Lecture notes on geometric algorithms

Jiří Matoušek

Version Dec/2001

These notes cover several geometric algorithms. The problems solved by them are of basic importance in computational geometry. But the techniques explained in these concrete examples is even more important. For a more extensive introduction to computational geometry consult, for example,

> M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf: *Computational Geometry: Algorithms and Applications.* Springer, Berlin, 1997.

We are going to consider incremental algorithms, and in particular, *randomized* incremental algorithms. An incremental algorithm inserts the input objects one by one, and after each insertion it updates the structure 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 some more complicated situations, an efficient algorithm is obtained by inserting the objects in random order.

1 Construction of a line arrangement

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

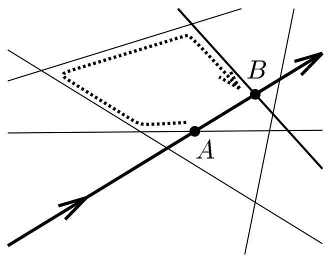
The arrangement can be represented, for instance, by specifying the incidence relations between vertices and edges and between edges and cells (more precisely, each vertex and each cell points to a list of its adjacent edges, and each edge points to its vertices and to its cells).

Alternatively, the arrangement can be stored as a planar graph (with an artificial vertex at infinity). Each vertex has a circular doubly-linked list of edges, and each edge points to its end-vertices.

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). The idea is to walk along the new line, find all intersected edges and cells of the old arrangement, and split them appropriately. To start the walk, we can (for instance) find the edge on ℓ_1 that is intersected by ℓ_{i+1}, in $O(i)$ time, and go from there.

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

It remains to determine the complexity of this algorithm. The cost of inserting the ℓ_{i+1} is proportional to i plus 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. The total running time is $O(n^2)$.

An arrangement of hyperplanes in $\mathbf{R}^d$ 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.

2 Convex hull 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 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 insert them into the current set; after the ith step, the convex hull of the points $p_1, p_2, \ldots, p_i$ has been constructed.

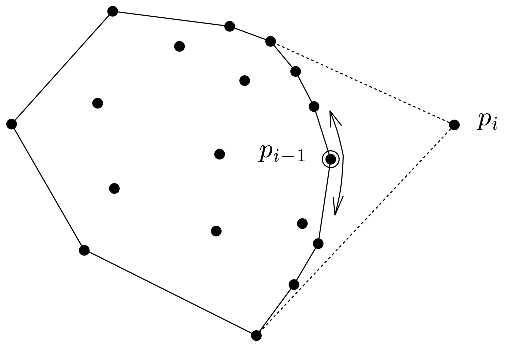


Figure 1: Adding a new point to the convex hull

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 that 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. 1). Note that p_i always sees the point p_{i-1}.

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. The 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, we 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). However, the above (seemingly simple-minded) solution is asymptotically 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 be deleted only once, and hence $\sum_{i=1}^{n} n_i < n$, and the total time is $O(n)$ as claimed. □

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 (using 2-dimensional linear programming).

3 Randomized QUICKSORT

In the planar convex hull algorithm (section 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. We begin with a variant of the well-known sorting algorithm QUICKSORT.

Algorithm QUICKSORT, having received a sequence $(x_1, x_2, \ldots, x_n)$ 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 the average". This is partially expressed in the following theorem.

3.1 Theorem. *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 has 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 3.1. Let $T_i = T_i(\pi)$ be the number of elements compared to the element $x_{\pi(i)}$ at the moment when $x_{\pi(i)}$ 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

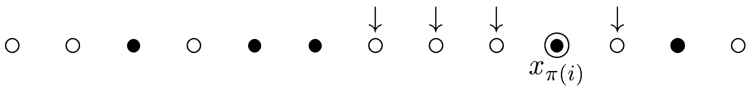


Figure 2: Illustration to the analysis of QUICKSORT

$\pi(1)-2$, and for $\pi(2)>\pi(1)$ we have $T_2=n-\pi(1)-1$. In general, T_i can be interpreted according to fig. 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.

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 when 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); that is, bad permutations are quite rare. We will not consider this here.

4 The 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 that, 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 nonintersecting segments in the plane (some of them may

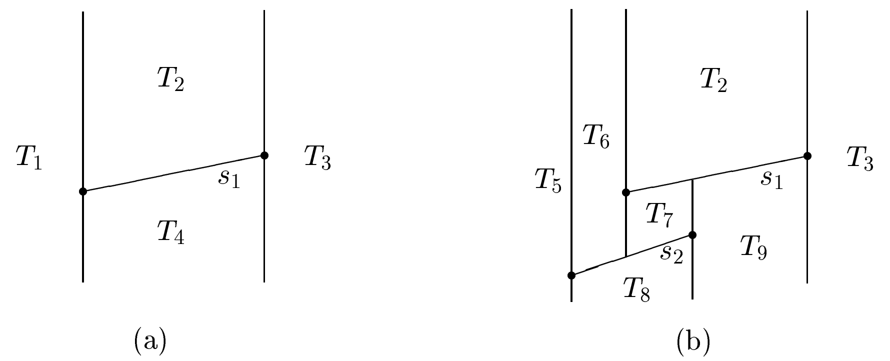


Figure 3: Trapezoidal decompositions after inserting s_1 (a), and after inserting s_2 (b).

share endpoints), build a data structure that, 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.

4.1 Definition. *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 this segment is the same for all points in one trapezoid, and it can be stored together with the trapezoid.

General position assumption. We assume that no segment is vertical and that no two distinct endpoints of the segments have the same x-coordinate. But it is allowed that several segments have a common endpoint.

Construction. Let $s_1, s_2, \ldots, s_n$ be the given nonintersecting 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. 3(a)). By inserting s_2 we can get a situation as in fig. 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 destroyed trapezoid we store pointers to the new trapezoids that arose in its territory. In the situation of fig. 3(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 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 do we locate a point using this information? We "walk through history". We follow whose territory the query point x belonged 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 that 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 the trapezoid containing x is changed by inserting the ith segment.

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. If none of such defining segments is deleted, then 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 the average point location time is $O(\log n)$ for every fixed point x. 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 and $O(n \log n)$ 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.

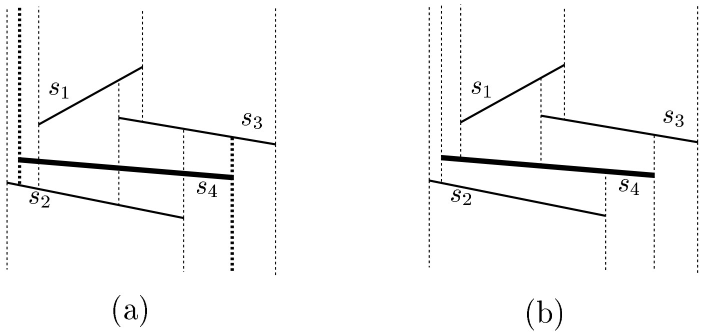


Figure 4: Subdividing old trapezoids intersected by s_i (a), and merging the adjacent new trapezoids (b).

Insertion of a new segment:

1. Locate the left 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 that share a full vertical side (see fig. 4).

The first step needs expected $O(\log i)$ time, 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 bounded above by the average number of trapezoids that 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 at most 4 defining segments. The average number of deleted trapezoids per segment is thus $O(1)$ (double counting). Therefore $\mathbf{E}[Y_i] = O(1)$ and $\sum_{i=1}^{n} \mathbf{E}[Y_i] = O(n)$. □

5 Linear programming in a fixed dimension

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

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

$$Ax \leq b, \tag{1}$$

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 $\mathbf{R}^d$ (each of them described by one of the inequalities in the system (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 that would solve the linear programming 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 randomized; 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).

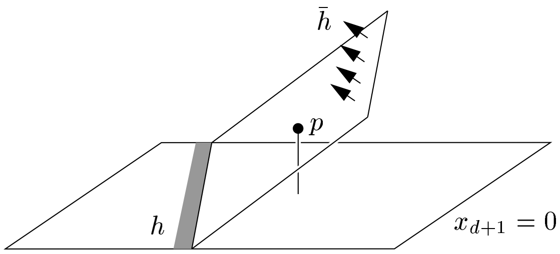
1. The intersection of our halfspaces is nonempty; that is, an admissible solution exists.

2. The objective function $\langle c, x \rangle$ is bounded from below on the set of admissible solutions, and 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 $\mathbf{R}^d$ with the $x_{d+1} = 0$ plane in $\mathbf{R}^{d+1}$, and for each constraint in the original problem, regarded as a halfspace $h \subset \mathbf{R}^d$, consider the halfspace $\bar{h} \subset \mathbf{R}^{d+1}$ 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 $\mathbf{R}^{d+1}$, 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 has 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; namely, 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 count only 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; that is, 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 (exercise).

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. It is also 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.

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