{k} \;\leq \lg n + \frac{i}{4k} \lg e \;\leq \lg n + \frac{i}{2k}.$$

(Here we have used the fact that $\lg e \approx 1.45 \leq 2$.) Therefore, $i/(2k) \leq \lg n - \lg k$, which implies that (assuming there is a hitting set of size k) the number of iterations i satisfies

$$i \;\leq\; 2k \lg \frac{n}{k},$$

and therefore, if the algorithm runs for more than $2\lg(n/k)$ iterations, we know that there cannot be a hitting set of size k. (At this point, the algorithm will double the value of k and try again.)

The total running time is polynomial in $n = |X|$ and $m = |\mathcal{R}|$. There are at most $O(n\log(n/k)) = O(n\log n)$ iterations. Each iteration involves the generation of an ε-net, which takes $O(k\log k)$ time. Also, for each of the m sets of $\mathcal{R}$, we take $O(k\log k)$ time to determine whether any of the elements of S lie within this set. This takes $O(km\log k) = O(nm\log n)$ time. Thus, the overall running time is (crudely) $O(n^2 m\log^2 n)$.

(Naive) Proof of ε-Sample Theorem: (Optional)

The proofs of the ε-Net and ε-Sample Theorems involve fairly straightforward applications of probability theory, but since we have not built up the necessary probability-theoretic background, it is not all that easy to present them. We will give a short (but not fully accurate) justification of the ε-Sample Theorem, which is due to Har-Peled (who presents the full proof in his book). This will give an idea of where the various expressions come from, even though the proof is not fully rigorous.

Before giving the proof, we will need to make use of a standard result from probability theory. This are well worth remembering, because it is used frequently in the analysis of randomized algorithms. To begin, let $Y_1, \ldots, Y_n$ be a set of n *Bernoulli trials*, that is, a sequence of independent "coin flips" having the value 1 with probability p_i and value 0 with probability $q_i = 1 - p_i$. Let $Y = \sum_i Y_i$, and let $\mu = \mathbb{E}[Y] = \sum_i p_i$, be its expected value. Clearly, we expect Y to be close to its expected value. What is the probability that it deviates from this by, say, a factor of $1 \pm \delta$ (where δ is a value that we will select based on our needs). The following result, called the *Chernoff bound*, credited to Herman Chernoff from 1952.

Theorem: (Chernoff Bound) For any $\delta > 2e - 1$:

$$\Pr[Y > (1+\delta)\mu] \;<\; \exp(-\mu\delta^2/4) \quad \text{and} \quad \Pr[Y < (1-\delta)\mu] \;<\; \exp(-\mu\delta^2/2).$$

Suppose we have a range space $(X, \mathcal{R})$ with VC-dimension d, and consider any range Q that contains a fraction p of the points of X, where $p \geq 2\varepsilon$. Consider any random sample S of s points from X (sampling with replacement so that the probabilities of lying within r do not change).

Let q_i be the ith sample point, and let Y_i be an indicator variable whose value is 1 if $q_i \in Q$ and 0 otherwise. Clearly, $(\sum_i Y_i)/s$ is an estimate for $p = |Q \cap X|/|X|$. To prove the ε-Sample Theorem, we want to determine how many samples s we should take so that this estimate is within a factor $1 \pm \varepsilon$ of p with confidence at least $1 - \varphi$.

Observe that the number of points of X we expect to see in Q is $\mu = ps$. A sample fails the sampling theorem if the actual count differs from the expected value by at least εs, that is

$$\left|\sum_{i=1}^{s} Y_i - \mu\right| \;>\; \varepsilon s \;=\; \frac{\varepsilon}{p} ps \;=\; \delta\mu,$$

where $\delta = \varepsilon/p$. Letting $Y = \sum_{i=1}^{s} Y_i$, this is equivalent to saying that Y either exceeds its expected value μ by a factor of $1 + \delta$ or falls short by a factor of $1 - \delta$. We can therefore

apply the Chernoff bound (combining both the upper and lower bounds), obtaining

$$\begin{aligned}\Pr\left[\left|\sum_{i=1}^{s} Y_i - \mu\right| > \delta\mu\right] &\leq \Pr[Y < (1-\delta)\mu] + \Pr[Y > (1+\delta)\mu] \\ &\leq \exp(-\mu\delta^2/2) + \exp(-\mu\delta^2/4) \\ &\leq 2\exp\left(-\frac{\mu\delta^2}{4}\right) = 2\exp\left(-\frac{\varepsilon^2}{4p}s\right).\end{aligned}$$

We want this probability of a bad sample to be at most φ. If we set φ to this value and solve for s, we obtain

$$s \geq \frac{4}{\varepsilon^2}\ln\frac{2}{\varphi} \geq \frac{4p}{\varepsilon^2}\ln\frac{2}{\varphi}.$$

This is pretty close to the value given in the ε-Sample Theorem, but strangely we never made use of the VC-dimension d. (Something must be wrong!) The problem with this proof is that we only applied it to a single range Q. To generalize it to all ranges, we need work a bit harder, and in doing so, we will introduce the VC-dimension. (We refer you to Har-Peled's book for the details.)

Lecture 18: Motion Planning

Motion planning: In this lecture we will discuss applications of computational geometry to the problem of motion planning. This problem arises in robotics and in various areas where the objective is to plan the collision-free motion of a moving agent in a complex environment.

Work Space and Configuration Space: The environment in which the robot operates is called its *work space*, which consists of a domain in which the robot can move and a set of *obstacles*, which the robot must avoid. Any overlap between the robot and an obstacle is called a *collision*. 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 (relative to some default placement). Combined, these two elements fully define the robot's exact position and shape in space.

For example, suppose that the robot is a triangle that can translate and rotate in the plane (see Fig. 125(a)). Its configuration may be described by the (x, y) coordinates of some reference point for the robot, and an angle θ that describes its orientation. Its geometric information would include its shape (say at some canonical position), given, say, as a simple polygon. Given its geometric description and a configuration (x, y, θ), this uniquely determines the exact position $\mathcal{R}(x, y, \theta)$ of this robot in the plane. Thus, the position of the robot can be identified with a point in the robot's *configuration space*.

A more complex example would be an *articulated arm* consisting of a set of *links*, connected to one another by a set of rotating *joints*. The configuration of such a robot might consist of a vector of joint angles $(\theta_1, \ldots, \theta_k)$ (see Fig. 125(b)). 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

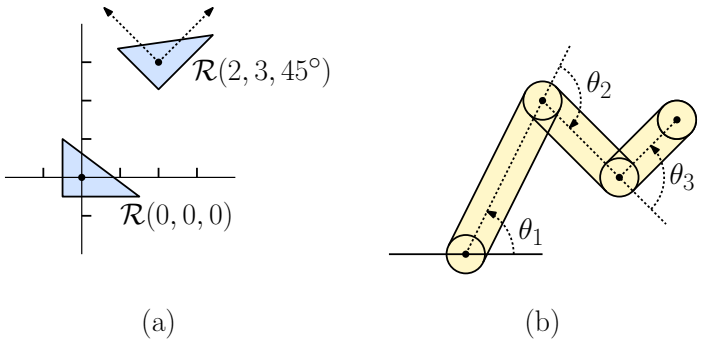


Fig. 125: Configurations of a translating and rotating robot.

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 the (x, y, z) coordinates of its 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 (see Fig. 125).

The problem of computing a collision-free path for the robot can be reduced to computing a path in the robot's configuration space. To distinguish between these, we use the term *work space* to denote the (standard Euclidean) space where the robot and obstacles reside (see Fig. 126(a)), and the *configuration space* to denote to the space in which each point corresponds to the robot's configuration (see Fig. 126(b)). Planning the motion of the robot reduces to computing a path in configuration space.

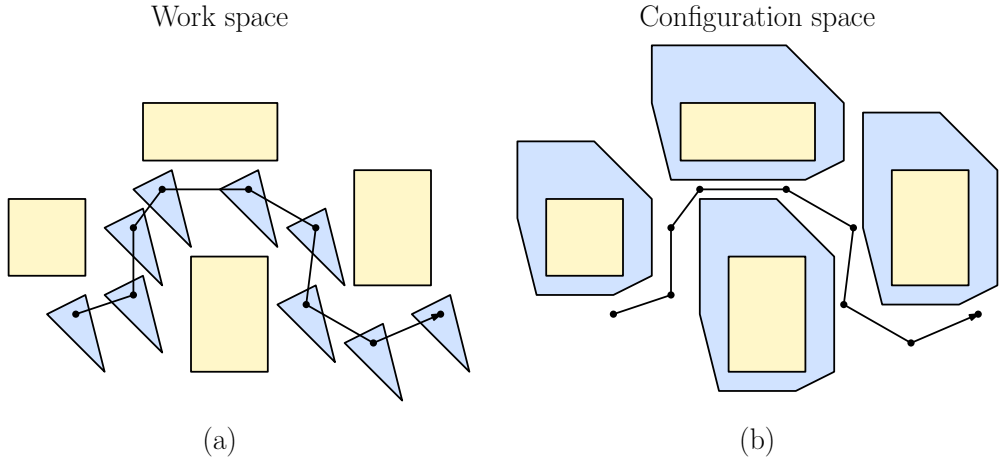


Fig. 126: (a) Translational motion in the work space amidst obstacles and (b) motion in the configuration space amidst collision obstacles.

A configuration that results in the robot to intersecting with one or more of the obstacles is called a *forbidden configuration*. The set of all forbidden configurations is denoted $C_{\text{forb}}(\mathcal{R}, S)$. All other placements are called *free configurations*, and the set of these configurations is denoted $C_{\text{free}}(\mathcal{R}, S)$, or *free space*.

Now consider the *motion planning* problem in robotics. Given a robot $\mathcal{R}$, an work space S, and initial configuration s and final configuration t (both points in the robot's free configuration

space), determine (if possible) a way to move the robot from one configuration to the other without intersecting any of the obstacles. This reduces to the problem of determining whether there is a path from s to t that lies entirely within the robot's free configuration space. Thus, we map the task of computing a robot's motion to the problem of finding a path for a single point through a collection of obstacles.

Configuration Obstacles: While motion occurs in the robot's work space, we perform motion planning in the robot's configuration space. For example, given a 2-dimensional robot that can translate and rotate amidst 2-dimensional polygonal objects, we determine collisions in the 2-dimensional space. But our motion plan is a path in 3-dimensional (x, y, θ) space, where (x, y) indicate the robot's translation and θ gives its rotation (about the reference point).

In order to determine which configurations are free and forbidden, we need to determine the meaning of an obstacle in configuration space. Let's suppose that the work space is d-dimensional and the configuration space is k-dimensional. Given an obstacle $P \subset \mathbb{R}^d$, its *configuration obstacle* or (or *C-obstacles* for short) is defined to be

$$\mathcal{C}_{\mathcal{R}}(P) \;=\; \{p \in \mathbb{R}^k : \mathcal{R}(p) \cap P \neq \emptyset\}.$$

(Note that the configuration p is in $\mathbb{R}^k$, but the determination of collision between $\mathcal{R}(p)$ and P is done in the work space $\mathbb{R}^d$.)

Configuration spaces are typically higher dimensional than the work space, and configuration obstacles can be quite complex. When rotation is involved, they are bounded by curved surfaces. The simplest case to visualize is that of translating a convex polygonal robot in the plane amidst a collection of polygonal obstacles, since both the work space and configuration space are two dimensional. Consider a reference point placed in the center of the robot. 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 (see Fig. 126(b)).

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 obstacles, and then we will consider the question of how to construct configuration spaces.

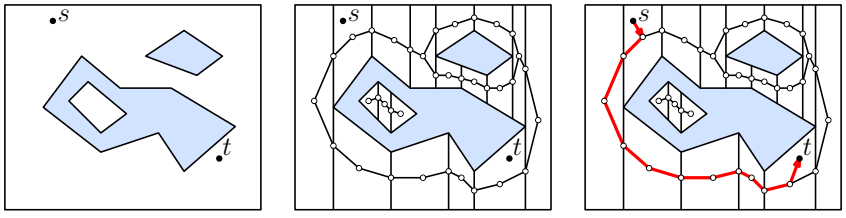


Fig. 127: Simple point motion planning through road maps.

Let us start with a very simple case in which the configuration space is 2-dimensional and the objects are simple polygons, possibly with holes (see Fig. 127(a)). To determine whether there is a path from one point s to another point t of free configuration space, we can subdivide

free space into simple convex regions. In the plane, we already know of one way to do this by computing a trapezoidal map. We construct a trapezoidal map for all of the line segments bounding the obstacles, then throw away any trapezoids that lie in the forbidden space (see Fig. 127(b)). 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 this subdivision. 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 (see Fig. 127(c)). 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.

Practical Considerations: While the trapezoidal map approach guarantees correctness, it is rather limited. If the configuration space is 2-dimensional, but the configuration obstacles have curved boundaries, we can easily extend the trapezoidal map approach, but we will generally need to insert walls at points of vertical tangency.

Higher-dimensional spaces pose a much bigger problem (especially when combined with curved boundaries). There do exist subdivision methods (one is called the *Collins cylindrical algebraic decomposition*, which can be viewed as a generalization of the trapezoidal map to higher dimensions and curved surfaces), but such subdivisions often can have high combinatorial complexity. Most practical road map-based approaches dispense with computing the subdivision, and instead simply generate a large random sample of points in free space. The problem is that if no path is found, who is to blame? Is there really no path, or did we simply fail to sample enough points? The problem is most extreme when the robot needs to navigate through a very narrow passage.

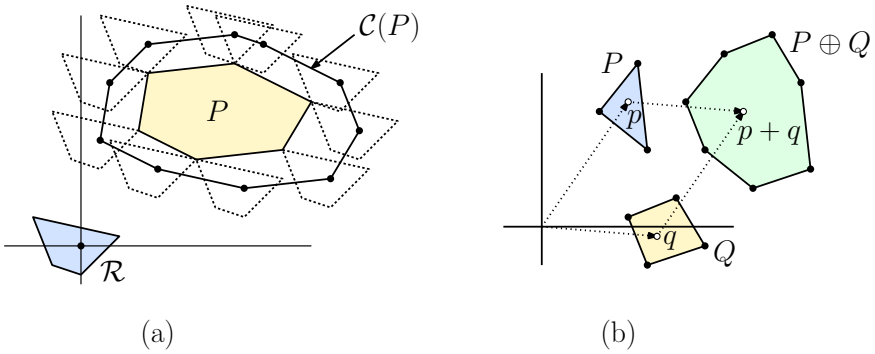
Another widely used heuristic is called the *rapidly-exploring random tree* (RRT). These trees provide a practical approach to sampling the configuration space and building a tree-based road map. While this method has good practical value, it can also fail when tight squeezes are necessary.

C-Obstacles via 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. One way to visualize $\mathcal{C}_{\mathcal{R}}(P)$ is to imagine "scraping" $\mathcal{R}$ along the boundary of P and seeing the region traced out by $\mathcal{R}$'s reference point (see Fig. 128(a)).

Given $\mathcal{R}$ and P, how do we compute the configuration obstacle $\mathcal{C}_{\mathcal{R}}(P)$? To do this, we first introduce the notion of a *Minkowski sum*. Let us think of points in the plane as vectors. Given any two sets P and Q in the plane, define their *Minkowski sum* to be the set of all pairwise sums of points taken from each set (see Fig. 128(b)), that is,

$$P \oplus Q \;=\; \{p + q \,:\, p \in P,\; q \in Q\}.$$

When taking the Minkowski sum of convex bodies, and useful alternative definition is in terms of *support functions*. Given a convex body P in $\mathbb{R}^d$, defined its *support function*,



(a) (b)

Fig. 128: Minkowski sum of two polygons.

denoted $h_P : \mathbb{R}^d \to \mathbb{R}$, by

$$h_P(u) \;=\; \sup_{p \in P}(u \cdot p),$$

where $(u \cdot p)$ is the standard dot product. When u is a unit vector, this is just the length of the maximum projection of P along direction u. A nice connection between support functions and Minkowski sums is that the support function of the Minkowski sum is the (arithmetic) sum of the support functions.

Lemma: Given two convex bodies P and Q in $\mathbb{R}^d$, $h_{P \oplus Q} = h_P + h_Q$.

In addition to sums of convex bodies, we also define

$$\alpha Q \;=\; \{\alpha q \,:\, q \in Q\} \qquad \text{and} \qquad -Q \;=\; \{-q \,:\, q \in Q\}.$$

(Intuitively, $-Q$ can be viewed as reflecting Q through the origin.) We introduce the shorthand notation $Q \oplus p$ to denote $Q \oplus \{p\}$, which is just the *translate* of Q by vector p. So, we can express the placement p of a translating robot $\mathcal{R}$ as $\mathcal{R}(p) = \mathcal{R} \oplus p$. The relevance of Minkowski sums to C-obstacles is given in the following claim.

Lemma: Given a translating robot $\mathcal{R}$ and an obstacle P, $\mathcal{C}_{\mathcal{R}}(P) = P \oplus (-\mathcal{R})$ (see Fig. 129).

Proof: Consider any translation vector t. Observe that $t \in \mathcal{C}_{\mathcal{R}}(P)$ iff $\mathcal{R}(t)$ intersects P, which is true iff there exist $r \in \mathcal{R}$ and $p \in P$ such that $p = r + t$ (see Fig. 129(a)), which is true iff there exist $-r \in -\mathcal{R}$ and $p \in P$ such that $t = p + (-r)$ (see Fig. 129(b)), which is equivalent to saying that $t \in P \oplus (-\mathcal{R})$. Therefore, $t \in \mathcal{C}_{\mathcal{R}}(P)$ iff $t \in P \oplus (-\mathcal{R})$, which means that $\mathcal{C}_{\mathcal{R}}(P) = P \oplus (-\mathcal{R})$, as desired.

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 $\mathcal{C}_{\mathcal{R}}(P)$ reduces to the problem of computing a Minkowski sum of two convex polygons. We'll show next 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.

Note that the above 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.

Computing the Minkowski Sum of Convex Polygons: Let's consider how to compute $P \oplus Q$ for two convex polygons P and Q, having m and n vertices, respectively. The algorithm is

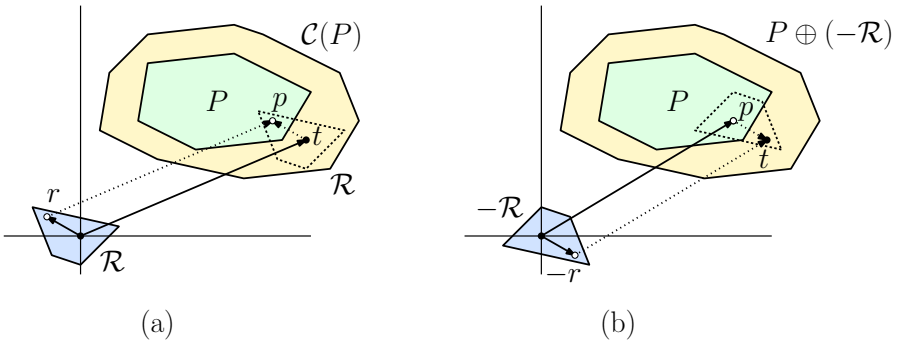


Fig. 129: Configuration obstacles and Minkowski sums.

based on the following observation. Given a vector u, We say that a point $p \in P$ is *extreme* in direction u if it defines the value of the support function $h_P(u)$, that is, it maximizes the dot product $p \cdot u$ over all $p \in P$. The following observation is a direct consequence of the above lemma about support functions.

Observation: Given two polygons P and Q, the set of extreme points of $P \oplus Q$ in direction u is the set of sums of points p and v that are extreme in direction u for P and Q, respectively.

This observation motivates an algorithm for computing $P \oplus Q$. We perform an angular sweep by sweeping a unit vector u counterclockwise around a circle. As u rotates, it is an easy matter to maintain the vertex or edge of P and Q that is extreme in this direction. Whenever u is perpendicular to an edge of either P or Q, 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*.

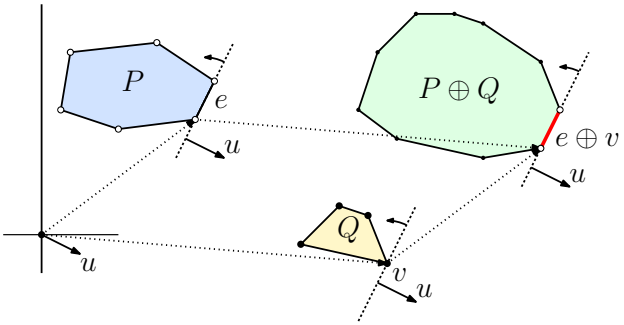


Fig. 130: Computing Minkowski sums.

Assuming P and Q are convex, observe that each edge of P and each edge of Q contributes exactly one edge to $P \oplus Q$. (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 can compute $P \oplus Q$ by directing the edges in a consistent manner (say, counterclockwise), treating each edge as a vector, merge the two sorted orders in $O(n+m)$ time, and then form the final polygon by joining these vectors tail to head in cyclic order.

Lemma: Given two convex polygons, P and Q, with n and m vertices respectively, their

Minkowski sum $P \oplus Q$ consist of at most $n+m$ edges and can be computed in $O(n+m)$ time.

Complexity of the Union of C-Obstacles: We have shown that free space for a translating robot among obstales $\{T_1, \ldots, T_k\}$ is the complement of a union of C-obstacles $P_i = \mathcal{C}_{\mathcal{R}}(T_i) = T_i \oplus (-\mathcal{R})$. If T_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? The complexity generally depends on what properties of the T_i's and $\mathcal{R}$. We will consider the various cases below.

Convex Robot and Convex Obstacles: Let's first consider the simplest case, where both $\mathcal{R}$ and the T_i's are convex. Recall that $P_i = T_i \oplus (-\mathcal{R})$ denotes the associated C-obstacle. Although the T_i's do not overlap, the P_i's may (see Fig. 131). A naive analysis would suggest that the boundary complexity of the union of such overlapping convex polygons $\bigcup_{i=1}^{k} P_i$ could be as large as $O(n^2m^2)$. We will show that the complexity will be much smaller when all the bodies are convex.

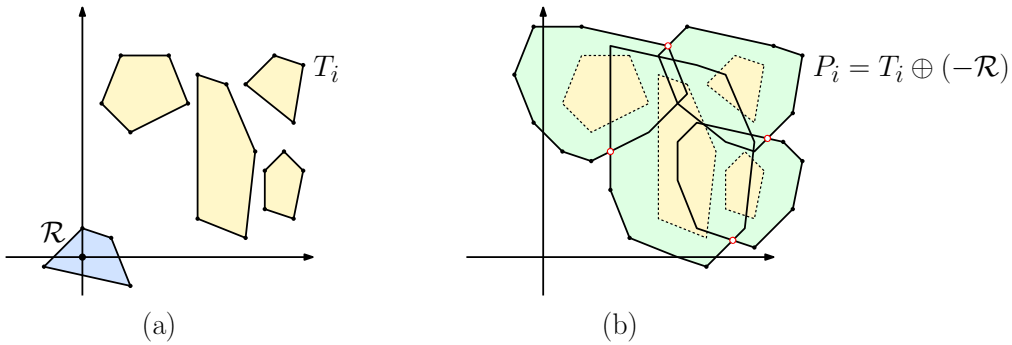


Fig. 131: The C-obstacles for a convex robot among convex obstacles.

We need some way to characterize the special geometric structure of the union of Minkowski sums of non-overlapping convex polygons. A set of convex bodies in the plane $\{P_1, \ldots, P_n\}$ is said to be a *collection of pseudodisks* if for any two distinct bodies P_i and P_j, both of the set-theoretic differences $P_i \backslash P_j$ and $P_j \backslash P_i$ are connected sets (see Fig. 132). If this is violated for any two objects, we say that these two objects have a *crossing intersection*. Note that the pseudodisk property is *not* a property of any single object. Rather, it is a property of the *entire collection* (or more properly, of all pairs of objects from the entire collection). The key observation is that if the original sets do not overlap, then the configuration obstacles form a collection of pseudodisks.

Lemma: Given a set of convex objects $\mathcal{T} = \{T_1, \ldots, T_k\}$ with pairwise disjoint interiors and convex Q, the set

$$\{T_i \oplus Q \,:\, 1 \leq i \leq k\}$$

is a collection of pseudodisks.

Proof: Consider two polygons T_1 and T_2 with disjoint interiors. We want to show that $T_1 \oplus Q$ and $T_2 \oplus Q$ do not have a crossing intersection. Given any directional unit vector u, recall that the *extreme* point of Q in direction u is the point $q \in Q$ that maximizes the dot product $(u \cdot q)$. By the lemma about support functions, the point of $T_1 \oplus Q$ that is

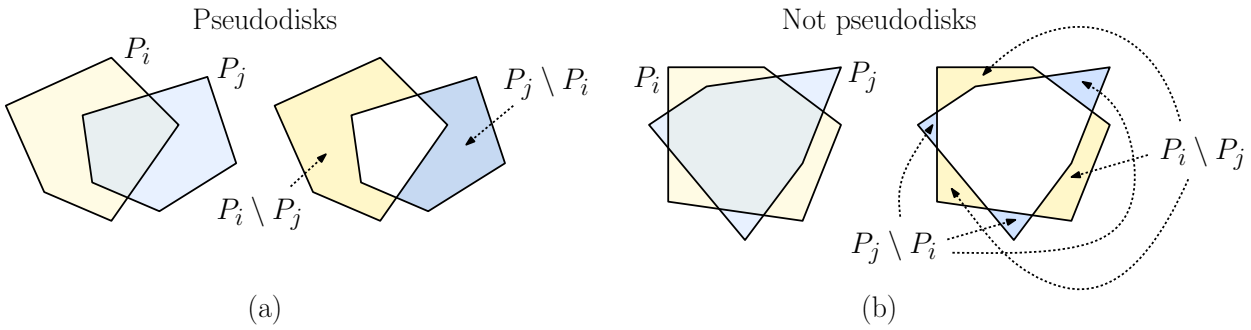


Fig. 132: Collections of pseudodisks.

extreme in direction u is the sum of the points t and q that are extreme for T_1 and Q, 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 u_1 and u_2 be the outward pointing perpendicular vectors for these tangents (see Fig. 133(a)). Because these polygons do not intersect, it follows easily that as the directional vector rotates from u_1 to u_2, T_1 will be the more extreme polygon, and from u_2 to u_1 T_2 will be the more extreme (see Fig. 133(b)).

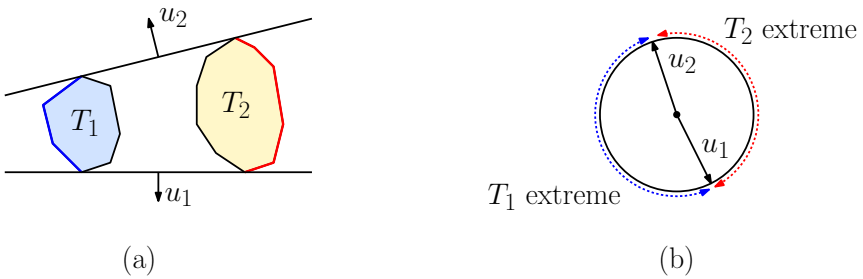


Fig. 133: Alternation of extremes.

Now, if to the contrary $T_1 \oplus Q$ and $T_2 \oplus Q$ 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 $(T_1 \oplus Q) \cup (T_2 \oplus Q)$ such that $p_1, p_3 \in T_1 \oplus Q$ and $p_2, p_4 \in T_2 \oplus Q$ (see Fig. 134(a)). First consider p_1. Because it is on the convex hull, consider the direction u_1 perpendicular to the supporting line here. Let q, t_1, and t_2 be the extreme points of Q, T_1 and T_2 in direction u_1, respectively. From our basic fact about Minkowski sums we have

$$p_1 = q + t_1 \qquad \text{and} \qquad p_2 = q + t_2.$$

Since p_1 is on the convex hull, it follows that t_1 is more extreme than t_2 in direction u_1, that is, T_1 is more extreme than T_2 in direction u_1. By applying this same argument, we find that T_1 is more extreme than T_2 in directions u_1 and u_3, but that T_2 is more extreme than T_1 in directions u_2 and u_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 Fig. 134(b)).

The important property of any collection of psuedodisks is that the number of vertices in boundary of their union does not grow quadratically in the original number of vertices, as

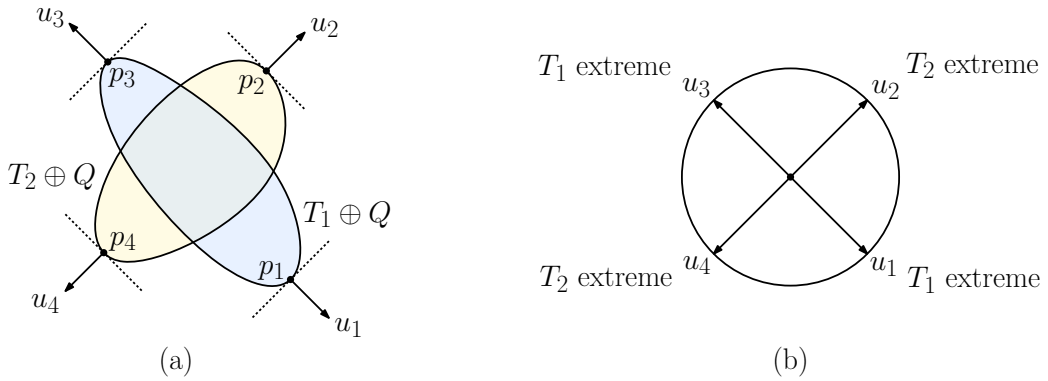


Fig. 134: Proof of the lemma on pseudodisks.

would be the case for any normal collection of polygons. Rather, it only grows linearly. We prove this in the following lemma.

Lemma: Consider a collection of polygonal pseudodisks $\{P_1, \ldots, P_m\}$ having a total of n vertices. Then the number of vertices in the polygonal domain $U = \bigcup_{i=1}^{n}$ is at most $2n$.

Proof: The proof is based on a charging scheme. Each vertex in U will be charged to one of the vertices the original pseudodisks P_i, such that no original vertex is charged more than twice.

There are two types of vertices that may appear on the boundary of U. The first are vertices from the original polygons. There can be at most n such vertices, and each is charged to itself. The more troublesome vertices are those that arise when the edges of two of the original ppolygons intersect each other. Suppose that two edges e_1 and e_2 of pseudodisks P_1 and P_2 intersect along the union at some vertex v. We shall see that, by the pseudodisk property, there is a vertex of some P_i in the interior of U that can be charged.

First, follow the edge e_1 into the interior of the pseudodisk P_2. Two things might happen. First, we might hit an endpoint v_1 of e_1 before leaving the interior P_2 (see Fig. 135(a)). In this case, we charge the intersection to v_1. Note that v_1 can be assessed at most two such charges, one from either of its two incident edges. On the other hand, if e_1 passes all the way through P_2 before coming to the endpoint, then let's instead try to apply the charge but this time with with the edge e_2. Again, if it hits its endpoint v_2 of e_2 before coming out of P_1, then charge to this endpoint (see Fig. 135(b)). Again, when this happens v_2 is assessed at most two charges.

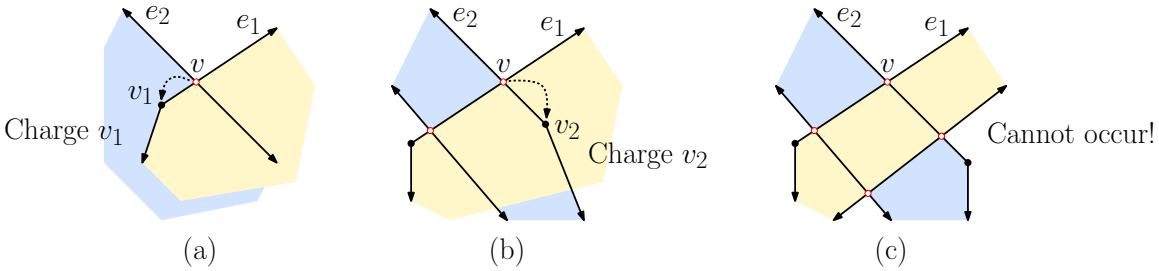


Fig. 135: Proof of the lemma on the complexity of pseudodisks.

What do we do if both charging attempts fail? This means that e_1 shoots straight

through P_2 and e_2 shoots straight through P_1 without encountering vertex to charge? In this case, it is easy to see that P_1 and P_2 must have a crossing intersection (see Fig. 135(c)). But this would violate the pseudodisk property, and therefore this cannot occur.

In conclusion, every vertex of the union U has been charged to one of the original n vertices, such that no original vertex receives more than two charges, implying that U has a total of at most $2n$ vertices.

We can now bound the total complexity of the configuration obstacle space in the convex-convex case.

Lemma: Given a set of disjoint convex polygonal obstacles $\mathcal{T} = \{T_1, \ldots, T_k\}$ with a total of n vertices and convex polygonal robot $\mathcal{R}$ with m vertices, the total boundary complexity of the configuration space $\mathcal{C}_{\mathcal{R}}(\mathcal{T})$ is $O(nm)$.

Proof: Let n_i denote the number of vertices of T_i. Letting $Q = -\mathcal{R}$ and $P_i = T_i \oplus Q$, it follows that P_i has at most $n_i + m$ vertices. Because the T_i's are disjoint, the P_i's form a collection of pseudodisks with a total number of vertices at most

$$\sum_{i=1}^{k}(n_i + m) \;=\; km + \sum_{i=1}^{k} n_i \;=\; km + n \;\leq\; nm + n \;=\; (n+1)m \;=\; O(nm)$$

vertices. By the above lemma, the union of these pseudodisks has total complexity at most $O(nm)$.

Convex Robot and Nonconvex Obstacles: Next, let's consider the case where the robot $\mathcal{R}$ is a convex m-sided polygon and there is a single simple obstacle polygon T with n sides. This can be reduced to the results of the previous section. We can triangulate T into $n-2$ triangles. Clearly, these triangles have disjoint interiors, and they have a total of $O(n)$ vertices. Hence by the results of the previous section, the overall complexity of the C-obstacle is $O(mn)$.

Theorem: Let $\mathcal{R}$ be a convex m-gon and T and simple n-gon, then C-obstacle $\mathcal{C}_{\mathcal{R}}(T) = T \oplus (-\mathcal{R})$ has total complexity $O(nm)$.

We claim that this $O(nm)$ bound is tight. To see this, let $\mathcal{R}$ be a regular m-gon, and define T to be a comb-like structure where the teeth of the comb are separated by a distance at least as large as the diameter of $\mathcal{R}$ (see Fig. 136(a)). 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 $\Omega(nm)$ (see Fig. 136(b)).

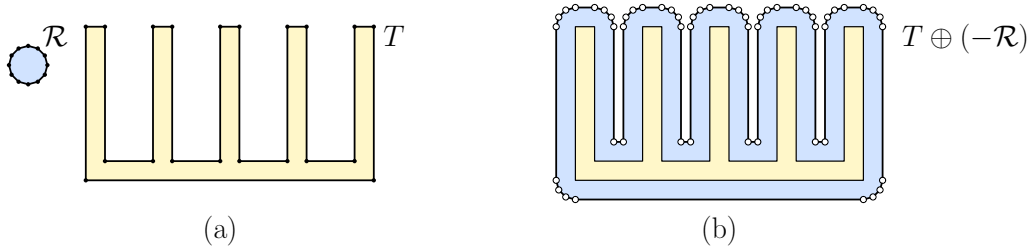


Fig. 136: Minkowski sum (simple-convex) of $O(nm)$ complexity.

Nonconvex Robot and Nonconvex Obstacles: Finally, let's consider the case where both $\mathcal{R}$ and T are general simple polygons with m and n sides, respectively. How many sides might there be in the C-obstacle $T \oplus (-\mathcal{R})$ in the worst case? We can derive a quick upper bound as follows. First observe that we can triangulate both T and $-\mathcal{R}$, decomposing them into $n-2$ and $m-2$ triangles, respectively. Thus,

$$T = \bigcup_{i=1}^{n-2} T_i \quad \text{and} \quad -\mathcal{R} = \bigcup_{j=1}^{m-2} R_j,$$

where the T_i's have pairwise disjoint interiors, as do the R_j's. It follows that

$$T \oplus (-\mathcal{R}) = \bigcup_{i=1}^{n-2} \bigcup_{j=1}^{m-2} (T_i \oplus R_j).$$

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 the complexity really be this high? Yes it could. Consider the two polygons in Fig. 137(a).

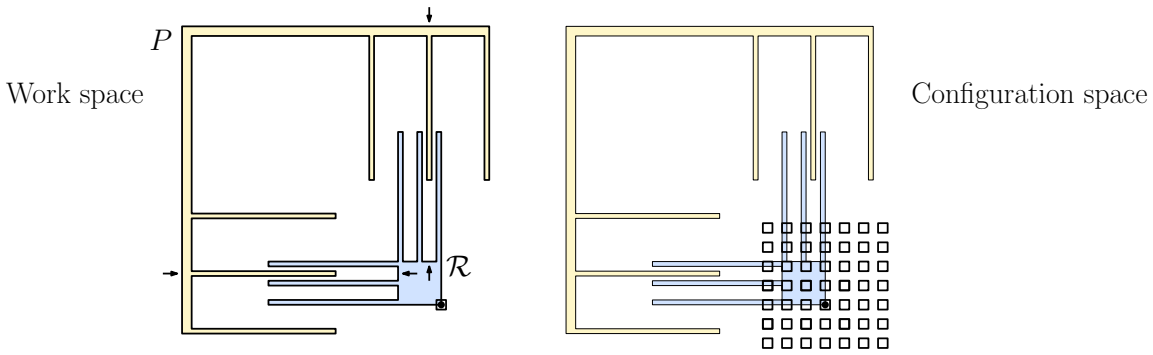


Fig. 137: Minkowski sum (simple-simple) of $O(n^2m^2)$ complexity.

Suppose that T and $\mathcal{R}$ have m and n "teeth", respectively. For each of independent choice of two teeth of T (one from the top and one from the side), and two gaps from $\mathcal{R}$ (one from the top and one from the side), there is a valid placement where these teeth fit within these gaps (see the arrows in Fig. 137(a)). However, as can be seen from the figure, it is impossible to move from one of these to another by translation without causing a collision. It follows that there are $\Omega(n^2m^2)$ connected components of the free configuration space, or equivalently in $T \oplus (-\mathcal{R})$ (see Fig. 137(b)).

You might protest that this example is not fair. While it is true that there are many components in the Minkowski sum, motion planning takes place within a single connected component of free space, and therefore the quantity that is really of interest is the (worst-case) combinatorial complexity of any *single* connected component of free space. (In the example above, all the components were of constant complexity.) This quantity is complicated to bound for general shapes, but later we will show that it can be bounded for convex shapes.

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

Supplemental Lectures

Lecture 19: Geometric Basics

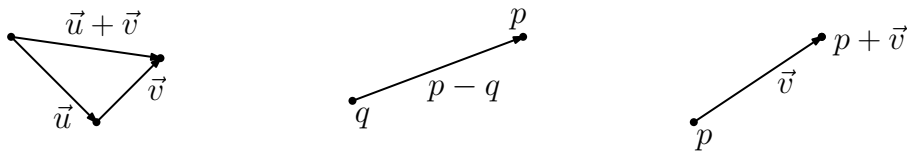
Geometry Basics: As we go through the semester, we will introduce much of the geometric facts and computational primitives that we will be needing. For the most part, we will assume that any geometric primitive involving a constant number of elements of constant complexity can be computed in $O(1)$ time, 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 (see Fig. 138). 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:

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



vector addition point subtraction point subtraction

Fig. 138: Affine operations.

A number of operations can be derived from these. For example, we can define the subtraction of two vectors $\vec{u} - \vec{v}$ as $\vec{u} + (-1) \cdot \vec{v}$ or scalar-vector division $\vec{v}/\alpha$ as $(1/\alpha) \cdot \vec{v}$ provided $\alpha \neq 0$.

There is one special vector, called the *zero vector*, $\vec{0}$, which has no magnitude, such that $\vec{v} + \vec{0} = \vec{v}$.

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

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

Note that the middle term of the above equation is not legal given our list of operations. But this is how the affine combination is typically expressed, namely as the weighted average of two points. The right-hand side (which is easily seen to be algebraically equivalent) is legal. An important observation is that, if $p_0 \neq p_1$, then the point $\text{aff}(p_0, p_1; \alpha_0, \alpha_1)$ lies on the line joining p_0 and p_1. As α_1 varies from $-\infty$ to $+\infty$ it traces out all the points on this line (see Fig. 139).

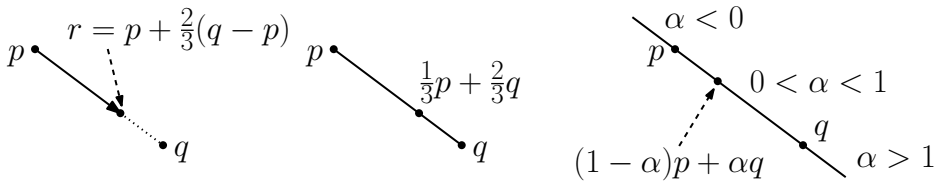


Fig. 139: Affine combination.

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 α_1 to α_0. The resulting operation is called a *convex combination*, and the set of all convex combinations traces out the line segment $\overline{p_0 p_1}$.

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

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

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 π) 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 (see Fig. 140). Note that orientation depends on the order in which the points are given.

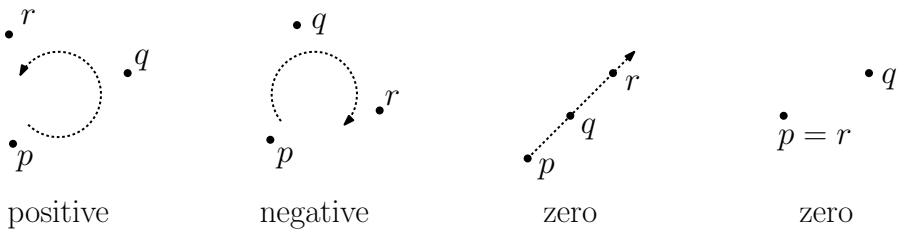


Fig. 140: Orientations of the ordered triple (p, q, r).

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 4×4 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 Fig. 141. The convex hull of a finite set of points in the plane is a bounded, closed, convex polygon.

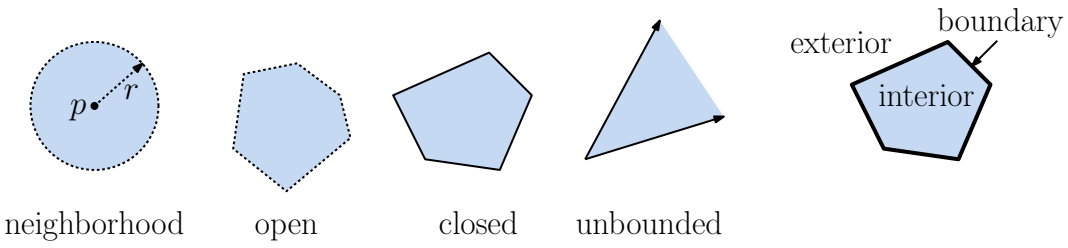


Fig. 141: Terminology.

Lecture 20: Computing Slope Statistics

Slope Statistics: 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 (see Fig. 142(a)). The hypothesis is that increasing the amount of treatment by Δa units results in an increase in therapeutic benefit of $\Delta b = s(\Delta a)$, where s is an unknown scale factor.

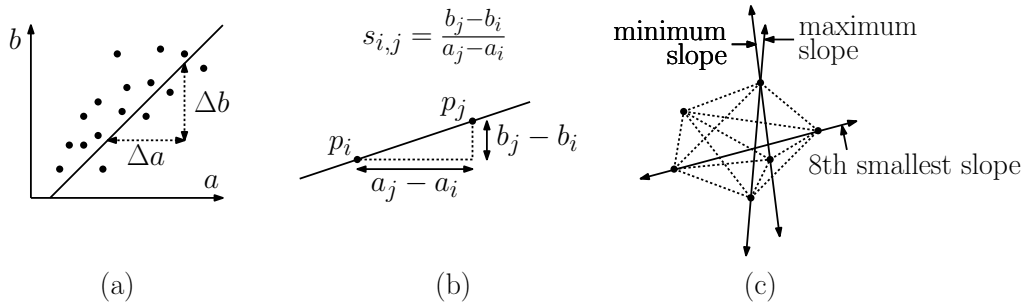


Fig. 142: (a) Slope analysis, (b) the slope $s_{i,j}$, and (c) the slope set $S = \{s_{i,j} \mid 1 \le i < j \le n\}$.

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. 142(b)). 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 the obvious $O(n^2)$ time.

Here are some natural questions we might ask about the set S (see Fig. 142(c)):

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. (This problem has been shown to be solvable in $O(n \log^2 n)$ time by D. Ajwani, S. Ray, R. Seidel, H. R. Tiwary, "On

Computing the Centroid of the Vertices of an Arrangement and Related Problems", WADS 2007.)

Range counting: Given a pair of reals $s^- \leq s^+$, 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 $1 \leq i < j \leq n$, $b_i > b_j$ if and only if $s_{i,j}$ is negative. For $1 \leq i < j \leq n$, 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. 143(a)).

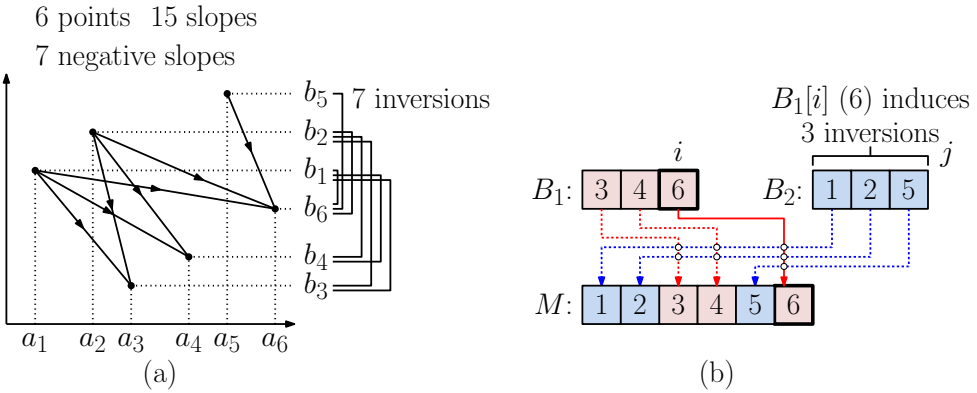


Fig. 143: Inversion counting and application to counting negative slopes.

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 split (e.g., down the middle) into a left and right subsequence, denoted B_1 and B_2, 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_1 and I_2 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_1, we compute the number of smaller elements in B_2, and add these to I. In the end, we return the total number of inversions, $I_1 + I_2 + I$.

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_1 and B_2. We repeatedly[28] copy the smaller of $B_1[i]$ and $B_2[j]$ to the merged sequence M. Because both subsequences are sorted, when we copy $B_1[i]$ to M, $B_1[i]$ is inverted with respect to the elements $B_2[1 \ldots j-1]$, whose values are smaller than it (see Fig. 143(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 particular, if $i \leq |B_1|$, we append the remaining $|B_1|-i+1$ elements of B_1 to M. Since these elements are all larger than any element of B_2, we add $(|B_1| - i + 1)|B_2|$ to the inversion counter. (When copying the remaining elements from B_2, 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.]

(0) If $|B| \leq 1$ then return an inversion count of zero;
(1) Split B into disjoint subsets B_1 and B_2, each of size at most $\lceil n/2 \rceil$, where $n = |B|$;
(2) $(B_1, I_1) \leftarrow \text{InvCount}(B_1)$;
    $(B_2, I_2) \leftarrow \text{InvCount}(B_2)$;
(3) Let $i \leftarrow j \leftarrow 1$; $I \leftarrow 0$; $M \leftarrow \emptyset$;
(4) While ($i \leq |B_1|$ and $j \leq |B_2|$)
    (a) if ($B_1[i] \leq B_2[j]$) append $B_1[i{+}{+}]$ to M and $I \leftarrow I + (j-1)$;
    (b) else append $B_2[j{+}{+}]$ to M;
    On exiting the loop, either $i > |B_1|$ or $j > |B_2|$.
(5) If $i \leq |B_1|$, append $B_1[i \ldots]$ to M and $I \leftarrow I + (|B_1| - i + 1)|B_2|$;
(6) Else (we have $j \leq |B_2|$), append $B_2[j \ldots]$ to M;
(7) return $(M, I_1 + I_2 + I)$;

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.

[28] More formally, we maintain the invariant that $B_1[i] > B_2[j']$ for $1 \leq j' \leq j-1$ and $B_2[j] \geq B_1[i']$ for $1 \leq i' \leq i-1$.

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. 144). (The reason for negating the b coordinate is now evident. Otherwise, we would get the negation of the slope.)

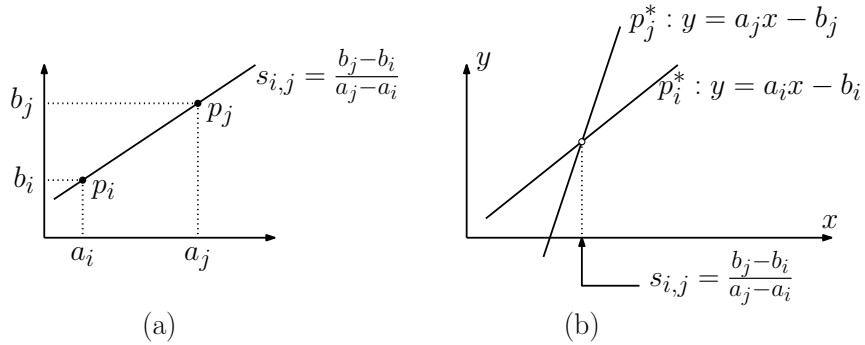


Fig. 144: Point-line duality and the relationship between the slope of a line between two points and the x-coordinate of the duals of the two points.

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. 145(a)).

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. 145(a)). Next, compute the order in which the (renumbered) lines intersect the right side of the slab. In particular, line

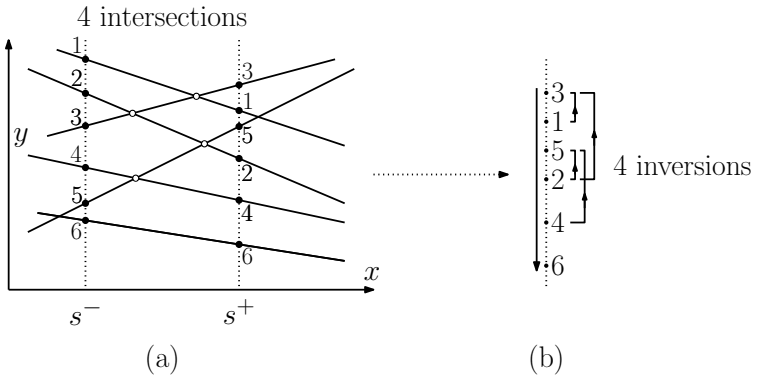


Fig. 145: Intersections in the vertical slab $[s^-, s^+]$ and inversion counting.

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 a simple 1-dimensional sorting algorithm. 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.

Lecture 21: Minimum Enclosing Ball

Minimum Enclosing Ball: 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 *Minimum Enclosing Ball Problem* (or MEB). We are given a set P of n points in $\mathbb{R}^d$, and we are asked to find the (closed) Euclidean ball of minimum radius that encloses all of these points. For the sake of simplicity, we will consider the problem in the plane, but the method readily generalizes to any (fixed) dimension. The algorithm is randomized, and the expected case running time (averaged over all random choices) is $O((d+1)!n)$ in $\mathbb{R}^d$. Under our usual assumption that the dimension d is fixed, this is $O(n)$.

Geometric Background: Let us recall some standard terminology. A *circle* is the set of points that are equidistant from some center point. In 3-space this is called a *sphere* and in general $\mathbb{R}^d$ space it is called a *hypersphere*. More formally, given a center point $c = (c_1, \ldots, c_d) \in \mathbb{R}^d$ and a positive radius $r \in \mathbb{R}$, the hypersphere is the set of points $p = (p_1, \ldots, p_d)$ such that

$$\sum_{i=1}^{d} (p_i - c_i)^2 \;=\; r^2.$$

(Note that because a hypersphere embedded in $\mathbb{R}^d$ is a $(d-1)$-dimensional surface, the term "k-dimensional hypersphere" usually refers to a sphere residing in $\mathbb{R}^{k+1}$.) The (closed) Euclidean *ball* is the set of points lying on or within the hypersphere, that is,

$$\sum_{i=1}^{d} (p_i - c_i)^2 \;\leq\; r^2.$$

In 2-dimensional space, this is often called a *disk*. (Note that the terms "ball" and "disk" refer to the solid object, while "circle," "sphere," and "hypersphere" refer to its boundary.) We will present an algorithm for the MEB problem in $\mathbb{R}^2$, and so we will use the terms "disk" and "ball" to mean the same things.

Before discussing algorithms, we begin with two useful geometric observations. First, three (noncollinear) points in the plane define a unique circle. We will not prove this, but it follows from standard results in algebra. The second observation is presented in the following claim.

Claim: Consider a finite set S of points in the plane such that no four points are cocircular. the minimum enclosing disk either has at least three points on its boundary, or it has two points, and these points form the diameter of the circle. If there are three points then they subdivide the circle bounding the disk into arcs of angle at most π.

Proof: Clearly if there are no points on the boundary the disk's radius can be decreased about its center until a single point lies on the boundary. If there is only one point on the boundary then the disk can be shrunken about this point until a second point is contacted (see Fig. 146(a)). If there are two points contacted, and they are not the diameter of the disk, then between them there must be arc of length greater than π. It follows that there is a family of disks whose centers lie on the perpendicular bisector of these two points. By moving the center closer to the midpoint of these points, we obtain a disk that is smaller and still contains all the points (see Fig. 146(b)). s

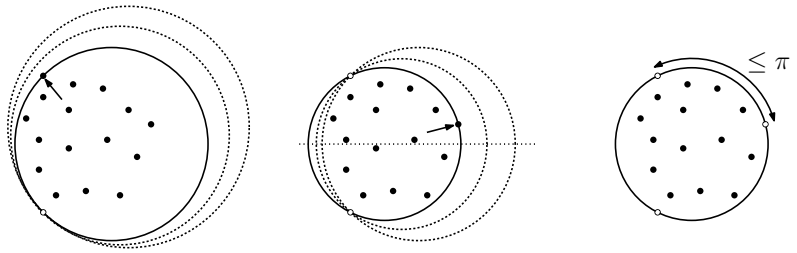


Fig. 146: Contact points for a minimum enclosing disk.

Thus, none of these configurations could be a candidate for the minimum enclosing disk. Also observe that if there are three points that define the minimum enclosing disk they subdivide the circle into three arcs each of angle at most π (see Fig. 146(c)). Because points are in general position we may assume there cannot be four or more cocircular points.

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

Linearization: We cannot solve the MEB problem by a direct reduction to LP. In this section we'll discuss an approach that "almost" reduces the planar MEB problem to a linear programming problem in 3-space. This serves to illustrate the similarity between this problem and LP.

Recall that in the MEB problem in $\mathbb{R}^2$ we are given a set $P = \{p_1, \ldots, p_n\}$, where $p_i = (p_{i,x}, p_{i,y})$. These points are contained within a circle centered at point c and radius r if and only if

$$(p_{i,x} - c_x)^2 + (p_{i,y} - c_y)^2 \leq r^2, \quad \text{for } 1 \leq i \leq n.$$

We are asked to determine whether there exists c_x, c_y and r (with r as small as possible) satisfying these n inequalities. 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. For each inequality, let us expand it and rearrange the terms, yielding:

$$\begin{aligned} p_{i,x}^2 - 2p_{i,x}c_x + c_x^2 + p_{i,y}^2 - 2p_{i,y}c_y + c_y^2 &\leq r^2 \\ 2p_{i,x}c_x + 2p_{i,y}c_y + (r^2 - c_x^2 - c_y^2) &\geq p_{i,x}^2 + p_{i,y}^2. \end{aligned}$$

Now, by introducing a new variable $R = r^2 - c_x^2 - c_y^2$, we have

$$(2p_{i,x})c_x + (2p_{i,y})c_y + R \;\geq\; (p_{i,x}^2 + p_{i,y}^2).$$

Observe that we now have n *linear inequalities* in three variables c_x, c_y and R. (We have effectively replaced r with R.)

Great! We can apply linear programming to find the solution—or can we? The problem is that the previous objective function was to minimize r. But r is no longer a parameter in the

new version of the problem. Observe that $r^2 = R + c_x^2 + c_y^2$, and minimizing r is equivalent to minimizing r^2, we could say that the objective is to minimize $R + c_x^2 + c_y^2$. Unfortunately, this is a *nonlinear* function of the variables c_x, c_y and R. In summary, we have introduced a change of variables that make the constraints linear, but the objective function is no longer linear. Thus, this is not an instance of LP, and it would seem that we are back to square-one.

Randomized Incremental Algorithm: Even though the linearized problem is not an instance of LP, we will show here that Seidel's randomized incremental algorithm can be adapted to solve it nonetheless.

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 B_{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 B_{i-1}. If so, then we set $B_i = B_{i-1}$ and go on to the next stage. If not, then we need to update the current disk to contain p_i, letting B_i denote the result. When the last point is inserted we output B_n.

How do we compute this updated disk? It might be tempting at first to say that we just need to compute the minimum 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 need some way to reduce the "dimensionality" of the problem.

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 $p_i \notin B_{i-1}$ then p_i is on the boundary of the minimum enclosing disk for the first i points, B_i.

Proof: The proof makes use of the following geometric observation. Given two intersecting disks B_1 and B_2 of radii r_1 and r_2, respectively, where $r_1 < r_2$, the portion of B_2's boundary that lies within B_1 is an arc of angle less than π. To see why, observe that if the arc were of angular extent greater than π it would contain two diametrically opposite points. But these points would be at distance $2r_2$ from each other, which exceeds B_1's diameter.

Now, suppose to the contrary that p_i is not on the boundary of B_i. Let r_{i-1} and r_i denote the radii of B_{i-1} and B_i, respectively. Because B_i covers a point that is not covered by B_{i-1} it follows that $r_{i-1} < r_i$. By the above observation, the portion of B_i's boundary that lies within B_{i-1} is an arc of angle less than π (the heavy curve in Fig. 147).

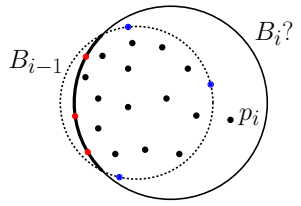


Fig. 147: The proof of the claim.

Since p_i is not on the boundary of B_i, the points defining B_i must be chosen from among the first $i-1$ points, from which it follows that they all lie within this arc (the red points

in Fig. 147). This would imply that between two of the points is an arc of angle greater than π (the arc not shown with a heavy line) which, by the earlier claim cannot be a minimum enclosing disk.

Aided with this observation, the we can derive an algorithm is similar in structure to the LP algorithm. First, we 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 solving the *1-point restricted MEB problem*, namely, we compute the MEB 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 lies on the boundary is analogous to the constraint used in linear programming that optimum vertex lie on the line supporting the current halfplane.) The procedure is called MinDiskWith1Pt(P, q), and is given a point set P and a constraint point $q \notin P$ that must be on the boundary of the final answer.

The constrained problem is solved in exactly the same manner, but with the change that whenever we detect a point p that lies outside the current disk, we invoke the *2-point restricted MEB problem*, namely, we compute the MEB that contains all the points $\{p_1, \ldots, p_{i-1}\}$ and is constrained to have both q and p_i on its boundary. The procedure is called MinDiskWith2Pt(P, q_1, q_2). Note that we do not need to define a 3-point restricted MEB problem, since three points uniquely determine a circle.

Minimum Enclosing Disk

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 B_2 be the minimum disk enclosing $\{p_1, p_2\}$.
(3) for $i \leftarrow 3$ to $|P|$ do
  (a) if $p_i \in B_{i-1}$ then $B_i \leftarrow B_{i-1}$.
  (a) else $B_i \leftarrow$ MinDiskWith1Pt$(P[1..i-1], p_i)$.

MinDiskWith1Pt(P, q) :

(1) Randomly permute the points in P. Let B_1 be the minimum disk enclosing $\{q, p_1\}$.
(2) for $i \leftarrow 2$ to $|P|$ do
  (a) if $p_i \in B_{i-1}$ then $B_i \leftarrow B_{i-1}$.
  (a) else $B_i \leftarrow$ MinDiskWith2Pts$(P[1..i-1], q, p_i)$.

MinDiskWith2Pts(P, q_1, q_2) :

(1) Randomly permute the points in P. Let B_0 be the minimum disk enclosing $\{q_1, q_2\}$.
(2) for $i \leftarrow 1$ to $|P|$ do
  (a) if $p_i \in B_{i-1}$ then $B_i \leftarrow B_{i-1}$.
  (a) else $B_i \leftarrow$ Disk(q_1, q_2, p_i).

LP-Type: The above reduction shows that the MEB problem is closely related to LP. There are in fact a number of related problems, like MEB, in which the incremental approach can be applied. This concept was described formally by Sharir and Welzl, in which they introduced the notion of *LP-type* problems. The input is given as a finite set S of elements, and there is an objective function f that maps subsets of S to values from a totally ordered set. (For example, think of f as the function that maps a set of points to the radius of their minimum enclosing disk.) The objective function is required to satisfy two key properties:

Monotonicity: For sets $A \subseteq B \subseteq S$, $f(A) \leq f(B) \leq f(S)$. That is, adding elements increases the objective function.

Locality: For sets $A \subseteq B \subseteq S$ and every $x \in S$, if $f(A) = f(B) = f(A \cup \{x\})$, then $f(A) = f(B \cup \{x\})$. Intuitively, if x is redundant for A, it is redundant for every superset of A. (For example, if x lies within the minimum disk enclosing the points of A, then it lies in the minimum disk enclosing any superset B of A.)

The randomized incremental LP algorithm (due to Seidel) that we introduced earlier can be readily generalized to handle LP-type problems.

Lecture 22: Kirkpatrick's Planar Point Location

Point Location: In *point location* we are given a polygonal subdivision (formally, a cell complex). The objective is to preprocess this subdivision into a data structure 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. Let n denote the number of vertices of the subdivision. By Euler's formula, the number of edges and faces are $O(n)$. 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.

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 (see Fig. 148). 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,

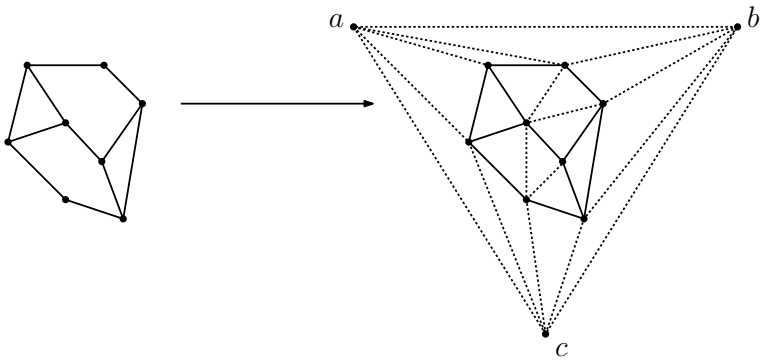


Fig. 148: Triangulation of a planar subdivision.

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

Let T_0 denote the initial triangulation. What Kirkpatrick's method does is to produce a sequence of triangulations, $T_0, T_1, T_2, \ldots, T_k$, where $k = O(\log n)$, such that T_k consists only of a single triangle (the exterior face of T_0), and each triangle in T_{i+1} overlaps a constant number of triangles in T_i.

We will see how to use such a structure for point location queries later, but for now let us concentrate on how to build such a sequence of triangulations. Assuming that we have T_i, we wish to compute T_{i+1}. In order to guarantee that this process will terminate after $O(\log n)$ stages, we will want to make sure that the number of vertices in T_{i+1} decreases by some constant factor from the number of vertices in T_i. In particular, this will be done by carefully selecting a subset of vertices of T_i and deleting them (and along with them, all the edges attached to them). After these vertices have been deleted, we need retriangulate the resulting graph to form T_{i+1}. The question is: How do we select the vertices of T_i to delete, so that each triangle of T_{i+1} overlaps only a constant number of triangles in T_i?

There are two things that Kirkpatrick observed at this point, that make the whole scheme work.

Constant degree: We will make sure that each of the vertices that we delete have constant ($\leq d$) degree (that is, each is adjacent to at most d edges). Note that the when we delete such a vertex, the resulting *hole* will consist of at most $d-2$ triangles. When we retriangulate, each of the new triangles, can overlap at most d triangles in the previous triangulation.

Independent set: We will make sure that no two of the vertices that are deleted are adjacent to each other, that is, the vertices to be deleted form an *independent set* in the current planar graph T_i. This will make retriangulation easier, because when we remove m independent vertices (and their incident edges), we create m independent *holes* (non triangular faces) in the subdivision, which we will have to retriangulate. However, each of these holes can be triangulated independently of one another. (Since each hole contains a constant number of vertices, we can use any 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 eight, 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 eight. 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 eight 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 three 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 (new) 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 (see Fig. 149).

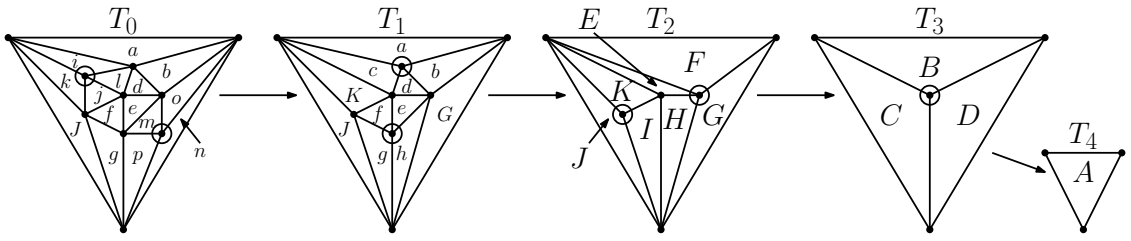


Fig. 149: Decomposing and triangulation by repeatedly removing an independent set and re-triangulating.

Each node corresponding to a triangle in triangulation T_{i+1}, stores pointers to all the triangles it overlaps in T_i. Since there are at most eight of these, the structure has bounded degree. 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 (see Fig. 150).

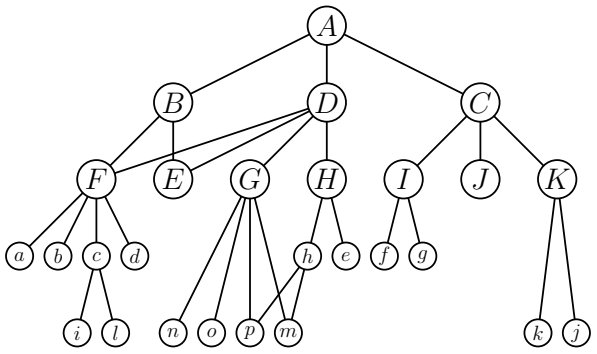


Fig. 150: Kirkpatrick's point location structure.

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 eight) 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.

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 eight point-in-triangle tests), thus the total query time is $O(\log n)$. The size of the data structure is the sum of sizes of the triangulations. Since the number of triangles in a triangulation is proportional to the number of vertices, it follows that the size is proportional to

$$n(1 + 17/18 + (17/18)^2 + (17/18)^3 + \ldots) \leq 18n,$$

(using standard formulas for geometric series). Thus the data structure size is $O(n)$ (again, with a pretty hefty constant).

The last thing that remains is to show how to construct the independent set of the appropriate size. We first present the algorithm for finding the independent set, and then prove the bound on its size.

(1) Mark all nodes of degree ≥ 9.

(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 six, there must be a lot of vertices of degree at most eight (otherwise the average would be unattainable). Second, whenever we add one of these vertices to our independent set, only eight 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 eight or less. To see why, suppose to the contrary that there were more than $n/2$ vertices of degree nine or greater. The remaining vertices must have degree at least three (with the possible exception of the three 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 eight or fewer, we mark at most nine new vertices (this node and at most eight 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 23: Topological Plane Sweep

Topological Plane Sweep: In previous lectures we have introduced arrangements of lines in the plane and how to construct them. In this lecture we present an efficient algorithm for sweeping an arrangement of lines. Since an arrangement of n lines has size $O(n^2)$, and since there are $O(n^2)$ events to be processed, each involving an $O(\log n)$ heap deletion, this typically leads to algorithms running in $O(n^2 \log n)$ time, using $O(n^2)$ space. It is natural to ask whether we can dispense with the additional $O(\log n)$ factor in running time, and whether we need all of $O(n^2)$ space (since in theory we only need access to the current $O(n)$ contents of the sweep line).

We discuss a variation of plane sweep called *topological plane sweep*. This method runs in $O(n^2)$ time, and uses only $O(n)$ space (by essentially constructing only the portion of the arrangement that we need at any point). Although it may appear to be somewhat sophisticated, it can be implemented quite efficiently, and is claimed to outperform conventional plane sweep on arrangements of any significant size (e.g. over 20 lines).

Cuts and topological lines: The algorithm is called *topological* plane sweep because we do not sweep a straight vertical line through the arrangement, but rather we sweep a curved *topological line* that has the essential properties of a vertical sweep line in the sense that this line intersects each line of the arrangement exactly once. The notion of a topological line is an intuitive one, but it can be made formal in the form of something called a *cut*. Recall that the faces of the arrangement are convex polygons (possibly unbounded). (Assuming no vertical lines) the edges incident to each face can naturally be partitioned into the edges that are *above* the face, and those that are *below* the face. Define a *cut* in an arrangement to be a sequence of edges $c_1, c_2, \ldots, c_n$, in the arrangement, one taken from each line of the arrangement, such that for $1 \le i \le n-1$, c_i and c_{i+1} are incident to the same face of the arrangement, and c_i is above the face and c_{i+1} is below the face (see Fig. 151).

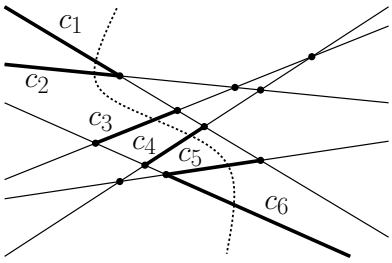


Fig. 151: Topological line and associated cut.

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 (see Fig. 152). Observe that on doing so these two lines swap in their order along the sweep line.

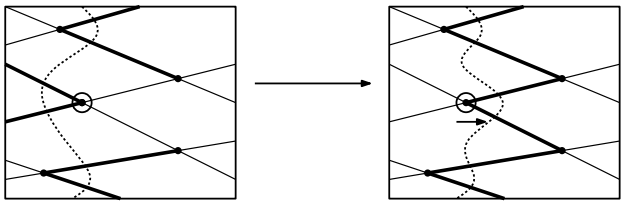


Fig. 152: Elementary step.

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.

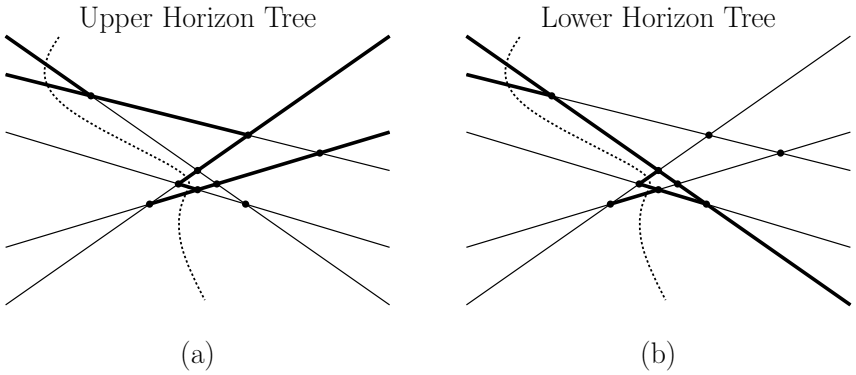


Fig. 153: Upper and lower horizon trees.

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 Fig. 154(a)). 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.

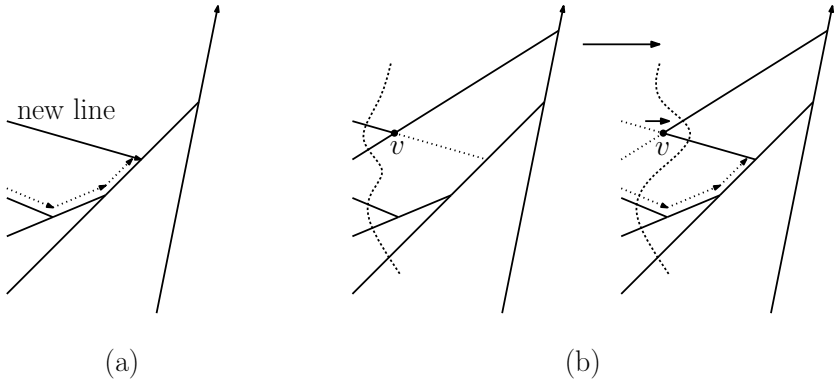


Fig. 154: Constructing (a) and updating (b) the UHT.

Next we show how to update the UHT after an elementary step. The process is quite similar (see Fig. 154(b)). 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 24: 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 (see Fig. 155(a) and (b)). (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.

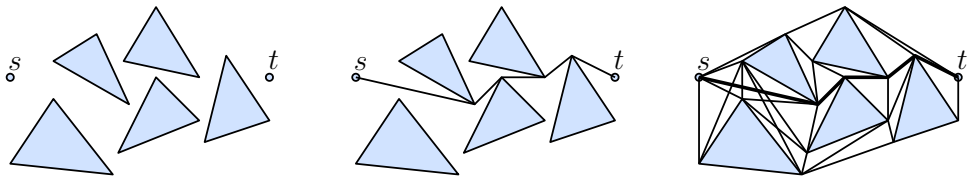


Fig. 155: Shortest paths and the visibility graph.

Claim: The shortest path between any two points that avoids a set of polygonal obstacles is a polygonal curve, whose vertices are either vertices of the obstacles or the points s and t.

Proof: We show that any path π that violates these conditions can be replaced by a slightly shorter path from s to t. Since the obstacles are polygonal, if the path were not a polygonal curve, then there must be some point p in the interior of free space, such that the path passing through p is not locally a line segment. If we consider any small neighborhood about p (small enough to not contain s or t or any part of any obstacle), then since the shortest path is not locally straight, we can shorten it slightly by replacing this curved segment by a straight line segment jointing one end to the other. Thus, π is not shortest, a contradiction.

Thus π is a polygonal path. Suppose that it contained a vertex v that was not an obstacle vertex. Again we consider a small neighbor hood about v that contains no part of any obstacle. We can shorten the path, as above, implying that π is not a shortest path.

From this it follows that the edges that constitute the shortest path must travel between s and t and vertices of the obstacles. Each of these edges must have the property that it does

not intersect the interior of any obstacle, implying that the endpoints must be visible to each other. More formally, we say that two points p and q are *mutually visible* if the open line segment joining them does not intersect the interior of any obstacle. By this definition, the two endpoints of an obstacle edge are not mutually visible, so we will explicitly allow for this case in the definition below.

Definition: The *visibility graph* of s and t and the obstacle set is a graph whose vertices are s and t the obstacle vertices, and vertices v and w are joined by an edge if v and w are either mutually visible or if (v, w) is an edge of some obstacle (see Fig. 155(c)).

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 (see the leftmost part of Fig. 156). Now, we imagine that angle θ continuously sweeps out all slopes from $-\infty$ to $+\infty$. Imagine that all the bullet lines attached to all the vertices begin to turn slowly counterclockwise. If we play the mind experiment of visualizing the rotation of these bullet paths, the question is what are the significant event points, and what happens with each event? As the sweep proceeds, we will eventually determine everything that is visible from every vertex in every direction. Thus, it should be an easy matter to piece together the edges of the visibility graph as we go.

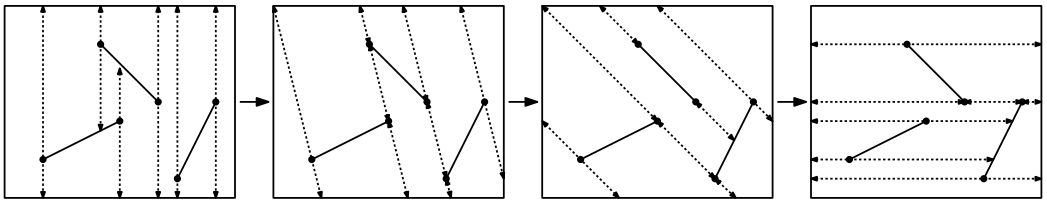


Fig. 156: Visibility graph by multiple angular sweep.

Let us consider this "multiple angular sweep" in greater detail.

It is useful to view the problem both in its primal and dual form. For each of the $2n$ segment endpoints $v = (v_a, v_b)$, we consider its dual line $v^* : y = v_a x - v_b$. Observe that a significant event occurs whenever a bullet path in the primal plane jumps from one line segment to another. This occurs when θ reaches the slope of the line joining two visible endpoints v and w. Unfortunately, it is somewhat complicated to keep track of which endpoints are visible and which are not (although if we could do so it would lead to a more efficient algorithm). Instead we will take events to be *all* angles θ between two endpoints, whether they are visible or not. By duality, the slope of such an event will correspond to the a-coordinate of the intersection of dual lines v^* and w^* in the dual arrangement. (Convince yourself of this.) Thus, by sweeping the arrangement of the $2n$ dual lines from left-to-right, we will enumerate all the slope events in angular order.

Next let's consider what happens at each event point. Consider the state of the angular sweep algorithm for some slope θ. For each vertex v, there are two bullet paths emanating from v along the line with slope θ. Call one the *forward bullet path* and the other the *backward bullet path*. Let $f(v)$ and $b(v)$ denote the line segments that these bullet paths hit, respectively. If either path does not hit any segment then we store a special null value. As θ varies the following events can occur. Assuming (through symbolic perturbation) that each slope is determined by exactly two lines, whenever we arrive at an events slope θ there are exactly two vertices v and w that are involved. Here are the possible scenarios:

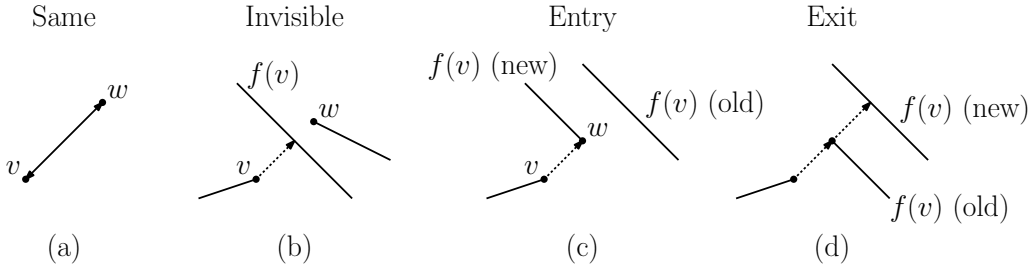


Fig. 157: Possible events.

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 (see Fig. 157(a)).

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 $f(v)$ (see Fig. 157(b)). (The case of $b(v)$ is symmetrical).

Compute the contact point of the bullet path shot from v in direction θ with segment $f(v)$. If this path hits $f(v)$ strictly before w, then we know that w is not visible to v,

and so this is a "non-event".

Segment entry: Consider the segment that is incident to w. Either the sweep is just about to enter this segment or is just leaving it. If we are entering the segment, then we set $f(v)$ to this segment (see Fig. 157(c)).

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 (see Fig. 157(d)).

In particular, since we are sweeping over w at the same time that we are sweeping over v. Thus we know that the bullet extension from w hits $f(w)$. All we need to do is to set $f(v) = f(w)$.

This is a pretty simple algorithm (although there are a number of cases). The only information that we need to keep track of is (1) a priority queue for the events, and (2) the $f(v)$ and $b(v)$ pointers for each vertex v. The priority queue is not stored explicitly. Instead it is available from the line arrangement of the duals of the line segment vertices. By performing a topological sweep of the arrangement, we can process all of these events in $O(n^2)$ time. (There a few technical details in the implementation of the topological plane sweep, but we will ignore them.)

Lecture 25: 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 (see Fig. 158(a)). 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, that is, it does not include it endpoints, and that each face is open, that is, it does not include its boundary. There is always at least one unbounded face, which 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 themselves contain holes). A subdivision is called a *convex subdivision* if all the faces (except the outer one) are convex (see Fig. 158(b)).

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 (see Fig. 159). The endpoints of the edges are *vertices*. For example, let $v_0, \ldots, v_n$ denote the set of $n+1$ vertices, and let $e_1, \ldots, e_n$ 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 $v_0 = v_n$.

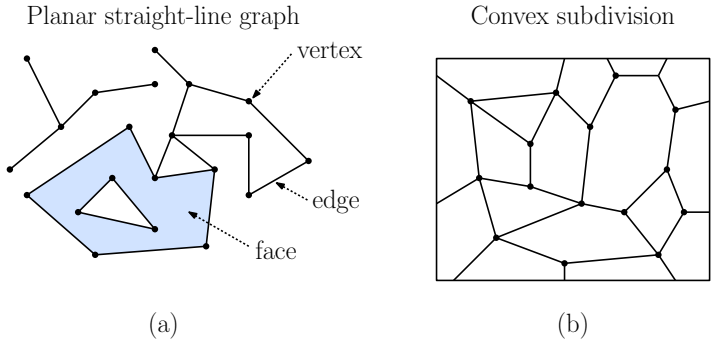


Fig. 158: Planar straight-line subdivision.

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.

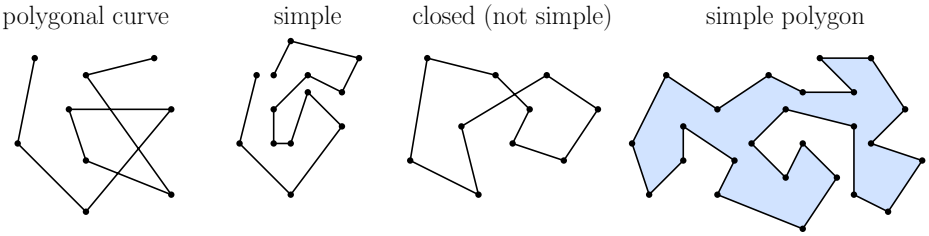


Fig. 159: Polygonal curves

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 *simple polygon* (or just *polygon*) to be the region of the plane bounded by a simple, closed 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. Such a set is called a *guarding set* (see Fig. 160(a)). Victor Klee posed the following question: Suppose we have an art gallery whose floor plan can be modeled as a polygon with n vertices. As a function of n, what is the minimum number of guards that suffice to guard such a gallery? Observe that are you are told about the polygon is the number of sides, not its actual structure. We want to know the fewest number of guards that suffice to guard *all* polygons with n sides.

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

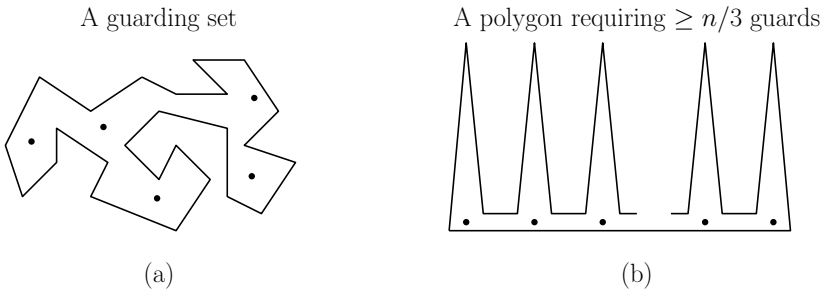


Fig. 160: Guarding sets.

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. Fig. 160(b)) shows that this bound is tight. Observe given such a polygon of this form with k "teeth", the number of vertices is $n = 3k$, and each guard can see into only one tooth. 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 (see Fig. 161(a)). 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 non-crossing diagonals.

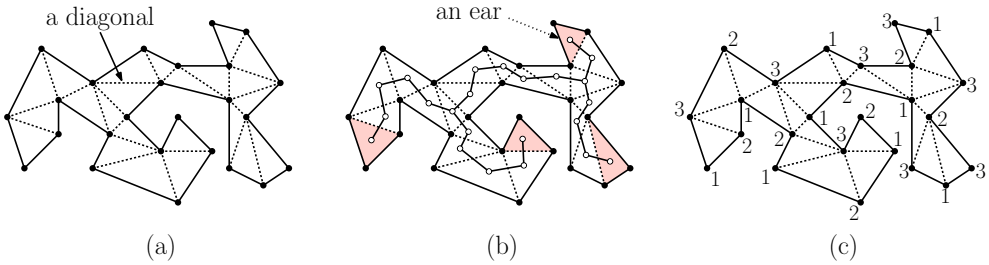


Fig. 161: (a) A polygon triangulation, (b) the dual tree (with ears shaded), and (c) the resulting 3-coloring.

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

We will leave the details as an exercise, but here is a quick sketch of the proof. We start with the observation that given any n-vertex polygon, with $n \geq 4$ it has at least one 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 to determine the number of of diagonals.

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

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

Proof: For every planar graph G there is another planar graph G^* called its *dual*. The dual G^* is the graph whose vertices are the faces of G, and two vertices of G^* are connected by an edge if the two corresponding faces of G share a common edge (see Fig. 161(b)).

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 three. (This would not be true if the polygon had holes.)

The coloring will be performed inductively. If the polygon consists of a single triangle, then just assign any three 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 (see Fig. 161(c)).

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 move each guard infinitesimally out from his vertex, and so guard the entire polygon.

Lecture 26: 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 a number of algorithms for computing Voronoi diagrams and Delaunay triangulations in the plane. These include:

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 Voronoi diagrams of line segments with some difficulty.

Randomized Incremental: (For DT.) 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 two roughly equal sized groups, e.g., by presorting the points by x-coordinate and selecting medians (see Fig. 162(a)). We compute the Voronoi diagram of the left side, and the Voronoi diagram of the right side (see Fig. 162(b)). Note that since each diagram alone covers the entire plane, these two diagrams overlap (see Fig. 162(c)). 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 $\text{Vor}(L)$ and one in $\text{Vor}(R)$. 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.

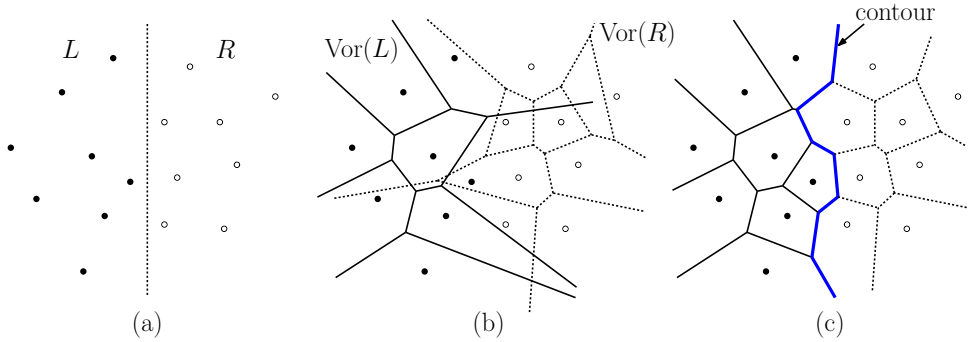


Fig. 162: Voronoi diagrams by divide-and-conquer.

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 two 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 Vor(L) and Vor(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 Vor(L) that is to the right of the contour, and the portion of Vor(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 four points are cocircular.

Lemma: The contour consists of a single polygonal curve (whose first and last edges are semi-infinite) 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 two 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 four points are cocircular, it follows that all vertices in the Voronoi diagram can have degree at most three. However, because the contour separates the plane into only two types of regions, it can contain only vertices of degree two. 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. We start tracing the contour between l_0 and r_0 (see Fig. 163). By walking along the boundary of $V(l_0)$ we can determine the edge that the contour hits 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.

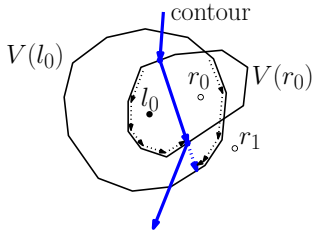


Fig. 163: Tracing the contour.

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 27: Multidimensional Polytopes and Convex Hulls

Polytopes: In this lecture we present basic facts about convex polytopes in dimensions three and higher. Although for beings dwelling in 3-dimensional space, spaces of high dimension may seem rather esoteric, but there are many problems in mathematics that can be reduced to the analysis of polytopes in dimensions much higher then the familiar three. Unfortunately for us, our intuitions about space have developed in these lower dimensions, and it requires a bit of imagination to see how familiar 3-dimensional concepts generalize to higher dimensions.

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.

A *simplex* (or *k-simplex*) is defined to be the convex hull of $k+1$ affinely independent points. 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 (see Fig. 164). Observe that a k-simplex is the smallest (in terms of number of vertices) convex polytope that is k-dimensional.

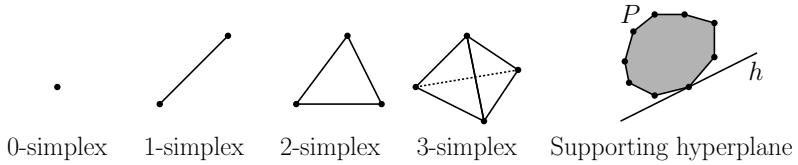
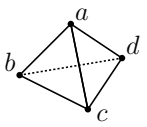


Fig. 164: Simplices and supporting hyperplane.

Faces: The boundary of a polyhedron in 3-space is bounded by vertices, edges, and faces. To generalize this to higher dimensions, let us first introduce a few definitions. 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 $\overline{h^+}$ (see Fig. 164). 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.)



Vertices: a, b, c, d
Edges: ab, ac, ad, bc, bd, cd
Faces: abc, abd, acd, bcd

Fig. 165: A tetrahedron and its proper faces.

The faces of dimensions 0 to $d-1$ are called *proper faces* (see Fig. 165). 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 on. We have the following useful obervation.

Observation: 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 can we represent the boundary structure of a polytope? In addition to the geometric 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 *DCEL*, *winged-edge*, *quad-edge*, and *half-edge* data structures.)

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. 166 for a tetrahedron. Note the similarity between this graph and the lattice of subsets based on inclusion relation.

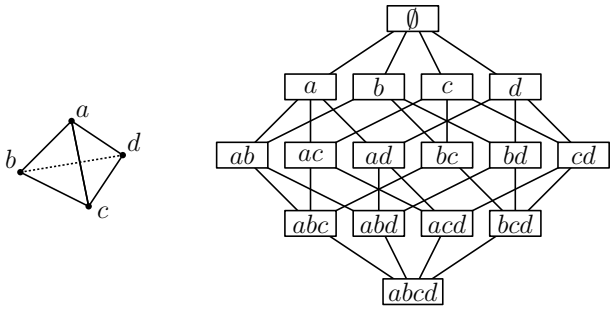


Fig. 166: The incidence graph for a tetrahedron.

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). As we shall see, these two concepts are essentially identical, and they are connected through the concept of the *polar transformation*, which maps points to hyperplanes and vice versa. (We have seem the projective dual transformation earlier this semester, which maps a point $p = (a, b)$ to the line $y = ax - b$. The polar is just another example of duality.)

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 $p - O$.) Given two vectors p and v, let $(p \cdot x)$ denote the standard vector *dot-product*: $(p \cdot x) = p_1x_1 + \cdots + p_dx_d$. The *polar hyperplane* of p, denoted p^* is defined to be the set

$$p^\circ \;=\; \{x \in \mathbb{R}^d \mid (p \cdot x) = 1\}.$$

Clearly, this is a linear equation in the coordinates of x, and therefore p° is a hyperplane in $\mathbb{R}^d$. 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 $\overrightarrow{Op}$. 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.

As with the projective dual, the polar transformation satisfies certain incidence and inclusion properties involving points and hyperplanes. Now, let h be any hyperplane that does not contain O. The *pole* of h, denoted h° is the point that satisfies $(h^\circ \cdot x) = 1$, for all $x \in h$ (see Fig. 167(a)).

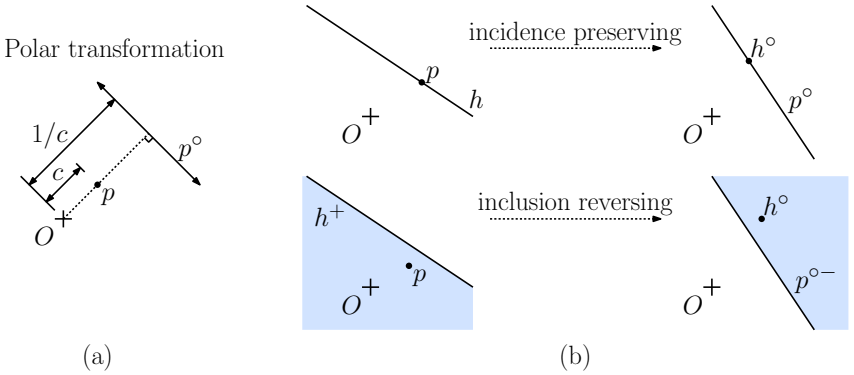


Fig. 167: The polar transformation and its properties.

Clearly, the polar transformation is an involution, that is, $(p^\circ)^\circ = p$ (assuming $p \neq O$) and

$(h^\circ)^\circ = h$ (assuming h does not pass through the origin). The polar transformation preserves important geometric relationships. Given a hyperplane h, define

$$h^- \;=\; \{x \in \mathbb{R}^d \mid (x \cdot h^\circ) < 1\} \qquad h^+ \;=\; \{x \in \mathbb{R}^d \mid (x \cdot h^\circ) > 1\}.$$

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 hyperplanes. That is, p belongs to h if and only if h° belongs to p° (see Fig. 167(b)).

Inclusion Reversing: The polarity transformation reverses relative position relationships in the sense that p belongs to h^- if and only if h° belongs to $(p^\circ)^+$, and p belongs to h^+ if and only if h° belongs to $(p^\circ)^-$ (see Fig. 167(b)).

(In general, any bijective transformation that preserves incidence relations is called a *duality*. The above claim implies that polarity is a duality.)

Convex Hulls and Halfspace Intersection: 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. Given an convex set $P \in \mathbb{R}^d$, define

$$P^\circ \;=\; \{x \in \mathbb{R}^d \mid (x \cdot p) \le 1, \forall p \in P\}.$$

The polar is always closed, convex, and contains the origin. If P is a convex body with O in its interior, then P° is a convex body containing O in its interior and $(P^\circ)^\circ = P$. Moreover, polarity reverses inclusion; if $P_1 \subseteq P_2$, then $P_2^\circ \subseteq P_1^\circ$.

This polarity of convex bodies is closely related to the pole operation defined earlier. The following is easily verified and is illustrated in Fig. 168.

Lemma: If P is a convex polytope containing the origin in its interior with vertex set $\{v_1, \ldots, v_j\}$ and whose facets lie on the hyperplanes $\{h_1, \ldots, h_k\}$ then P° is a convex polytope with vertex set $\{h_1^\circ, \ldots, h_k^\circ\}$ and whose facets lie on the hyperplanes $\{v_1^\circ, \ldots, v_j^\circ\}$.

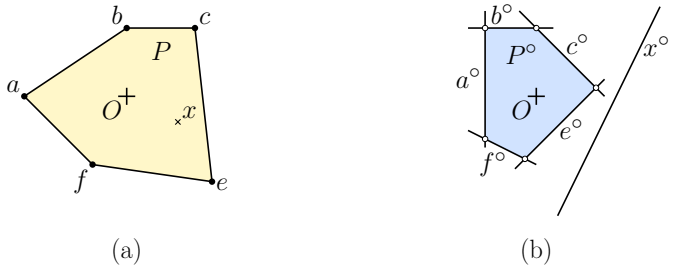


Fig. 168: The polar image of a convex polytope.

Observe that, following the order-reversing property, a point $x \in P$ is mapped through the polar to a hyperplane x° that lies entirely outside of P°, and vice versa (see Fig. 168). 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!)

Simple and Simplicial Polytopes: Our next objective is to investigate the relationship between the number of vertices and number of facets on a convex polytope. Earlier in the semester we saw that a 3-dimensional polyhedron with n vertices has $O(n)$ edges and faces. This was a consequence of Euler's formula. In order to investigate the generalization of this to higher dimensions, we begin with some definitions. A polytope is *simplicial* if all its proper faces are simplices (see Fig. 169(a)). 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. (For example, in $\mathbb{R}^3$ a face with four vertices would imply that these four points are coplanar, which would violate general position.)

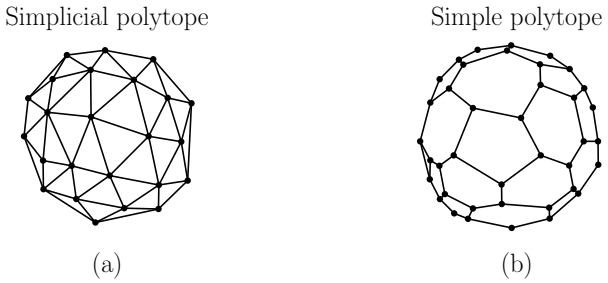


Fig. 169: Simplicial and simple polytopes.

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 (see Fig. 169(b)). 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.

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 half-spaces 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 at 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. 170 for an example in $\mathbb{R}^5$.) 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.)

Since the local neighborhood about v is a simplex, there is a face of dimension at least $\lceil d/2 \rceil$ that spans these edges and is incident to v (this is the 3-face lying above v in Fig. 170). Therefore, v is either the lowest or highest vertex for this face. We assess v's charge to this face. 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).

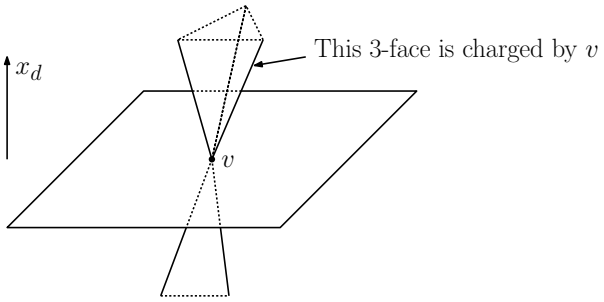


Fig. 170: Proof of the Upper Bound Theorem in $\mathbb{R}^5$. In this case the three edges above v span a 3-face whose lowest vertex is v.

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 28: Orthogonal Range Searching and kd-Trees

Range Searching: In this lecture we will discuss a new data structure problem,, called *range searching*. We are given a set of n points $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$ and a class of range shapes (e.g., rectangles, balls, triangles, halfplanes). The points of P are to be preprocessed and stored in a data structure (whose structure may be based on the knowledge of the allowable range shapes). Given a query range Q from this class, the objective is identify the points of P that lie within Q. The term "identify" can mean a number of things:

Emptiness: If $P \cap Q = \emptyset$, return "empty" else "non-empty"

Counting: Return the count $|P \cap Q|$ of the number of points of P within Q

Weighted counting: Each point $p_i \in P$ has an associated weight $w(p_i)$, and we are to return the weighted sum of points in Q, $\sum_{p \in Q} w(p_i)$.

Semigroup weighted counting: Why limit to addition? For example, how about computing the product of weights or the maximum of the weights?

Generally, weighted counting makes sense as long as you have an operator that is both commutative and associative. Your data structure may also take advantage of special properties of your operator. For example, if the operator has an *inverse*, you have the flexibility both add and subtract weights. If your operator is *idempotent* (that is, $a \oplus a = a$) then there is no harm in counting the same item multiple times.

Reporting: Return a list containing all the points of $P \cap Q$.

Top-k: Report the (up to) k points of $P \cap Q$ based on weight.

Complexity Bounds: Observe that all of the above queries can be answered in $O(n)$ time trivially, by just running through all the points and testing one-by-one whether they lie within Q. You want to think of n as being huge (e.g., millions or more), but you want a query time that is much smaller (say tens to hundreds). Data structures for range searching are usually analyzed in terms of two quantities, space and query time. Ideally, one would like to achieve $O(n)$ space and $O(\log n)$ query time, since this matches the best you can expect in 1-dimensional space. If you cannot achieve this "gold standard," you might hope to do well with one criteria or the other. For example, the query time might be $O(\log n)$, but the space is $O(n^2)$. Alternatively,

the space might be $O(n)$, but the query time is $O(\sqrt{n})$. (Remember that from the perspective of asymptotics, logarithmic query times are always better than polynomial times, thus $\log^c n$ is better than n^b for positive numbers c and b, not matter how large c or how small b.)

Ideally, the query time does not depend on the number of points that lie within the query range. It doesn't matter whether your range contains no point or all the points, the (worst-case) running time depends only on n. The exception is reporting queries, since you need to take time to place the elements in the list. For example, the query time for range reporting might be expressed as $O(k + \log n)$, where $k = |P \cap Q|$ and $n = |P|$.

Another issue is the amount of time that it takes to construct the data structure. Ideally, the construction time should be about the same as the space, perhaps larger by a factor of $O(\log n)$.

Orthogonal Range Queries: In this lecture we will focus on perhaps the most common range search, called an *orthogonal range searching*. In this case a range is defined by axis-parallel rectangles in $\mathbb{R}^2$ and d-dimensional hyperrectangles in $\mathbb{R}^d$. An important property of orthogonal ranges is that they can be expressed as the product of 1-dimensional ranges. Let's assume that a point $p \in \mathbb{R}^d$ is expressed as its coordinate vector $(p_1, \ldots, p_d)$. Given two points $a, b \in \mathbb{R}^d$, where $a_i < b_i$ for $1 \leq i \leq d$, they define an axis-parallel rectangle (see Fig. 171(a))

$$R(a, b) \;=\; \{p \in \mathbb{R}^d : a_i \leq p_i \leq b_i\} \;=\; [a_1, b_1] \times \cdots \times [a_d, b_d]$$

Given any set of points $P = \{p_1, \ldots, p_n\} \in \mathbb{R}^d$ (see Fig. 171(b)), the objective is to preprocess these points into a data structure so that, given any query range $R = R(a, b)$, we can quickly count (or report) the points of $P \cap R$.

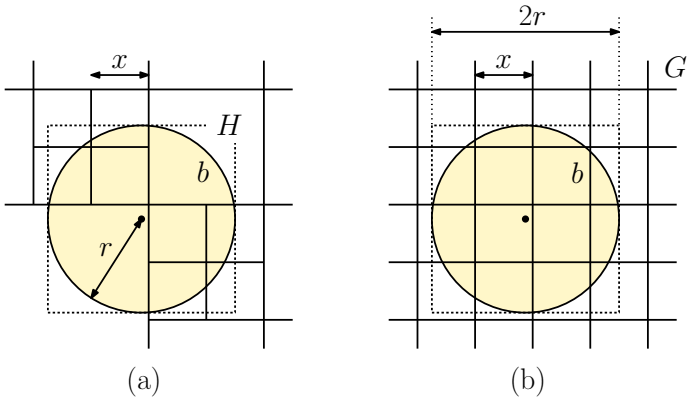


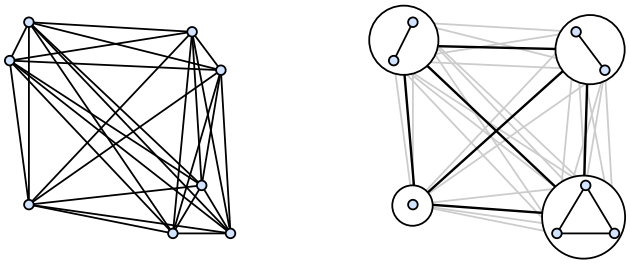
Fig. 171: Orthogonal range query.

Canonical Subsets: A common approach used in solving almost all range queries is to represent P as a collection of *canonical subsets* $\{P_1, \ldots, P_k\}$, 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 (see Fig. 172(a)). Canonical subsets are for conceptual purposes, and need not be stored in the data structure.



28 pairs 11 well-separated pairs

Fig. 172: (a) Canonical subsets for interval queries and (b) internal nodes labeled with the largest key of its left subtree and answering the range query $[x_{lo}, x_{hi}] = [2, 23]$. (Canonical subsets are usually not stored explicitly.)

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 $P = \{p_1, p_2, \ldots, p_n\}$ 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.

Let us consider a different approach, which will generalize to higher dimensions. We sort the points of P in increasing order and store them in the leaves of a balanced binary search tree[29] (see Fig. 172(a)). There are $n - 1$ internal nodes in the tree, and each internal node is labeled with the largest key appearing in its left child. Each internal node is associated with its canonical subset, that is, the subset of points stored in the leaves of the subtree rooted at this node.

Note that the canonical subsets are usually *not* stored explicitly with each node of the tree. If we want to answer range counting queries, we can simply store the count (or weighted count) of the canonical subset in each node. If we want to answer range reporting queries, we can simply traverse the tree to obtain the canonical subset.

[29]For the 1-dimensional case, a tree is not really needed. We could store the points in an array and perform binary search. Unfortunately, this will not generalize to higher dimensions.

Answering range queries: Observe 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 u of the kd-tree is associated with the following quantities:

`u.cut-dim`	the cutting dimension (e.g., $x = 0$ and $y = 1$)
`u.cut-val`	the cutting value (a real number)
`u.weight`	the number (or generally, total weight) of points in u's subtree

In dimension d, the cutting dimension may be represented as in integer ranging from 0 to $d-1$. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to `u.cut-val` are stored in the left subtree and the remaining points are stored in the right subtree (see Fig. 173). 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 `u.point` is this point.

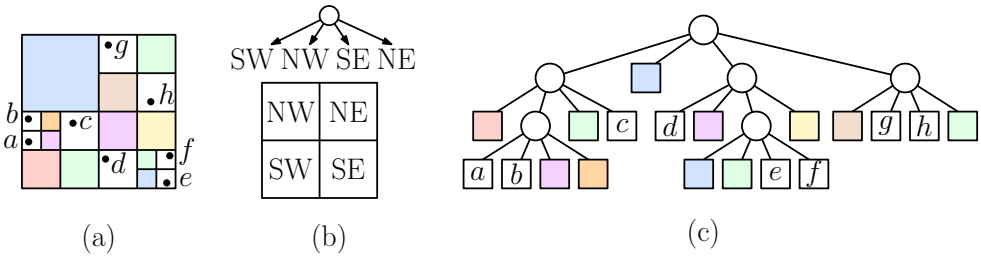


Fig. 173: (a) Point set and kd-subdivision, (b) kd-tree structure, and (c) example of node cells.

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 (see Fig. 173(a)).

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. 173.) 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 *cell*, denoted `u.cell`, which is the associated rectangular region in the hierarchical subdivision. (Cells can either be computed as part of preprocessing or 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 rangeCount(Range Q, KDNode u) {
    if (u.isLeaf)                              // hit the leaf level?
        if (u.point in Q) return u.weight      // count if point lies in range
        else return 0
    else                                       // u is internal
        if (u.cell does not overlap Q)         // disjoint?
            return 0
        else if (u.cell is contained in Q)     // contained in Q?
            return u.weight                    // return total subtree weight
        else return
            rangeCount(Q, u.left) +            // count left side
            rangeCount(Q, u.right)             // ...and right side
}
```

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. 174 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.

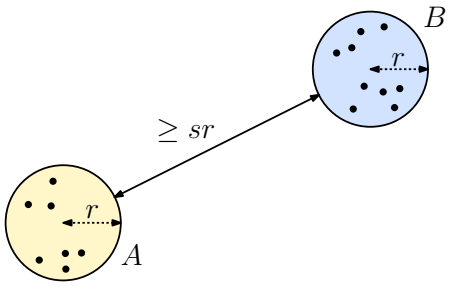


Fig. 174: Range search in a kd-tree. (Note: This particular tree was not generated by the algorithm describeds above.)

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.

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

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 29: Orthogonal Range Trees

Orthogonal Range Trees: In the previous lecture 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*. 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.

1-dimensional Range Tree: Before discussing 2-dimensional range trees, let us first recall the 1-dimensional case. Given a set $P = \{p_1, \ldots, p_n\}$ of scalars, we wish to preprocess them so that given an interval $Q = [Q_{\text{lo}}, Q_{\text{hi}}]$, we can count (or report) all the points that lie in this interval. We begin by storing all the points of our data set in the external nodes (leaves) of any balanced binary search tree sorted by x-coordinates (e.g., an AVL tree). We assume that if an internal node contains a value x_0 then the leaves in the left subtree are strictly less than x_0, and the leaves in the right subtree are greater than or equal to x_0. Each node u in this tree is implicitly associated with it *canonical subset* $P(u) \subseteq P$ of elements of P that are in the leaves descended from u. We assume that for each node u, we store the number of leaves that are descended from u, denoted u.size. Thus u.size is the number of elements in $P(u)$.

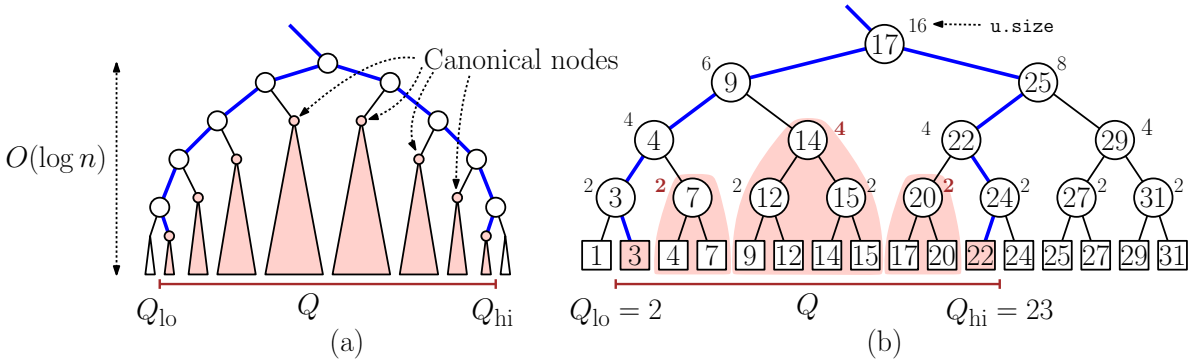


Fig. 175: The canonical subtrees (shaded in red) for the query $[2, 23]$.

Given the query interval $Q = [Q_{\text{lo}}, Q_{\text{hi}}]$, we identify all the maximal subtrees u such that the canonical set associated with u lies entirely within Q (see Fig. 175(a)). We return the sum of sizes of all these subtrees (see Fig. 175(b))

We claim that the nodes identifying the canonical subsets can be identified in $O(\log n)$ time from this structure. Intuitively, we search the tree to find the leftmost leaf u whose key is greater than or equal to Q_{lo} and the rightmost leaf v whose key is less than or equal to Q_{hi}. Clearly all the leaves between u and v (including u and v) constitute the points that lie within the range. Since these two paths are of length at most $O(\log n)$, there are at most $O(2 \log n)$ such trees possible, which is $O(\log n)$. To form the canonical subsets, we take the subsets of all the *maximal subtrees* lying between u and v.

To perform this search, we maintain for each node u a *cell* $C = [x_0, x_1]$, which in this 1-dimensional case is just an interval running from the leftmost leaf key to the rightmost leaf key in this subtree.

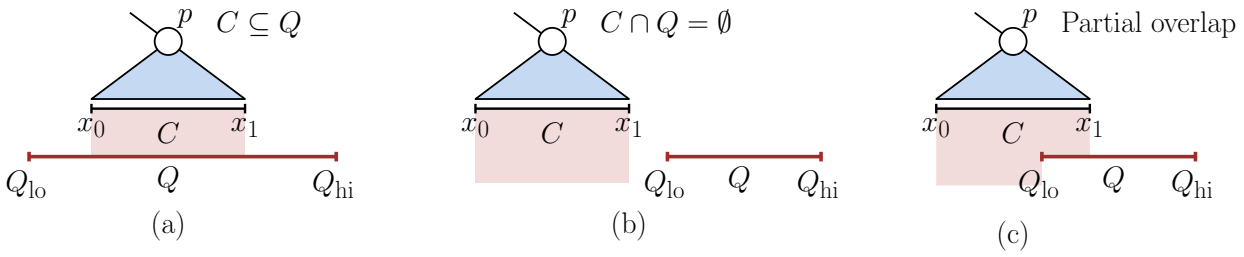


Fig. 176: Cases arising in 1-dimensional range searching.

The recursive search procedure is shown in the code block below. Its arguments to the

procedure are the current node u, the range interval Q, and the current cell C. Let $C_0 = [-\infty, +\infty]$ be the initial cell for the root node (or generally, any interval large enough to contain all the points). The initial call is `range1D(root, Q, C0)`. As with the kd-tree, there are three cases. If $Q \cap C = \emptyset$, there is no overlap and we return 0. If $C \subseteq Q$, then all the points of the canononical set are included in the count (see Fig. 176). Otherwise, we recurse on the left and right subtrees.

1-Dimensional Range Counting Query

```
int range1Dx(Node u, Range Q, Interval C=[x0,x1]) {
    if (u is a leaf)                              // hit the leaf level?
        return (Q contains u.point ? 1 : 0)       // count if point in range
    else if (Q contains C )                       // Q contains entire cell?
        return u.size                             // return entire subtree size
    else if (Q is disjoint from C)                // no overlap
        return 0
    else
        return range1Dx(u.left,  Q, [x0, u.x]) + // count left side
               range1Dx(u.right, Q, [u.x, x1])   // and right side
}
```

Lemma: Given a (balanced) 1-dimensional range tree and any query range Q, in $O(\log n)$ time we can compute a set of $O(\log n)$ canonical nodes u, such that the answer to the query is the disjoint union of the associated canonical subsets $P(u)$.

Multi-layered Search Trees: The orthogonal range-tree data structure is a nice example of a more general concept, called a *multi-layered 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. 177). 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. 177, $P(u_6) = \{p_5, \ldots, p_8\}$.) 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. 177). 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.

Now, given a query range $Q = Q_1 \cap Q_2$, where $Q_i \in \mathcal{R}_i$, we answer queries as follows. Recall

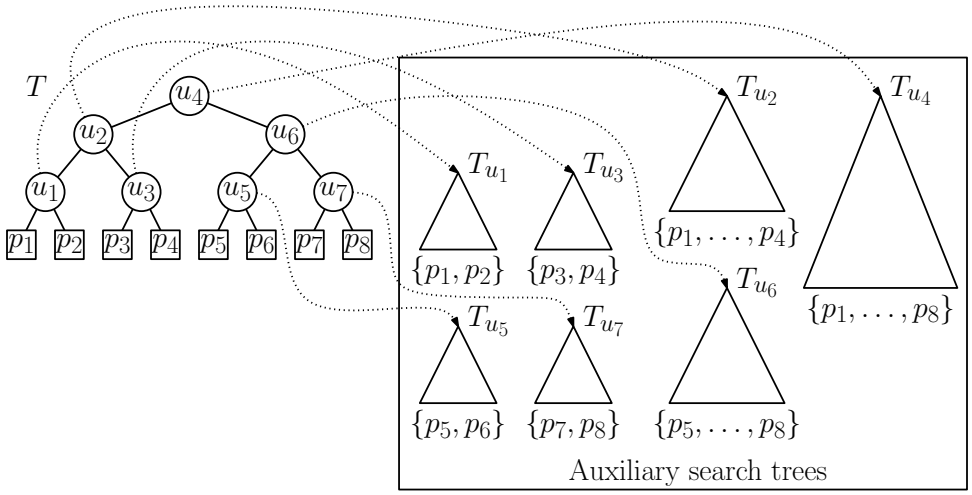


Fig. 177: Multi-layered search: A search the primary tree T identifies auxiliary trees T_{u_i} to search.

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 $u \in T$, whose associated canonical subsets form a partition of $P \cap Q_1$. 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. 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.

Orthogonal Range Trees: Now let us consider how to answer range queries in 2-dimensional space. We first create 1-dimensional tree T as described in the previous section sorted by the x-coordinate (see the left side of Fig. 178). For each internal node u of T, recall that $P(u)$ denotes the canonical set of points associated with the leaves descended from u. For each node u of this tree we build a 1-dimensional range tree for the points of $P(u)$, but sorted on y-coordinates (see the right side of Fig. 178). This called the *auxiliary tree* associated with u. Thus, there are $n - 1$ auxiliary trees, one for each internal node of T.

Notice that there is a significant amount of duplication here. Each point in a leaf of the x-range tree arises in the sets $P(u)$ for all of its ancestors u in the x-range tree. We will analyze the space usage below.

Query Processing: Now, when a 2-dimensional range is presented we do the following. First, we invoke a variant of the 1-dimensional range search algorithm to identify the $O(\log n)$ canonical nodes. (These are shown in blue in the left side of Fig. 178.) For each such node u, we know that all the points of the set lie within the x portion of the range, but not necessarily in the y part of the range. So, for each of the nodes u of the canonical subtrees, we search the associated 1-dimensional auxiliary y-range and return a count of the resulting points. These counts are summed up over all the auxiliary subtrees to obtain the final answer.

The algorithm given in the code block below is almost identical the previous one, except that we make explicit reference to the x-coordinates in the search, and rather than adding u.size to the count, we invoke a 1-dimensional version of the above procedure using the y-coordinate

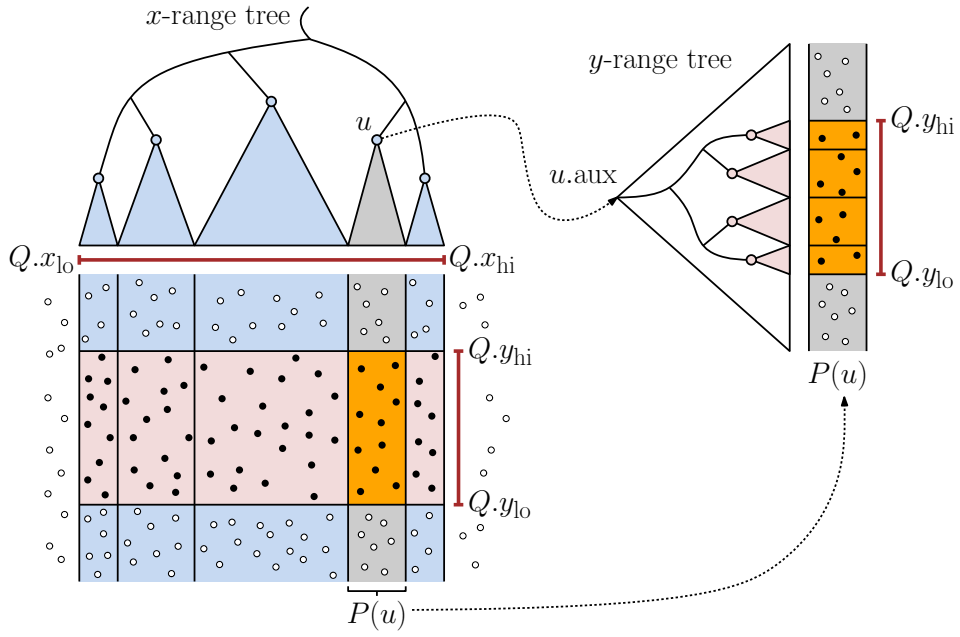


Fig. 178: 2-Dimensional Range tree.

instead. Let $Q.x$ denote the x-portion of Q's range, consisting of the interval $[Q_{\text{lo}}.x, Q_{\text{hi}}.x]$. The procedure `range1Dy()` is the same procedure described above, except that it searches on y-coordinates rather than x.

2-Dimensional Range Counting Query

```
int range2D(Node u, Range2D Q, Range1D C=[x0,x1]) {
    if (u is a leaf)                                // hit the leaf level?
        return (Q contains u.point ? 1 : 0)         // count if point in range
    else if (Q's x-range contains C) {              // Q's x-range contains C
        [y0, y1] = [-infinity, +infinity]           // initial y-cell
        return range1Dy(u.aux, Q, [y0, y1])         // search auxiliary tree
    }
    else if (Q.x is disjoint from C)                // no overlap
        return 0
    else
        return range2D(u.left,  Q, [x0, u.x]) +     // count left side
               range2D(u.right, Q, [u.x, x1])       // and right side
}
```

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 $O(n \log n)$.

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.

Lemma: The total size of an 2-dimensional range tree storing n keys is $O(n \log n)$, and it can be constructed in time $O(n \log n)$.

Query Time: It takes $O(\log n)$ time to identify the canonical nodes in the x-range tree. For each of these $O(\log n)$ nodes we make a call to a 1-dimensional y-range tree. When we invoke this on the subtree rooted at a node u, the running time is $O(\log |P(u)|)$. But, $|P(u)| \leq n$, so this takes $O(\log n)$ time for each auxiliary tree search. Since we are performing $O(\log n)$ searches, each taking $O(\log n)$ time, the total search time is $O(\log^2 n)$. As above, we can replace the counting code with code in `range1Dy()` with code that traverses the tree and reports the points. This results in a total time of $O(k + \log^2 n)$, assuming k points are reported.

Higher Dimensions: Thus, each node of the 2-dimensional range tree has a pointer to a auxiliary 1-dimensional range tree. We can extend this to any number of dimensions. At the highest level the d-dimensional range tree consists of a 1-dimensional tree based on the first coordinate. Each of these trees has an auxiliary tree which is a $(d-1)$-dimensional range tree, based on the remaining coordinates. A straightforward generalization of the arguments presented here show that the resulting data structure requires $O(n \log^d n)$ space and can answer queries in $O(\log^d n)$ time.

Theorem: Given an n-element point set in d-dimensional space (for any constant d) orthogonal range counting queries can be answered in $O(\log^d n)$ time, and orthogonal range reporting queries can be answered in $O(k + \log^d n)$ time, where k is the number of entries reported.

Faster Queries by Cascading Search: 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-layered 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. 179(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. 179(b)).

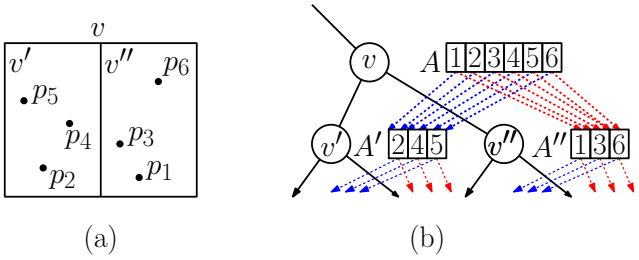


Fig. 179: Cascaded search in range trees.

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.

We call this *cascading search*. 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 30: 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*. We are given a set of orthogonal line segments S (see Fig. 180(a)), which have been preprocessed. Given an orthogonal query rectangle W, we wish to count or report all the line segments of S that intersect W (see Fig. 180(b)). We will assume that W is a 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, in Fig. 180(b) the query would report the segments that are shown with heavy solid lines, and segments with broken lines would not be reported.

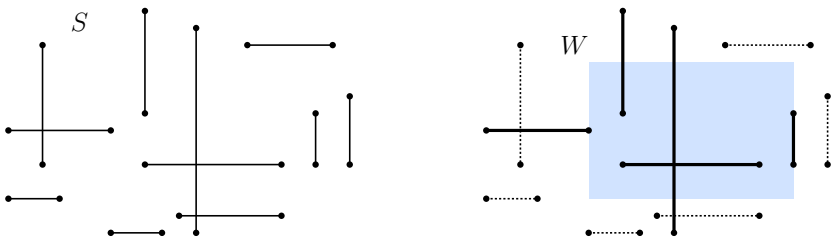


Fig. 180: Window Query.

Window Queries for Orthogonal Segments: We will present a data structure, called the *interval tree*, which (combined with a range tree) can answer window counting queries for orthogonal line segments in $O(\log^2 n)$ time, where n is the number line segments. It can report these segments in $O(k + \log^2 n)$ time, where and k is the total number of segments reported. The interval tree uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.

We will consider the case of range reporting queries. (There are some subtleties in making this work for counting queries.) We will derive our solution in steps, starting with easier subproblems and working up to the final solution. To begin with, observe that the set of

segments that intersect the window can be partitioned into three types: those that have no endpoint in W, those that have one endpoint in W, and those that have two endpoints in W.

We already have a way to report segments of the second and third types. In particular, we may build a range tree just for the $2n$ endpoints of the segments. We assume that each endpoint has a cross-link indicating the line segment with which it is associated. Now, by applying a range reporting query to W we can report all these endpoints, and follow the cross-links to report the associated segments. Note that segments that have both endpoints in the window will be reported twice, which is somewhat unpleasant. We could fix this either by sorting the segments in some manner and removing duplicates, or by marking each segment as it is reported and ignoring segments that have already been marked. (If we use marking, after the query is finished we will need to go back an "unmark" all the reported segments in preparation for the next query.)

All that remains is how to report the segments that have no endpoint inside the rectangular window. We will do this by building two separate data structures, one for horizontal and one for vertical segments. A horizontal segment that intersects the window but neither of its endpoints intersects the window must pass entirely through the window. Observe that such a segment intersects any vertical line passing from the top of the window to the bottom. In particular, we could simply ask to report all horizontal segments that intersect the left side of W. This is called a *vertical segment stabbing query*. In summary, it suffices to solve the following subproblems (and remove duplicates):

Endpoint inside: Report all the segments of S that have at least one endpoint inside W. (This can be done using a range query.)

Horizontal through segments: Report all the horizontal segments of S that intersect the left side of W. (This reduces to a vertical segment stabbing query.)

Vertical through segments: Report all the vertical segments of S that intersect the bottom side of W. (This reduces to a horizontal segment stabbing query.)

We will present a solution to the problem of vertical segment stabbing queries. Before dealing with this, we will first consider a somewhat simpler problem, and then modify this simple solution to deal with the general problem.

Vertical Line Stabbing Queries: Let us consider how to answer the following query, which is interesting in its own right. Suppose that we are given a collection of horizontal line segments S in the plane and are given an (infinite) vertical query line $\ell_q : x = x_q$. We want to report all the line segments of S that intersect ℓ_q (see Fig. 181(a)). 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.

As is true for all our data structures, we want some balanced way to decompose the set of intervals into subsets. Since it is difficult to define some notion of order on intervals, we instead will order the endpoints. Sort the interval endpoints along the x-axis. Let $\langle x_1, x_2, \ldots, x_{2n} \rangle$ be the resulting sorted sequence. Let x_{med} be the median of these $2n$ endpoints. Split the intervals into three groups, L, those that lie strictly to the left of x_{med}, R those that lie strictly to the right of x_{med}, and M those that contain the point x_{med} (see Fig. 181(b)). 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.

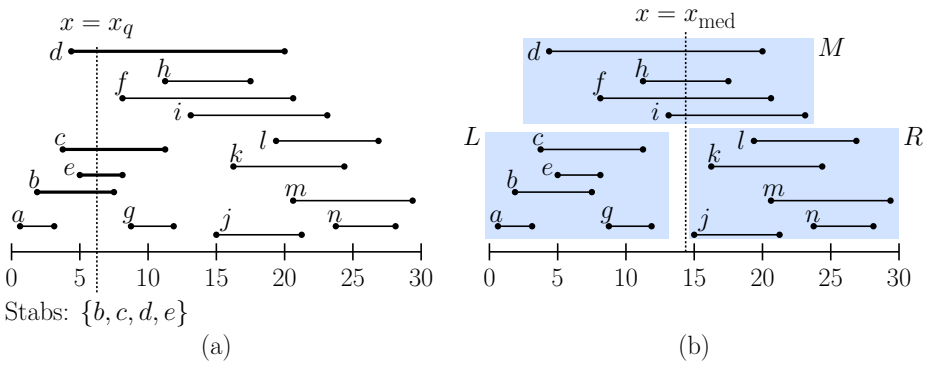


Fig. 181: Line Stabbing Query. (We have organized the horizontal segments into groups according to their y-coordinates, but the y-coordinates can be arbitrary.)

But how do we handle the intervals of M that contain x_{med}? We want to know which of these intervals intersects the vertical line ℓ_q. At first it may seem that we have made no progress, since it appears that we are back to the same problem that we started with. However, we have gained the information that all these intervals intersect the vertical line $x = x_{\text{med}}$. How can we use this to our advantage?

Let us suppose for now that $x_q \leq x_{\text{med}}$. How can we store the intervals of M to make it easier to report those that intersect ℓ_q. The simple trick is to sort these lines in increasing order of their left endpoint. Let M_L denote the resulting sorted list. Observe that if some interval in M_L does not intersect ℓ_q, then its left endpoint must be to the right of x_q, and hence none of the subsequent intervals intersects ℓ_q. Thus, to report all the segments of M_L that intersect ℓ_q, we simply traverse the sorted list and list elements until we find one that does not intersect ℓ_q, that is, whose left endpoint lies to the right of x_q. As soon as this happens we terminate. If k' denotes the total number of segments of M that intersect ℓ_q, then clearly this can be done in $O(k' + 1)$ time.

The case $x_q > x_{\text{med}}$ is symmetrical. We simply sort all the segments of M in a sequence, M_R, which is sorted from right to left based on the right endpoint of each segment. Thus each element of M is stored twice, but this will not affect the size of the final data structure by more than a constant factor. The resulting data structure is called an *interval tree*.

Interval Trees: The general structure of the interval tree was derived above. Each node of the interval tree has a left child, right child, and itself contains the median x-value used to split the set, x_{med}, and the two sorted sets M_L and M_R (represented either as arrays or as linked lists) of intervals that overlap x_{med}. We assume that there is a constructor that builds a node given these three entities. The following code block presents 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 $(x_{\text{lo}}, x_{\text{hi}})$. See Fig. 182(a)) for an example.

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

```
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
}
```

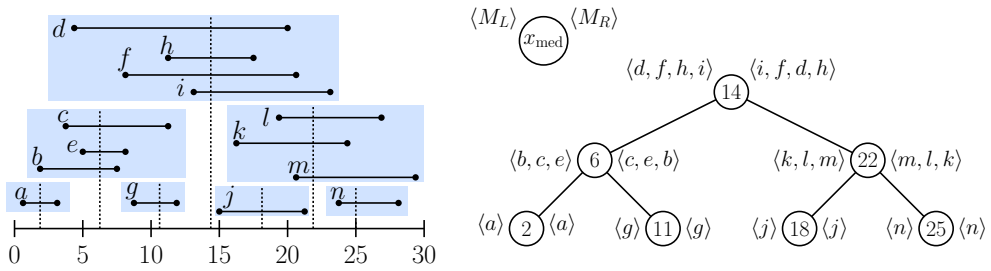


Fig. 182: Interval Tree.

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 present this algorithm in the following code block. Let x_q denote the x-coordinate of the query line.

Line Stabbing Queries for an Interval Tree

```
stab(IntTreeNode t, Scalar xq) {
    if (t == null) return                       // fell out of tree
    if (xq < t.xMed) {                          // left of median?
        for (i = 0; i < t.ML.length; i++) {     // traverse ML
            if (t.ML[i].lo <= xq) print(t.ML[i])// ..report if in range
            else break                          // ..else done
        }
        stab(t.left, xq)                        // recur 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)                       // recur on right
    }
}
```

This procedure actually has one small source of inefficiency, which was intentionally included to make code look more symmetric. Can you spot it? Suppose that $x_q = t.x_{\text{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that are strictly to the right of x_{med} and so is a waste of effort. However it does not affect the asymptotic running time.

As mentioned earlier, the time spent processing each node is $O(1 + k')$ where k' is the total number of points that were recorded at this node. Summing over all nodes, the total reporting time is $O(k+v)$, where k is the total number of intervals reported, and v is the total number of nodes visited. Since at each node we recur on only one child or the other, the total number of nodes visited v is $O(\log n)$, the height of the tree. Thus the total reporting time is $O(k+\log n)$.

Vertical Segment Stabbing Queries: Now let us return to the question that brought us here. Given a set of horizontal line segments in the plane, we want to know how many of these segments intersect a vertical line segment. Our approach will be exactly the same as in the interval tree, except for how the elements of M (those that intersect the splitting line $x = x_{\text{med}}$) are handled.

Going back to our interval tree solution, let us consider the set M of horizontal line segments that intersect the splitting line $x = x_{\text{med}}$ and as before let us consider the case where the query segment q with endpoints (x_q, y_{lo}) and (x_q, y_{hi}) lies to the left of the splitting line.

The simple trick of sorting the segments of M by their left endpoints is not sufficient here, because we need to consider the y-coordinates as well. Observe that a segment of M stabs the query segment q if and only if the left endpoint of a segment lies in the following semi-infinite rectangular region (see Fig. 183).

$$\{(x, y) \mid x \leq x_q \text{ and } y_{\text{lo}} \leq y \leq y_{\text{hi}}\}.$$

Observe that this is just an orthogonal range query. (It is easy to generalize the procedure given last time to handle semi-infinite rectangles.) The case where q lies to the right of x_{med} is symmetrical.

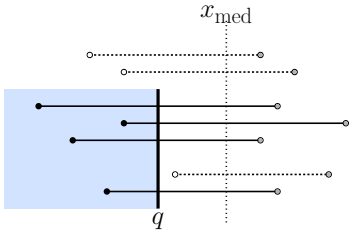


Fig. 183: The segments that stab q lie within the shaded semi-infinite rectangle.

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 31: 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 Fig. 184).

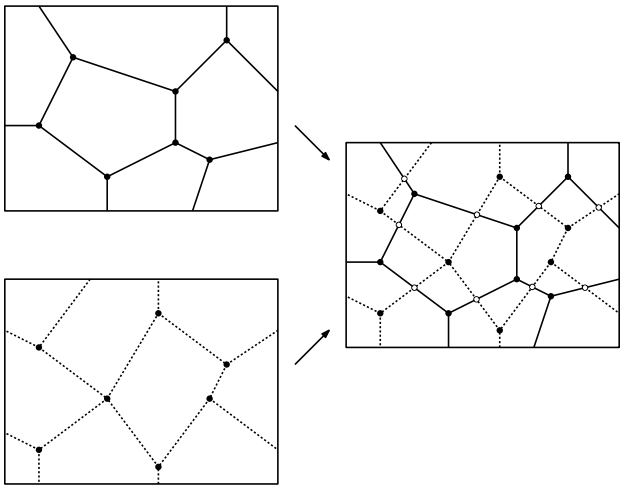


Fig. 184: Red-blue line segment intersection. The algorithm outputs the white intersection points between segments of different colors. The segments of each color are pairwise disjoint (except possibly at their endpoints).

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 Fig. 185(a)).

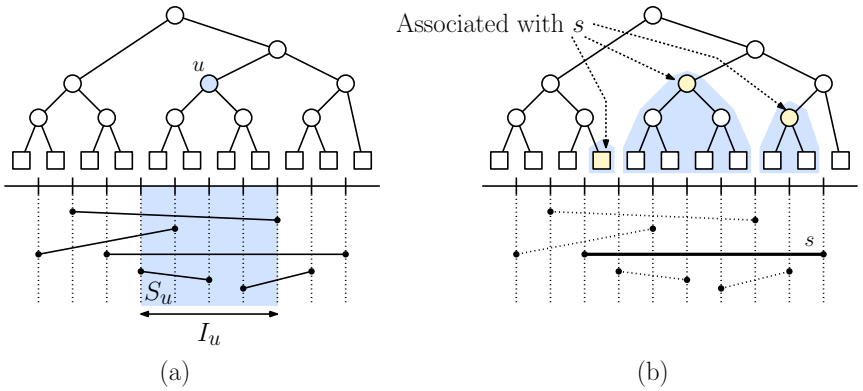


Fig. 185: Hereditary Segment Tree: Intervals, slabs and the nodes associated with a segment.

We associate a segment s with a set of nodes of the tree. A segment is said to *span* interval I_u if its projection covers this interval. We associate a segment s with a node u if s spans I_u but s does not span I_p, where p is u's parent (see Fig. 185(b)).

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.)

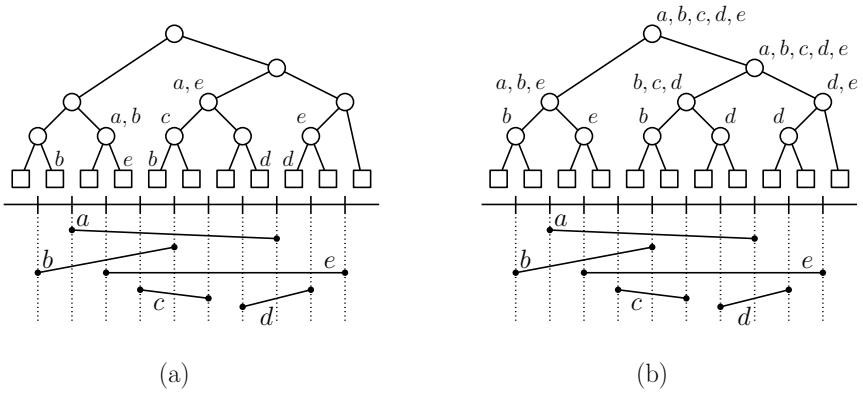


Fig. 186: Hereditary Segment Tree with standard lists (left) and hereditary lists (right).

Each node u is also associated with a list B^*_u, called the *blue hereditary list*, which is the union of the B_v for all proper descendents v or u. The red hereditary list R^*_u is defined analogously. (Even though a segment may occur in the standard list for many descendants, 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 Fig. 187). 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 $O(N \log N)$.

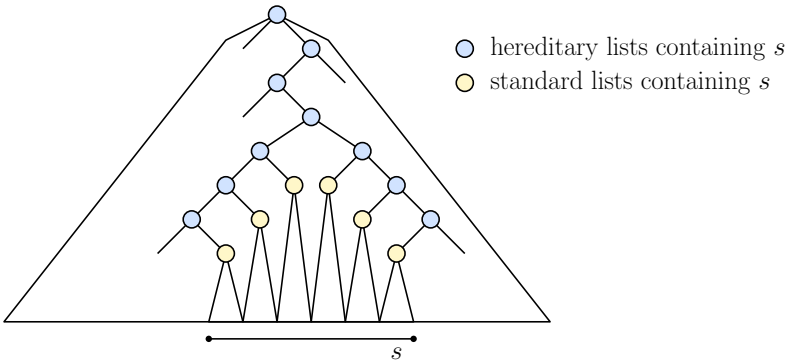


Fig. 187: Standard and hereditary lists containing a segment s.

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: Our approach follows along the lines of the inversion counting procedures we have seen earlier in the semester. First, 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 $\langle b_1, \ldots, b_{m_u} \rangle$ and $\langle r_1, \ldots, r_{n_u} \rangle$ 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 Fig. 188(a)). On the other hand, if $i \geq j$ then b intersects the red segments $r_{j+1}, \ldots, r_i$ (see Fig. 188(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 Fig. 188(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 $\{r_1, r_2, r_3\}$.

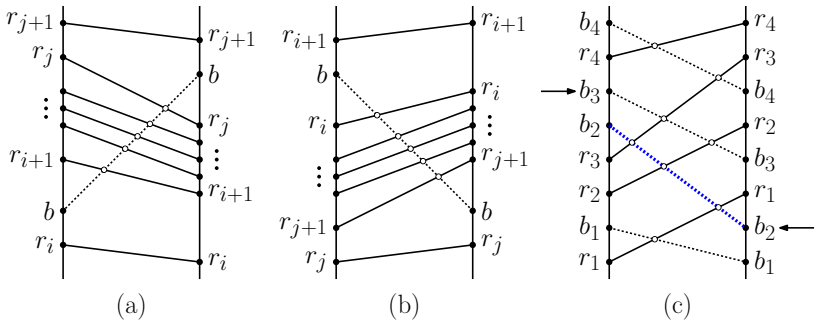


Fig. 188: Red-blue intersection counting/reporting. Long-long intersections.

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 $r_i, r_{i+1}, \ldots, r_j$.

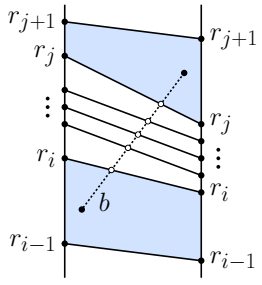


Fig. 189: Red-blue intersection counting/reporting: Long-short intersections.

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 complexity of

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

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

Lecture 32: Ham-Sandwich Cuts

Ham Sandwich Cuts of Linearly Separated Point Sets: In this short lecture, we consider an application of duality and arrangements, namely computing a Ham-Sandwich cut of two linearly separable 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 (see Fig. 190(a)). It is a well-known theorem from mathematics that such a simultaneous bisector exists for any pair of sets (even for shapes, where bisection is in terms of area).

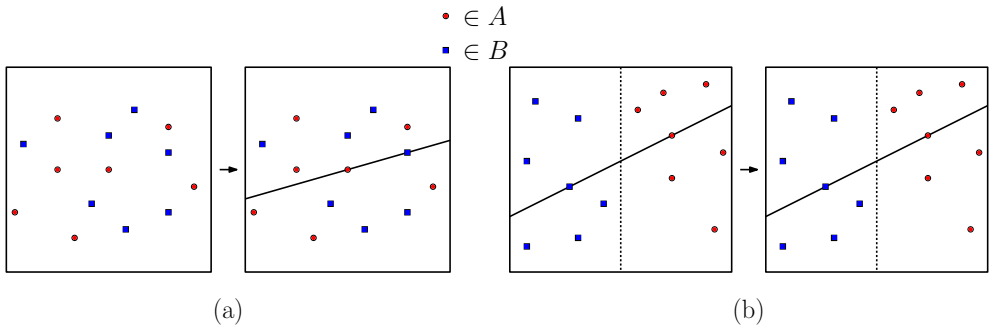


Fig. 190: Ham sandwich cuts (a) general and (b) linearly-separable.

This problem can be solved in $O(n^2)$ time through the use of duality and line arrangements, but we will consider a restricted version that can be solved much faster. In particular, let us assume that the two sets can be separated by a line (see Fig. 190(b)). We may assume that the points have been translated and rotated so the separating line is the y-axis. Thus all the red points (set A) have positive x-coordinates, and all the blue points (set B) have negative x-coordinates. 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.

Ham-Sandwich Cuts in the Dual: 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? Suppose that we dualize the points using the standard dual transformation, where a point $p = (p_a, p_b)$ is mapped to the line $p^* : y = p_a x - p_b$. We obtain 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 (see Fig. 191(a)), 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 (see Fig. 191(b)). 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 (see Fig. 191(c)).

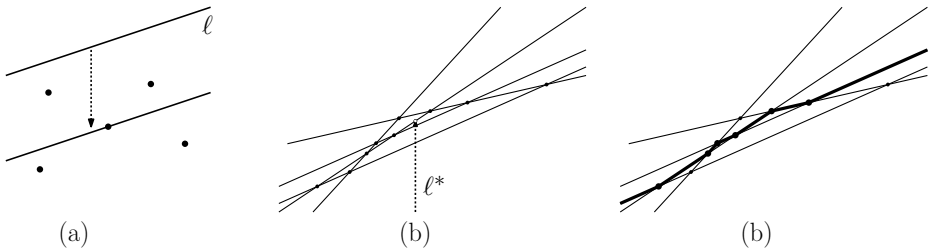


Fig. 191: The (a) median line, (b) median point, and (c) median level.

Thus, the set of bisecting lines for set A in dual form consists of a polygonal curve. Because all the points of A have positive x-coordinates, their dual lines have positive slopes (see Fig. 192(a)). Because this curve is formed from edges of the dual lines in A, and because all lines in A have positive slope, the median level $M(A)$ is monotonically increasing (see Fig. 192(b)). Similarly, the median level 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 (see Fig. 192(c)).

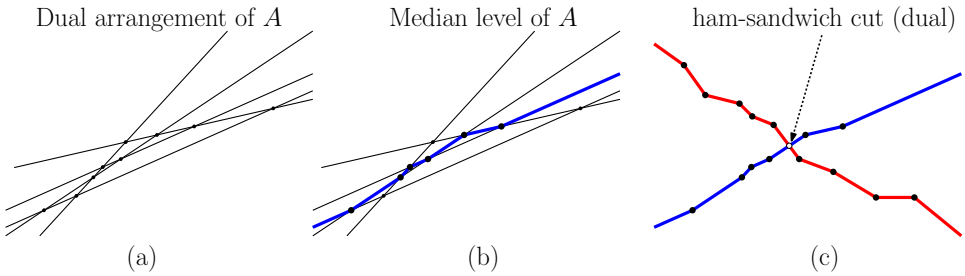


Fig. 192: Ham sandwich: Dual formulation.

Computing the Ham-Sandwich Cut by Prune and Search: We could compute the intersection of these two curves in $O(n^2)$ time by a simultaneous topological plane sweep of both

arrangements (even if the points were not linearly separable). However because of linear separability, 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 33: Introduction to Computational Topology

What is Topology? We are all familiar with Euclidean spaces, especially the plane $\mathbb{R}^2$ where we draw our figures and maps, and the physical space $\mathbb{R}^3$ where we actually live and move about. Our direct experiences with these spaces immediately suggest a natural *metric structure* which we can use to make useful measurements such as distances, areas, and volumes. Intuitively, a metric recognizes which pairs of locations are close or far. In more physical terms, a metric relates to the amount of energy it takes to move a particle of mass from one location to another. If we are able to move particles between a pair of locations, we say the locations are *connected*, and if the locations are close, we say they are *neighbors*. In every day life, we frequently rely more upon the abstract notions of *neighborhoods* and *connectedness* if we are not immediately concerned with exact measurements. For instance, it is usually not a big deal if we miss the elevator and opt to take the stairs, or miss an exit on the highway and take the next one; these pairs of paths are equivalent if we are not too worried about running late to an important appointment.

How do we develop our understanding of spaces without a metric structure? This brings to mind the more familiar setting of *graph theory*, which deals with abstract networks of nodes connected by edges. While we might picture a certain configuration of the nodes and their interconnections, we are not too fixated on the exact *positions* of the nodes nor their relative *distances*. Despite the underspecified *shape* or *realization* of the graph, we are still aware of other qualitative properties, such as the adjacency relation and the number of connected

components, which are again easy to describe in terms of *neighborhoods* and *connectedness*. Specifically, those qualitative properties are *invariant* under arbitrary *deformations* as long as they preserve the *neighborhood structure*, i.e., the adjacency relation, of the graph.

Eulerian Paths. The foundations of graph theory are largely credited to Euler who established its first result by resolving the well-known *Eulerian path* problem in 1735. In its original form, the problem simply asked to find a path that crossed each of seven bridges exactly once; see Figure 193(left).

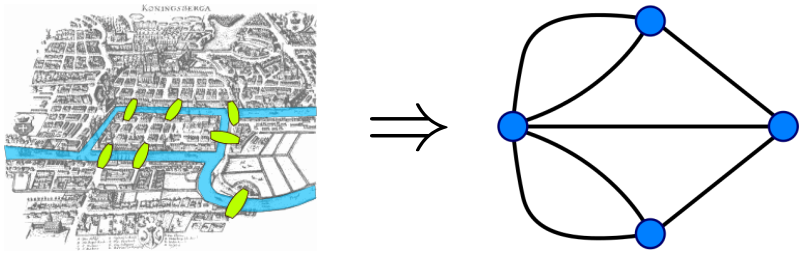
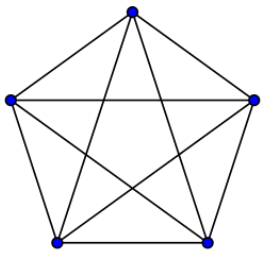


Fig. 193: Seven Bridges of Königsberg, and the origin of graph theory. (Figures from [1, 2])

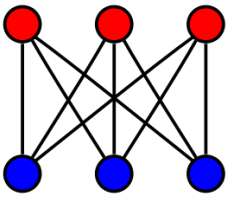
Euler's *topological insight* was to recognize that the subpaths within each land mass are irrelevant to the solution. This allows one to consider the abstract setting provided by the usual graph model; see Figure 193(right). Next, observing how the path first enters into a node through an edge before leaving through a different edge, Euler correctly identified the issue with vertices of *odd degree*. In particular, an Eulerian path exists if and only if the graph has exactly zero or two nodes of odd degree. Euler later published the result under the title "The solution of a problem relating to the geometry of position," where the *geometry of position* indicates that it is about something more general than *measurements and calculations*.

As hinted in the previous example, one of the main uses of topological ideas is to identify an *obstruction* to the existence of an object.

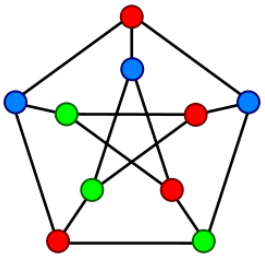
Forbidden Graph Characterizations. If we cannot solve a problem on a given graph H, chances are we cannot solve it on any other graph G whenever G contains *something* that *looks like* H. To formalize this notion, define a *contraction* as the merging or *identification* of two adjacent vertices. We say that H is a *minor* of G if H can be obtained by a sequence of contractions, edge deletions, and deletion of isolated vertices. The equivalent theorems of Kuratowski (1930) and Wagner (1937) essentially state that a graph G is planar if and only if its minors include neither K_5 nor $K_{3,3}$, i.e., the complete graph on five nodes and the complete bipartite graph on six vertices; see Figure 194(a) and (b). Hence, the existence of a K_5 or $K_{3,3}$ minor is an *obstruction* to planarity. The Petersen graph shown in Figure 194(c), which serves as counterexample for many claims in graph theory, contains K_5 and $K_{3,3}$ as minors. Hence, the Petersen graph is not planar.



(a) K_5.



(b) $K_{3,3}$.



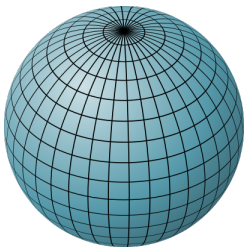
(c) 3-coloring of the Petersen graph.

Fig. 194: Graph minors and coloring. (Figures from [3, 4, 5])

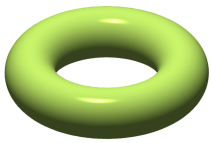
Another example of obstruction is provided in the context of *graph coloring*, which has many applications in scheduling and distributed computing. Recall that a t-coloring of a graph is an assignment of one of t colors to each vertex such that no two adjacent vertices get the same color; see Figure 194(c). Clearly, a coloring of K_t requires at least t colors. One of the deepest unsolved problems in graph theory is the *Hadwiger conjecture* (1943) postulating that K_t minors are the only *obstruction* to the existence of colorings with fewer than t colors.

Beyond the discrete spaces commonly studied in graph theory, a *topological space* can be any set endowed with a *topology*, i.e., a *neighborhood structure*. The mathematical subject of *topology* is the formal study of properties of topological spaces which are *invariant* under *continuous functions*. Such properties are simply referred to as *topological invariants*.

Genus. Intuitively, the genus of a connected and orientable surface is the number of *holes* or *handles* on it; see Figure 195. It is a traditional joke that a topologist cannot distinguish his coffee mug from his doughnut; as both have genus one, they may be (continuously) deformed into one another and are in that sense *topologically equivalent*. In contrast, the mismatch in the genus is an *obstruction* to the existence of continuous mappings from spheres to tori.

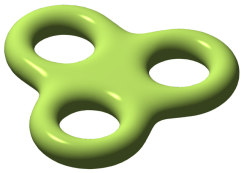


(a) $g = 0$.



(b) $g = 1$.

(c) $g = 2$.



(d) $g = 3$.

Fig. 195: Genus of orientable surfaces. (Figures from [6, 7, 8, 9])

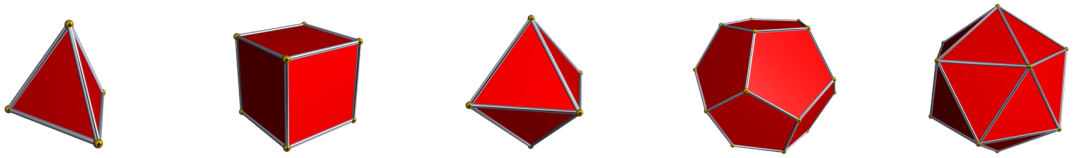
In relation to the previous examples, the *genus of a graph* is the smallest g such that the graph can be drawn without crossing on an orientable surface of genus g. Because the Earth is (locally) flat, *planar graphs* can be drawn on the sphere implying they have genus zero. More generally, the genus is one of the measures of complexity of the graph, and can be exploited to obtain faster algorithms for graphs with small genus. Alas, deciding whether a given graph has genus g is NP-complete.

While we may be interested in studying surfaces, or other topological spaces, we need simpler discrete structures to keep computations easy or at all feasible. This workaround does not allow us to compute everything we might have wanted, but it does provide very useful information. For example, the previous example showed how the genus can be used to classify surfaces. It turns out there is a closely related topological invariant which is more amenable to computation.

Euler's Polyhedron Formula. The following remarkable formula by Euler is considered, together with his resolution of the Seven Bridges of Königsberg problem, as the first two theorems in topology. Consider a *polyhedron* $P \subset \mathbb{R}^3$, and denote the number of vertices, edges, and faces of P by V, E, and F, respectively. The *Euler characteristic* χ is defined as:

$$\chi = V - F + E. \tag{4}$$

For any convex polyhedron P, we have that $\chi = 2$; see Figure 196. As the Euler characteristic is a topological invariant, one correctly anticipates that it also evaluates to 2 for the sphere.

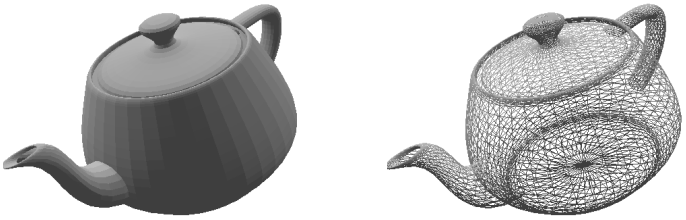


(a) $4 - 6 + 4$. (b) $8 - 12 + 6$. (c) $6 - 12 + 8$. (d) $20 - 30 + 12$. (e) $12 - 30 + 20$.

Fig. 196: Convex polyhedra with $\chi = 2$. (Figures from [10, 11, 12, 13, 14])

The previous example confirms the intuition that convex polytopes are suitable as *discrete approximations* to the sphere. In order to approximate arbitrary surfaces, which may not be convex, we are going to need more flexible structures.

Simplicial Complexes. You are probably familiar with triangular subdivisions of planar shapes, and the three-dimensional models suitable for rendering pipelines in computer games. Just a collection of vertices and connecting edges suffice to define a bare-bones *wireframe* that still captures the salient features of a shape; see Figure 197.

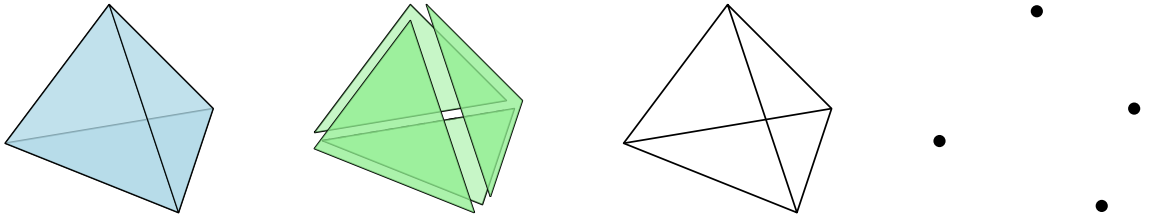


(a) Rendering all triangles. (b) Wireframe, edges only.

Fig. 197: The Utah teapot, arguably the *most important object in computer graphics history.*

It will prove useful to use a notation that easily generalizes to higher dimensions. We start with a set of points $S \subset \mathbb{R}^d$, for $d \geq 0$. We define a *p-simplex* σ as a subset of $p + 1$ points in S, and we say that σ has *dimension* $\dim \sigma = p$. For a *geometric realization*, the simplex σ is the *convex hull* of $p + 1$ affinely-independent points; see Figure 198. We write this as

$\sigma = \text{conv}\{v_0, \ldots, v_p\}$. To capture the structure of the simplex, we define a *k-face* of σ is a simplex τ with (1) $\tau \subseteq \sigma$, and (2) $\dim \tau = k$ for $-1 \leq k \leq p$; we write this as $\tau \preceq \sigma$ and call σ a *coface* of τ. We say a (co)face of σ is proper if its dimension is different from $\dim \sigma$, and write $\partial\sigma$ for the proper faces of σ. Finally, the interior of σ is defined as $|\sigma| = \sigma - \partial\sigma$.



(a) A 3-simplex. (b) Four 2-simplices. (c) Six 1-simplices. (d) Four 0-simplices.

Fig. 198: The simplicial structure of a tetrahedron.

Suppressing realizations for a moment, we define an *abstract simplicial complex* K as a collection of simplices with the following *closure property*. Whenever a simplex σ appears in K, all faces of σ also appear in K. Similarly, we say that K is a *p-complex* with *dimension* $\dim K = \max_{\sigma \in K} \dim \sigma$ and *underlying space* $|K| = \cup_{\sigma \in K} |\sigma|$. For a geometric realization, we additionally require that for any two simplices $\sigma, \tau \in K$, we have that $\sigma \cap \tau \in K$. When this extra condition holds, we say that K is a *simplicial complex*. Every abstract simplicial complex of dimension p has a geometric realization, as a proper simplicial complex, in R^{2p+1}.

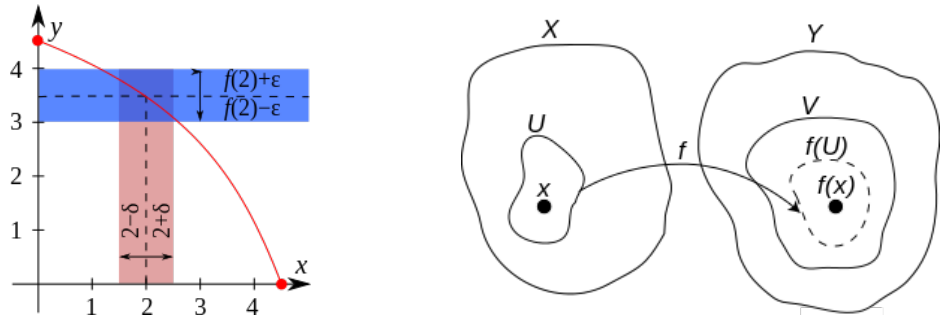
Before we can use simplicial complexes as proxies of topological spaces, we also need to approximate the continuous maps between such spaces through their simplicial proxies. We start by building some intuition as to how continuous maps act on spaces.

Continuous Maps. Imagine we have two surfaces X and Y, and a mapping $f : X \to Y$. In this case, f takes a point $x \in X$ to the corresponding point $y = f(x) \in Y$. Visually, y is where x ends up after going through some deformation that takes X to Y; see Figure 199. When do we consider such mappings to be continuous? Imagine you label two nearby points x_1 and x_2 on X and trace where they end up on Y. Once you identify the point $y_1 = f(x_1)$, where would you expect $y_2 = f(x_2)$ to be? For example, take x_1 and x_2 to be the the eyes of the cow.

Fig. 199: A continuous deformation of a cow model (X) into a ball (Y). (Figure from [15])

You are probably familiar with the notion of continuous functions from calculus which suggests we use an ε-neighborhood $V \subseteq Y$ around y and show that there is a corresponding $\delta = \delta(\varepsilon)$ such that all points in an δ-neighborhood $U \subseteq X$ around x are mapped by f into V, i.e., $f(U) \subseteq V$. In the particularly familiar context of a univariate function $g : \mathbb{R} \to \mathbb{R}$, the

neighborhoods in question are immediately realized as *open intervals* of the form $(a, b) \subset \mathbb{R}$, where there is no shortage of intervals in the *continuum* which is $\mathbb{R}$ for us to choose from. Specifically, we require that $\lim_{\varepsilon \to 0} \delta(\varepsilon) = 0$; see Figure 200(a).



(a) Continuity at $x = 2$ by ε-δ. (b) Continuity at $x \in X$ using neighborhoods.

Fig. 200: Essentially equivalent definitions of continuous functions. (Figures from [16, 17])

It is plausible to conclude that neighborhoods, rather than the ε and δ, are all we need for continuity. Only that for general spaces, such as the surfaces X and Y, we have to work with their particular neighborhoods as specified by their respective topologies; see Figure 200(b). While not all topologies furnish neighborhoods as convenient as the intervals on the real line, a meaningful version of continuous maps can be defined for spaces with similar topologies.

Given our enhanced understanding of continuity, it is about time to formalize what we mean by *topologically equivalent* and *simplicial proxy*.

Homeomorphisms and Triangulations. A *homeomorphism* $f : X \to Y$ is a continuous function with a continuous inverse $f^{-1} : Y \to X$. Whenever such a homeomorphism f exists, we say that X and Y are *homeomorphic*, which literally translates to having the same shape. Applying this precise notion to our simplicial proxies, we say that a simplicial complex $\widehat{X}$ is a *triangulation* of X if its underlying space $|\widehat{X}|$ is homeomorphic to X.

We can now proceed to approximate a continuous mapping $f : X \to Y$ by a discrete mapping between triangulations $\widehat{f} : \widehat{X} \to \widehat{Y}$. But, what does it mean for such a mapping $\widehat{f}$ to be continuous?

Simplicial Neighborhoods. Take a point x in the underlying space $|\widehat{X}|$. To examine the continuity of $\widehat{f}$ at x, we need to consider the neighborhood of x on $|\widehat{X}|$. While x may belong to many simplices of $\widehat{X}$, there is a unique simplex that contains x in its *interior*; let us denote this simplex by $\sigma(x)$. If another point $x' \in \widehat{X}$ is a neighbor of x, it might be the case that $x' \in |\sigma(x)|$. However, we need to allow x' to go outside $|\sigma(x)|$ and reach farther parts of $\widehat{X}$.

Let us consider what lies beyond $|\sigma(x)|$. For example, if $\dim \widehat{X} = 3$ and $\sigma(x)$ is an edge with $\dim \sigma(x) = 1$, x' could start at x in $|\sigma(x)|$ and wander into a different simplex. Recalling the geometric realization, we consider an ε-neighborhood around x. We allow this neighborhood to expand over nearby *interior* points of adjacent simplices without crossing any boundaries, i.e., mimicking open intervals from calculus. For example, we do not connect x to the interior of an adjacent edge e unless there is a path through the interior of a common face or tetrahedron, i.e., a coface. As such, the neighborhood of x is contained in the cofaces of $\sigma(x)$.

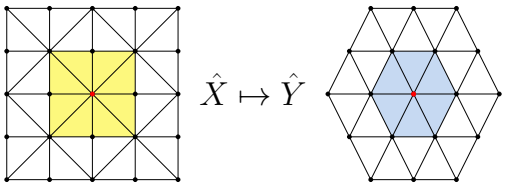


Fig. 201: One star in each of $\widehat{X}$ and $\widehat{Y}$. The star condition includes the image of one into the other.

The cofaces of a simplex $\sigma \in K$ constitute its *star*; we write this as $\mathrm{St}_K(\sigma) = \{\tau \in K \mid \sigma \preceq \tau\}$. Taking the union of all interior points, we define the *star neighborhood* as $N_K(\sigma) = \cup_{\tau \in \mathrm{St}_K(\sigma)} |\tau|$; see Figure 201. It will suffice for our purposes to consider the neighborhoods of vertices in $\widehat{X}$ and $\widehat{Y}$.

Star Condition. Recalling the definition of continuity, we require our maps $\widehat{f} : |\widehat{X}| \to |\widehat{Y}|$ to satisfy $\widehat{f}(N_{\widehat{X}}(v)) \subseteq N_{\widehat{Y}}(u)$ for all vertices $v \in \widehat{X}$ and some vertex $u = \phi(v) \in \widehat{Y}$; see Figure 201. This *star condition* has the following important consequence. Fix a point $x \in |\widehat{X}|$, and let $\sigma \in \widehat{X}$ and $\tau \in \widehat{Y}$ denote the unique simplices containing x and $\widehat{f}(x)$, respectively, in their interiors. Assuming σ is the p-simplex $[v_0, \ldots, v_p]$, we have by the definition of the star that $x \in N_{\widehat{X}}(\sigma) \subseteq N_{\widehat{X}}(v_i)$, for all $0 \leq i \leq p$; in fact $N_{\widehat{X}}(\sigma) = \cap_{i=0}^{p} N_{\widehat{X}}(v_i)$. Passing through $\widehat{f}$, the star condition implies that $\widehat{f}(x) \in \widehat{f}(N_{\widehat{X}}(\sigma)) \subseteq \cap_{i=0}^{p} N_{\widehat{Y}}(\phi(v_i)) \neq \emptyset$. By the same token, we get that $[\phi(v_0), \ldots, \phi(v_p)]$ is a simplex in $\widehat{Y}$ which must coincide with $\tau \ni \widehat{f}(x)$, i.e., $\widehat{f}(\sigma) = \tau$.

Instead of arbitrary continuous maps, there is great appeal to working with piecewise-linear maps on triangulations. In fact, the star condition was chosen to provide exactly that.

Simplicial Approximations. Assume that $\widehat{f} : |\widehat{X}| \to |\widehat{Y}|$ satisfies the star condition, and let $\phi_{\widehat{f}} : \mathrm{Vert}\, \widehat{X} \to \mathrm{Vert}\, \widehat{Y}$ be the associated *vertex map*. Fixing a p-simplex $\sigma = \mathrm{conv}\{v_0, \ldots, v_p\}$, we may express any $x \in |\sigma|$ as a *linear combination* of the vertices. Using the so-called *barycentric coordinates*, we write $x = \sum_{i=0}^{p} \lambda_i v_i$, where $\lambda_i > 0\ \forall i$. The expression can be extended to all vertices of K; letting $b_i(x) = \lambda_i$ for $0 \leq i \leq p$ and $b_i(x) = 0$ otherwise, we may write $x = \sum_i b_i(x) v_i$. Passing through $\phi_{\widehat{f}}$, we get that $\sum_i b_i(x) \phi_{\widehat{f}}(v_i) \in \phi_{\widehat{f}}(\sigma)$, where $\phi_{\widehat{f}}(\sigma)$ is a simplex in $\widehat{Y}$. As such, the vertex map $\phi_{\widehat{f}}$ induces a continuous, piecewise-linear *simplicial map* $x \mapsto \sum_i b_i(x) \phi_{\widehat{f}}(v_i)$. We will denote the induced simplicial map as $\widehat{f}_\Delta : \widehat{X} \to \widehat{Y}$. As both $\widehat{f}(x)$ and $\widehat{f}_\Delta(x)$ belong to the same simplex in $\widehat{Y}$, we call $\widehat{f}_\Delta$ a *simplicial approximation*, i.e., there is a smooth interpolation (or *homotopy*) to gradually change $\widehat{f}_\Delta$ into $\widehat{f}$.

While the star condition seems to provide all we need, it is only a convenient assumption we had to introduce. What about continuous maps from $|\widehat{X}|$ to $|\widehat{Y}|$ that the assumption fails to capture?

Subdivisions. Assume that a continuous map $\widehat{f} : |\widehat{X}| \to |\widehat{Y}|$ does not satisfy the star condition. Then, there must be a vertex $v \in \widehat{X}$ such that $\widehat{f}(N_{\widehat{X}}(v))$ is not contained in $N_{\widehat{Y}}(u)$ for *any* vertex $u \in \widehat{Y}$. Equivalently, $N_{\widehat{X}}(v)$ is not contained in any $\widehat{f}^{-1}(N_{\widehat{Y}}(u))$. In other words, $N_{\widehat{X}}(v)$ is relatively *too large*. Can we make the star of v *smaller* without changing $\widehat{f}$?

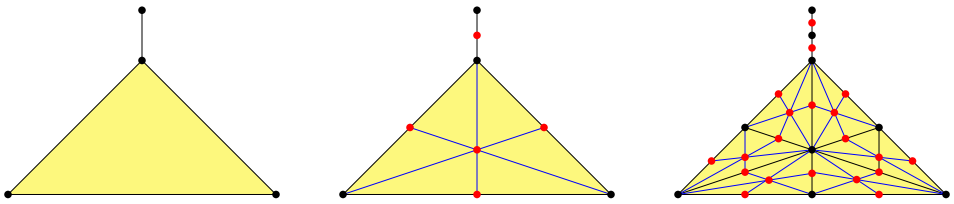


Fig. 202: Barycentric subdivisions of a triangle with an incident edge. New elements are highlighted.

It becomes clear that we need to keep $|\widehat{X}|$ intact, so that $\widehat{f}$ remains essentially the same, while making some stars smaller to satisfy the star condition. As the stars are defined by $\widehat{X}$, we seek a *finer* triangulation of $|\widehat{X}|$. One way to achieve that is to *subdivide* every simplex σ into smaller ones $\{\sigma'_i\}$ such that $|\sigma| = \cup\sigma'_i$. In particular, we make use of the *barycenter* of each simplex in $\widehat{X}$, which is defined as the average of its vertices. For $p = 1$ to $\dim \widehat{X}$, we insert the barycenter σ_c of each p-simplex σ as a new vertex, and form new p-simplices σ'_i by connecting σ_c to each $(p-1)$-simplex of the (subdivided) $(p-1)$-faces of $\partial\sigma$; denote this *barycentric subdivision* by Sd. A simple induction shows that every p-simplex is replaced by $(p+1)!$ new p-simplices; see Figure 202. More importantly, for any p-simplex σ, $\mathrm{diam}(\sigma'_i) \leq \frac{p}{p+1}\mathrm{diam}(\sigma)$. By repeating as needed, the diameters of all simplices are rapidly reduced such that the star neighborhoods of all vertices in $\mathrm{Sd}^k\widehat{X}$ are covered by the pre-image of some vertex in $\widehat{Y}$. Specifically, $\mathrm{Sd}^k\widehat{X}$ satisfies the star condition for a finite $k \geq 0$ and a simplicial approximation can then be defined on $\mathrm{Sd}^k\widehat{X}$; this is known as the *simplicial approximation theorem*.

Having established triangulations as a viable discrete representation to approximate the topological spaces we will be studying, we now proceed to the computation of topological invariants. As in Euler's polyhedron formula, this computation boils down to a simple *counting*. However, as the structure of simplicial complexes is more complicated compared to polyhedra, we make use of a few tools from *algebra* to help keep track of our counts.

Simplicial Counting. Take a simplicial complex K, and let σ_1 and σ_2 be 2-simplices in K. In computing the Euler characteristic, we would need to count the triangles σ_1 and σ_2 before *subtracting* the number of edges. Now, it might be the case that σ_1 and σ_2 have an edge in common. An added difficulty is that a single triangle introduces *three* edges as its boundary.

To facilitate the counting and representation of boundaries, we will use special sets of simplices enhanced with two convenient operations.

Chains. We define the *p-chains* C_p as so-called *formal sums* of p-simplices: a p-chain c is written as $c = \sum_i a_i\sigma_i$, where σ_i ranges over all p-simplices in K and a_i simply indicates whether σ_i is included in c or not. To facilitate the counting of simplices, we define an *addition* operation. The *sum* of two chains $c_1 + c_2$ is the chain with all simplices in either c_1 or c_2, but not both, i.e., their symmetric difference. In other words, we choose a_i as *modulo 2 coefficients*.

The algebraic framework we are about to develop will compensate for the lack of geometric visuals with greater expressive power, as will prove essential to our computations.

Algebra I. A *group* $(A, \bullet)$ is a set A together with a *binary operation* $\bullet : A \times A \to A$, meaning that A is *closed* under the action of $\bullet$. We further require that $\bullet$ is *associative* so that for all $\alpha, \beta, \gamma \in A$ we have that $\alpha \bullet (\beta \bullet \gamma) = (\alpha \bullet \beta) \bullet \gamma$. Finally, we require an *identity element* $\omega \in A$ such that $\alpha + \omega = \alpha$ for all $\alpha \in A$. If, in addition, $\bullet$ is *commutative*, we have that $\alpha \bullet \beta = \beta \bullet \alpha$ for all $\alpha, \beta \in A$, and we say the group $(A, \bullet)$ is *abelian*.

Using this new language, we say that $(C_p, +)$ is an abelian group. In particular, if K has n_p p-simplices, then $(C_p, +)$ is (*isomorphic* to) the set of binary vectors of length n_p with the usual exclusive-or operation $\oplus$. Hence, $(C_p, +)$ is not just any group; *it is a vector space!*

Boundary Maps. By our definition of chains, any p-simplex $\sigma \in K$ also belongs to the chain group C_p. As the boundary of σ is a collection of $(p-1)$-simplices, it will be convenient to express the boundary *in one shot* as an element in C_{p-1}. Letting $\sigma = [v_0, \ldots, v_p]$ we write:

$$\partial_p \sigma = \sum_{i=0}^{p} [v_0, \ldots, \widehat{v_i}, \ldots, v_p], \tag{5}$$

where $\widehat{v_i}$ indicates that v_i is excluded in the corresponding $(p-1)$ face. We can also take the boundary of a collection of p-simplices, i.e., a p-chain, to obtain the sum of their boundaries as a single $(p-1)$-chain. We denote this mapping by $\partial_p : C_p \to C_{p-1}$, and write $\partial_p c = \sum_i a_i \partial_p \sigma_i$.

Naturally, every chain group C_p gets its own boundary map ∂_p, though we often drop the subscript of ∂_p as we have been doing already with addition and summation. The combined action of those two operators gives rise to a rich algebraic structure essential to our computations.

Algebra II. A mapping $\delta : (A, \bullet) \to (B, \odot)$ is called a *homomorphism* if it commutes with the group operation, i.e., $\delta(\alpha \bullet \alpha') = \delta\alpha \odot \delta\alpha'$ for all $\alpha, \alpha' \in A$. (Note the switch from $\bullet$ to $\odot$.)

It is easy to verify that $\partial_p(c + c') = \partial_p c + \partial_p c'$ for all $c, c' \in C_p$, i.e., ∂_p is a homomorphism. Recalling the simplicial structure depicted in Figure 198, we use the boundary homomorphisms to arrange our p-chain groups into a *chain complex* that we write as

$$\ldots \xrightarrow{\partial_{p+2}} C_{p+1} \xrightarrow{\partial_{p+1}} C_p \xrightarrow{\partial_p} C_{p-1} \xrightarrow{\partial_{p-1}} \ldots \xrightarrow{\partial_0} 0. \tag{6}$$

Effectively, we have replaced the simplicial complex with a sequence of algebraic modules, i.e., the chain complex. The added algebraic structuring of our chains quickly becomes useful for computation. Building upon the familiar language of vector algebra, we obtain a particularly convenient expression.

Boundary Matrices. Letting n_p denote the number of p-simplices, we saw how we can think of the chain group $(C_p, +)$ as the vector space $(\{0,1\}^{n_p}, \oplus)$. As the mapping $\partial_p : C_p \to C_{p-1}$ is well-defined, we can think of it in turn as a mapping $\partial_p : \{0,1\}^{n_p} \to \{0,1\}^{n_{p-1}}$ between vectors. Whenever a p-simplex is included in a p-chain c, we know that all $(p-1)$-simplices on its boundary contribute to the $(p-1)$-chain $\partial_p c$. Letting $\{\sigma_i\}_i$ and $\{\tau_j\}_j$ denote the sets of p-simplices and $(p-1)$-simplices, respectively, we write $c = \sum_i a_i \sigma_i$ and $\partial_p c = \sum_i a_i \partial_p \sigma_i = \sum_j b_j \tau_j$. Rearranging, we get that $b_j = \sum_i \partial_p^{j,i} a_i$, where $\partial_p^{j,i}$ is 1 if $\tau_j \prec \sigma_i$ and 0 otherwise. If we think of $[\partial_p^{j,i}]_i$ as a *column vector* for each j and the p-chain c as another column vector $[a_i]_i$, we recognize b_j as an *inner product*. Collecting all the vectors $[\partial_p^{j,i}]_{i,j}$ into a *boundary matrix*, we realize the boundary mapping as a *linear transformation* between vector spaces.

$$\partial_p c = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_{p-1}} \end{bmatrix}, \qquad \partial_p = \begin{bmatrix} \partial_p^{1,1} & \partial_p^{1,2} & \cdots & \partial_p^{1,n_p} \\ \partial_p^{2,1} & \partial_p^{2,2} & \cdots & \partial_p^{2,n_p} \\ \vdots & \vdots & \ddots & \vdots \\ \partial_p^{n_{p-1},0} & \partial_p^{n_{p-1},2} & \cdots & \partial_p^{n_{p-1},n_p} \end{bmatrix}, \qquad c = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n_p} \end{bmatrix} \tag{7}$$

With the aid of these algebraic tools, we can now start using chains and see what comes out.

Boundaries and Cycles. Unlike the whole convex polytopes considered in Euler's equation, the chains we defined may correspond to an entire simplicial complex or just a subset of its simplices. While some of the chains carry useful information about the complex, many do not. Let us examine the 1-chains on triangulations of surfaces like the ones shown in Figure 195. There are many chains that cannot help us distinguish the sphere from any of the tori, e.g., the boundary of a single triangle. In contrast, other types of chains only arise if there is a handle; they include edges that *wrap around* one or more handles. We call that latter type *cycles*. How do we extract the number of handles from these cycles?

It is easy to see a cycle, but our computations will benefit from an algebraic characterization. If α is a 1-cycle, it consists of a set of vertices each shared by two edges. It follows that $\partial_1 \alpha$ counts each vertex twice *modulo 2* yielding 0. But, the same could be said about the boundary of any set of triangles whether it wraps around a handle or not. Hence, we distinguish two subsets of p-chains that have *no boundary*: those that arise as the boundary of some $(p+1)$-chain under the action of ∂_{p+1} are the *p-boundaries* B_p, and the rest are the *p-cycles* Z_p.

As outlined above, the *fundamental lemma of homology* asserts that $\partial_p \partial_{p+1} c = 0$ for every integer p and all chains $c \in C_{p+1}$. Furthermore, as ∂_p commutes with addition, both B_p and Z_p are *subgroups* of C_p, where B_p is in turn a subgroup of Z_p.

Multiplicity of Representation. There would typically be multiple 1-cycles that wrap around a single handle. Some of those cycles are *minimal*, containing only the edges that wrap around the handle, while others contain extra 1-boundaries that carry no additional information. Namely, for any $\alpha \in Z_p$ and $\beta \in B_p$, we have that $\alpha' = \alpha + \beta \in Z_p$.

The above discussion suggests that we need to recognize *equivalent* cycles while ignoring the contribution of any boundaries. To formalize this notion, we need a few more tools from algebra.

Algebra III. Given a group $(A, \bullet)$ and a subgroup B, we define an *equivalence relation* that identifies a pair of elements $\alpha, \alpha' \in A$ whenever $\alpha' = \alpha \bullet \beta$ for some β in B. The equivalence relation partitions A into *equivalence classes* or *cosets*; the coset $[\alpha]$ consists of all the elements of A identified with α. Then, the collection of cosets, together with the operator $\bullet$, give rise to the *quotient group* A/B of the elements in A *modulo* the elements in B.

Before we apply quotients, we recall that the *order* of a group is the total number of elements, and for abelian groups, like p-chains, the *rank* is the cardinality of a maximal linearly independent subset, i.e., the number of p-simplices.

Homology Groups. We define the p-th *homology group* H_p as Z_p/B_p. Now, to count the number of p-holes, we seek to compute the rank of H_p; this rank is known as the p-th *Betti number*

$$\beta_p = \text{rank } H_p = \text{rank } Z_p - \text{rank } B_p. \tag{8}$$

The computation of the Betti numbers relies on the following fundamental theorem in algebra.

Algebra IV. Let V and W be vector spaces and $T : V \to W$ be a *linear transformation*. We define the *kernel* of T as the subspace of V, denoted $\text{Ker}(T)$ of all vectors v such that $T(v) = 0$. The remaining elements $v \in V$ for which $T(v) \neq 0$ are mapped to a subspace of W; the *image* of T. The *rank-nullity theorem* states that $\dim V = \dim \text{Image}(T) + \dim \text{Ker}(T)$.

In the context of p-chains, we get that Z_p is the kernel of ∂_p, while B_{p-1} is its image. Hence, rank C_p = rank Z_p + rank B_{p-1}. Note that $B_{-1} = 0$, and for a d-dimensional complex $Z_{d+1} = 0$.

The Euler Characteristic (Redux). Recalling the alternating sum in Euler's polyhedron formula, we can now use the Betti numbers to derive the generalized *Euler-Poincaré formula*.

$$\begin{aligned}
\chi &= \sum_{p\geq 0}(-1)^p \text{ rank } C_p = \sum_{p\geq 0}(-1)^p(\text{rank } Z_p + \text{rank } B_{p-1}) \\
&= (\text{rank } Z_0 + \cancel{\text{rank } B_{-1}}) - (\text{rank } Z_1 + \text{rank } B_0) + (\text{rank } Z_2 + \text{rank } B_1) - \dots \\
&= (\text{rank } Z_0 - \text{rank } B_0) - (\text{rank } Z_1 - \text{rank } B_1) + (\text{rank } Z_2 - \text{rank } B_2) - \dots \\
&= \sum_{p\geq 0}(-1)^p(\text{rank } Z_p - \text{rank } B_p) \\
&= \sum_{p\geq 0}(-1)^p\beta_p. \qquad (9)
\end{aligned}$$

A remarkable fact is that homology groups do not depend on the triangulation used, i.e., they are indeed *topological invariants*. Hence, the sequence of Betti numbers reveals important qualitative features of the underlying space. Now, all that remains is to compute the ranks as in Equation 8.

Matrix Reduction. As discussed above, recognizing Z_p as $\text{Ker}(\partial_p)$ we seek to compute the rank of the matrix ∂_p of dimensions rank $C_{p-1} \times$ rank C_p; see Equation 7. Using essentially the *Gaussian elimination* algorithm, we can *reduce* the matrix ∂_p *without changing its rank* by a series of transformations, i.e., *row and column operations*, into the *Smith normal form*; see Figure 203. As we work with modulo 2 coefficients, we obtain an initial segment of the diagonal being 1 and everything else being 0. Namely, the leftmost rank B_{p-1} columns have 1 in the diagonal, and the rightmost rank Z_p columns are zero. By processing all boundary matrices, we can extract the Betti numbers as the differences between the ranks β_p = rank Z_p − rank B_p. By keeping track of the reducing transformations, we can also obtain the bases of the boundary and cycle groups as subspaces of their respective chain groups.

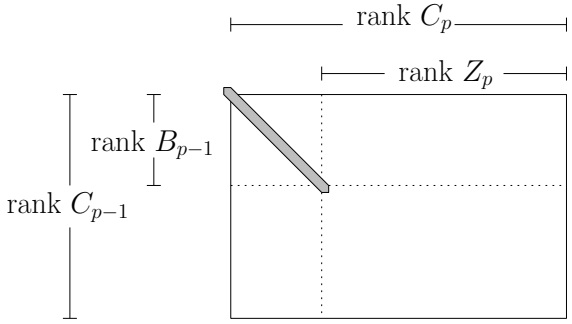


Fig. 203: Reducing the boundary matrix ∂_p to the Smith normal form.

Beyond the topological invariants of the spaces themselves, topology is also concerned with the invariants of maps between spaces.

Functoriality. Given two simplicial complexes $\widehat{X}$ and $\widehat{Y}$, a simplicial map $\widehat{f}_\Delta : \widehat{X} \to \widehat{Y}$ induces a map from the p-chains of $\widehat{X}$ to the p-chains of $\widehat{Y}$, which we denote by $\widehat{f}_\# : C_p(\widehat{X}) \to C_p(\widehat{Y})$.

Letting $\partial_{\widehat{X}}$ and $\partial_{\widehat{Y}}$ denote the boundary maps for $\widehat{X}$ and $\widehat{Y}$, respectively, we get that the induced map commutes with the boundary maps, i.e., $\widehat{f}_{\#} \circ \partial_{\widehat{X}} = \partial_{\widehat{Y}} \circ \widehat{f}_{\#}$. This can be expressed as a *commutative diagram* where all directed paths from one node to another are equivalent.

$$\begin{array}{ccccccccc} \ldots & \xrightarrow{\partial_{\widehat{X}}} & C_{p+1}(\widehat{X}) & \xrightarrow{\partial_{\widehat{X}}} & C_p(\widehat{X}) & \xrightarrow{\partial_{\widehat{X}}} & C_{p-1}(\widehat{X}) & \xrightarrow{\partial_{\widehat{X}}} & \ldots \\ & & \downarrow{\scriptstyle \widehat{f}_{\#}} & & \downarrow{\scriptstyle \widehat{f}_{\#}} & & \downarrow{\scriptstyle \widehat{f}_{\#}} & & \\ \ldots & \xrightarrow{\partial_{\widehat{Y}}} & C_{p+1}(\widehat{Y}) & \xrightarrow{\partial_{\widehat{Y}}} & C_p(\widehat{Y}) & \xrightarrow{\partial_{\widehat{Y}}} & C_{p-1}(\widehat{Y}) & \xrightarrow{\partial_{\widehat{Y}}} & \ldots \end{array} \tag{10}$$

As the induced map $\widehat{f}_{\#}$ commutes with the boundary maps, it maps boundaries to boundaries and cycles to cycles. Consequently, $\widehat{f}_{\#}$ induces a map between homology groups, which we denote by $H(\widehat{f}) : H_p(\widehat{X}) \to H_p(\widehat{Y})$. This map $H(\widehat{f})$ on homology is an *algebraic reflection* of the continuous map $\widehat{f} : |\widehat{X}| \to |\widehat{Y}|$; it is a form of *functoriality* as studied in category theory.

Important applications of functoriality involve a map $f : Y_1 \to Y_2$ that *factors through a map* to X as shown in Figure 204. If the homologies of Y_1 and Y_2 are known, then we can use the induced homomorphisms to make inferences about the homology of X.

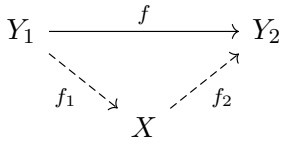


Fig. 204: $f : Y_1 \to Y_2$ with $f = f_2 \circ f_1$ where $f_1 : Y_1 \to X$ and $f_2 : X \to Y_2$.

To demonstrate this powerful proof technique, we present a remarkable and far-reaching result.

Brouwer Fixed-Point Theorem. Consider a self-map of the closed unit disc $f : \mathbb{D} \to \mathbb{D}$. A *fixed point* of f is any point $x \in \mathbb{D}$ such that $f(x) = x$. We will show that every *continuous* self-map of $\mathbb{D}$ must have a fixed point!

Assume for contradiction that $f : \mathbb{D} \to \mathbb{D}$ is continuous and has no fixed points. It would follow that for any $x \in \mathbb{D}$ there is a well-defined line passing through x and $f(x) \neq x$. Define $r(x)$ as the intersection of the ray from x towards $f(x)$ and $\partial\mathbb{D}$, i.e., the unit circle bounding the disk $\mathbb{D}$; see Figure 205. Hence, we implicitly defined $r : \mathbb{D} \to \partial\mathbb{D}$ using the self-map f. As f is continuous, so is r. In addition, $r(x) = x$ for all $x \in \partial\mathbb{D}$, i.e., r is a *retraction*. Denoting the inclusion of $\partial\mathbb{D}$ into $\mathbb{D}$ by $\iota : \partial\mathbb{D} \to \mathbb{D}$, we obtain the diagram in the middle. Passing through homology, we see that $H_1(\partial\mathbb{D})$ is isomorphic to $\mathbb{F}_2$, i.e., it has rank 1, while $H_1(\mathbb{D}) = 0$. But, as shown to the right, identity on the homology of $\partial\mathbb{D}$ maps each element to itself, while the second map on the bottom is injective, mapping 0 to exactly one element. Hence, the diagram to the right does not commute, i.e., $H_1(r) \circ H_1(\iota)(1) \neq \mathrm{Id}$, and we obtain a contradiction.

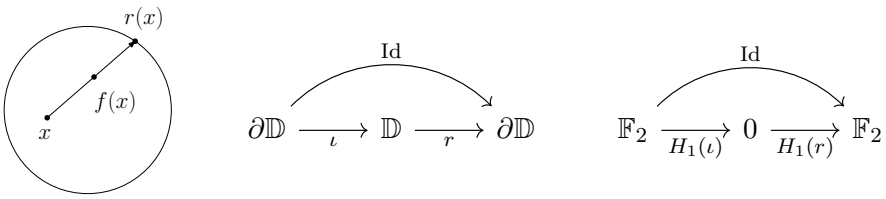


Fig. 205: Self-maps on the disk $\mathbb{D}$ with no fixed points, and a contradiction through functoriality.

The proof above generalizes to higher dimensions; for $\mathbb{D}^n$ we use H_{n-1}. This theorem is closely related to the *hairy ball theorem* establishing the impossibility of continuous and everywhere non-vanishing tangent vector fields on an even-dimensional sphere: *you can't comb the hair on a coconut!*

Beyond the surfaces we have been using in our elementary introduction, data analysis applications frequently deal with *sample points* assumed to be drawn from an *unknown* underlying manifold embedded in high dimensional space $\mathbb{R}^d$. We conclude with a brief discussion of how the techniques from above can be applied in this context, as has recently been in rapid development in the computational topology and topological data analysis research communities.

A major thrust in the development of topological approaches to data analysis is to achieve robustness against errors and imprecision in data measurements, collectively referred to as *noise*. As we have seen, topological properties are less sensitive to exact distances, which can be useful in the derivation of robust qualitative descriptors.

Homotopy Equivalence. An intuitive way to *hallucinate* the whole of a shape from a collection of sample points is to *grow* a ball around each sample and take the union of those balls. While this seems to work visually, there is a technical complication we need to consider. For example, if we take a dense sample on a 1-dimensional curve, grow a ball at each sample and take the union, we obtain a *thick* version of the curve; a tube of sorts. While the tube can be deformed continuously into the original curve, it would not be possible to define a continuous inverse, since many points on the tube will have to be mapped to the same point on the curve. Still, we would expect the union of balls to capture the topology of the original shape. This generalized notion of topological equivalence is called *homotopy equivalence*; while it is weaker form of equivalence compared to homeomorphism, it can be much more convenient.

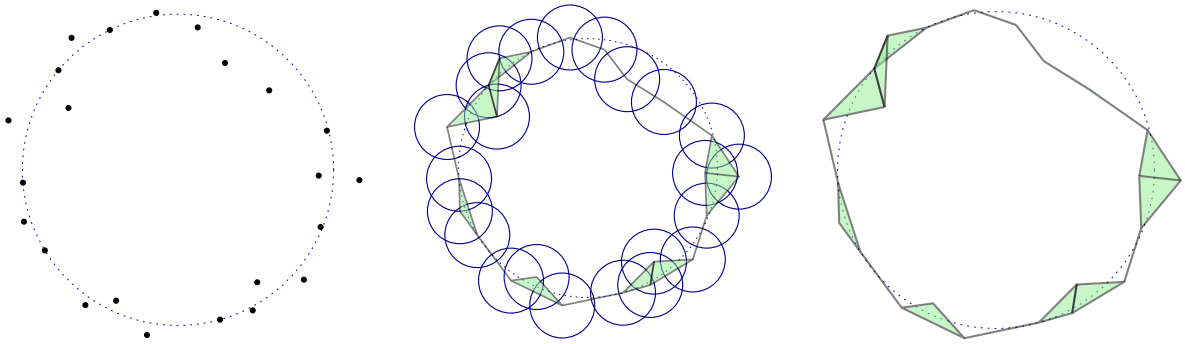


Fig. 206: Noisy samples from a circle, the union of balls and its nerve, and the Cêch complex.

Applying the idea outlined above, we work with the union of balls of a suitable radius. Unsurprisingly, we replace the geometric object represented by the union of balls by an algebraic object amenable to processing, i.e., a (abstract) simplicial complex.

Nerves. Taking the collection of balls centered at each sample point, we associate a vertex with each ball and a p-simplex with each subset of $(p+1)$ balls with a non-empty intersection. This type of complex is known as the *nerve* of the collection of objects; the balls in this case as shown in Figure 206. The homotopy-equivalence of this Cêch complex and the union of balls follows from the *nerve lemma*, which requires that the intersections of any set of objects is *contractible*, i.e., homotopy-equivalent to a point. This turns out to be the case for any collection of closed convex objects, not only balls, which can be related to *Helly's theorem*.

There remains the issue of choosing a suitable radius. In addition, the choice of radius is not completely separate from the density of the sample and the shape of the manifold. As the choice of radius impacts the topology type observed through the union of balls, how do we identify the most likely topology? This type of question motivated the recent development of a rich and exciting theory that came to be known as *persistent homology*.

To appreciate the issue of scale, we consider different choices of radii for the example in Figure 206.

Filtrations. As seen in Figure 207 below, a very small radius results in a disconnected union of balls, while an overly large radius yields a single blob with the hole *filled in*. Now, imagine a *continuous process* of growing the radii from $r = 0$ to $r = \infty$, where we think of r as a function of time t. As this process results in a sequence of nested shapes, it is called a *filtration*, with t being the *filtration parameter*. At some point, say $r(t_0) = a$, we recover the topology of the circle for the first time. Then, at a later point $t_1 > t_0$, with $r(t_1) = b > a$, the hole is filled in. Along the way, the number of connected components decreases as r goes from 0 to a.

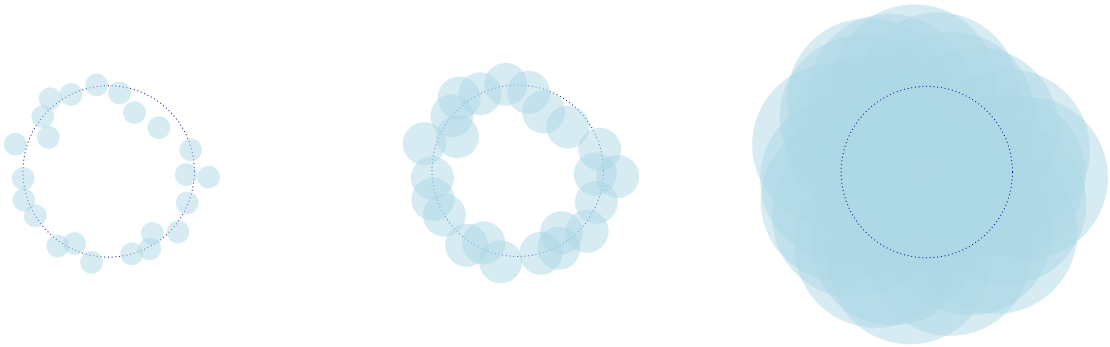


Fig. 207: Three different scales to estimate the topology from a noisy samples.

While filtrations provide a dynamical model of the evolution of topological features, we still need a way to extract the salient topological features as they appear and ultimately disappear. In addition, we would like to discard extraneous features arising due to the sampling and noise.

Persistence. We define the *birth* and *death* of a topological feature as the values of the filtration parameter when it first appears and when it gets filled in, respectively. Then, the *persistence* of the feature is the difference between the two. In the example above, the persistence of the 1-cycle is $t_1 - t_0$. Under reasonable conditions on the sampling, features with larger persistence are more likely to capture salient aspects of the underlying *shape of the data*, while features with small persistence can be disregarded, e.g., the many connected components in Figure 207.

Outlook. For this brief introduction, we did not cover the algorithmic aspects of computational topology. The matrix reduction algorithm was simply presented as a variant of Gaussian elimination, and we did not discuss the extensions needed to compute persistent homology. Other important considerations involve more compact complexes than the Cêch complex, e.g., the Vietoris-Rips complex, or the simplification of simplicial complexes to reduce their sizes without changing their homotopy type, as would be beneficial for efficient computation. Finally, the extracted persistent homology features can be summarized into convenient topological signatures, known as *persistence diagrams* and *barcodes*, with an associated metric structure such that it can be used to efficiently compare two data sets using their salient topological properties. We hope the reader will be excited to further explore these topics, while catching up on all the technical details that could not be presented here in more depth.