FUNDAMENTALS

HANG SI

Contents

In order to well-understood the discussion and the analysis of algorithms in this chapter, it is necessary to understand the fundamental geometry and combinatorics of triangulations. We will first present the basic and related materials in this chapter.

1. Mathematical Backgrounds

1.1. Convex sets, convex hulls.

1.1.1. *Linear, affine, and convex combinations.* The space $\mathbb{R}^d$ is a vector space. A *linear subspace* of $\mathbb{R}^d$ is closed under addition of vectors and under multiplication of (scalar) real numbers. For example, the linear subspaces of $\mathbb{R}^2$ are the origin $\mathbf{0}$, all lines passing through origin, and $\mathbb{R}^2$ itself.

Basic geometric objects are points, lines, planes and so forth, which are *affine subspaces*, also called *flats*. In an affine subspaces, lines are not passing through $\mathbf{0}$. An affine subspace of $\mathbb{R}^d$ has the form $\mathbf{x} + L$ where $\mathbf{x}$ is some vector and L is a subspace of $\mathbb{R}^d$. Thus non-empty affine subspaces are the translates of linear subspaces.

Let $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$ be points in $\mathbb{R}^d$, and α_1, α_2, ..., α_n be (scalar) real value in $\mathbb{R}$, then a *linear combination* of $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$ is a point:

$$\alpha_1\mathbf{a}_1 + \alpha_2\mathbf{a}_2 + \cdots + \alpha_n\mathbf{a}_n \in \mathbb{R}^d.$$

In particular, it is

- an *affine combination* if $\sum_{i=1}^n \alpha_i = 1$, or
- a *conical combination* if $\alpha_i \geq 0$, $\forall i$, or
- a *convex combination* if $\sum_{i=1}^n \alpha_i = 1$, and $\alpha_i \geq 0$, $\forall i$.

The set of all linear combination of $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$ is a linear subspace of $\mathbb{R}^d$. The set of all affine combination of $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$ is an affine subspace of $\mathbb{R}^{d-1}$. It is also called the *affine span*. The *affine hull* of a set $X \subset \mathbb{R}^{d-1}$, denoted as $\text{aff}(X)$, is the intersection of all affine span of $\mathbb{R}^d$ containing X. The *dimension* of $\text{aff}(X)$ is the dimension of the corresponding linear subspace. A set of $k+1$ points $\mathbf{a}_0, ..., \mathbf{a}_k \in \mathbb{R}^d$ is *affinely independent* if its affine hull has dimension k. Affine subspaces of dimension 0, 1, 2, and $d-1$ in $\mathbb{R}^d$ are called *points*, *lines*, *planes*, and *hyperplanes*, respectively.

1.1.2. *Convex set.* A point set $K \subseteq \mathbb{R}^d$ is *convex* if with any two points $\mathbf{x}, \mathbf{y} \in K$ it also contains the straight line segment $[\mathbf{x}, \mathbf{y}] = \{\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \ : \ 0 \leq \lambda \leq 1\}$ between them.

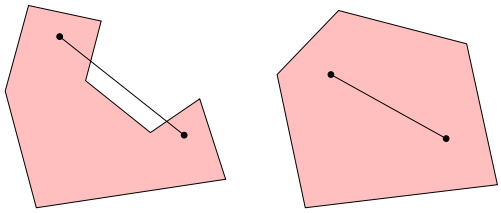

FIGURE 1. A non-convex (left) and a convex set (right).

Given two convex sets X and Y, it is easy to show that there intersection $X \cap Y$ is also a convex set, which may be $\emptyset$. Indeed, this is true for arbitrary number of convex sets. For any family $\{X_i\}, i \in I$ of convex sets, the intersection $\bigcap_{i \in I} X_i$ is also convex.

1.1.3. *Convex hull.* Given an arbitrary not necessary convex set $X \subseteq \mathbb{R}^d$, there may exist many convex sets that contain X. The *convex hull* of X, denoted as $\text{conv}(X)$, is the smallest convex set containing X.

There are many ways to define the convex hull. The following two definitions are equivalent. The convex hull of a set X is the convex combination of all points of X, i.e.,

$$\text{conv}(X) = \left\{ \sum_{i=1}^{n} a_i \mathbf{a}_i \;\middle|\; \mathbf{a}_i \in X, a_i \in \mathbb{R}, a_i \geq 0, \sum_{i=1}^{n} a_i = 1 \right\}.$$

It is also the intersection of all convex sets that contains X, i.e.,

$$\text{conv}(X) = \bigcap \{C \,|\, C \text{ is a convex set, and } X \subseteq C\}.$$

Let V be a finite set of $k+1$ points. The *affine hull* of V, $\text{aff}(V)$, is the smallest affine subspace that contains V, i.e.,

$$\text{aff}(V) = \left\{ \sum_{i=0}^{k} \alpha_i \mathbf{v}_i \;\middle|\; \mathbf{v}_i \in V, \alpha_i \in \mathbb{R}, \sum_{i=0}^{k} \alpha_i = 1 \right\}.$$

The *dimension* of $\text{aff}(V)$ is the dimension of the corresponding linear subspace. A set of $k+1$ points $\mathbf{v_0}, ..., \mathbf{v_k} \in \mathbb{R}^{\mathbf{d}}$ is *affinely independent* if its affine hull has dimension k.

The dimension of the convex hull $K = \text{conv}(X)$, $\dim(K)$, is the smallest affine subspace that contains K.

1.2. **Simplicial complexes, triangulations.** We use simpical complex as the fundamental tool to model geometric shapes and spaces. A complex is essentially a collection of certain types of basic elements satisfying some properties (more precise form will follow later). In the case of simplicial complex, these basic elements are simplices. Because of their combinatorial nature, simplical complexes are perfect data structure for geometric modelling algorithms. We first introduce the simplices and simplical complex in a geometric setting. Then we show that we can also consider it more abstractly (which makes the concepts more powerful in practice).

1.2.1. *Geometric simplicial complexes.* Recall that a set of points $\{\mathbf{p}_1, \cdots, \mathbf{p}_n\} \subseteq \mathbb{R}^d$ is *affinely independent* if $\sum_{i=1}^n \alpha_i \mathbf{p}_i = \mathbf{0}$ implies that $\alpha_i = 0$ for all i.

A *geometric k-simplex* σ in $\mathbb{R}^d$ is the convex hull of a collection of $k+1$ affinely independent points in $\mathbb{R}^d$. The *dimension* of σ is $\dim(\sigma) = k$. For example, the (-1)-simplex is the empty set ($\emptyset$), a 0-simplex is a vertex (or point), a 1-simplex is an edge, a 2-simplex is a triangle, and a 3-simplex is a tetrahedron, see Figure 2.

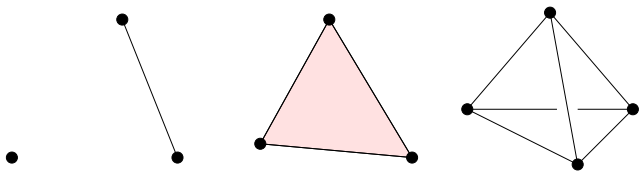

FIGURE 2. From left to right are a 0-simplex (a point), a 1-simplex (an edge), a 2-simplex (a triangle), and a 3-simplex (a tetrahedron).

A *face* τ of σ is the convex hull of any subset of the vertices of σ. It is again a simplex. $\tau = \emptyset$ and $\tau = \sigma$ are the two trivial faces of σ, all others are *proper* faces of σ. We write $\tau \leq \sigma$ or $\tau < \sigma$ if τ is a face or a proper face of σ. The number of l-faces of σ is equal to the number of ways we can choose $l+1$ from $k+1$ points, which is $\binom{k+1}{l+1}$. The total number of faces is

$$\sum_{l=-1}^{k} \binom{k+1}{l+1} = 2^{k+1}.$$

The union of all proper faces of a simplex σ is called the *boundary* $\mathrm{bd}(\sigma)$ of σ. The *interior* $\mathrm{int}(\sigma)$ of σ is $\sigma - \mathrm{bd}(\sigma)$, see Figure 3. Note that a point (a 0-simplex) has no proper face, i.e., the boundary of a point is $\emptyset$. Therefore the interior of a point is the point itself.

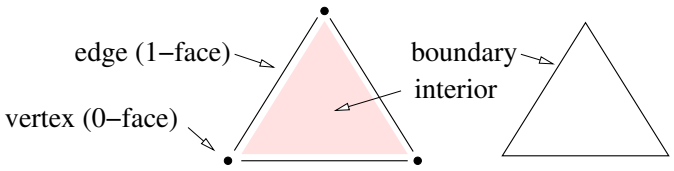

FIGURE 3. The boundary and interior of a 2-simplex (a triangle).

A *geometric simplicial complex* $\mathcal{K}$ in $\mathbb{R}^d$ is a finite collection of simplices, such that any two simplices are either disjoint or meet in a common face which is also in $\mathcal{K}$. More formally,

(i) any face of a simplex $\sigma \in \mathcal{K}$ is also in $\mathcal{K}$, and
(ii) the intersection of any two simplices $\sigma, \tau \in \mathcal{K}$ is a face of both σ and τ.

The first property implies $\emptyset \in \mathcal{K}$. The second property implies that any two different simplices in $\mathcal{K}$ have disjoint interiors, i.e., $\text{int}(\sigma) \cap \text{int}(\tau) = \emptyset$. The *dimension* $\dim(\mathcal{K})$ of $\mathcal{K}$ is the largest dimension of a simplex of $\mathcal{K}$. The *vertex set* $\text{vert}(\mathcal{K})$ of $\mathcal{K}$ is the set of all vertices of $\mathcal{K}$. Without the second property, $\mathcal{K}$ is an *abstract simplical complex*. Figure **??** illustrates a 3-dimensional geometric simplical complex and an abstract simplical complex, respectively.

A *subcomplex* is a subset of $\mathcal{K}$ that is a simplicial complex itself. Observe that every subset of a simplicial complex satisfies Condition (ii). To enforce Condition (i), we add faces and simplices to the subset. The *closure* of a subset $\mathcal{L} \subset \mathcal{K}$ is the smallest subcomplex that contains $\mathcal{L}$,

$$\text{Cl } \mathcal{L} = \{\tau \in \mathcal{K} \mid \tau \in \mathcal{L}\}.$$

A particular subcomplex is the *i-skeleton* $\mathcal{K}^{(i)}$ of $\mathcal{K}$, which consists of all simplices $\sigma \in \mathcal{K}$ whose dimension is i or less. Hence $\mathcal{K}^{(0)} = \text{vert}(\mathcal{K}) \cup \emptyset$.

We use special subsets to talk about the local structure of a simplical complex. These subsets may or may not be closed. The *star* of a complex τ consists of all simplices that contain τ, and the *link* consists of all faces of simplices in the star that do not intersect τ, i.e.,

$$\begin{aligned} \text{St } \tau &= \{\sigma \in K \mid \tau \leq \sigma\} \\ \text{Lk } \tau &= \{\sigma \in \text{Cl St}\tau \mid \tau \cap \sigma = \emptyset\} \end{aligned}$$

Figure 4 illustrates these two definitions. Note that the star of τ is not a sub complex (check it), while the link of τ is (validate it).

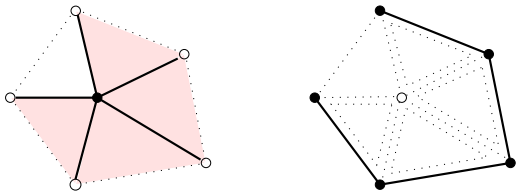

FIGURE 4. Star and link of a vertex. Left: the solid edges and the shaded triangles belong to the star of the solid vertex. Right: the solid edges and vertices belong to the link of the hollow vertex.

1.2.2. *Underlying spaces, triangulations.* Let $\mathcal{K}$ be a simplicial complex. The *underlying space* $|\mathcal{K}|$ of $\mathcal{K}$ is the union of all simplices of $\mathcal{K}$, i.e., $|\mathcal{K}| = \bigcup_{\sigma \in \mathcal{K}} \sigma$, see Fig. 5 for examples. Note that the underlying space of any (geometric) simplical complex is a geometric domain (such as a polygon in the plane or a polyhedron in 3d). This domain consists of a point set which are from the simplices of the given simplicial complex.

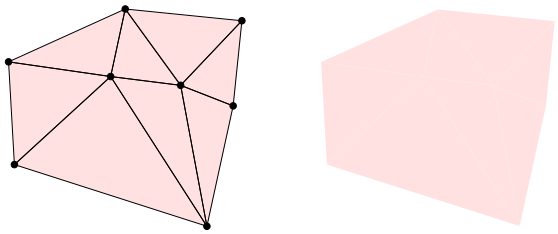

FIGURE 5. A simplicial complex (left) and its underlying space (right).

We can give each simplex its natural topology as a subspace of $\mathbb{R}^d$. We can then give $|\mathcal{K}|$ a natural topology defined as: a subset A of $|\mathcal{K}|$ is closed iff $A \cap \sigma$ is closed for any $\sigma \in \mathcal{K}$. Alternatively, if $\mathcal{K}$ is finite, since $|\mathcal{K}|$ is embedded, we can consider the subspace topology on $\mathcal{K}$ induced from $\mathbb{R}^d$. Note that these two topologies are the same only if $\mathcal{K}$ is finite. Otherwise, the first one is finer and more general than the second one. However, in this class, we will talk about only finite simplicial complexes. Hence from now on we consider the underlying space $|\mathcal{K}|$ of any finite simplicial complex equipped with the natural induced homology form $\mathbb{R}^d$ (where K embeds into)

There are many possible simplicial complexes that have the same underlying space. A simplicial complex gives a combinatorial structure on its underlying space. It is then a *triangulation* of that space. We will use this idea to define triangulations of geometric objects, such as point sets and polygonal domains.

Let S be a finite point set in the plane, a *triangulation* of S is a 2-dimensional simplical complex $\mathcal{T}$ such that

(i) the vertices of $\mathcal{T}$ are in S; and
(ii) the underlying space $|\mathcal{T}|$ is the convex hull of S.

Figure 1.2.2 shows some triangulations of a point set. Note a triangulation does not need to use all vertices of the point set.

1.2.3. *Abstract simplicial complexes. ...*

1.3. **Planar graphs, Euler's formula.** A *graph* $\mathcal{G} = (V, E)$ is a set V of vertices, and a set E of edges, each a pair of vertices of V. There are many drawings of a graph in the plane, some with and some without crossings. As illustrated in 7. A graph is *planar* if it has an embedding in the plane without crossing edges. Only graphs with relatively few edges can be drawn without crossings in the plane.

Planar graphs have applications in circuit layout and are helpful in displaying graphical data such as program flow charts, organizational charts, and scheduling conflicts.

A two-dimensional triangulation is a planar graph. Theory of planar graphs are particular important to study complexity of triangulations. In this section, we introduce the famous Euler's formula for planar graphs.

A graph is *simple* if every edge has two distinct vertices and no two edges have the same vertices. A simple graph is *connected* if there is a *path* (a sequence of edges) that connecting every pair of its vertices. The smallest connect graph are *trees*, which are characterised by having a unique simple path between every pair of vertices. Removing

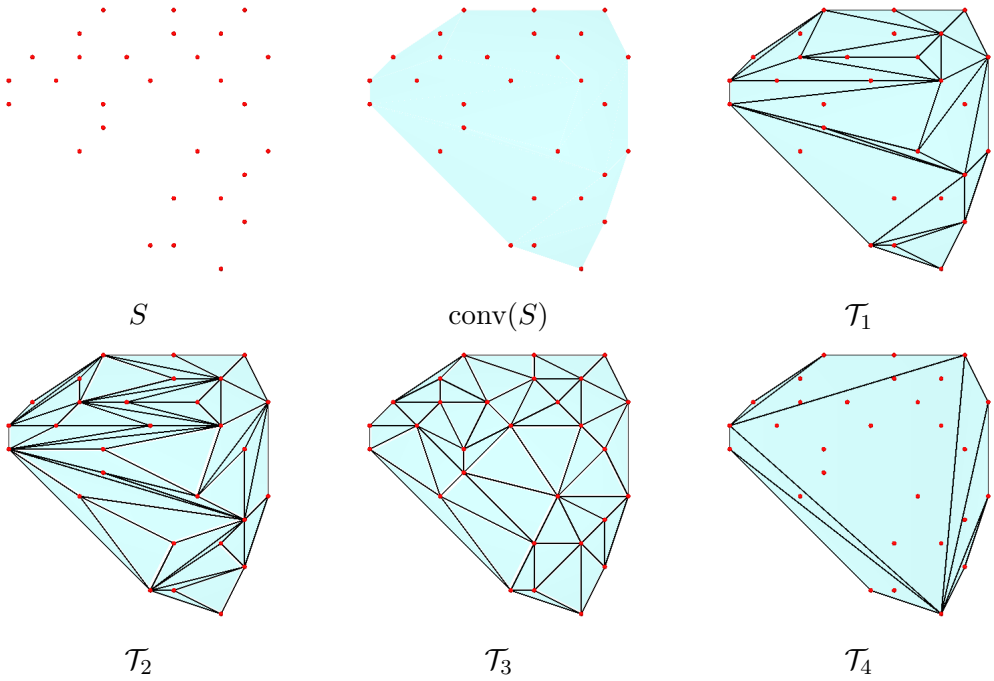

FIGURE 6. A 2d point set S, the convex hull $\mathrm{conv}(S)$, and different triangulations $\mathcal{T}_1, \ldots, \mathcal{T}_4$ of S.

any one edge disconnects the graph. A *spanning tree* of $\mathcal{G} = (V, E)$ is a tree (V, T) with $T \subset E$. It has the same vertex set as the graph and uses a minimal set of edges necessary to be connected. A graph is connected if and only if it has a spanning tree.

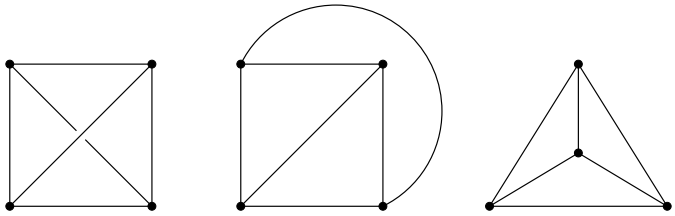

FIGURE 7. From left to right: a drawing that is not an embedding, and embedding with one curved edge, and a straight-line embedding.

Let $\mathcal{G} = (V, E)$ be a planar graph. It decomposes the plane into a set of regions, which are called *faces* of $\mathcal{G}$. Let v, e, and f denote the number of vertices, edges, and faces of $\mathcal{G}$. Euler formula is a linear relation between these numbers.

Theorem 1.1 (Euler Formula)**.** *Every connected planar graph $\mathcal{G} = (V, E)$ satisfies*

$$v - e + f = 2.$$

This formula is well-known as the Euler formula for planar graphs and convex 3d polytopes. There are plenty of proofs, see a collection of different proofs of this formula

by D. Eppstein's "Geometry Junkyard" [1]. Here we show a proof based on the dual graph and spanning trees.

Proof. Let $\mathcal{G} = (V, E)$ be a connected planar graph. Define the *dual graph* $\mathcal{G}^* = (V^*, E^*)$ of $\mathcal{G}$, such that every vertex in V^* corresponds to a face of $\mathcal{G}$, and every edge in E^* corresponds to two adjacent faces of $\mathcal{G}$. Let F be the set of faces of $\mathcal{G}$, V^* and F are bijective, and E^* and E are bijective, see Figure 8.

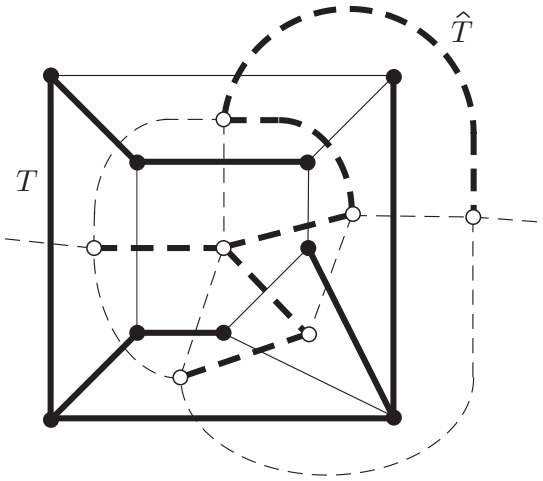

FIGURE 8. A proof of Euler formula using dual graph and spanning trees (Figure from D. Eppstein).

Choose any spanning tree $\mathcal{T}$ of $\mathcal{G}$. It has v vertices, and $v-1$ edges. The dual edges of $(\mathcal{G}-\mathcal{T})^*$ is also a spanning tree of $\mathcal{G}^*$. The two spanning trees together have $v-1+f-1$ edges, i.e., $v-1+f-1=e$, which the above formula follows. □

Euler formula is a topological and combinatorial property of that graph. It can be used to show many interesting properties of planar graphs as well as three-dimensional convex polytopes.

A triangulation is a planar graph, but it may not be maximally connected. The face outside the convex hull of this triangulation is a k-polygon, and $k \geq 3$. Using the Euler formula, we can get upper bounds on the number of edges and faces of a triangulation in terms of the number of vertices of its vertex set.

Let n, e, and f be the number of vertices, edges, and triangles of a triangulation $\mathcal{T}$ of a point set S. In a triangulation $\mathcal{T}$, every triangle have three edges, every interior edge is shared by two faces, and every convex hull edge is shared by one face, we then have:

$$3f = 2e - k,$$

where k is the number of edges on the convex hull of $\mathcal{T}$. Since $k \geq 0$, then,

$$3f \leq 2e.$$

[1] `https://www.ics.uci.edu/~eppstein/junkyard/euler/`

Using the Euler formula and the above inequality, we can bound the total number of edges and faces of $\mathcal{T}$, which are

$$\begin{aligned} e &\leq 3n-6, \\ f &\leq 2n-4. \end{aligned}$$

The *degree* (or *valency*) of a vertex u in a planar graph $\mathcal{G}$ is the number of edges at this vertex. Every edge of $\mathcal{G}$ has two distinct vertices (endpoints). The sum of vertex degrees is twice the number of edges, which must less than $6n-12$. It follows that every planar graph has a vertex whose degree is less than 6.

If a planar graph is not connected, i.e., it has more than one connected component. The Euler formula does not hold anymore. In general, the *Euler characteristic* of a d-dimensional simplical complex $\mathcal{K}$ is the alternating sum of the number of simplices

$$\chi = s_0 - s_1 + s_2 - \cdots + (-1)^d s_d,$$

where d is the dimension of $\mathcal{K}$ and s_i is the number of i-simplices in $\mathcal{K}$. It is common to omit the $(-1) - simplex$ from the sum.

Let $\mathcal{K}$ be a two-dimensional simplicial complex, $\chi = v - e + f$. We have seen that for any simply connected planar graph, $\chi = 2$. However, if a planar graph is not simply connected, i.e., it has more than one connected component, then the right hand side of the Euler formula is not necessarily 2 anymore.

2. Algorithm Backgrounds

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. It is not enough to write down algorithms. It is important to show: correctness, analysis of runtime, memory usage, etc.

Here are some very nice resources:

- Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein. Introduction to Algorithms, third edition. MIT Press/McGraw-Hill, 2009.
- Jeff Erickson: Lecture notes on Algorithms, `http://jeffe.cs.illinois.edu/teaching/algorithms`.
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. Mathematics for Computer Science, unpublished lecture notes, most recent revision January 2013. `http://opendatastructures.org/LLM.pdf`

Computational geometry emerged from the field of algorithms design and analysis in the late 1970s. It has grown into a recognized discipline with its own journals, conferences, and a large community of active researchers. The success of the field as a research discipline can on the one hand be explained from the beauty of the problems studied and the solutions obtained, and, on the other hand, by the many application domains?computer graphics, geographic information systems (GIS), robotics, and others?in which geometric algorithms play a fundamental role.

- H. Edelsbrunner. Algorithms in Combinatorial Geometry. Springer-Verlag, Heidelberg, Germany, 1987.
- Mark de Berg Otfried Cheong Marc van Kreveld Mark Overmars, Computational Geometry, Algorithms and Applications, Third Edition, Springer-Verlag Berlin Heidelberg, Germany, 2008, 2000, 1997.

2.1. **Algorithmic foundations.**

2.1.1. *A Computational model.* A commonly-used (and simple) computational model is the random access machine (RAM) model. It models a single-processor computer with a random access memory. A RAM consists of a read-only input tape, a write-only output tape, a program and a (random acess) memory. The memory consists of registers each capable of holding a real number of arbitrary precision. There is also no upper bound on the memory size. All computations take place in the processor. A RAM can access (read or write) any register in the memory in one time unit when it has the correct address of that register.

The following operations on real numbers can be done in unit time by a random access machine:

- Arithmetic operations, ...
- Comparisons
- Indirect access

2.1.2. *Asymptotic notations.* To analysis an algorithm requires the evaluation/estimation of its runtime and the amount of memory used.

A simple and generic way to analyze algorithms is to use a mathematical function f related to the size n of the inputs of the algorithm, where for instance, n is the number of points to be inserted in the triangulation.

Finding the exact value of $f(n)$ might be very difficult or even impossible. However, we are not usually interested in this value, but rather a rough estimate. For example, the question like "how does the running time scale with the size of the input?". This is called *asymptotic analysis*, and the idea is that we will ignore low-order terms and constant factors, focusing on the shape of the running time curve.

Asymptotic notation is a shorthand used to give a quick measure of the behavior of a function $f(n)$ as n grows large. We typically use n to denote the size of input, and $T(n)$ to denote the running time of our algorithm on an input of size n

Several ways of quantifying this value exist. The usual notations are $f(n) = \Theta(g(n))$, or $\Omega(g(n))$, or $O(g(n))$, or $o(g(n))$.

- (Big Oh): $T(n) \in O(f((n))$ if there exists constants $c > 0$, $n_0 > 0$, such that $T(n) \leq cf(n)$ for all $n > n_0$.

  Informally, we can view this as "$T(n)$ is proportional to $f(n)$, or better, as n gets large." For example, $3n^2 + 17 \in O(n^3)$. Obviously, $3n^2 - 2n \in O(n^2)$. Now let us show that $3n^2 + 17 \in O(n^2)$, we need to show that

  $$3n^2 + 17 < cn^2,$$

  for $n > n_0$. We pick the constant $c = 3 + 17 = 20$, and $n_0 = 1$, then for $n > n_0$, we have $3n^2 + 17 < 3n^2 + 17n^2 = 20n^2$.

  Big Oh is the most frequently used asymptotic notation. It is used to give an upper bound on the growth of a function, such as the running time of an algorithm.
- $T(n) \in \Omega(f(n))$ if there exists $c, n_0 > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

  Informally, we can view this as "$T(n)$ is proportional to $f(n)$, or worse, as n gets large." For example, $3n^3 \in \Omega(n^2)$. Obviously, $3n^2 + 17 \in \Omega(n^2)$ by choosing

$c = 1, n_0 = 1$. Let us show that $3n^2 - 2n \in \Omega(n^2)$. We need to show that

$$0 < cn^2 < 3n^2 - 2n,$$

for all $n > n_0$. If $c < 3 - \frac{2}{n}, n > 1$, the above inequality holds. We could choose $c = 1.0$, $n_0 = 1$, then for $n > n_0$, we have $n^2 < 3n^2 - 2n$. In fact, choose any constant $0 < c < 1$ and $n_0 = 1$ will be a proof.

This notation is especially useful for lower bounds.

- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.

  Informally we can view this as "T(n) is proportional to $f(n)$ as n gets large." For example, both $3n^2 - 2n$ and $3n^2 + 17$ are in $\Theta(n^2)$.

  This notation is used to express an asymptotically tight bound.
- $T(n) \in o(f(n))$ if for all constants $c > 0$, there exists $n_0 > 0$ such that $T(n) < cf(n)$ for all $n > n_0$.

Very informally, O is like $\leq$, Ω is like $\geq$, Θ is like $=$, and o is like $<$. Although these notations, e.g, $O(f(n))$ represents a set of functions, we usually write $T(n) = O(f(n))$ to mean that $T(n)$ is in $O(f(n))$.

2.1.3. *Complexity of algorithms.* Suppose X is an algorithm (to solve a problem). The running time and memory usage are the two main factors, which decide the efficiency of X. The running time is measured by counting the key operations such as the comparisons in sorting algorithms. The memory usage is measured by the maximum memory required by the algorithm.

There are two types of running time analysis of algorithms: worst case and expected case.

For the worst case analysis, we seek the maximum amount of time used by the algorithm for all possible inputs.

For the expected analysis we normally assume a certain probabilistic distribution on the input and study the performance of the algorithm for any input drawn from the distribution.

Mostly, we are interested in the asymptotic analysis, i.e., the behavior of the algorithm as the input size approaches infinity.

Let A be an algorithm. The time complexity of A is $O(f(n))$ if there exists a constant c such that for every integer $n \geq 0$, the running time of A is at most $c \cdot f(n)$ for all input of size n.

2.1.4. *Complexity of problems.* The time complexity of a know algorithm for a problem gives us the information about *at most* how much time we need to solve the problem. We would also like to know the *minimum* amount of time we need to solve the problem.

A function $u(n)$ is an *upper bound* on the time complexity of a problem $\mathcal{P}$ if there is an algorithm A solving $\mathcal{P}$ such that the running time of A is $u(n)$. A function $l(n)$ its a *lower bound* on the time complexity of a problem $\mathcal{P}$ if *any* algorithm solving $\mathcal{P}$ has time complexity at least $l(n)$.

For example, the lower bound for sorting is $\Omega(n \log n)$. It is proven by the *decision tree model* of computing, in which only comparisons can be performed. It has been proven that the complexity of the convex hull problem in plane is the same as sorting [**?**].

2.2. **Incremental construction.** In this section, we introduce a common and simple approach for building geometric structures such as convex hulls as well as triangulations. It is called *incremental construction*. In such an algorithm objects are added one at a time and the structure is updated with each new insertion.

We introduce the basic framework of incremental construction through a basic geometry problem – construction of the convex hull of a given point set in the plane.

2.2.1. *Convex hull construction.* Recall that the *convex hull* of a (not necessarily convex) set of points K is the smallest convex set that contains the points. It is denoted as $\text{conv}(K)$. The **convex hull problem** is: Given a finite set of points S in $\mathbb{R}^d$, return $\text{conv}(S)$. In this section, we consider the convex hull problem in the plane.

A *polygon* is a planar figure that is described by a finite number of straight line segments connected to form a closed path. In general, the path of a polygon may cross itself. A *simple polygon* is a polygon whose path does not intersect itself.

The Jordan curve Theorem [**?**, Jordan 1887] shows that any simple polygon divides the plane into two regions, the region inside it and the region outside it. The convex hull of a two-dimensional point set is a simple polygon.

The line segments of a simple polygon are called *edges*, and two edges meet at a *vertex*. A simple polygon with n vertices has n edges. It is also called an *n-gon*. The graph of a simple polygon is a cyclic sequence of vertices (or edges). It can be represented by a doubly-linked data structure.

There are plenty of methods for constructing convex hulls, see, e.g., [2, Chapter 19]. We will introduce an incremental algorithm. The basic idea of an incremental convex hull algorithm is that points (objects) are added one at a time, and the convex hull (geometric structure) is updated with each new insertion.

The input of this algorithm is a set S of n vertices in the plane. For simplifying our describing the basic scheme of the incremental algorithm, we assume the input point set satisfies a **condition**: no three of its points lie on a common line. Any point set satisfies this condition is said in *general position*. This assumption avoids lots of *degenerated cases* (or *special cases*), such as two points share the same location, or three or more points lie on the same line. In the latter case, the convex hull with collinear edges must be merged. With the general position assumption, the basic algorithm is given in 10.

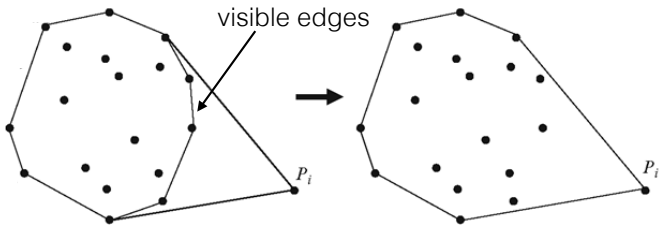

FIGURE 9. Incremental construction of convex hull.

The algorithm initialises a triangle, which is the convex hull of the first three points.

Algorithm: IncrementalConvexHull(S)
Input: $S = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$ is a set of n points in the plane;
Output: conv(S);
1 Initialise conv(S_3) := conv($\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$);
2 **for** $i = 4$ to n **do**
3 **if** $\mathbf{p}_i$ lies outside conv(S_{i-1}); **then**
4 let Q be the set of all *visible edges* of $\mathbf{p}_i$;
5 Q is a chain of edges, let $\mathbf{p}_a$, $\mathbf{p}_b$ be the endpoints of Q;
6 insert edges $\{\mathbf{p}_a\mathbf{p}_i\}$ and $\{\mathbf{p}_b\mathbf{p}_i\}$ into conv(S_{i-1});
7 remove edges in Q from conv(S_{i-1});
8 **endif**
9 **endfor**

FIGURE 10. The incremental convex hull algorithm.

Now consider adding a new point to the convex hull. If the point lies in the interior of the convex hull, then the convex hull does not change, one can simply skip this point, and go to process the next one.

Assume the point lies outside the convex hull. Then the convex hull needs to be updated. Every edge of a convex hull defines a unique *supporting line* which contains this edge and the convex hull lies in only at one side of this line. We call an edge of the convex hull *visible* by a point if this point lies in the other side of its supporting line which does not contain the convex hull, in other words, the supporting line of this edge separates the convex hull and this point, see Figure 2.2.1.

The following steps are performed to update the convex hull: (1) locate a *visible edge* of the convex hull for the point; (2) construct a cone from the new point to all of its visible edges, see Figure 2.2.1 Left, and (3) delete all visible edges of this point in Step 2, and enlarge the convex hull by adding the two cone edges thus forming the convex hull with the new point and the previously processed points, see Figure 2.2.1 Right.

With the assumption that S is in general position. conv(S_3) is a triangle (line 1). In line 5, $\mathbf{P}_a$ and $\mathbf{P}_b$ are not collinear with $\mathbf{p}_i$.

2.2.2. *The* `orient2d` *predicate.* We need a method to check whether a convex hull edge is visible by a newly added point. This is a geometric query (called *predicate*), denoted as `orient2d`, which takes three ordered points $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ in the plane, and test whether they follow a counterclockwise or clockwise direction, see Figure 2.2.2, likewise, it tells that the last point $\mathbf{r}$ lies in left or right side of the oriented edge $\mathbf{pq}$. Note that the orientation depends on the order in which the points are given.

This predicate can be answered by taking the sign of the signed area of the triangle with vertices $\mathbf{p} := (p_x, p_y)$, $\mathbf{q} := (q_x, q_y)$ and $\mathbf{r} := (r_x, r_y)$, i.e,

$$\texttt{orient2d}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \text{sign}(\det(A)),$$

where

$$A = \begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}$$

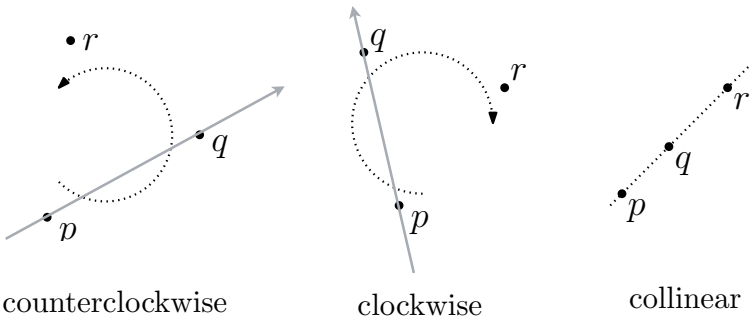

FIGURE 11. Orientations of three points in the plane.

By our choice of the matrix A, we have

$$\begin{array}{lcll} \texttt{orient2d}(\mathbf{p},\mathbf{q},\mathbf{r}) & > & 0 & \longrightarrow \mathbf{p},\mathbf{q},\mathbf{r} \text{ are counterclockwise,} \\ \texttt{orient2d}(\mathbf{p},\mathbf{q},\mathbf{r}) & < & 0 & \longrightarrow \mathbf{p},\mathbf{q},\mathbf{r} \text{ are clockwise} \\ \texttt{orient2d}(\mathbf{p},\mathbf{q},\mathbf{r}) & = & 0 & \longrightarrow \mathbf{p},\mathbf{q},\mathbf{r} \text{ are collinear.} \end{array}$$

The result of $\det(A)$ is twice of the signed area of the triangle defined by the three points.

Remark. If two columns of A in above are interchanged, e.g.,

$$B = \begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix},$$

then we have $\det(A) = -\det(B)$. The signs of the above `orient2d` test must be also inverted.

2.2.3. *Correctness.* We can show that the above convex hull algorithm is correct as long as the point set S is in general position.

In order to show that an incremental algorithm is correct, we need to establish the invariant that holds after each iteration of the loop.

Claim: After the insertion of point $\mathbf{p}_i$, the convex hull of S_i is constructed.

2.2.4. *Basic analysis of running time.* The complexity of this algorithm can be estimated in terms of n, the total number of points, and h, the number of points (edges) on the convex hull.

The cost to compute the new convex hull of a new point $\mathbf{p}_i$ lies in two parts:

(1) find all the visible edges to form Q; and
(2) update the convex hull by adding two edges $\mathbf{p}_a\mathbf{p}_i$ and $\mathbf{p}_b\mathbf{p}_i$.

We first consider (2), which is clearly related to the size of Q, i.e., the number of visible edges for point $\mathbf{p}_i$. Since $\mathbf{p}_i$ can be an arbitrary point in S, and the intermediate convex hull $\text{conv}(S_{i-1})$ is also arbitrary, it is difficult to know exactly the size of Q for $\mathbf{p}_i$. The key fact is that once an edge appear in Q, it will be deleted after inserting $\mathbf{p}_i$, and it will never reappear later. Only two new edges are created at $\mathbf{p}_i$. By the Euler's formula, the overall size for all Q's is $O(n)$. Hence (2) only requires $O(n)$ time.

Consider (1). What is the time we need to find all visible edges for $\mathbf{p}_i$. A easy way to search all visible edges for $\mathbf{p}_i$ is simply check all edges in the current convex hull. Let h_{i-1} be the number of edges in the convex hull of $\text{conv}(S_{i-1})$. Then the total cost of searching n convex hulls is: $h = h_3 + h_4 + \cdots + h_{n-1}$. Let h_c be the maximum number in $\{h_3, \cdots, h_{n-1}\}$. The running time is clearly $O(nh_c)$. If h_c is $O(1)$, i.e., it is bounded by a constant independent of n, then the total cost is $O(n)$. However, if h_c is large, such that $h_c = \Omega(n)$, then total cost of this step is $O(n^2)$.

In summary, depending on the input point set and the sequence of insertion, the IncremetalConvexHull algorithm may run in $O(n)$ and $O(n^2)$ time. Hence the worst case running time is $O(n^2)$.

2.2.5. *Improving the running time by sorting.* Indeed, we just need to consider how to find the first visible edge e of $\mathbf{p}_i$. Then the other visible edges can be found by searching the doubly linked list both clockwise and counterclockwise starting from e to find the complete list of visible edges. This only takes $O(1)$ time.

We can clearly, improve this algorithm by pre-sorting the given set S. A simple way is to sort the point set along a fixed direction, for example, the x- or y-axis, then insert points using the sorted sequence. This guarantees that each newly added point is outside the current hull. Moreover, the last inserted point, e.g., $\mathbf{p}_{i-1}$ must be connected by a visible edge of $\mathbf{p}_i$. Hence the cost of location a visible edge for $\mathbf{p}_i$ also takes $O(1)$ time. It is known that the sorting of a set of vertices along a sweep line can be done in time $\Theta(n \log n)$. Thus this incremental algorithm with a line sweep sorting constructs the convex hull in $\Theta(n \log n)$ time.

The `quickhull` [1] algorithm can be though of a special way of sorting points which is suitable for constructing convex hulls. The basic idea is following: it maintains a *outside set* for each convex hull facet through out the process. A point is in a facet's outside set if it is above the facet. An unprocessed point is always in exactly one outside set. The `quickhull` algorithm [2] choses the furthest point in an outside set and to insert it. It is intuitive, by this way of choosing inserting points, most of the interior points are skipped, hence the temporary side of the convex hull will be reduced. However, there is no guarantee that this is always the best choice of new vertex.

It turns out, if we choose new point randomly, this will always result a good expected runtime. We will discuss this in the next section.

2.2.6. *A line-sweep triangulation algorithm.* By slightly modifying the incremental convex hull algorithm in Figure 10 one can obtain a triangulation of S as well. Instead of creating a cone (in line 5) and deleting the visible edges (in line 6), we create triangles by joining each visible edge in Q and the point $\mathbf{p}_i$. This resulting a triangulation (instead of the convex hull) of $\{\mathbf{p}_1, \ldots, \mathbf{p}_i\}$.

We then have a simple and efficient algorithm using line-sweeping [3] to construct triangulations for a set of points. The basic idea is to sort the point set along a fixed direction (for example, the x-axis), then use a (vertical) line that sweeps over the plane from left to right. The triangulation is created online during the line sweeping. An invariant is: at any moment in time, the partial triangulation contains all points to the left of the line. When the line hits a new vertex (an event), the triangulation is

[2]The source code of `quickhull` is freely available at `http://www.quickhull.org`.

augmented by creating new triangles connecting to this new vertex. The algorithm is given in Figure 12.

Algorithm: LineSweep(S, $\mathbf{s}$)
Input: A set S of n points in $\mathbb{R}^2$, $\mathbf{s}$ is the normal of sweep line;
Output: A triangulation $\mathcal{T}$ of S;
1 sort the points in S into a sequence $L := \{\mathbf{p}_1, \dots \mathbf{p}_n\}$ along $\mathbf{s}$;
2 initialize $\mathcal{T}$ with only one triangle $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$;
3 **for** $i = 4$ to n **do**
4 let Q be the set of all visible edges of $\mathbf{p}_i$;
($Q \neq \emptyset$ since $\mathbf{p}_i$ locates outside conv(S_{i-1}).)
5 create new triangles to $\mathcal{T}$ by each edge in Q and $\mathbf{p}_i$;
6 **endfor**

FIGURE 12. The sweep line triangulation algorithm.

In line 1, the vertices in L are ordered in such a way, that no conflict will occur during the algorithm. A simple order is the lexicographic (dictionary) order along the x- or y-axis. For the initial triangle to be valid, it is necessary to assume the general position, i.e., $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ are not collinear. To efficiently obtain the set Q (in line 4), one could start from the last newly created triangle, which must contain a hull edge e. Then the set of all visible edges of $\mathbf{p}_i$ can be collected by a breadth search from e. Figure 13 illustrates an example of this algorithm.

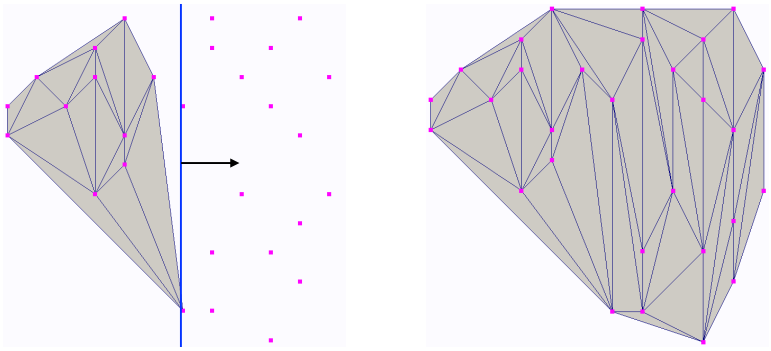

FIGURE 13. The line-sweep algorithm yo construct a triangulation.

The sort of a set of vertices along the sweep line can be done in time $O(n \log n)$. The number of visible edges of $\mathbf{p}_i$ is a constant independent of n. The total number of newly created triangles is less than $2n - 4$. Thus this line sweep algorithm constructs a triangulation in $O(n \log n)$ time.

2.3. **Randomized algorithms.** A Randomised algorithm uses a random number generator as a key input for decision making. They are simple and frequently with good expected running time.

2.3.1. *The basics of probability analysis.* Consider rolling two dice and observing the results. It could be that the first die comes up 1 and the second comes up 2, or that the first comes up 2 and the second comes up 1, and so on. There are $6 \times 6 = 36$ possible outcomes. Each of these outcomes has probability $1/36$ (assuming these are fair dice). Suppose we care about some quantity such as "what is the probability the sum of the dice equals 7?" We can compute that by adding up the probabilities of all the outcomes satisfying this condition (there are six of them, for a total probability of $1/6$).

In the language of probability theory, any probabilistic setting is defined by a *sample space* $\mathcal{S}$ The points of the sample space are called *outcome*. In the above example, the sample space S for the pair of dice has 36 outcomes. Each outcome of the sample space has a probability, e.g, $1/36$. A *probability function* on a sample space S is a total function $\Pr : \mathcal{S} \to \mathbb{R}$ such that

- $\Pr[\omega] \geq 0$ for all $\omega \in \mathcal{S}$, and
- $\sum_{\omega \in \mathcal{S}} \Pr[\omega] = 1$.

A sample space together with a probability function is called a *probability space*.

An *event* is a subset of the sample space. For instance, one event we might care about is the event that the first die comes up 1. Another is the event that the two dice sum to 7. For any event $E \subseteq \mathcal{S}$, the *probability of an event* is just the sum of the probabilities of the outcomes contained inside it, i.e,

$$\Pr[E] := \sum_{\omega \in E} \Pr[\omega].$$

A *random variable* X is a function from a probability space to a number, i.e., it maps every outcome of the sample space $\mathcal{S}$ to an integer or a real. For instance, another way we can talk formally about these dice is to define the random variable X_1 representing the result of the first die, X_2 representing the result of the second die, and $X = X_1 + X_2$ representing the sum of the two. We could then ask: what is the probability that $X = 7$?

One property of a random variable we often care about is its *expectation*. The *expected value* of a (discrete) random variable X over sample space $\mathcal{X}$ is:

$$\mathbf{E}[X] := \sum_{\omega \in S} \Pr[\omega]\, X(\omega).$$

In other words, the expectation of a random variable X is just its average value over $\mathcal{S}$, where each outcome ω is weighted according to its probability. For instance, if we roll a single die and look at the outcome. The random variable $X : \mathcal{S} \to \mathbb{R}$ is simply the value of the outcome die (between 1 and 6). Assume every outcome has equal probability, which is $1/6$, then the expected value is:

$$\mathbf{E}[X] := 1\frac{1}{6} + 2\frac{1}{6} + \cdots + 6\frac{1}{6} = 3.5.$$

Another example is toss a coin. Use H means Head, and T means Tail, respectively. The sample space has only two outcomes, $\mathcal{S} = \{H, T\}$, each with probability $1/2$ (assume it is a fair coin). Define a random variable R such that $R(H) = 1$ and $R(T) = 3$. Then the expected value of R is:

$$\mathbf{E}[R] := 1\frac{1}{2} + 3\frac{1}{2} = 2.$$

An *indicator random variable* is a random variable which maps each outcome to either 0 or 1. Indicator random variables are useful to handle events. In particular, an indicator random variable partitions the sample space into those outcomes mapped to 1 and those outcomes mapped to 0. More generally, for any partition of the probability space into disjoint events $A_1, A_2, \cdots$ we can rewrite the expectation of random variable X as:

$$\mathbf{E}[X] := \sum_i \sum_{\omega \in A_i} \Pr[\omega]\, X(\omega) = \sum_i \Pr[A_i] \mathbf{E}[X|A_i],$$

where $\mathbf{E}[X|A_i]$ is the expected value of X given A_i, defined to be

$$\mathbf{E}[X|A_i] := \frac{1}{\Pr[A_i]} \sum_{\omega \in S} \Pr[\omega]\, X(\omega).$$

An important fact about expected values is Linearity of Expectation: for any two random variables X and Y,

$$\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y].$$

This fact is incredibly important for analysis of algorithms because it allows us to analyse a complicated random variable by writing it as a sum of simple random variables and then separately analysing these simple ones.

2.3.2. *Randomised incremental convex hull construction.* As an example, we describe a randomized incremental convex hull algorithm and analyze its expected running time.

This algorithm is basically the same as the incremental algorithm given in Figure 10. The only difference is how to compute the *visible edges* for an outside new point $\mathbf{p}_i$, i.e., in line 4. The key idea in computing visible edges efficiently is to precompute them: for each point not yet inserted, keep track of one edge of the current convex hull that is visible from it. With this edge available, the set of all visible edges can be found in $O(1)$ in the doubly linked list.

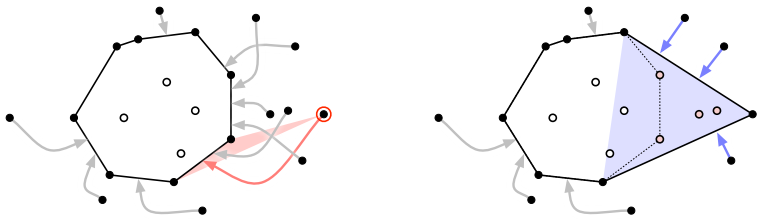

FIGURE 14. Left: Each uninserted point maintains pointer to a visible edges on the convex hull. Right: some visible edges will be changed and updated during the incremental construction of the convex hull. Courtesy of Jeff Erickson.

One way to realise this is for each uninserted point $\mathbf{p}_i$ maintain a pointer to one of its visible edges on the current (growing) convex hull, see Figure 14 Left. Initially, for all points $\mathbf{p}_4, \cdots, \mathbf{p}_n$, we assign one of the edges of the triangle $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ to them, computed by the `Orient2d()` predicate. Later on, at the insertion of an outside point $\mathbf{p}_i$, some old convex hull edges will be removed, and two new convex hull edges containing $\mathbf{p}_i$ are added to the convex hull. The edges assigned to those uninserted points need to be

updated, see Figure 14 Right. The question is: How much will be the total cost to update the visible edges for all points? It certainly depends on the insertion order of points. It is possible if we insert points in a bad order, each insertion may cause $O(n)$ points whose visible edges need to be updated. Such an example is shown in Figure 15. Hence the total cost is $O(n^2)$. It turns out, if the points $\mathbf{p}_4, \cdots, \mathbf{p}_n$ are randomly permuted, the expected cost is $O(n \log n)$.

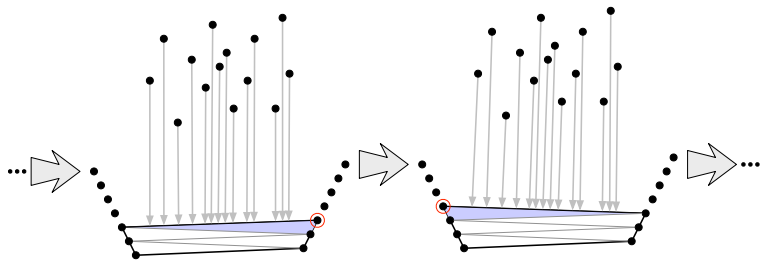

FIGURE 15. Inserting points in a bad order will have a total cost $\Omega(n^2)$. Courtesy of Jeff Erickson.

We apply backward analysis [4] to show the expected running time of assigning and reassigning future points to edges. Whenever we add a new edge, we pay a cost equal to the number of future points assigned to that edge. We consider the incremental algorithm backwards: on each iteration, remove a random point $\mathbf{p} \in \{\mathbf{p}_4, \cdots, \mathbf{p}_n\}$. If $\mathbf{p}$ is inside the current convex hull, pay nothing. Otherwise, remove the two edges of the convex hull that contain $\mathbf{p}$, and pay cost equal to the number of "future" points which are outside these two edges (where "future" is technically the past now). If there are i points remaining, then each "future" point has probability at most $2/i$ of contributing to the cost of this iteration. By linearity of expectation, the expected cost of this iteration is $(n-3-i) * 2/i$ (we do not count the first 3 points of the initial triangle). Summing over all i gives an expected total cost of

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

Thus the total expected running time of the randomised incremental convex hull algorithm is $O(n \log n)$.

EXERCISES

1. Let $X \subset \mathbb{R}^d$ be a set of n points. Show the following definitions of the convex hull of X are equivalent:
   (1.1) The set of all points that are convex combinations of all points in X.
   (1.2) The intersection of all convex sets that include X.
2. Convex hull questions:
   (2.1) Is empty set convex?
   (2.2) Is a set with only one point convex?
3. The interior of a simplex is defined as the simplex without its boundary.

(3.1) What is the minimum number of points to create a simplex with non-empty interior?

4. Let $\mathcal{K}$ be a two-dimensional simplicial complex. Claim: The interior of simplices of $\mathcal{K}$ partition the underlying space of $\mathcal{K}$.
   (4.1) Is this claim true or false?
   (4.2) Show your reasons for your answer of (4.1).
5. Asymptotic notations. Show the following equations hold.
   (5.1) $n^2 + 3n + 20 = O(n^2)$.
   (5.2) $n \log n - 2n + 13 = \Omega(n \log n)$;
   (5.3) $n^2 + 5n + 7 = \Theta(n^2)$.

References

[1] C. Bradford Barber, David P. Dobkin, and Hannu T. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Tranactions on Mathematical Software*, 22(4):469–483, 1996.

[2] Jocob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry and Its Applications*. CRC, Boca Raton, Raton, Florida 33431, 2000.

[3] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM*, 25(10):739–747, October 1982.

[4] Raimund Seidel. *Backwards Analysis of Randomized Geometric Algorithms*, pages 37–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.