Chapter 9

Incremental algorithms

An incremental algorithm inserts the input objects one by one, and after each insertion it updates the object being constructed. For instance, for a construction of the convex hull of points $p_1, \ldots, p_n$, an incremental algorithm successively computes the convex hulls of the sets $\{p_1\}$, $\{p_1, p_2\}, \ldots$, $\{p_1, \ldots, p_n\}$. The efficiency may depend essentially on the order of insertion. First we discuss simpler examples, in which either the order of insertion doesn't matter, or where a good insertion order can easily be described. Later on, we show that in numerous more complicated situations, an efficient algorithm is obtained by inserting the objects in random order.

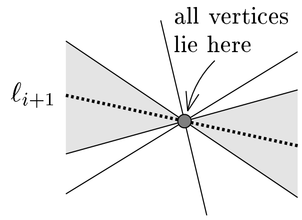
9.1 Construction of a line arrangement

Problem 9.1.1 *Given a set L of n lines in the plane, we want to construct the arrangement $\mathcal{A}(L)$.*

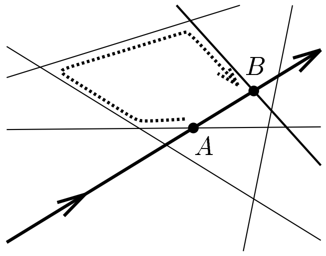
The arrangement can be represented, for instance, by specifying the incidence relations between vertices and edges and between edges and 2-faces (more precisely, each vertex and each 2-face points to a list of its adjacent edges, and each edge points to its vertices and to its 2-faces).

A simple solution. For each line, compute the intersections with all other lines, and sort the intersections along the line. This yields edges and vertices, and it is a simple exercise to get the above representation from this information. The running time is dominated by the n sortings, which together need $O(n^2 \log n)$ steps. Next, we describe an incremental algorithm with $O(n^2)$ running time.

An incremental algorithm. Suppose that i input lines have already been processed (i.e. their arrangement has been constructed), and we are now inserting a line ℓ_{i+1} (the order of insertion is arbitrary). First, we want to find the 2-face containing the left unbounded ray of the new line. We can afford to spend $O(i)$ time on that, and hence we can for instance inspect all the unbounded faces of the arrangement of the first i lines (these have $O(i)$ total complexity by the Zone theorem 6.3.1). A simpler and more effective solution is possible: note that if we look at a line arrangement from a very large distance, it looks like a collection of lines intersecting at a single point, and hence an unbounded face intersected by ℓ_{i+1} can be found by comparing the slope of ℓ_{i+1} with the slopes of the other lines.



Further we trace the new line ℓ_{i+1} through the cells of the arrangement. For instance, having reached a point A as in the following picture



we walk along the boundary of the 2-face (as indicated by arrows) until the point B. For each edge on the boundary of the current 2-face, we test whether it already intersects our line ℓ_{i+1}. In this way we find out how should the old 2-face be split into two new 2-faces. Having found the point of exit of the new line from the current 2-face, we split the 2-face and the exit edge and we update the incidences appropriately. Then we proceed to the next 2-face, until we reach the right unbounded ray of ℓ_{i+1}.

It remains to determine the complexity of this algorithm. The cost of inserting the $(i+1)$st line is proportional to the number of edges we walk through. We note that these edges all lie in the zone of ℓ_{i+1} in the arrangement of the first i lines. The complexity of the zone is $O(i)$ by the Zone theorem 6.3.1. The total running time is $O(n^2)$.

An arrangement of hyperplanes in a higher dimension can be constructed similarly (the details are somewhat more complicated, though). The resulting complexity of the algorithm is $O(n^d)$, again by the Zone theorem.

9.2 Convex hull computation in the plane

We are given n points in the plane, and we want to compute (a suitable representation of) their convex hull. If the reader has seen no such algorithm yet, we recommend him very strongly to design at least one planar convex hull algorithm by himself before reading further!

A very simple incremental convex hull algorithm proceeds as follows.

1. Sort the points by their x-coordinate (in $O(n \log n)$ steps). Let p_i denote the ith leftmost point (for simplicity, assume that no two input points have the same x-coordinate and no three of them are collinear).

2. Consider the points from left to right, and construct the convex hull of the already encountered points; after the ith step, the convex hull of the points $p_1, p_2, \dots, p_i$ will have been constructed.

The convex hull is first a single point, then a segment, then a triangle. We need a general method for adding the new point to the current convex hull. Since the points are sorted by their x-coordinate, the newly inserted point p_i is always a vertex of the convex hull after the ith step (since it has the maximum x-coordinate).

We have to detect (and delete) the points which became interior points of the convex hull because of the newly inserted point. These are the vertices of the previous convex hull "visible" from the newly added point p_i but such that they do not form an edge of the new convex hull together with p_i (see fig. 9.1). Note that p_i always sees the point p_{i-1}.

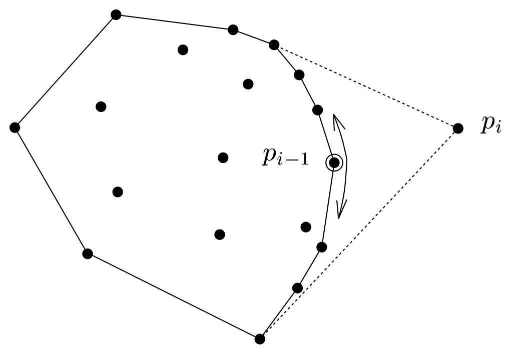


Figure 9.1: Adding a new point to the convex hull

To find the points to be deleted, we go from p_{i-1} in both directions until we reach the last point visible from p_i. Tthe visibility is tested using the angle as seen from p_i: For instance, if we walk from p_{i-1} upwards, the angle first increases for some time, and then it starts decreasing, which marks the last visible point.

Having determined the last visible points, delete all the points lying between them, and insert the point p_i instead. This finishes the algorithm's description.

It might seem that if many points are deleted, lots of time is wasted by walking through all of them (instead, one could try to locate the last visible points by some kind of binary search). It turns out, however, that the above (seemingly simple-minded) solution is optimal, and there is no need to complicate the algorithm!

Claim. *The second part of the algorithm (after the initial sorting) runs in $O(n)$ time.*

Proof: The current convex hull (which is a convex polygon) can be represented by a circular doubly-linked list of its vertices, and we always have a pointer to the vertex inserted last. Hence the vertices deleted in the ith step can be found and deleted in time $O(1 + n_i)$, where n_i is the number of vertices deleted in the

ith step. But each point can only be deleted once, hence $\sum_{i=1}^{n} n_i < n$, and the total time is $O(n)$ as claimed. $\square$

Remark. It can be shown that computing the convex hull of n points requires at least $n \log n$ operations in the worst case (so we really need the initial sorting). On the other hand, if the resulting convex hull has h vertices, it can be computed in $O(n \log h)$ time (this is indicated in Exercise 9.8.5).

Exercises

1. *Suppose that there is an algorithm which computes the convex hull of any given n points in the plane in time at most T. Using it as a subroutine, construct an algorithm which sorts n given real numbers in time $T + O(n)$.
2. Design an algorithm which, given n points in the plane, decides whether their convex hull is a triangle, and if yes, finds it, all that in $O(n)$ time.

9.3 Randomized QUICKSORT

In the planar convex hull algorithm (section 9.2) we insert points from left to right. For an incremental convex hull algorithm in higher dimensions no such simple rule for a "good" insertion order is known. It turns out that a random insertion order works well on the average. We show an analysis of algorithms of this type on several simpler examples first. We begin with a variant of a well-known sorting algorithm, QUICKSORT.

Algorithm QUICKSORT, having received a sequence of elements $(x_1, x_2, \ldots, x_n)$ as input, proceeds as follows. The elements $x_2, x_3, \ldots, x_n$ are compared to x_1 and divided into two groups: those smaller than x_1 and those at least as large as x_1. In both groups, the order of elements remains the same as in the input sequence. Each group is then sorted by a recursive invocation of the same method. The recursion terminates with trivially small groups (at most one-element ones, say).

This algorithm may need about n^2 steps in the worst case (the worst thing we can do to the algorithm is to present it with an already sorted sequence). In practice, however, QUICKSORT is very popular, and it is considered one of the fastest sorting algorithms. It behaves very well "on average". This is partially expressed in the following theorem.

Theorem 9.3.1 *Let $x_1 < x_2 < \ldots < x_n$ be the input elements written in the sorted order. Let π be a permutation of the set $\{1, 2, \ldots, n\}$, and let $T(\pi)$ be the number of comparisons (of pairs of elements) made by QUICKSORT for the input sequence $(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$. Then the arithmetic average of $T(\pi)$ over all possible permutations π, denoted by $\mathbf{E}T$, is at most $2n \ln n$.*

The notation $\mathbf{E}T$ comes from probability theory. If the permutation π is chosen at random (in other words, if the input elements are randomly rearranged), $\mathbf{E}T$ is the *expectation* of the number of comparisons.

Reasons for the $O(n \log n)$ average behavior of the algorithm are not difficult to see. When the input elements are randomly ordered, we expect that the first element usually divides the remaining elements into two groups of roughly comparable size (it is unlikely that one group is much smaller than the other one). If this happens in most cases, the recursion in the algorithm will have roughly $\log n$ levels, and on each level of recursion we need $O(n)$ comparisons in total.

This may sound quite convincing, but certainly it is not a proof. Next, we give a rigorous proof, based on a rather different idea.

Proof of Theorem 9.3.1. Let $T_i = T_i(\pi)$ be the number of elements compared to the element $x_{\pi(i)}$ at the moment $x_{\pi(i)}$ is the dividing element. For instance, we always have $T_1 = n-1$, since $x_{\pi(1)}$ is the first element in the input sequence, and all the other elements are compared to it. If $\pi(2) < \pi(1)$, T_2 is $\pi(1) - 2$, and for $\pi(2) > \pi(1)$ we have $T_2 = n - \pi(1) - 1$. In general, T_i can be interpreted according to fig. 9.2. The small circles in this figure depict the elements $x_1, x_2, \ldots, x_n$ in the sorted order. The full circles are the elements with indices $\pi(1), \pi(2), \ldots, \pi(i-1)$, (the first $i-1$ elements in the input sequence). The element $x_{\pi(i)}$ is marked by a double circle, and the remaining elements have empty circles. It is not difficult to see that T_i is exactly the number of empty circles "seen" by the element $x_{\pi(i)}$, if the full circles are considered opaque.

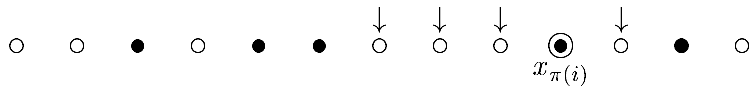


Figure 9.2: Illustration to the analysis of QUICKSORT

We will investigate $\mathbf{E}T_i$, the expected value of T_i (i.e. the arithmetic average of $T_i(\pi)$ over all permutations π). Imagine that we consider the indices i in the backward order $n, n-1, \ldots, 1$, and we look at the corresponding picture with full and empty circles. Initially, all the circles are full. The random permutation π is not defined yet; we create it by random choices as we go along. First we choose one full circle at random and make it empty; the index of this circle becomes $\pi(n)$. Then we randomly select another full circle and make it empty, which determines $\pi(n-1)$, etc. At the moment i full circles remain, we choose one of them at random, which corresponds to $x_{\pi(i)}$. The quantity T_i is the number of empty circles seen by $x_{\pi(i)}$ at this moment.

To estimate the expected value of T_i, we use a simple double counting. Each *empty* circle sees at most two full circles. Hence the total number of pairs (full circle,empty circle) whose members see each other is at most $2(n-i)$. On the average, one of the i full circles sees at most $\frac{2(n-i)}{i}$ empty circles, and hence $\mathbf{E}T_i \leq \frac{2(n-i)}{i}$. Since $T(\pi) = \sum_{i=1}^{n} T_i(\pi)$, we get

$$\mathbf{E}T = \sum_{i=1}^{n} \mathbf{E}T_i \leq \sum_{i=1}^{n} \frac{2(n-i)}{i} = 2n \sum_{i=1}^{n} \frac{1}{i} - 2n \leq 2n \ln n\,.$$

□

Remark. It can be shown that the probability of the running time of QUICKSORT deviating from the expectation by a significant amount is quite small (exponentially small), i.e. bad permutations are quite rare. We will not consider this here.

Exercises

1. Consider one round of a long jump competition. Suppose that there are n participants and they jump in a random order (each of them jumps once). For simplicity, assume that a given participant always jumps the same length, and that these lengths are distinct for distinct participants. Determine the expected number of times the current jump is longer than all the preceding ones. [3 points]

9.4 Point location problem

We are given a subdivision of the plane into n regions (such as a map of the world, or the Voronoi diagram of some n-point set, or an arrangement of about $\sqrt{n}$ lines, etc.). We want to construct a data structure which, given a given query point x, quickly determines the region containing x.

A solution to this problem can easily be obtained from a solution to another problem: Given n non-intersecting segments in the plane (some of them may touch), build a data structure which, given a query point x, quickly determines the segment lying vertically above x (if one exists). Yet another version of this problem, which is the one we will actually discuss, uses the notion of trapezoidal decomposition for the given set of segments.

Definition 9.4.1 *A* trapezoidal decomposition *for n given segments in the plane is constructed as follows: From each endpoint of each of the segments, erect a vertical segment upwards and downwards, and extend it in both directions until it hits another segment (if no segment is hit continue up or down to infinity).*

Let S be a set of n segments. If we can detect the trapezoid of the trapezoidal decomposition of S containing a query point x, then we can also tell which segment lies immediately above x, since for all points in one trapezoid is this segment the same and can be stored together with the trapezoid.

Construction. Let $s_1, s_2, \ldots, s_n$ be the given segments in a random order. We insert them one by one and maintain the trapezoidal decomposition; after the ith step we have the trapezoidal decomposition for $s_1, \ldots, s_i$. We start with the trapezoidal decomposition for the empty set of segments, which consists of a single trapezoid T_0, namely the whole plane.

After s_1 has been inserted, T_0 is replaced by trapezoids T_1, T_2, T_3, T_4 (see fig. 9.3(a)). By inserting s_2 we can get a situation as in fig. 9.3(b), say. In general, by inserting s_i, some trapezoids are destroyed and new ones are created in the area occupied by the destroyed ones. We do not throw away the destroyed trapezoids; we store them, and with each of them we store pointers to the new trapezoids which arise in the territory of the destroyed one. In the situation of fig. (b), T_0 has pointers to T_1, T_2, T_3, and T_4. The trapezoid T_1 has pointers to T_5, T_6, T_8, and the trapezoid T_4 points to T_7, T_8, T_9. We obtain an oriented

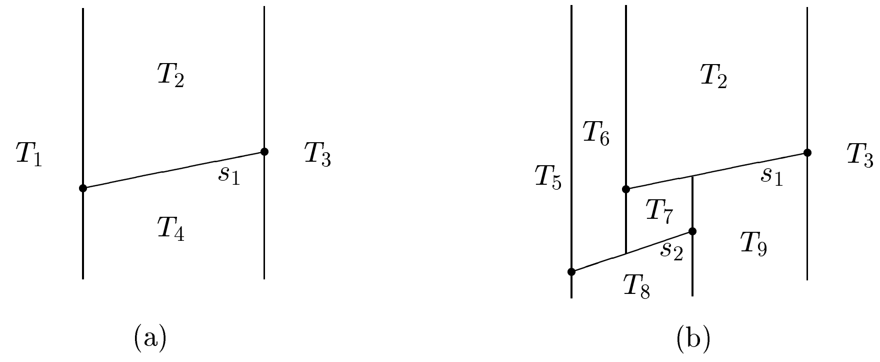


Figure 9.3: Trapezoidal decompositions after inserting s_1 (a), and after inserting s_2 (b).

acyclic graph. The source node is T_0, and the sink nodes are the currently existing trapezoids. After all segments have been inserted, this graph becomes our point location structure.

How to locate a point using this information? We "walk through history". We follow whose territory did the query point x belong to during the history of the incremental building of the trapezoidal decomposition. Initially, x belonged to T_0 (the whole plane). If x lived in T_j and T_j was destroyed, we look which trapezoids arose in the territory of T_j after T_j disappeared (recall that T_j has pointers to these), and we test which of them x lies in. Having reached a trapezoid which still exists in the final trapezoidal decomposition, we have located x.

Observation: By inserting a new segment, at most 4 new trapezoids can arise on the territory of a single existing trapezoid. (This is checked by analyzing several possible cases.) The one containing x can be found by at most three tests (a test means comparing the position of x against some line).

Hence the number of tests needed for locating x is proportional to the number of times the trapezoid containing x changed during the history. In the subsequent analysis, we consider x arbitrary but fixed, and we estimate the expected number of these changes; the expectation is with respect to a random insertion order of the segments.

We introduce the following random variables:

$$X_i = \begin{cases} 1 & \text{if the trapezoid containing } x \text{ changes by inserting the } i\text{th segment} \\ 0 & \text{otherwise.} \end{cases}$$

(The ith segment refers to the insertion ordering; thus the role of the ith segment can be played by any of the input segments, depending on the insertion order.) The total time for locating x is proportional to $X = \sum_{i=1}^{n} X_i$, and we have $\mathbf{E}X = \sum \mathbf{E}X_i$. The expectation $\mathbf{E}X_i$ is the probability that by inserting the ith segment, the trapezoid containing x is changed.

We use the "backwards analysis" trick. Time flows backwards, and we delete the segments one by one in random order. Now only $s_1, \ldots, s_i$ remain. What is the probability that the trapezoid T containing x disappears by deleting a randomly chosen segment? This T is defined by at most four segments: the ones defining the top side and the bottom side, and maybe the one(s) whose endpoint(s) determine the left and right vertical sides. (Here we need to assume that the segments are in general position!) If none of these defining segments is deleted the trapezoid remains. Hence $\mathbf{E}X_i \leq 4/i$, and $\mathbf{E}X \leq 4\sum_{i=1}^{n} 1/i = O(\log n)$.

This analysis shows that for any fixed point x is the average point location time $O(\log n)$. With some more work it can be shown that with high probability, the location time is $O(\log n)$ *for all points simultaneously.*

It remains to analyze the expected storage required for the data structure, and the expected time needed for its construction. Results: linear storage, $O(n \log n)$ construction time, which is asymptotically optimal.

The construction in more detail. We represent the trapezoidal decomposition suitably; every trapezoid points to its left and right neighboring trapezoids (under the general position assumption, there can be at most two left neighbors and two right neighbors) and to the segment defining its top side.

Insertion of a new segment:

1. Locate the endpoint of the newly inserted segment in the current trapezoidal decomposition, using the part of the data structure built until now. (The structure is not yet finished but it works already!)

2. Walk along the new segment through the trapezoidal decomposition. Detect the trapezoids intersected by it, and slice them into new trapezoids. Finally merge the neighboring new trapezoids which share a full vertical side (see fig. 9.4).

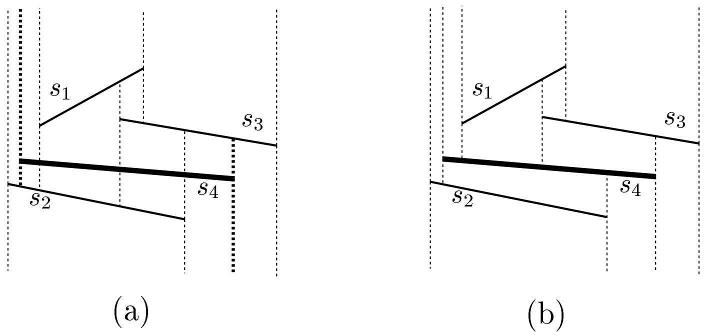


Figure 9.4: Subdividing old trapezoids intersected by s_i (a), and merging the adjacent new trapezoids (b).

The first step needs $O(\log i)$ time on the average, by the above analysis. This gives $O(n \log n)$ total time for all the n segments (as we will see, the remaining steps of the algorithm need expected $O(n)$ time only). The second step can be implemented in time proportional to the total number of deleted trapezoids (there are at most 4 new trapezoids per old trapezoid). Since each trapezoid must have been created before it can be deleted, it suffices to account for all created trapezoids.

Let Y_i be the number of trapezoids newly created in step i. Then the total storage requirement and the total time for everything except for the location of the endpoints is at most proportional to $Y = \sum_{i=1}^{n} Y_i$.

Using backwards analysis again, we see that $\mathbf{E}Y_i$ is upper-bounded by the average number of trapezoids which disappear by deleting a random segment among some i segments. A trapezoidal decomposition for i segments has at most $6i$ vertices, and we can view it as a drawing of a planar graph, hence the total number of trapezoids is $O(i)$.

Any given trapezoid can only disappear by deleting one of its at most 4 defining segments. The average number of deleted trapezoids per segments is thus $O(1)$ (double counting). Therefore $\mathbf{E}Y_i = O(1)$ and $\sum_{i=1}^{n} \mathbf{E}Y_i = O(n)$. $\square$

Exercises

1. Determine precisely the maximum possible number of trapezoids in the trapezoidal decomposition for a set of n segments (in general position).

9.5 Trapezoidal decomposition for a simple polygon

FILL IN

9.6 What if the segments intersect?

FILL IN

9.7 Convex hull in higher dimensions

In this section, we explain a randomized incremental algorithm for computing the convex hull of n points in $\mathbb{R}^d$. There are many possible variants of such an algorithm; we present one of them.

The dimension d is considered fixed (i.e. any function of d is treated as a constant).

It seems more intuitive to consider the problem in a dual setting. Here we are given n halfspaces in $\mathbb{R}^d$, and we want to compute their intersection; the resulting H-polytope is dual to the convex hull of the original points.

We assume that the input points are in general position. In the dual setting, this implies that the intersection of any subset of the halfspaces is a simple polytope; in particualr, each its vertex is incident to at most d edges. This assumption is essential; without it, the analysis as presented below doesn't work.

It will also be technically convenient to have a set Σ consisting of $d+1$ of the given n halfspaces whose intersection is a simplex (i.e. it is bounded). The halfspaces of Σ will be called the *special halfspaces*. They can be obtained as follows: In the original n-point set (i.e. in the primal setting), we find a subset S of $d+1$ affinely independent points. This can be done in $O(n)$ time by a simple linear algebra; if no $d+1$ affinely independent points exist, all points lie in a hyperplane and the dimension of the problem can be reduced. Then the origin o of the duality is chosen as an interior point of the convex hull of S, and Σ consists of the halfspaces containing o and bounded by the hyperplanes dual to the points of S.

So assume a set Σ of $d+1$ special halfspaces has been fixed, and let $h_1, h_2, \ldots, h_{n-d-1}$ be the remaining halfspaces in a random order. Let $\mathcal{P}_r$ denote the intersection $\left(\bigcap_{\sigma\in\Sigma} \sigma\right) \cap h_1 \cap h_2 \ldots \cap h_r$. The algorithm begins with the simplex $\mathcal{P}_0$, and then it inserts the remaining halfspaces one by one, maintaining the current intersection $\mathcal{P}_r$. The insertion of a new halfspace h_r "slices off" a portion of $\mathcal{P}_{r-1}$. We have to describe how is this done.

9.7.1 Inserting a new halfspace in dimension 3

First we describe and partially analyze the insertion step for the case $d = 3$, which is somewhat simpler and where we can rely on intuition. We have computed the polytope $\mathcal{P}_{r-1}$, and now we want to insert the halfspace h_r and compute $\mathcal{P}_r$.

First suppose that we know some vertex v of $\mathcal{P}_{r-1}$ which doesn't lie in h_r and will thus be sliced off. (If no such vertex exists, the halfspace h_r contains $\mathcal{P}_{r-1}$ completely and thus $\mathcal{P}_r = \mathcal{P}_{r-1}$.) Call such a v a *starting conflict vertex* for h_r; later on we will discuss how to find it.

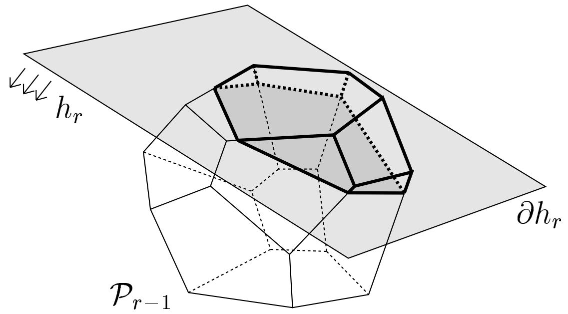


Figure 9.5: Slicing off the cap of h_r

Walking through the cap. Let us call the portion of $\mathcal{P}_{r-1}$ cut off by h_r the *cap*. We want to find all vertices of the cap (these will be deleted). The vertices and the edges of $\mathcal{P}_{r-1}$ lying in the cap define a graph G_r. We are going to visit all vertices of G_r by some graph-traversal algorithm (depth-first search, say), beginning from the starting conflict vertex v. First we need to know that all vertices of the cap can be reached from v:

Lemma 9.7.1 *The graph G_r defined by the vertices and edges lying in the cap is connected.*

Proof. We may assume that the boundary of h_r is not parallel to any edge of $\mathcal{P}_{r-1}$ (if it were parallel, we can tilt h_r a little without changing the set of vertices and edges lying in the complement of h_r).

Define a direction "upwards" as one perpendicular to the boundary of h_r and pointing out of h_r. From each vertex but the highest one, there is an edge going upwards; hence from each vertex outside h_r there is a path going to the topmost vertex. $\square$

Having found all vertices of the cap by graph traversal, it remains to change $\mathcal{P}_{r-1}$ into $\mathcal{P}_r$. Before we give details of this, let's be more specific about the representation of the current polytope.

Polytope representation. We store the vertices and the edges of the current polytope $\mathcal{P}_r$ (in a suitable geometric representation; in principle, it would suffice to represent a vertex, say, by the triple of its defining hyperplanes, but for the sake of efficiency, we will probably want to store the coordinates of the vertex as well). Recall that each vertex has 3 adjacent edges; we store pointers to these edges at each vertex, and each edge points to its end-vertices. Finally, with each edge we record the two facets adjacent to it. We need not represent the facets explicitly; we only assume they are numbered by small integers, say, and for two given edges, we can tell whether they share a facet or not. If needed, a geometric description of the facets can be easily reconstructed after the algorithm finishes.

Changing $\mathcal{P}_{r-1}$ into $\mathcal{P}_r$. We already know which vertices and edges should be deleted. The new vertices arise as intersections of edges with ∂h_r (the boundary of h_r). The relevant edges are actually discovered during the traversal of G_r: these are the edges whose one endpoint lies in the cap and the other doesn't. These edges are shortened appropriately.

A bit more subtle is the construction of the new edges arising as intersections of ∂h_r with facets of $\mathcal{P}_{r-1}$. From a new wertex in ∂h_r, we walk along edges in the cap around a facet (the pointers from edges to facets allow us to follow the facet boundary), until we reach another vertex in ∂h_r. In this way, we find pairs of vertices forming new edges in ∂h_r. All these edges are made adjacent to the new facet created within the plane ∂h_r. This finishes the construction of $\mathcal{P}_r$.

Running time analysis. Here we will analyze the expected running time of all steps of the algorithm, except for finding the starting conflict vertex for each inserted halfspace.

First, we claim that the time needed for insertion of h_r is at most proportional to the number of vertices of $\mathcal{P}_{r-1}$ cut off by h_r (the cap vertices). Since each vertex has degree 3 (by simplicity), the number of newly created vertices is at most $3\times$ the number of the cap vertices. The number of shortened edges and new edges is each bounded by the number of new vertices, and the total work is proportional to the number of deleted faces plus the number of new (or modified) faces.

By this claim, it suffices to bound the total number of vertices deleted in all steps. This is dominated by the total number of vertices created during the algorithm. Let X_r be the number of vertices created by inserting h_r. We employ the usual analysis with time running backwards.

We have a polytope $\mathcal{P}_r$ defined by some $r+4$ halfspaces, and hence with $O(r)$

vertices. Among these halfspaces, 4 are special and they cannot be removed. Among the remaining r halfspaces, we choose a random one, call it h_r, and remove it. Then $\mathbf{E}X_r$ is at most the expected number of vertices destroyed by this. For each vertex v of $\mathcal{P}_r$, define $X_{r,v}$ as 1 if v disappears by removing h_r, and as 0 otherwise. Should the vertex v disappear, the boundary of h_r has to be one of the 3 planes defining v; hence the probability of v disappearing is at most $3/r$. This yields $\mathbf{E}X_r = \sum_v \mathbf{E}X_{r,v} \leq O(r).3/r = O(1)$. The expectation of the total running time, not counting the time for finding the starting conflict vertices, is thus $O(n) + \sum_{r=1}^{n} \mathbf{E}X_r = O(n)$.

Exercises

1. Give an example of n halfspaces in $\mathbb{R}^3$ and an order of their insertion such that the total number of vertices created by the incremental algorithm described in this section is of the order n^2.

9.7.2 Finding the starting conflict vertices

We say that a vertex v of some $\mathcal{P}_i$ is *in conflict with* a halfspace h_j if $v \notin h_j$. For each newly inserted halfspace h_r, we need to find a vertex v of $\mathcal{P}_{r-1}$ in conflict with h_r — this is the part of the algorithm we have left out so far. Similarly to the point location algorithm from section 9.4, we will search the history. We imagine the algorithm runs once again from the beginning until step $r-1$, and we maintain a vertex of the current polytope which is in conflict with h_r. Initially, $\mathcal{P}_0$ is a tetrahedron, so we can find some its vertex v_0 in conflict with h_r in constant time.

When inserting some h_i, $1 \leq i < r$, we have the current conflict vertex v_{i-1} of $\mathcal{P}_{i-1}$. If v_{i-1} remains a vertex of $\mathcal{P}_i$, we can use it in the next step too, i.e. we set $v_i = v_{i-1}$. What to do if v_{i-1} is cut off by h_i? We look for the new conflict vertex v_i among the vertices newly created by inserting h_i.

We search the cap cut off by h_i by a graph-traversal algorithm. During this search, we never enter vertices lying in h_r (this is important for the running time analysis). As soon as we find a vertex on the boundary of h_i lying outside h_r, we declare it to be the new v_i.

Let us argue that this search always succeeds, unless all of $\mathcal{P}_i$ lies in h_r. (There is some subtlety in this, since the graph formed by the vertices and edges of $\mathcal{P}_{i-1}$ lying both outside h_r and outside h_i need not be connected!) Suppose that v' is a vertex of $\mathcal{P}_i$ in conflict with h_r (see fig. 9.6). This v' is either a vertex of $\mathcal{P}_{i-1}$ or it lies on an edge of $\mathcal{P}_{i-1}$. Since the graph formed by vertices and edges of $\mathcal{P}_{i-1}$ cut off by h_r is connected by Lemma 9.7.1, there is a path π of edges of $\mathcal{P}_{i-1}$ connecting v_{i-1} to v' whose all vertices and edges lie outside h_r. This path π starts outside h_i and ends inside it or on its boundary. If we follow π from v_{i-1}, the first vertex of $\mathcal{P}_i$ in which we encounter the boundary of h_i is the desired v_i.

In an actual implementation, we do not repeat the whole algorithm, of course; we only store relevant information from its execution and use it for finding the starting conflict vertices. For each inserted halfspace h_i, we store

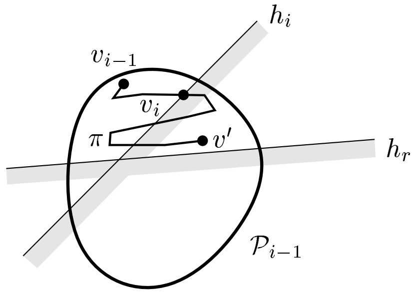


Figure 9.6: Existence of the new conflict vertex v_i.

the cap cut off by it (i.e. the graph G_i), and with each vertex v of this cap, we store a pointer to h_i — the halfspace which "killed" v.

For the initial conflict vertex v_0 of $\mathcal{P}_0$ we consider the halfspace h_{i_1} which killed it. We search the cap cut off by this h_{i_1} as described above (avoiding vertices in h_r), and obtain a new conflict vertex v_{i_1}. Again we look which halfspace killed it, etc., until we find a vertex which is still alive — this is the desired starting conflict vertex for h_r.

Running time analysis. Let $Y_{r,i}$ denote the number of vertices in the cap cut off by h_i which are in conflict with h_r. The total time needed for searching the starting conflict vertex for h_r is at most proportional to $1 + \sum_{i=1}^{r-1} Y_{r,i}$. Let us estimate the expectation of $Y_{r,i}$. To understand the analysis, regard the symbols $h_1, h_2, \ldots$ as placeholders which are assigned specific halfspaces by the random choice of the insertion order. Imagine that we first choose the *set* of $i+1$ halfspaces (call it H_0) which are going to fill the placeholders $h_1, h_2, \ldots, h_i$ and h_r, and then we choose a random assignment of the halfspaces from H_0 to these placeholders. Suppose that some H_0 has been fixed; we estimate the expectation of $Y_{r,i}$ with respect to the random assignment of halfspaces of this H_0 to $h_1, \ldots, h_i$ and h_r.

Let B denote the set of the boundary planes of the halfspaces of $H_0 \cup \Sigma$, where Σ is the set of the 4 special halfspaces defining the initial simplex $\mathcal{P}_0$. Clearly, any vertex contributing to $Y_{r,i}$ is a vertex of the arrangement of B, and moreover it is in conflict with precisely two halfspaces in $H_0 \cup \Sigma$, namely with h_r and h_i. In the terminology of section 6.2, this is a vertex of level 2 in the arrangement of B (where the origin o is chosen as an interior point of the intersection of all halfspaces of $H_0 \cup \Sigma$). Let V_2 denote the set of all vertices of level 2 in the arrangement of B. By Theorem 6.2.1, we get $|V_2| = O(|H_0 \cup \Sigma|) = O(i+1)$.

Let us consider a fixed vertex $v \in V_2$, and ask what is its expected contribution to $Y_{r,i}$. In other words, if we choose two halfspaces of H_0 at random and call them h_r and h_i, what is the probability that v is in conflict precisely with h_r and h_i? Since v is in conflict with exactly two halfspaces of H_0, h_r and h_i must be exactly these two halfspaces, in some order. Hence there are 2 favorable choices of h_r and h_i among the $(i+1)i$ possible ones, and so the expected contribution of v to $Y_{r,i}$ is at most $\frac{2}{(i+1)i}$. Then $\mathbf{E}Y_{r,i} \leq |V_2| \frac{2}{(i+1)i} = O(1/i)$.

Hence the total expected time needed for searching all the starting conflict vertices in the algorithm is at most proportional to

$$\sum_{r=1}^{n-4}\left(1+\sum_{i=1}^{r-1}\frac{1}{i}\right)=\sum_{r=1}^{n-4}(1+O(\log r))=O(n\log n)\,.$$

We should also analyze the expected amount of storage required by the algorithm. But this is at most proportional to the total number of vertices created during the algorithm, and this quantity has already been analyzed (in section 9.7.1) — its expectation is $O(n)$. This concludes the analysis of the 3-dimensional convex hull algorithm. We have shown:

Theorem 9.7.2 *The above described randomized incremental algorithm computes the convex hull of n points in general position in $\mathbb{R}^3$ in $O(n\log n)$ expected time, using $O(n)$ expected storage.*

Remark. It is also possible to incorporate the proof of the special case of the theorem about level $\leq k$ (Theorem 6.2.1) into the above analysis; we actually suffice with the $k=1$ case. The whole proof is not much more complicated than the one shown above, but we consider it conceptually cleaner to separate the level $\leq k$ part.

Exercises

1. Describe and analyze a randomized incremental algorithm for computing the Voronoi diagram of n points in general position in the plane. Use the algorithm for intersection of halfspaces in $\mathbb{R}^3$ as a basis, but try to work directly in the plane. [2–4 points]
2. (a) Design an algorithm with $O(n\log n)$ expected running time, which finds, for given n points in general position in the plane, the pair of points with the smallest possible distance. [2–4 points]

   (b) A similar problem, but we are looking for the pair with maximum distance, i.e. the diameter of the set. [2–4 points]

   (c) Design an algorithm with $O(n\log n)$ expected time, which finds, for each of n given points, the farthest point. [3–4 points]

9.7.3 Higher dimensions

The three-dimensional convex hull algorithm presented above, as well as its analysis, can be generalized into any fixed dimension d without any essential new ideas. In fact, most of the description remains the same.

The current polytope $\mathcal{P}_r$ is again represented by storing its vertices, edges, and 2-faces, exactly as was described for the 3-dimensional case. We need not store faces of higher dimensions; if desired, they can be reconstructed in time proportional to the total number of vertices of the polytope (see Exercise 1 — but beware that this only works for simple polytopes!). Having detected all vertices and edges of the cap, and the new vertices created inside ∂h_r, we can find the new edges in ∂h_r by traversing around the boundaries of 2-faces inside the cap, as in the 3-dimensional case. For accounting the running time of this operation, we need the fact that in a simple polytope, any edge is adjacent to a bounded number of 2-faces only.

An additional algorithmic step required is the reconstruction of the new 2-faces created within ∂h_r, and this is a special case of Exercise 1 (we need to reconstruct the 2-faces of a $(d-1)$-dimensional polytope in ∂h_r from its vertices and edges). The procedure for finding the starting conflict vertices, and also the argument for its correctness, remain literally the same.

Thanks to the general position assumption, the number of created edges and 2-faces is at most proportional to the number of created vertices. Hence in all running time estimates, it suffices to count vertices (as we did in the 3-dimensional case). The 3-dimensional analysis was based on the fact that the intersection of n halfspaces in $\mathbb{R}^3$ has $O(n)$ complexity. If we use an asymptotic version of the Upper bound theorem 4.5.1 instead, giving $O(n^{\lfloor d/2 \rfloor})$ vertices for the intersection of n halfspaces in $\mathbb{R}^d$, we arrive at $O(n^{\lfloor d/2 \rfloor})$ expected complexity of the algorithm (we leave the details to Exercise 2). We thus get

Theorem 9.7.3 *The indicated randomized incremental algorithm computes the convex hull of given n points in general position in $\mathbb{R}^d$, where $d \geq 4$ is arbitrary but fixed, in expected time and storage $O(n^{\lfloor d/2 \rfloor})$.*

Concluding remarks. We have given a convex hull algorithm with expected $O(n^{\lfloor d/2 \rfloor})$ running time for n points. This is asymptotically optimal in the worst case (for the cyclic polytope, say). But in practice one mostly encounters polytopes with n vertices and with much fewer than $n^{\lfloor d/2 \rfloor}$ faces. For point sets in general position, the algorithm seems to behave mostly "reasonably", meaning that the running time will not be of much larger order of magnitude than the number of faces of the convex hull. However, constructions are known of n-point sets where the convex hull of all points has few faces, but the convex hull of any $n-1$ points has very many faces. On such sets, the algorithm behaves badly (since it must compute the "intermediate" convex hulls). Highly degenerate point sets (which occur is some practical applications) present another problem. The above algorithm is only applicable to sets in general position, and hence we have to use an infinitesimal perturbation to the original point set. Such a perturbation may create a large number of faces not present in the original convex hull, which will have to be computed by the algorithm. A number of algorithms for the convex hull problem are known, based on different ideas, but currently none of them can handle all inputs (especially highly degenerate ones) satisfactorily.

Exercises

1. The k-skeleton of a convex polytope is the set of all its faces of dimension at most k, together with their incidence relations, i.e. for each j-face we have a list of the $(j-1)$-faces contained in it, and (for $j < k$) a list of the $(j+1)$-faces containing it. Let $\mathcal{P}$ be a simple polytope in $\mathbb{R}^d$ (d fixed), and suppose that its k-skeleton is given, for some $k \geq 1$. Give an algorithm which computes the $(k+1)$-skeleton of $\mathcal{P}$, in time proportional to the total number of vertices of $\mathcal{P}$. (Try the case $k = 1$ first.)

2. Go through the description and analysis of the 3-dimensional algorithm, and discuss in detail all modifications needed for the d-dimensional case.

9.8 Linear programming in small dimension

The linear programming problem has numerous variants in formulation. Here we consider the following formulation:

Problem 9.8.1 *Given a real matrix A with n rows and d columns, a vector $b \in \mathbb{R}^n$, and a vector $c \in \mathbb{R}^d$, find a point $x \in \mathbb{R}^d$ satisfying the system of n linear inequalities*

$$Ax \leq b\,, \tag{9.1}$$

and such that the value of the linear function $\langle c, x\rangle$ is smallest possible.

This problem has numerous applications in many areas. Geometrically, we can interpret it as follows: We are given n halfspaces in $\mathbb{R}^d$ (each of them described by one of the inequalities in the system (9.1)). Their intersection is an H-polytope. In this H-polytope, we are looking for a point lying "as far as possible" in a given direction. More exactly: if we translate a hyperplane orthogonal to the vector c in the direction $(-c)$, at which point will the hyperplane touch the polytope the last time?

This problem is most often solved by the simplex method (or dual simplex method) in practice. This is practically quite satisfactory in most cases, but it has a serious drawback from a theoretical point of view: These methods may require exponentially many steps in the worst case (for certain artificially constructed inputs). For this reason, other algorithms have been sought, which would guarantee a polynomial-time complexity in the worst case. And polynomial algorithms were indeed discovered (the first one was the so-called *ellipsoid method*), but polynomial only in a certain weak sense. Their complexity depends on the number of bits in the representation of the numbers in the matrix A; the higher input precision, the more steps may be needed. In particular, the number of arithmetic operations executed by these algorithms cannot be bounded in terms of n and d alone. It is a big open problem to find an algorithm which would solve the linear programing problem, for any input, using a number of arithmetic operations bounded by a polynomial function of n and d (a so-called *strongly polynomial* algorithm).

In computational geometry, the linear programming problem has been viewed as a geometric problem in dimension d, where d was considered small compared to n. As we will see, algorithms exist solving the linear programming problem in any fixed dimension in time $O(n)$. The multiplicative constant usually grows very fast with the dimension, however. For the best known algorithm in this respect, the constant grows with the dimension roughly as $\exp(\sqrt{d})$ (i.e. much more slowly than the exponential function but faster than any polynomial). This algorithm is a randomized one; it was developed on the basis of a simpler algorithm described below.

Seidel's algorithm. This is a randomized incremental algorithm. It is built by induction on dimension.

For $d = 1$: Linear programming on the real line can be reduced to finding the interval of admissible solutions. This can easily be done in $O(n)$ time. Then we select one of the endpoints (depending on whether the direction c is positive or negative).
For $d > 1$: Suppose than an $O(n)$ algorithm in dimension $d-1$ exists. We are going to give a linear-time algorithm in dimension d. First we make several simplifying assumptions (later on we show how to avoid them).

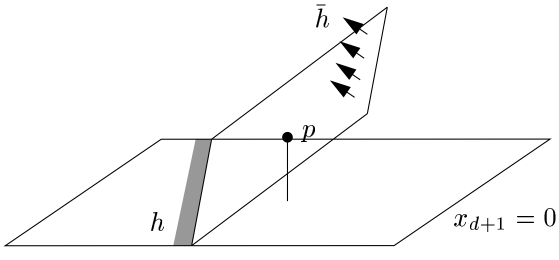
1. The intersection of our halfspaces is nonempty, i.e. an admissible solution exists.

2. The objective function $\langle c, x\rangle$ is bounded from below on the set of admissible solutions, hence an optimal solution exists.

3. The vector c is in general position with respect to the halfspaces, which means that it is not parallel to any face of dimension ≥ 1 in the arrangement of the hyperplanes bounding the input halfspaces. This implies that the optimal solution is unique (and it is a vertex of the polytope of admissible solutions).

If the problem doesn't satisfy some of these assumptions, here is how to proceed:

ad 1. If there is no admissible solution we can finish, so we only need an algorithm for deciding whether an admissible solution exists or not. The following simple construction shows that this can be checked by finding the optimum of a linear program in dimension $d+1$ which always has an admissible solution. Namely, identify the space $\mathbb{R}^d$ with the $x_{d+1} = 0$ plane in $\mathbb{R}^{d+1}$, and for each constraint in the original problem, regarded as a halfspace $h \subset \mathbb{R}^d$, consider the halfspace $\bar{h} \subset \mathbb{R}^{d+1}$ containing h and such that its boundary contains the boundary of h and the point $p = (0, \ldots, 0, 1)$.



This yields the constraints of a linear programming problem in $\mathbb{R}^{d+1}$, and we minimize x_{d+1}. This new linear program always has an admissible solution (namely p) and it is easy to see that the optimum is ≤ 0 if and only if the original linear program had an admissible solution.

ad 2. We add several "artificial" halfspaces in such a way that the objective function becomes bounded from below. For instance, we can enclose the solution space in a gigantic cube, i.e. we add the halfspaces "$x_i \geq -K$"

and "$x_i \leq K$", $i = 1, 2, \ldots, d$, where K is a very large number (we need not specify its value, since we can use it as a formal quantity in calculations).

ad 3. If more than one optimal solution exists, we select one with the lexicographically smallest coordinate vector. This guarantees that the solution is always unique. Geometrically, it corresponds to a small perturbation of the vector c.

These are admittedly not the most efficient ways to handle the unpleasant situations, but they simplify the algorithm's description.

The algorithm in dimension d starts by inserting the "artificial" bounding halfspaces (introduced in 2.). Then the input halfspaces are inserted one by one in a random order, and the current optimal solution is maintained. Having inserted all the halfspaces, we obtain the optimal solution.

Let x_{i-1} be the optimal solution for the first $i-1$ inserted halfspaces. We are going to insert a halfspace h_i. We distinguish two cases.

(a) If x_{i-1} lies in h_i, then we have $x_i = x_{i-1}$, since the previous optimal solution remains admissible, and there is no better admissible solution (the set of admissible solutions keeps shrinking).

(b) If x_{i-1} doesn't lie in h_i, we have to find a new optimal solution, since x_{i-1} is not admissible anymore. We claim that the new optimal solution has to lie in the boundary ∂h_i of the halfspace h_i.

> *Proof:* Suppose that x_i lies somewhere inside h_i. Then the segment connecting x_{i-1} and x_i all lies in the polytope of admissible solutions for the first $i-1$ halfspaces. The boundary of h_i intersects this segment at some point y between x_{i-1} and x_i. We have $\langle c, x_i\rangle \geq \langle c, x_{i-1}\rangle$, for otherwise x_{i-1} wouldn't be optimal in the previous step. But the objective function is monotone along the segment $x_{i-1}x_i$, whence $\langle c, x_i\rangle \geq \langle c, y\rangle$. So y is at least as good as x_i but it lies on the boundary of h_i.

The new optimal solution x_i is thus a point from the intersection of the boundary ∂h_i of h_i with the admissible polytope for the first $i-1$ halfspaces, and it can be found using the linear programming algorithm in dimension $d-1$: We project the vector c into the hyperplane of ∂h_i, we intersect the first $i-1$ halfspaces with ∂h_i, and we solve a linear programming problem within ∂h_i. Its optimal solution is the desired x_i.

This finishes the description of the algorithm. It remains to prove that the algorithm runs in linear expected time. For simplicity, we will only count the number of tests "$x_{i-1} \in h_i$" executed in the algorithm (including the recursive calls). One can check that the total running time is at most proportional to the number of these tests. Let $T_d(n)$ denote the maximum of the expected number of these tests executed in the algorithm in dimension d for n input halfspaces (the maximum is with respect to the set of input halfspaces, i.e. we take the worst possible input, and the expectation is with respect to the random ordering of the halfspaces).

By the inductive hypothesis, we have $T_{d-1}(i) \leq C_{d-1}i$ for some constant C_{d-1}. In the algorithm, we insert a new halfspace n times. If the optimal solution doesn't change by an insertion, we only execute a single test in that step. If a new optimal solution is computed in the ith step, we use the algorithm in dimension $d-1$ with $i-1$ halfspaces, which needs at most $C_{d-1}i$ tests on the average. This leads to the recursion

$$T_d(n) \leq \sum_{i=1}^{n}(1 + p_i C_{d-1} i)\,,$$

where p_i is the probability that the optimal solution changes in the ith step. The subsequent bound for p_i is based on the fact that the optimal solution is always defined by at most d halfspaces. More precisely: If the optimum value of the objective function $\langle c, x\rangle$ for the halfspaces $h_1, h_2, \ldots, h_i$ equals some number m_i, then there exist at most d halfspaces among $h_1, \ldots, h_i$, for which the optimum value is also m_i — see Exercise 1.

We use backwards analysis. If we imagine the algorithm running backwards, then p_i is the probability that the optimal solution changes by deleting one (randomly chosen) halfspace among i halfspaces. The optimum certainly will not change if none of the d halfspaces defining it is deleted (since these d halfspaces alone are sufficient to enforce that the optimum cannot be better[1]). Therefore $p_i \leq \frac{d}{i}$, and so we get

$$T_d(n) \leq \sum_{i=1}^{n}(1 + dC_{d-1}) = (1 + dC_{d-1})n\,.$$

The expected running time is thus indeed linear in n, and it is easy to check by induction that $d! \leq C_d \leq (d+1)!$ (if we start with $C_1 = 1$). Hence the dependence of the bound on the dimension is exponential. Such a behavior of the algorithm has also been confirmed by implementation experience.

Remarks. Seidel's algorithm is important mainly because one can prove good bounds on its running time (for any input) in a simple way. Also, it is a starting point for algorithms with a better theoretical running time bound. From a practical point of view, the dual simplex method is usually much faster than Seidel's algorithm.

Exercises

1. Let H be a finite set of closed halfspaces in $\mathbb{R}^d$ such that $\bigcap H \neq \emptyset$ and that the function $x \mapsto \langle c, x\rangle$ ($c \in \mathbb{R}^d$ is some fixed vector) attains a minimum value m on the intersection $\bigcap H$. Prove that there exists a subset $H_0 \subseteq H$ of at most d halfspaces such that the mentioned function also has minimum m on the intersection $\bigcap H_0$. (Use Helly's theorem.) [3 points]

2. Let P and Q be two convex polygons in the plane. Show that the problem of finding the maximum number $\lambda > 0$ such that some translate of λQ is contained in P can be formulated as a linear programming problem with a small (fixed) number of variables. [2 points]

[1] Here we use the assumption that the optimum is unique for any subset of our halfspaces!

3. Let P be a convex polytope in $\mathbb{R}^3$. Formulate the problem of finding the largest ball inscribed in P as a linear programming problem with a small (fixed) number of variables. [2 points]

4. An *annulus* is the difference of two concentric circular disks in the plane, i.e. a set of the form $\{(x,y) \in \mathbb{R}^2; r_1^2 \le x^2 + y^2 \le r_2^2\}$ for some $r_1, r_2 > 0$. Let P be an n-point set in the plane. Show that the problem of finding an annulus of the smallest possible area containing P can be formulated as a linear programming problem with a small (fixed) number of variables. [3 points]

5. (a) Let A, B be point sets in the plane separated by a vertical line v. We want to find a line lying above all points of $A \cup B$ and intersecting v as low as possible. Formulate this as a linear programming problem. [2 points]

   (b) Using (a), formulate a divide-and-conquer type algorithm for computing the upper part of the convex hull of an n-point set in the plane. If the resulting upper hull consists of h segments, the algorithm should have $O(n \log(h+2))$ expected running time. [5 points]

9.9 The Sharir-Welzl algorithm and LP-type problems

Chapter 10

Algorithms for line arrangements

10.1 Computing line-point incidences

Problem 10.1.1 *We are given n lines and m points in the plane. We want to find all incidences between the points and the lines (or, in another version of the problem, to decide whether at least one incidence exists at all).*

A trivial algorithm would test each point against each line, in time $O(mn)$. We derive an asymptotically much better algorithm using methods of section 7.6.

Algorithm for $n << m$. First, we assume that the number of points is much larger than the number of lines. We will proceed in two steps. We preprocess the lines, and then we test the points one by one.

In the first step, we compute the arrangement of the given lines. Then we build a point location structure for this arrangement (regarded as a subdivision of the plane into convex cells). The arrangement can be constructed in time $O(n^2)$ (see section 9.1). Then it can be preprocessed for point location by the method of section 9.4. A direct application that method yields $O(n^2 \log n)$ expected preprocessing time, but for the special case of a line arrangement this can be improved to $O(n^2)$ — see Exercise 1. For a given point, the a data structure allows us to find all the lines incident to the point in $O(\log n)$ time. Hence for m points, the total complexity of the algorithm is $O(n^2 + m \log n)$.

Algorithm for $n >> m$. By geometric duality (transforming lines into points and points into lines and preserving incidences), the above algorithm can also be used to detect all incidences between n lines and m points in expected time $O(m^2 + n \log m)$, which is efficient if the number of lines is much larger than the number of points.

Algorithm for $n = m$. What to do if the number of lines is about the same as the number of points, say? We proceed as in the proof of the Szemerédi-Trotter theorem 7.1.1 via cuttings (section 7.6). We subdivide the plane into triangles in a suitable way, and we solve the problem in each triangle separately.

We choose a parameter r (adjusted so that the resulting asymptotic complexity is as small as possible). We construct a $(1/r)$-cutting for the given set L of n lines.

How to construct such a $(1/r)$-cutting? Looking into the proof of Lemma 7.7.1, we see that if we choose s lines of L at random, with $s = 6r \ln n$, and triangulate their arrangement, we get a $(1/r)$-cutting with probability close to 1. If, by chance, we did not get a $(1/r)$-cutting, we can repeat the attempt — the expected number of such repetitions is only sightly bigger than 1. How can we tell whether we have succeeded, i.e. whether we have a $(1/r)$-cutting? For every triangle Δ_i of the cutting we find the set L_i of lines of L intersecting its interior (we will need these sets anyway), and we check whether $|L_i| \leq n/r$. It is advantageous to process one line of L by one, and "track" each line through the triangles of the cutting (similarly as he have inserted a new line into an arrangement in section 9.1). The total time for this will be proportional to the number of incidences line/triangle's interior, which for a $(1/r)$-cutting is bounded by $O(nt/r)$, where t is the number of triangles in the $(1/r)$-cutting. With our method, we have $t = O(s^2) = O(r^2 \log^2 n)$. As soon as the number n/r of lines for some triangle is exceeded, we can stop and re-start the algorithm, since the current cutting is not a $(1/r)$-cutting.

In this way, we find a $(1/r)$-cutting consisting of $O(r^2 \log^2 n)$ triangles, and for each triangle Δ_i we also get its set L_i of lines, all that in total expected time $O(nr \log^2 n)$. (Better, and more complicated, methods are known for finding a $(1/r)$-cutting consisting of $O(r^2)$ triangles in time $O(nr)$.)

Now we preprocess the constructed $(1/r)$-cutting for point location, and we divide the given m points among the triangles. Points which fall into vertices or sides of the triangles can be processed separately — we leave the details to Exercise 2.

In this way we get, for each triangle Δ_i, a set P_i of points and a set L_i of lines, where $|L_i| \leq n/r$ and $\sum_{i=1}^{t} |P_i| \leq m$. If some triangle contains more than the average number of points, i.e. $|P_i| > m/t$, we further subdivide P_i into several subsets of size at most m/t; the number of subsets arising from P_i is at most $|P_i|(t/m) + 1$. Altogether, we get at most $2t$ point sets of size at most m/t each, and for each point set we have the corresponding set of at most n/r lines. Each such subproblem, with at most n/r lines and at most m/t points, is solved using the above discussed algorithm for the case $m << n$, in time $O\left((m/t)^2 + (n/r)\log m\right)$. For all the at most $2t$ subproblems plus the initial cutting construction, we thus need time

$$O\left(\frac{m^2}{t} + \frac{tn}{r}\log m + nr\log^2 n\right) = O\left(\frac{m^2}{(r\log n)^2} + nr\log^2 n \log m\right).$$

Now is the time to fix the parameter r; we choose it so that both terms in the last formula have the same order of magnitude (for simplicity, we only consider the case $m = n$; the approach for other values of m, n is the same). We get $r = n^{1/3}\log^{-5/3} n$, and the total expected complexity of the algorithm is $O(n^{4/3}\log^{4/3} n)$.

It is believed that no algorithm can solve this problem faster in the worst case than in time $\Omega(n^{4/3})$, but no nontrivial lower bound is known. Proving one is a big challenge in computational geometry.

The algorithm for line/point incidences is a prototype of numerous computational geometry algorithms which use $(1/r)$-cuttings and their analogues for asymptotically efficient divide-and-conquer. From a practical point of view, these algorithms currently appear less important than, say, the randomized incremental approach, mainly because the application of cuttings seems to give very large constants of proportionality in the resulting algorithms.

Exercises

1. Show that the arrangement of n lines can be preprocessed for efficient point location in expected time $O(n^2)$ (in such a way that the expected query time for each point is $O(\log n)$). Proceed similarly as in section 9.4, but instead of inserting the segments (edges) in a random order, insert whole lines at once in random order, and maintain a trapezoidal decomposition of the arrangement. [4 points]
2. Describe in detail, how can one process points falling on triangle vertices or edges in the algorithm for line/point incidences described in this section (in such a way that no incidence is possibly neglected and the asymptotic complexity of the algorithm doesn't get worse). [3 points]
3. Let S be a set of n segments in the plane (in general position, say). The goal of this exercise is to design an algorithm for counting the number of intersections of the segments of S in time roughly $n^{4/3}$.

   (a) Design an $O(n \log n)$ algorithm for the following problem: Given a triangle Δ and a set L of n segments in it, each having both endpoints on the boundary of Δ, count the number of intersections of the segments. (It may be helpful to read the beginning of section 10.2 first). [4 points]

   (b) Given a set L in Δ as in (a), plus a set S of m segments contained in Δ (but not necessarily with endpoints on the boundary), design an algorithm counting the number of intersections of the segments of L with the segments of S in expected time $O(m^2 + n \log m)$. [4 points]

   (c) Complete the following algorithm sketch (and its analysis). Given S as in the beginning of this exercise, extend the segments to lines and compute a $(1/r)$-cutting for these lines, with an r to be chosen later. For each triangle Δ_i of this $(1/r)$-cutting, find the segments of S intersecting it, and classify them into "long for Δ_i" and "short for Δ_i": the short ones have a vertex in Δ while the long ones cut through completely. For each Δ_i, use (a) for counting long-long intersections, (b) for long-short intersections, and do short-short intersection by brute force.

10.2 Slope selection

Here we develop several algorithms for the following problem:

Problem 10.2.1 *Let P be a set of n points in the plane in general position. These points determine $\binom{n}{2}$ lines; assume these all have distinct slopes. The goal is to find the line with the kth smallest slope (where k is a given integer, $1 \leq k \leq \binom{n}{2}$).*

Application (in statistics): We have several measurements of two physical quantities which should be linearly related. We want to pass a line through the measured points. If some of the measurements are off by a large amount (we omitted a digit in typing them into a computer, say), "classical" methods (such as the method of least squares) give distorted results. We want a method which is "robust", which means that it is not badly influenced by few atypical points. One such method is to take a line whose slope is the median among all slopes determined by pairs of the measured points; this method turns out to be fairly robust in the just discussed sense.

We will discuss the problem of selecting the kth smallest slope in a dual plane. Here is a dual re-formulation of the original problem:

Problem 10.2.2 *Consider n lines in the plane in general position (none parallel to the x-axis). These lines determine $\binom{n}{2}$ intersections. Assume that their x-coordinates are all distinct. The goal is to compute the intersection with the kth smallest x-coordinate.*

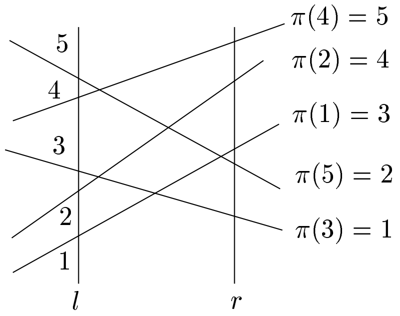
This problem can easily be solved in $O(n^2)$ time: Compute all the intersections, and use a linear-time selection algorithm. We will be able to reduce this to $O(n \log n)$ expected time. As a subroutine, we will need an algorithm for the following problem:

Problem 10.2.3 *Given the n lines as above, count the number of intersections in a given vertical strip, i.e. the number of intersections with x-coordinate lying in an interval $(\ell, r]$ for given real numbers ℓ, r.*

Remarks. An algorithm for Problem 10.2.3 allows us to check the correctness of an answer for problem 10.2.2. To simplify notation, we let the symbol ℓ stand also for the vertical line $\{x = \ell\}$, and similarly for r.

Lemma 10.2.4 *Let the lines be numbered by increasing y-coordinate of their intersections with the vertical line ℓ. Then the ordering of the intersections with the vertical line r determines a permutation π, and the number of inversions of this permutation equals the number of intersections in the vertical strip $(l, r]$. (An inversion of a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a pair (i, j), where $i < j$ and $\pi(i) > \pi(j)$, i.e. a pair for which π reverses ordering. Note that we regard a permutation as a bijective mapping.)*

Proof. Two lines intersect in the strip $(l, r]$ if and only if their order on ℓ is different from the order on r. This corresponds to the definition of an inversion of a permutation. See a figure:



□

Remark. If the vertical line r passes through an intersection, we define the permutation using the ordering on the line $r + \varepsilon$ for a very small $\varepsilon > 0$ (and similarly for the ordering on ℓ).

We describe an algorithm for counting the inversions of a given permutation in time $O(n \log n)$. (A somewhat faster algorithm is possible, but since we need $n \log n$ operations for sorting the intersections with the vertical lines ℓ and r to get the permutation π, an $O(n \log n)$ inversion counting is good enough.)

Algorithm for inversion counting. Let π be a permutation of the set $\{1, 2, \ldots, n\}$. We can interpret it also as an n-term sequence $(\pi(1), \pi(2), \ldots, \pi(n))$. We define *small numbers* as the numbers $1, 2, \ldots, \lfloor n/2 \rfloor$ and *large numbers* as the numbers $\lfloor n/2 \rfloor + 1, \ldots, n$. An inversion of the permutation π appears in the sequence as a pair of terms in which the first term is larger than the second term. We divide the inversions into three classes: with two small numbers, with two large numbers, and with a large number and a small number. The inversions of the first two types are counted by a recursive application of the same algorithm. For instance, for the sequence $(6, 4, 1, 3, 7, 8, 2, 5)$ we would call the algorithm recursively on the subsequence $(4, 1, 3, 2)$ of small numbers and the subsequence $(6, 7, 8, 5)$ of large numbers. The inversions of the last class, with one small number and one large number, can be computed directly in $O(n)$ time. The number of inversions of this class in which a given small number i participates is the number of large numbers preceding i in the sequence. Imagine that each large number is replaced by the symbol $\bigcirc$ and each small number by $\circ$ (for our example, we get $(\bigcirc, \circ, \circ, \circ, \bigcirc, \bigcirc, \circ, \bigcirc)$). We traverse the sequence from left to right, we count how many $\bigcirc$ we passed, and for each $\circ$ we add the current value of the $\bigcirc$ counter to some global counter of inversions.

It is easy to see that the total time of this algorithm, including the recursion, is $O(n \log n)$.

Remarks. By a modification of this algorithm, we can also pick a random inversion of the given permutation in $O(n \log n)$ time (all inversions have the same probability). For a given permutation, the algorithm defines a certain fixed ordering of the set of inversions. For instance, we put all the inversions with one small number and one large number first, ordered lexicographically (first by their small number, then by their large number). Then the inversions with two small numbers follow (sorted by a recursive application of the same method), and then the inversions with two large numbers. Hence "the kth inversion" has a well-defined meaning, and the algorithm can be modified so that it select the kth inversion for a given k in $O(n \log n)$ time (see Exercise 1 for details). Hence if I is the total number of inversions, we can first choose a random number k in range from 1 to I, and then select the kth inversion.

By a similar method, one can even select p random independent inversions in time $O(n \log n + p)$: First we generate p random numbers and then we select the corresponding inversions; see Exercise 1.

Next, we describe several strategies for finding the kth intersection of the given lines.

Binary search. First we find the intersection with the smallest and the largest x-coordinate, x_{min} and x_{max}. These can be computed in time $O(n \log n)$, since the pairs of lines defining them have consecutive slopes (so that there are only $O(n)$ pairs to check).

We set $\ell = x_{min} - 1$ and $r = x_{max} + 1$. Further set $c = (r - \ell)/2$, and let p be the number of intersections in the interval $(\ell, c]$. If $p > k$ we will look for the kth intersection in the interval $(\ell, c]$, and if $p < k$ we will look for the $(k - p)$-th intersection in the interval $(c, r]$. In the case $p = k$ we have the desired intersection already.

After h iterations we know the x-coordinate of the kth intersection with accuracy $(r - l)/2^h$, which is quite sufficient for most applications. But if we want to determine the pair of lines defining the kth intersection, this strategy need not be satisfactory, since the x-coordinates of the intersections can theoretically be cluttered in an arbitrarily small interval.

Randomized halving. This is a modification of the previous approach. The difference is that we don't choose c as the geometric center of the current interval, but as the x-coordinate of some intersection lying in the current interval. This allows us to avoid bad cases when one half of the interval contains all the intersections and the other half contains none.

We select the intersection for the interval splitting at random, using the above mentioned algorithm for selecting a random inversion of a permutation. It can be shown that in expected $O(\log n)$ steps of the binary search, we reach an interval containing only one intersection (see Exercise 2). Each step needs $O(n \log n)$ time, and hence we get an expected total complexity of $O(n \log^2 n)$.

Randomized interpolation. This strategy uses the fact that p independent random intersections (inversions) can be selected in $O(n \log n + p)$ time. Let M be the number of intersections in the current interval $(\ell, r]$. Let us select n random independent intersections. How many of them are, on the average, before the kth intersection? (Here and in the rest of this section, by the kth intersection we mean the kth leftmost among all the M intersections lying in the interval $(\ell, r]$. On the other hand, the jth selected intersection means the jth leftmost among the n randomly selected intersections; if some intersection is selected several times, it is counted with the appropriate multiplicity.)

The expected number of selected intersections lying to the left of the kth intersection is $n\frac{k}{M}$. Hence we expect that the jth selected intersection, where $j = \lfloor n\frac{k}{M} \rfloor$, should be a good approximation for the position of the kth intersection. A more detailed analysis shows that with a quite high probability, the kth intersection lies between the j^-th and the j^+th selected intersection, where $j^- = \lfloor n\frac{k}{M} - 3\sqrt{n}\rfloor$ and $j^+ = \lceil n\frac{k}{M} + 3\sqrt{n}\,\rceil$ (we prove this in a moment).

One step of the algorithm goes as follows. Let ℓ' be the x-coordinate of the j^-th selected intersection, and let r' be the x-coordinate of the j^+th selected intersection. We check the following conditions:

(A) The kth intersection lies between ℓ' and r'.

(B) The interval $(\ell', r']$ contains at most $6M/\sqrt{n}$ intersections.

If both these conditions are satisfied, we replace the current interval $(\ell, r]$ by the interval $(\ell', r']$ (and update the value of k appropriately). If either (A) or (B) are violated, we repeat the selection of n random intersections.

If the algorithm is started with all the $\binom{n}{2}$ intersections in the current interval, after the first successful step we have an interval with $O(n^{3/2})$ intersections (by the condition (B)), and after the second successful step an interval with $O(n)$ intersections only. Among these, we can already select the desired intersection directly, in time $O(n \log n)$. We claim that each of the two required steps uses expected time $O(n \log n)$, which implies that the total expected time of the algorithm is $O(n \log n)$. To this end, it suffices to prove

Lemma 10.2.5 *For a random choice of n independent intersections among the M intersections lying in the considered interval $(\ell, r]$, the probability that conditions (A) and (B) are both satisfied is at least $\frac{1}{2}$.*

Proof. We use the so-called Chernoff inequality, in our case the following its version: Let $X_1, X_2, \ldots, X_n$ be independent random variables, each attaining value 0 with probability p and value 1 with probability $1-p$. If we put $X = X_1 + \cdots + X_n$, then for any $\lambda \geq 0$ we have

$$\mathrm{Prob}\left[X \geq \mathbf{E}X + \lambda\sqrt{n}\right] \leq e^{-\lambda^2/2} \tag{10.1}$$

(where $\mathbf{E}X = pn$), and also

$$\mathrm{Prob}\left[X \leq \mathbf{E}X - \lambda\sqrt{n}\right] \leq e^{-\lambda^2/2}. \tag{10.2}$$

Let us consider the condition (A). If, for instance, the kth intersection lies behind r', it means that at least j^+ among the n randomly selected intersections fall among the first k intersections. Imagine that the random intersections are selected one by one, by n independent random trials. Define a random variable X_i whose value is 1 if the intersection selected in the ith trial lies among the first k leftmost intersections, and 0 otherwise. Then $X = X_1 + \cdots + X_n$ is the number of selected intersections falling before the kth intersection. We have $\mathbf{E}X = n\frac{k}{M}$, and we can use (10.1):

$$\mathrm{Prob}\left[X \geq j^+\right] = \mathrm{Prob}\left[X \geq \mathbf{E}X + 3\sqrt{n}\right] \leq e^{-3^2/2} < \frac{1}{9}\,.$$

Similarly, using (10.2) we find that the kth leftmost intersection lies to the left of ℓ' with probability at most $\frac{1}{9}$. Analogously we get that the condition (B) is violated with probability at most $\frac{2}{9}$. □

Exercises

1. (a) Modify the algorithm for inversion counting described in the text in such a way that it finds the kth inversion for a given k in time $O(n \log n)$. [2 points]

   (b) Extend the algorithm in such a way that for given $k_1, k_2, \ldots, k_p$, it lists the inversions with these numbers in time $O(n \log n + p)$. [3 points]

2. Show that the strategy of random halving described in text requires $O(\log n)$ expected steps (one step = selecting a random intersection in the current interval and taking the left or the right part of the interval). [3 points]

10.3 Slope selection again: Parametric search

FILL IN

10.4 Ham-sandwich cut in the plane

Let A be a finite set in $\mathbb{R}^d$. We say that a hyperplane h *bisects* A if each of the open halfspaces defined by h contains at most $\lfloor |A|/2 \rfloor$ points of A. (Note that if A has an odd number of points, then any hyperplane bisecting A must contain a point of A.)

We will consider an algorithmic version of the following remarkable theorem.

Theorem 10.4.1 (Ham-sandwich theorem) *Let $A_1, A_2, .., A_d \subset \mathbb{R}^d$ be finite point sets. Then there exists a hyperplane h simultaneously bisecting all the A_i.*

The name of the theorem comes from the following popular interpretation of the 3-dimensional version: "For every sandwich made of bread, ham and cheese, the bread, the cheese and the ham can all be halved by a single straight cut." The Ham-sandwich theorem is usually proved by topological methods, and we will not give a general proof here. We will consider an algorithm for the planar version, i.e. for the following problem.

Problem 10.4.2 *Given finite point sets A_1 and A_2 in the plane, find a ham-sandwich cut line for them, i.e. a line bisecting both A_1 and A_2.*

We will also obtain an existence proof for the ham-sandwich cut as a by-product of the discussion of the algorithm. For simplicity, we assume that the set $A_1 \cup A_2$ is in general position, and that both A_1 and A_2 have an odd number of points. As was noted above, any ham-sandwich cut must then contain a point of A_1 and a point of A_2. Hence there are only $O(n^2)$ candidate lines for the ham-sandwich cut, and each of them can be checked in $O(n)$ time; this yields an obvious $O(n^3)$ algorithm. We will derive an algorithm with $O(n)$

expected running time. The problem of finding a ham-sandwich cut has some applications in other geometric problems, and we can also demonstrate several general techniques in the algorithm.

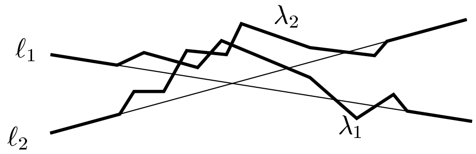
We will work in a dual setting (using the duality $\mathcal{D}$ defined in section 4.1). We thus obtain two sets $L_1 = \mathcal{D}(A_1)$ and $L_2 = \mathcal{D}(A_2)$ of non-vertical lines in general position. We suppose that both $n_1 = |L_1|$ and $n_2 = |L_2|$ are odd. We also put $n = n_1 + n_2$.

Consider a point dual to a ham-sandwich cut line (for brevity, call such a point a *ham-sandwich cut*). Such a point is an intersection of a line of L_1 and a line of L_2, and it has precisely $k_1 = \lfloor n_1/2 \rfloor$ of the L_1-lines below it and $k_2 = \lfloor n_2/2 \rfloor$ of the L_2-lines below it.

Recall from sections 6.2 and 7.8 that the level k_1 of the arrangement of L_1 is the polygonal line formed by all edges of level k_1, where the level of an edge is the number of lines lying strictly below some its interior point. Denote the level k_1 of the arrangement of L_1 by λ_1, and the level k_2 of the arrangement of L_2 by λ_2. In this terminology, any intersection of λ_1 and λ_2 is a ham-sandwich cut.

Lemma 10.4.3 *The levels λ_1 and λ_2 intersect in an odd number of points. In particular, a ham-sandwich cut exists.*

Proof. First we observe that the left unbounded ray and the right unbounded ray of λ_1 lies on the same line $\ell_1 \in L_1$ (the one with the median slope). Similarly the unbounded rays of λ_2 are parts of some line $\ell_2 \in L_2$. One of these lines, say ℓ_1, has smaller slope. This means that far enough to the left, λ_1 is above λ_2, while far to the right, λ_1 is below λ_2:



Since the levels λ_1 and λ_2 are continuous polygonal curves with at most finitely many intersections, they intersect an odd number of times. □

Next, we note the following:

Observation 10.4.4 *If V is an open vertical strip in the plane, we can test whether the number of ham-sandwich cuts within V is odd or even, in $O(n)$ time.*

Proof. Let ℓ be the left boundary line of V and r the right boundary. We may assume that neither r nor ℓ pass through any intersection in the arrangement of $L_1 \cup L_2$ (since V is open, we can always shrink it a little without changing the number of ham-sandwich cuts in V). In linear time, we can compute the intersections of lines of L_1 with ℓ, and using a linear-time selection algorithm, we can select the k_1th intersection from below, which gives us the intersection of λ_1 with ℓ; let us denote this point by $\lambda_1(\ell)$. Similarly we find the intersections

$\lambda_2(\ell)$, $\lambda_1(r)$, and $\lambda_2(r)$. Now λ_1 and λ_2 intersect an odd number of times in V if and only if the vertical order of $\lambda_1(\ell)$ and $\lambda_2(\ell)$ is opposite to the vertical order of $\lambda_1(r)$ and $\lambda_2(r)$. □

A suboptimal algorithm. The first idea of the algorithm for finding a ham-sandwich cut is a binary search, similar to the randomized halving strategy from section 10.2. Initialize V to an open vertical strip containing all intersections of the arrangement of $L_1 \cup L_2$. Then select a random intersection of the arrangement of $L_1 \cup L_2$ within V, and let c be the vertical line passing through that intersection. If c contains a ham-sandwich cut we are done. Otherwise, we consider the open strips V_1 and V_2 into which is V divided by c. Since V has an odd number of ham-sandwich cuts, exactly one of the strips V_1 and V_2 has an odd number of ham-sandwich cuts. We can decide which one is it by Observation 10.4.4 and narrow the search to it. In expected $O(\log n)$ of these search steps, we have to hit a ham-sandwich cut by one of the subdividing lines c (otherwise we would reach a strip with no intersection in it which is impossible). Each step can be performed in $O(n \log n)$ time — this is the time needed for selecting a random intersection in the current strip, and hence we have ham-sandwich cut algorithm with an $O(n \log^2 n)$ expected running time.

Pruning. To improve this time to linear, we use a general strategy called *prune and search*. The improved algorithm will be divided into *phases*. In each phase, we discard at least $\frac{1}{6}$ of the current lines (any positive fixed fraction instead of $\frac{1}{6}$ would do). Each phase takes time linear in the current number of lines, and since the number of lines decreases geometrically, the total running time is also linear.

At the beginning of each phase, some lines of L_1 and L_2 have already been discarded; only subsets $L_1' \subseteq L_1$ and $L_2' \subseteq L_2$ remain. We also have a current open vertical strip V and integers k_1 and k_2, such that the following invariant holds:

> The level k_1 in the arrangement of L_1' and the level k_2 in the arrangement of L_2' intersect an odd number of times in the strip V, and each such intersection is a ham-sandwich cut (for the original problem, i.e. for L_1 and L_2).

Initially, of course, we have $L_1' = L_1$, $L_2' = L_2$, $k_1 = \lfloor n_1/2 \rfloor$, $k_2 = \lfloor n_2/2 \rfloor$ and V is a vertical strip containing all vertices of the arrangement of $L_1 \cup L_2$.

Let us describe how a phase works. Call the bigger set of L_1' and L_2' the *green lines* and the smaller one the *yellow lines* (if the cardinalities are equal choose arbitrarily). Let g, y be the corresponding indices for "green" and "yellow" (i.e. if L_1' are the green lines then $g = 1$ and $y = 2$, etc.).

We will discard at least $\frac{1}{3}$ of the green lines, i.e. at least $\frac{1}{6}$ of the total number of lines. The green lines will play the main role in the phase; there will be only few references to the yellow lines.

Let λ_g be the level k_g in the arrangement of the green lines, and similarly for λ_y. Choose 200 random independent pairs of green lines (the constant 200 is chosen somewhat arbitrarily; by a more careful argument, one could show that a smaller constant works as well). Compute the 200 corresponding green intersections, and find those falling into the current strip V. Erect vertical lines through them; this divides V into strips $V_0, V_1, \ldots, V_t$, $t \leq 200$. We will need the following lemma, whose proof is left to Exercise 3:

Lemma 10.4.5 *With probability at least $\frac{1}{2}$, no V_i contains more than $\frac{1}{20}$ of the $\binom{n_g}{2}$ green intersections.*

As in Observation 10.4.4, we find some V_i with an odd number of intersections of λ_g and λ_y in it. (In practice, we would not test each V_j separately, rather we would use a binary search strategy to locate a good V_i.)

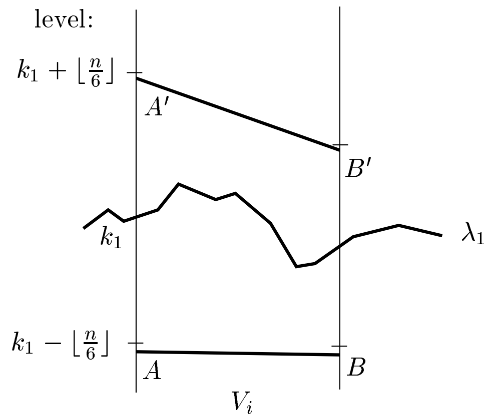


Figure 10.1: The trapezoid T.

Next, we construct a trapezoid $T = ABB'A'$ in V_i (see fig. 10.1). The points A on the left boundary of V_i and B on the right boundary lie just below the level $k_g - \lfloor n_g/6 \rfloor$ of the green lines, and the points A', B' lie just below the level $k_g + \lfloor n_g/6 \rfloor$ (if $k_g - \lfloor n_g/6 \rfloor \leq 0$, say, then A and B are chosen below all green lines, and similarly for the top side). We check if both the top and bottom sides of the trapezoid T are intersected by at most $n_g/3$ green lines each. If not, we restart the whole phase from the beginning (the choice of the 200 random intersections etc.). If yes, we pass to the next phase of the algorithm. In this next phase, V_i will be the current strip. The set of yellow lines and k_y will remain the same. The green lines not intersecting the trapezoid T will be discarded, and k_g will be replaced by $k_g - b$, where b is the number of the green lines passing completely below T. This finishes the description of the algorithm, and it remains to show that everything works.

First we note that a successful phase indeed discards at least $\frac{1}{3}$ of the green lines: Each side of T is intersected by $\leq n_g/3$ of the green lines, hence we have $\frac{4}{3}n_g$ intersections altogether, and each line is counted twice in this. Hence at least $n_g/3$ green lines miss T.

To verify that the invariant of the algorithm holds also for the next phase, we show that no intersection of the levels λ_g and λ_y within the strip V_i is neglected by discarding the green lines missing the trapezoid T. To this end, we show that within V_i, the level λ_g lies completely inside T. So suppose that λ_g breaks below the side AB, say; then some point C on this side has level larger than k_g. Since both A and B have level $k_g - \lfloor n_g/6 \rfloor$, both the segments AC and CB have to be intersected by more than $n_g/6$ green lines. But this contradicts the condition that the side AB is intersected by $\le n_g/3$ green lines.

Finally we show that expected number of times a phase is restarted is bounded by a constant. To this end, we prove that if the strip V_i contains $\le \frac{1}{20}\binom{n_g}{2}$ green intersections (which happens with probability at least $\frac{1}{2}$ by Lemma 10.4.5), then the top and bottom sides of T are intersected by $\le n_g/3$ green lines each. Let u be the number of green lines intersecting the side AB "upwards", i.e. having A above and B below, and d the number of green lines intersecting AB "downwards". Since A and B have the same level in the arrangement of the green lines, we have $u = d$. But every "upwards" line intersects every "downwards" line in the strip V_i, and hence V_i contains at least ud intersections. If this should be at most $\frac{1}{20}\binom{n_g}{2} \le n_g^2/40$, we get $u + d < n_g/3$ as claimed. This shows that each phase works in expected time linear in the current number of lines, and hence the algorithm has a linear total expected time.

Exercises

1. The algorithm in this section was formulated for input sets with an odd number of points. Suggest how to modify it so that it handles also point sets with an even number of points. [1 point]

2. The planar ham-sandwich theorem essentially claims that any two finite point sets can be simultaneously divided by a line in ratio 1:1. Show that this is the only ratio for which such a result can hold in general; for example, it is not true that any two sets can each be divided in the ratio 1:2 by a single line. (Suggest a suitable exact formulation first.) [3 points]

3. Let the numbers $1, 2, \ldots, N$ be white. Choose one of them at random and color it black; repeat 200 times (independent attempts). Show that after this, with probability at least $\frac{1}{2}$, the longest interval of consecutive white numbers has length $\le N/20$. [4 points]

4. Let $A_1, A_2 \subset \mathbb{R}^2$ be finite point sets separated by a line ℓ (i.e. A_1 lies in one of the open halfplanes defined by ℓ and A_2 in the other one). Prove that a ham-sandwich cut line for A_1 and A_2 is unique. [3 points]