Backwards Analysis of Randomized Geometric Algorithms

Raimund Seidel*
Computer Science Division
University of California Berkeley
Berkeley CA 94720
USA

August 20, 1992

Abstract

The theme of this paper is a rather simple method that has proved very potent in the analysis of the expected performance of various randomized algorithms and data structures in computational geometry. The method can be described as "analyze a randomized algorithm as if it were running backwards in time, from output to input." We apply this type of analysis to a variety of algorithms, old and new, and obtain solutions with optimal or near optimal expected performance for a plethora of problems in computational geometry, such as computing Delaunay triangulations of convex polygons, computing convex hulls of point sets in the plane or in higher dimensions, sorting, intersecting line segments, linear programming with a fixed number of variables, and others.

1 Introduction

The curious phenomenon that randomness can be used profitably in the solution of computational tasks has attracted a lot of attention from researchers in recent years. The approach has proved useful in such diverse areas as number theory, distributed computing, combinatorial algorithms, complexity theory, and others. For surveys see [34, 47, 55]. It is interesting that in Rabin's seminal 1976 paper [46] which initiated the study of algorithmic uses of randomness one of the two example problems considered was a computational geometry problem, namely

*Supported by NSF Presidential Young Investigator Award CCR-9058440. Email address: seidel@cs.berkeley.edu

the Euclidean closest pair problem. Of course computational geometry as a field came about a number of years later. Shamos's thesis [52] appeared in 1978. Talking about possible future research directions in the epilogue Shamos mentions "probabilistic algorithms" and writes *"This approach seems to be able to yield geometric algorithms of startling efficiency."* However, it was to take almost another decade until randomized or probabilistic methods were investigated in computational geometry in more detail.

In the mid 80's Ken Clarkson started to create and apply his random sampling technique, which in the mean time has developed into a surprising general framework with numerous applications [15, 16, 17, 18]. Around the same time Haussler and Welzl published their important paper [32] that introduced ε-nets and the VC-dimension, which have become very useful and versatile tools in the design and analysis of randomized algorithms. In a series of papers [42, 43, 44] Mulmuley introduced and studied a number of probabilistic games that allow a rather tight analysis of the expected behaviour of a number of geometric algorithms. At the same time a steady stream of papers of a more specialized nature started to appear, dealing with randomized solutions for a wide range of particular problems ([7, 8, 9, 30, 1, 50, 12] is a non-exhaustive, random(?) list of references).

The purpose of this paper is to popularize a rather simple trick for analyzing the expected performance of certain randomized algorithms:

Analyze an algorithm as if it was running backwards in time, from output to input.

This is based on the observation that often the cost of the "last" step of an algorithm can be expressed as a function of the complexity of the "final product" output by the algorithm.

In this paper we apply this "backwards analysis" to a number of problems and algorithms. We start with demonstrating the idea of backwards analysis on a simple algorithm due to Paul Chew [14] for constructing the Delaunay triangulation of the vertices of a convex polygon and show that it has linear expected running time. As far as we know Chew was the first to apply this type of analysis to a computational geometry algorithm.

Next we apply backwards analysis to an algorithm due to Mulmuley for determining all intersection pairs among a set of line segments in the plane [42, 43]. Mulmuley's original analysis of the algorithm was based on probabilistic games and was rather involved. The "backwards" view leads to a considerable simplification, and also applies equally well to a more general version of the problem, where the segments need not be straight and can intersect each other more than once.

Some claim — tongue in cheek — that any method of value in computational geometry must be also applicable to the planar convex hull problem. Thus we present a planar convex hull algorithm along with the analysis of its $O(n \log n)$ expected running time. The algorithm is somewhat reminiscent of QUICKSORT. Thus we apply the principle of backwards analysis to QUICKSORT and with little effort we derive the *exact* value for the expected number of comparisons made by QUICKSORT. Moreover, backwards analysis turns out to provide a particularly easy approach for bounding the probability that the running time of QUICKSORT

significantly exceeds its expectation.

Next we give a negative example. We consider a triangulation problem along with an algorithm that is a straightforward generalization of QUICKSORT. Interestingly enough, backwards analysis can apparently not be applied to this algorithm.

Following this we turn our attention to linear programming when the dimension is small. We present a very simple algorithm for solving linear programs with m constraints in d variables that has expected running time $O(d!m)$. Again the analysis via the backwards view is very straightforward. We then describe an adaption of this method due to Welzl [56] for the problem of findingthe smallest enclosing balls for a finite set of points in $\mathbb{R}^d$.

Finally we turn our attention to the problem of constructing the convex hull of n points in $\mathbb{R}^d$. We consider a randomized incremental algorithm and show that for $d > 3$ there is a simple variant that via backwards analysis can easily be shown to have optimal $O(n^{\lfloor d/2 \rfloor})$ expected running time. Then we consider the "conflict graph" based algorithm due to Clarkson and Shor [18] and present a new backwards analysis due to Clarkson [21] that shows that this algorithm has "optimal" expected running time for all dimensions d.

2 Delaunay Triangulations of Convex Polygons

Let S be a set of n points in the plane. The *Delaunay triangulation of* S, for short $DT(S)$ is a plane graph whose vertices are the points in S and that connects two points $p, q \in S$ by a straight edge iff there is an open disk that has p and q on its boundary but that contains no points of S. When S is non-degenerate in the sense that no four points of S are co-circular and not all of S lies on one straight line, $DT(S)$ is always a triangulation. The bounded faces of $DT(S)$ are then exactly those triangles of points in S for which the smallest circumscribed disk contains no point of S in its interior.

Delaunay triangulations along with their dual structures, which are known as Voronoi diagrams, have been studied intensively in computational geometry. Their efficient construction and the recognition of their multifarious useful properties by Shamos and Hoey [53] were instrumental in getting the field of computational geometry started. By now these structures along with numerous generalizations are standard fare in the field (see [45, 25, 3, 36]).

Shamos's and Hoey's big feat was an $O(n \log n)$ algorithm for the construction of Delaunay triangulations. It was also very soon realized that the $O(n \log n)$ bound was asymptotically worst case optimal for reasonable models of computation. However, for a long time the question remained whether $DT(S)$ could be computed in $o(n \log n)$ time when the point set S has some special structure. In particular one was interested in the question whether an $O(n)$ time bound was possible when S consists of the vertices of a convex polygon, given in order around the polygon. An affirmative answer was eventually given in 1986 by Aggarwal, Guibas, Saxe, and Shor [2] using an ingenious but rather involved algorithm.

In the mean time, almost unnoticed, Paul Chew had discovered a very simple *randomized* algorithm for this problem with linear expected running time [14]. Chew employed backwards

anlysis, and as far as we know this was the first time that this trick was used in computational geometry. His algorithm and analysis shall serve as a first example for the concept of backwards analysis.

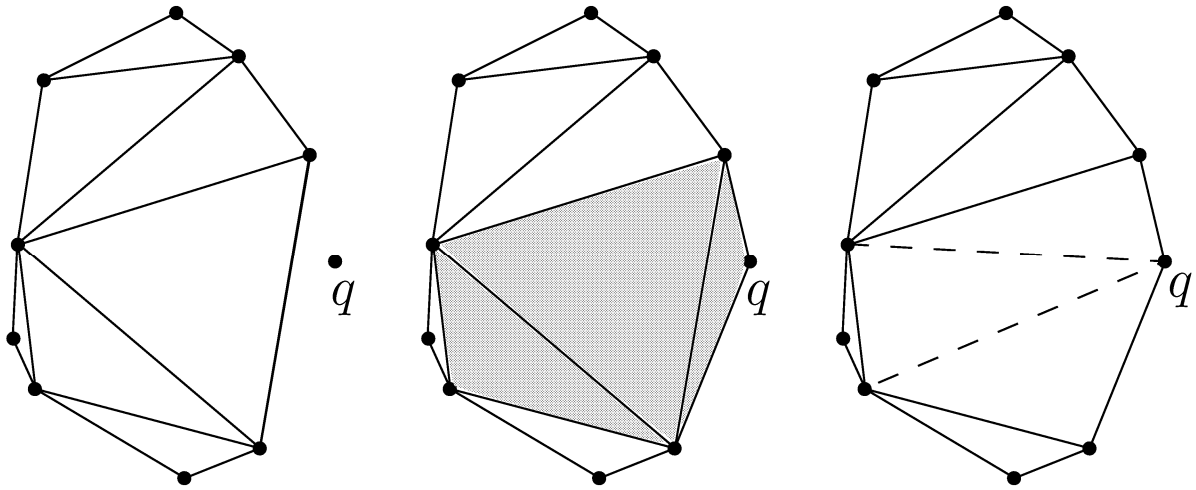
So let S be the n vertices of a convex polygon P given in order around P. Assume that no four points in S are co-circular, a condition that could easily be simulated using standard perturbation techniques [25, pp. 185]. Chew's algorithm proceeds as follows:

> If S consists of only three points, then the triangle spanned by them forms $DT(S)$.
>
> If S contains more than three points, then choose a random point $q \in S$, let p and r be its two neighboring vertices around P, and let $S' = S \setminus \{q\}$.
>
> Recursively compute $DT(S')$, attach the triangle p, q, r to this triangulation, and then update this triangulation D of S as follows to obtain $DT(S)$:
>
> First identify all "bad" triangles of D, namely p, q, r and all triangles of $DT(S')$ whose circumscribed disk contains q. This is done by performing a depth-first search in the dual graph G of D whose nodes are the triangles of D and that has two triangles adjacent iff they share a common edge of D. This depth-first search is to start at the triangle p, q, r. Since the set of bad triangles is known to form a connected subgraph of G (a fact not proven here) and since G has maximum degree 3, all bad triangles can thus be identified in time proportional to their number.
>
> Finally remove from D all edges that have bad triangles on both sides, and retriangulate the resulting face by introducing all diagonals that have q as an endpoint. (See Figure 1 for an illustration.)

What is the expected running time of this algorithm? The question really is, what is the expected running time of this procedure with the recursive call excluded. It is not hard to see that this cost is proportional to the number of bad triangles. So what is the expected number of bad triangles?

This question seems difficult to answer, especially if we fix our attention on one particular point q. The trick now is to express this cost not in terms of $DT(S')$ and q, but *in terms of the resulting structure* $DT(S)$. It is not hard to see that the number of bad triangles is exactly one more than the number of diagonals with endpoint q that are introduced in the last step of the algorithm. Or in other words, the number of bad triangles is proportional to the degree of q in $DT(S)$. But of course, if q is chosen from S uniformly at random, then the expected degree of q in $DT(S)$ is the sum of the degrees of all vertices in S divided by n, which is twice the number of edges of $DT(S)$ divided by n, which, since $DT(S)$ is an outer-planar graph, is $2 * (2n - 3)/n = 4 - 6/n$.

Thus the expected time necessary to perform the body of the procedure outlined above without the recursive call is constant. From this it follows immediately that the overall expected time necessary is $O(n)$.



$DT(S \setminus \{q\})$ "bad" triangles shaded $DT(S)$

Figure 1

Let us point out once more that the decisive part in the analysis of the algorithm is to express the cost of the "last step" (which we abstracted out as the body of a head-recursive procedure) as a function of the produced output. If we were to run the algorithm backwards starting with $DT(S)$ and repeatedly deleting random points of S at cost proportional to their degree, the expected cost of a deletion could be expressed as a function of the "input" (namely the average degree of the current triangulation) and it would be clear that this is constant.

3 Intersecting Line Segments

Let S be a set of n straight line segments in the plane. We are interested in finding all intersecting pairs of segments in S.

The first non-trivial algorithm for solving this problem was given by Bentley and Ottmann in 1979 [5]. It was based on the sweep paradigm and achieved a worst case running time of $O((K+n)\log n)$, where K is the output size, namely the number of intersecting pairs of segments in S. Since it is possible that all segments of S intersect each other, this algorithm can have an $O(n^2 \log n)$ running time, which is inferior to the $O(n^2)$ time of the trivial method of checking every pair in S. Thus the question arose, whether a bound of the form $O(K+n\log n)$ was possible. Since it is easy to show by reduction from element uniqueness that $\Omega(n \log n)$ is a lower bound to the segment intersection problem, and since $\Omega(K)$ is also a lower bound as at least this much time has to be spent on output, one could not hope for anything better than $O(K + n \log n)$.

In 1983 Chazelle [10] came close to this goal with a rather complicated algorithm whose worst case running time was $O(K + n\log^2 n/\log\log n)$. Finally, five years later he and Edelsbrunner [11] designed an even more complicated deterministic algorithm that did achieve the

$O(K + n \log n)$ worst case running time. Around the same time independently Mulmuley [42] as well as Clarkson and Shor [18] developed rather simple randomized algorithms with $O(K + n \log n)$ expected running time. Clarkson and Shor based the analysis of the running time of their algorithm on the general theory of random sampling. They even managed to come up with a version of the algorithm that required only $O(n)$ space. Mulmuley analyzed the performance of his algorithm via probabilistic games that he developed for this purpose. His analysis is reasonably complex, however it yields rather tight constants.

In this section we present Mulmuley's algorithm and give a very simple analysis of its expected performance that is based on our backwards view.

For the sake of ease of presentation let us assume we are dealing with a set S of n segments that is non-degenerate in the sense that no two segments of S have the same endpoint, no three segments intersect in a common point, no two segment endpoints have the same x-coordinate, and that no segment endpoint lies in the relative interior of some other segment. As usual such non-degeneracy could be simulated by standard perturbation methods [25, pp. 185], or also the algorithm could easily be modified so that none of these assumptions are necessary.

Mulmuley's algorithm does more than just determine which pairs of segments in S intersect. It constructs what we call the *trapezoidal decomposition* induced by S. This decomposition $\mathcal{T}(S)$ can be intuitively defined as follows: First draw a sufficiently large axis-parallel rectangle frame F that contains in its interior all segments of S. Next draw all segments of S in the rectangle F. Finally, from each intersection point and from each segment endpoint draw its *vertical extensions*, i.e. start drawing two vertical rays, one going up, the other going down, that extend until they hit a segment of S or the boundary of F (see Figure 2). Thus F is decomposed into trapezoids that each have two vertical sides (one of which can have length 0). Using a sweep argument it is not hard to prove that $\mathcal{T}(S)$ contains exactly $3(n + K) + 1$ trapezoids; thus, when viewed as a planar graph, $\mathcal{T}(S)$ has $O(K+n)$ faces, edges, and vertices.

Let us define for any subset $R \subset S$ the trapezoidal decomposition $\mathcal{T}_S(R)$ in a similar way as follows: Draw all segments of R in the rectangle frame F, and for each intersection point of segments in R as well as for each endpoint of a segment in S (note: *in* S) draw its vertical extensions. But now the rays that form the vertical extensions extend only until they hit a segment of R or the boundary of the rectangle F (see Figure 3). The decomposition $\mathcal{T}_S(R)$ partitions F into $2n + r + 3K_R + 1$ trapezoids, where $r = |R|$ and K_R is the number of pairs of intersecting segments in R, and thus $\mathcal{T}_S(R)$ as a planar graph has $O(n + K_R)$ faces, edges, and vertices.

Mulmuley's algorithm for computing $\mathcal{T}(S) = \mathcal{T}_S(S)$ is very simple: First compute $\mathcal{T}_S(\emptyset)$ and then insert the segments of S into the trapezoidation in random order to compute $\mathcal{T}_S(R)$ for an every increasing $R \subset S$.

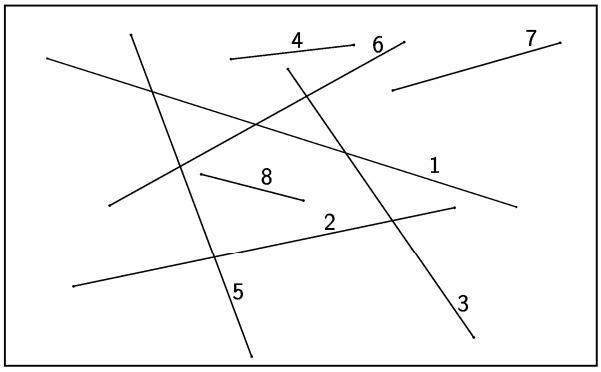


Figure 2a: Set S of 8 segments in a frame

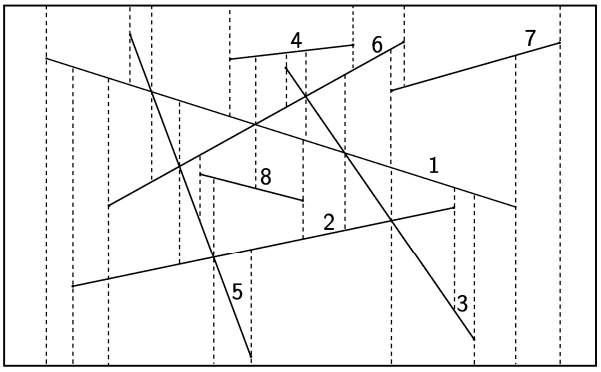


Figure 2b: Trapezoidation $\mathcal{T}(S)$

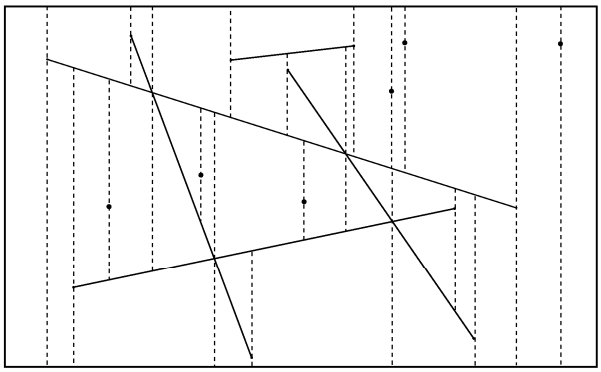


Figure 3: $\mathcal{T}_S(R)$ for $R = \{1, 2, 3, 4, 5\}$

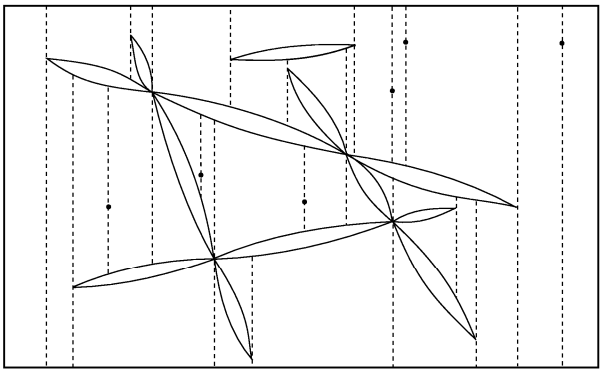


Figure 4: Introduction of zero-width faces for pieces yields $G_S(R)$

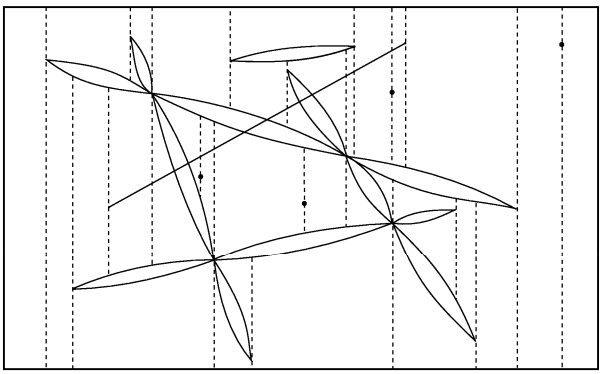


Figure 5a: Inserting segment 6

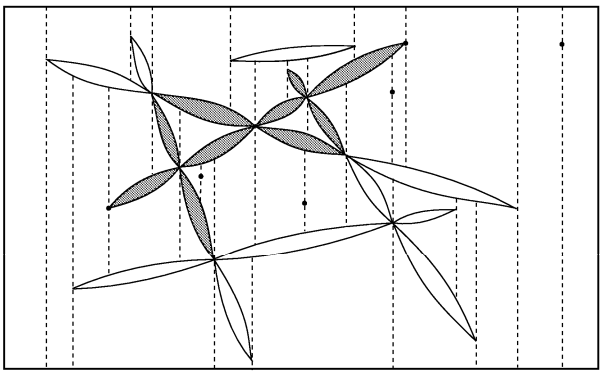


Figure 5b: $\mathcal{P}_R(6)$ shaded

We will express his algorithm for computing $\mathcal{T}_S(R)$ recursively:

> If $R = \emptyset$ then compute $\mathcal{T}_S(R)$ directly by sorting the endpoints of S by their x-coordinates.
>
> Otherwise randomly pick a segment $s \in R$, recursively compute $\mathcal{T}_S(R \setminus \{s\})$, and introduce s into this trapezoidation to obtain $\mathcal{T}_S(R)$.

We still have to specify how a segment s is introduced into the current trapezoidation. In order to do this we need to give more detail about the representation of trapezoidations. Mulmuley chooses a somewhat idiosyncratic representation that essentially works as follows: Consider $\mathcal{T}_S(R)$, and consider some segment $s \in R$. Assume s is intersected by i other segments in R. Thus s is partitioned into $i + 1$ *pieces*. Conceptually Mulmuley makes each piece of each segment in R into a narrow zero-width face, and obtains this way from $\mathcal{T}_S(R)$ a plane graph that we denote by $G_S(R)$. Figure 4 should make the idea clear. Note that $G_S(R)$ has the same number of vertices as $\mathcal{T}_S(R)$ and thus it also has complexity $O(n + K_R)$. Also note that in the graph $G_S(R)$ faces that correspond to trapezoids of $\mathcal{T}_S(R)$ have at most six edges around them, only the zero-width faces can have more than a constant number of edges around them.

How does one now introduce segment s into $G_S(R')$ to obtain $G_S(R)$, where $R' = R \setminus \{s\}$? It works in two phases (that of course could be combined into one). In the first phase one determines which faces and edges of $G_S(R')$ are intersected by s. In the second phase the new faces of $G_S(R)$ are created. This involves splitting the faces of $G_S(R')$ that are intersected by s, creating the zero-width faces for the pieces of s, introducing the vertical extensions from the intersection points of s with the other segments of R, and merging faces that are separated by vertical edges that are not part of vertical extensions any more because of the introduction of s. We leave the details of the second phase to the reader.

The first phase can simply be done as follows: Since the two endpoints of s are vertices of $G_S(R')$ already, we can determine in constant time, say, the leftmost face of $G_S(R')$ that is intersected by s. Now we "thread" s through $G_S(R')$ in the usual way, similar to the incremental line arrangement construction algorithm [27, 13]: We walk along the segment s; assume we just entered a face f through some edge e; we determine through which edge the segment s leaves f and which new face the segment enters by simply testing all edges of f (say, in clockwise order starting after e). Now we have reached a new face, and we repeat. Obviously, this procedure can also tell when we have reached the right endpoint of s.

What is the time necessary for introducing segment s? It is not hard to see that the cost of phase one dominates the cost of phase two. Thus it suffices to consider just the cost of phase one. This cost is clearly the sum of the degrees of all the faces of $G_S(R')$ that are intersected by s (here the degree of a face f of $G_S(R')$, for short $deg(f, G_S(R'))$, is the number of edges of $G_S(R')$ incident to f). However, what does this evaluate to for a random segment $s \in R$?

Now let us apply backwards analysis. The first important step is to express the cost of phase one of introducing a segment s into $G_S(R')$ in terms of the result graph $G_S(R)$ and not in terms of $G_S(R')$. It is not hard to see that this cost is given by $\sum_{f \in \mathcal{P}_R(s)} deg(f, G_S(R))$,

where $\mathcal{P}_R(s)$ is the set of all zero-width faces of $G_S(R)$ that either derive from pieces of s in $G_S(R)$ or from those pieces of other segments in R that are incident to intersection points with s (see Figure 5).

It follows that if s is randomly chosen from the r segments in R, then the expected cost of adding s to $G_S(R \setminus \{s\})$ is proportional to

$$\frac{1}{r} \sum_{s \in R} \sum_{f \in \mathcal{P}_R(s)} deg(f, G_S(R)) \ .$$

But in this double sum every piece of a segment in R contributes at most three times. Thus if $\mathcal{P}(R)$ denotes the set of all faces that derive from pieces of segments in R, then this double sum is at most

$$\frac{3}{r} \sum_{f \in \mathcal{P}(R)} deg(f, G_S(R)) \ .$$

Since $G_S(R)$ is a planar graph, this sum is clearly proportional to the complexity of the graph, and thus it is $O(n + K_R)$. It follows that the expected cost of introducing the last segment of R is $O(\frac{n}{r} + \frac{K_R}{r})$.

But what is the expected value of K_R? By the way the algorithm proceeds it is clear that R is a random subset of S of size r. Now if $\{s, t\}$ is one of the K pairs of intersecting segments in S, what is the probability that they are both in R? Clearly $\frac{r(r-1)}{n(n-1)}$. It follows that the expected number of pairs of intersecting segments in R, i.e. the expectation K_r of K_R is $\frac{r(r-1)}{n(n-1)} K$.

Thus the expected cost of introducing the last segment into $G_S(R)$ is $O(\frac{n}{r} + \frac{r-1}{n(n-1)})$. To obtain the expected cost for all recursive calls of the entire algorithm one clearly only needs to sum this expression for $1 \leq r \leq n$, which yields

$$O(nH_n + K),$$

where $H_n = 1 + 1/2 + \ldots + 1/n \approx \log n$. Since computing $G_S(\emptyset)$ just amounts to sorting $2n$ numbers it follows that the expected running time of the entire algorithm is $O(K + n \log n)$.

Remarks: As an exercise the reader may want to try this type of analysis on a version of this algorithm that does not use zero-width faces and uses instead of $G_S(R)$ simply a planar graph representation of $\mathcal{T}_S(R)$. Thus the trapezoids can have arbitrarily many edges around them.

In the presentation of Mulmuley's algorithm we assumed non-degeneracy. This is not too much of an issue, except in cases where many segments intersect in one point. In such a situation it would be desirable to obtain an expected running time of $O(I + n \log n)$, where is I is the number of intersection points between segments. (Note that if all segments intersect in one point, then $I = 1$ but $K = \binom{n}{2}$.) By fairly obvious modifications of the algorithm outlined above it is possible to achieve this $O(I + n \log n)$ bound. The only complications arise in the analysis, since I_r, the analogue of K_r, seems to be difficult to express in a nice

closed form. However, the following can be shown and saves the analysis: If X is the set of intersection points between segments of S and if for $p \in X$ the number of segments of S that intersect in p is denoted by $d(p)$, then[1]

$$\sum_{1 \le r \le n} \frac{I_r}{r} = \sum_{p \in X} (H_{d(p)} - 1) \; .$$

But since $\sum_{p \in X} d(p) = O(I + n)$, clearly $\sum_{p \in X} (H_{d(p)} - 1)$ and therefore also $\sum_{1 \le r \le n} I_r / r$ is $O(I + n)$.

Finally we should point out that the algorithm described here does not exploit at all the straightness of the segments in S or the fact that any two segments intersect at most once. With very straightforward modifications the algorithm can be adapted to construct the trapezoidal decomposition induced by a set S of "segments," where each member $s \in S$ is a bounded x-monotone curve (i.e. every vertical line intersects s at most once), where every pair $s, s' \in S$ intersect in a finite number of points, and where for any vertical line ℓ one can determine in $O(1)$ time the "first" intersection point between s and s' to the right of ℓ. No changes in the analysis are necessary at all. It is still $O(I + n \log n)$, where I now stands for the number of intersection points and can be arbitrarily large.

4 Constructing Planar Convex Hulls

The convex hull *conv* S of a set S of n points in the plane is the smallest convex polygon that contains S. Computing such a convex hull amounts to determining the circular sequence of points in S that constitute the corners around the polygon *conv* S.

The planar convex hull problem has received an extraordinary amount of attention in the computational geometry literature. We will not attempt to give a complete history here, but just list three mile stones: in 1972 Graham gave the first $O(n \log n)$ algorithm [29]; in 1978 Bentley and Shamos showed that for a large class of geometric distributions the convex hull of a set of n points drawn according to such a distribution could be computed in $O(n)$ expected time [6] (here the expectation is with respect to the input distribution); in 1983 Kirkpatrick and Seidel gave an algorithm whose worst case running time also depends on the output-size and comes to $O(n \log H)$, where H is the number of corners of the output polygon [35].

The attractiveness of the planar convex hull problem to computational geometers stems partly from the fact that most computational paradigms can be successfully applied to this

[1] The derivation of this formula — at least as done by the author — is not completely straightforward and requires some massaging of sums involving quotients of binomial coefficients. In particular one needs to show that

$$\sum_{1 \le r \le n} \frac{1}{r} \left[1 - \frac{\binom{n-d}{r}}{\binom{n}{r}} - d \frac{\binom{n-d}{r-1}}{\binom{n}{r}} \right] = H_d - 1 \, ,$$

where the quantity in the square brackets is the probability that an intersection point among d segments of S exists in a random sample of r segments.

problem. This is also the case with our paradigm of backwards analysis. In this section we present a randomized algorithm and anlyze its expected running time exploiting this backwards view. I am not sure whom to attribute this algorithm to. It certainly owes a lot to the conflict graph based algorithm of Clarkson and Shor [18], and it seems to have grown out of discussions among several researchers at a DIMACS workshop in the fall of 1989.

For the sake of ease of presentation we will again assume that we are dealing with a set S of n points in non-degenerate position, which in this case is to mean that no three points in S are colinear. Moreover, we will assume that $n > 2$. Our algorithm again works incrementally. It puts the points of S in a random order and then computes the convex hulls of ever increasing subsets of S. We will again describe the algorithm recursively. In this description we will assume the existence of the following kind of oracle: If for $T \subset S$ we know $P_T = conv\, T$, then the oracle can tell for any point $q \in S \setminus T$ whether q is contained in P_T, and if q is not contained, the oracle can tell one edge of P_T that is "visible" from q, i.e. the straight line through that edge separates P_T and q.

Here is the algorithm for computing $P_R = conv\, R$, where $R \subset S$.

> If $|R| = 3$ then P_R is the triangle formed by the three points in R.
>
> If $|R| > 3$ then randomly choose a point q from R, and let $R' = R \setminus \{q\}$.
>
> Recursively compute $P_{R'} = conv\, R'$, and then "insert" point q as follows to obtain $P_R = conv\ R$:
>
> Query the oracle about q and $P_{R'}$. If q is contained in $P_{R'}$, then $P_R = P_{R'}$ and nothing needs to be done.
> Otherwise, the oracle returns an edge e of $P_{R'}$ that is visible form q. Starting at e perform a search to determine all edges of $P_{R'}$ that are visible from q. These edges form a chain. To obtain P_R replace that chain by a new chain of two edges that have q as common endpoint (see Figure 6).

If one is allowed to disregard the costs incurred by the oracle, then by the following amortization argument the running time of this algorithm is easily seen to be $O(n)$. The cost of computing P_R from $P_{R'}$, provided they are different, is clearly proportional to the number of edges of $P_{R'}$ that are found to be visible from q. There can be a large number of such edges. However, as they are all deleted and never re-appear, one can charge this cost to the creation of those edges. But whenever a point of S is added at most two new edges are created, and thus the overall cost for all creations and deletions and hence for the entire algorithm (without the cost of the oracle calls) is $O(n)$.

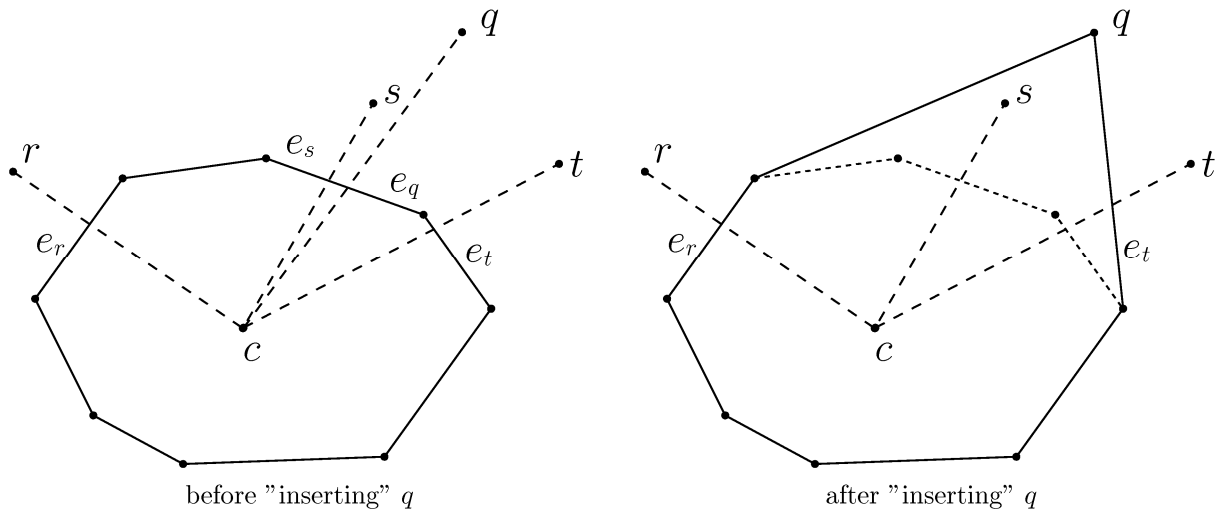


Figure 6

How can one implement the oracle? The idea is to maintain for each point p in $S \setminus R$ a canonical edge e_p of P_R that is visible form p (or to note that no visible edge exists for p). This canonical edge is defined as follows: Let c be a fixed point (not necessarily in S) in the interior of P_R. For each point $p \in S \setminus R$ that lies outside of P_R, its visible "conflict" edge e_p is the unique edge of P_R that is intersected by the straight line segment that connects c and p (uniqueness can be enforced by considering each corner of P_R as part of exactly one or its two incident edges). With this information clearly each oracle call can be answered in constant time. It remains to specify how this information is maintained.

So assume that in the procedure outlined above the recursive call has produced for each point $p \in S \setminus R'$ its canonical conflict edge e_p of $P_{R'}$. If for $p \in S \setminus R$ the conflict edge e_p is not visible from q, then this edge is also an edge of P_R and remains to be the conflict edge for p. (Note that if c is in $P_{R'}$, then it is also in P_R.) If, on the other hand, e_p is visible from q and gets deleted, then the only candidates for being the new conflict edge for p are the two new edges of P_R that are incident to the new point p. Assuming that for each edge e of the current convex hull we maintain the set of all p for which $e_p = e$, it should be clear that the cost of producing the conflict information with respect to P_R given the conflict information for $P_{R'}$ is proportional to the number points in $S \setminus R$ for which the conflict edge changes.

We estimate the expected cost of maintaining the conflict information by considering each point $p \in S$ individually. Since in case of a change p's new conflict edge can be determined in constant time, it suffices to estimate the expected number of changes for p. So what is the probability that for $p \in S$ the conflict edge e_p changes when computing P_R from $P_{R'}$? Backwards analysis suggests that we should express this probability in terms of the "output" P_R. Clearly e_p is new for p iff q is one of the two endpoints of e_p. But since the algorithm chose q to be a random element of R, the probability that q happens to be one of the two endpoints of e_p is $2/r$. Thus the expected number of conflict edge changes for a point $p \in S$

when computing P_R from $P_{R'}$ is at most $2/r$ (note that p could be already in R in which case no change can occur any more). Summing over all $r \leq n$ now yields that the expected total number of conflict edge changes for a point $p \in S$ is at most $2H_n$, which is $O(\log n)$.

Observing that creating the initial conflict information in the "bottoming out" case where $|R| = 3$ takes $O(n)$ time, we can now conclude that the entire maintenance of the conflict information and also the entire algorithm takes expected time $O(n \log n)$.

Remarks: Using the techniques described in the next section it is possible to show that the probability that the running time of the convex hull algorithm presented here exceeds its expected value by a multiplicative factor of c is only $O(n^{-c(\log c - 1)})$.

5 Backwards Analysis of QUICKSORT

QUICKSORT constitutes the archetypical example of a randomized[2] algorithm. Invented by Hoare in 1960 [33], it has since been amply analyzed (see for instance Sedgewick's book [49]) and with its various versions it has become the maybe most frequently used sorting algorithm in practice.

We will consider a somewhat different version of QUICKSORT that is more amenable to backwards analysis than the usual version. However, we also show that both versions have exactly the same running time distribution. Thus the results of our analysis carry over to ordinary QUICKSORT.

Let S be a set of n distinct keys. (The presence of non-distinct keys does not increase the running time of QUICKSORT.) Our algorithm at first puts those keys into a random order $p_1, \ldots, p_n$. For $0 \leq r \leq n$ let $S_r = \{p_i | i \leq r\}$. Our algorithm will make n iterations (or "rounds"), maintaining the following invariant I_r upon completion of each round r:

$$I_r : \begin{cases} \text{The } r \text{ keys in } S_r \text{ have been sorted correctly; say, their order is} \\ \qquad q_0 = -\infty < q_1 < q_2 < \cdots < q_r < \infty = q_{r+1} \ . \\ \text{The remaining keys in } S \setminus S_r \text{ have been partitioned into } r+1 \text{ sets } B_0, B_1, \ldots, B_r, \text{ where} \\ \qquad B_j = \{q \in S \setminus S_r | q_j < q < q_{j+1}\} \ . \end{cases}$$

At the beginning of execution obviously invariant I_0 holds with $B_0 = S$. In the end invariant I_n must hold, which implies that the set $S = S_n$ has been correctly sorted.

What needs to happen in round r, so that, assuming invariant I_{r-1}, one can establish I_r? At the beginning of the round key p_r has to be contained in some B_j. Thus p_r lies between q_j and q_{j+1}, and hence we now know the sorted order of S_r, as desired. To establish the second part of invariant I_r we only need to split the set $B_j \setminus \{p_r\}$ into two subsets comprising the keys smaller than "pivot" p_r and larger than "pivot" p_r, respectively (see Figure 7).

[2]Of course the original version of QUICKSORT was deterministic and it was probabilistically analyzed with respect to an assumed input distribution, namely all permutations of the output occur equally likely as input.

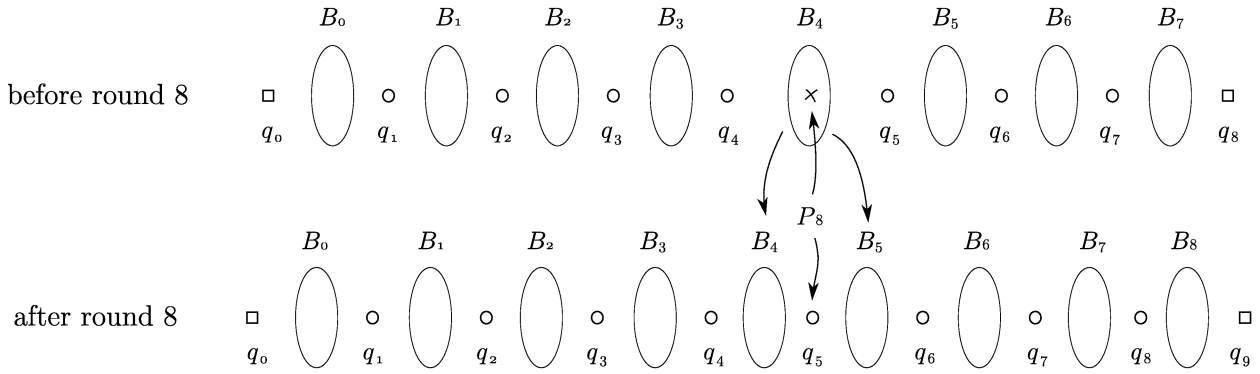


Figure 7

This already completes our description of the algorithm. In this description we have omitted all data structuring aspects. This is justified since we don't intend this algorithm to be implemented but rather to serve as a vehicle for analyzing the number of key comparisons that happen in the usual version of QUICKSORT.

Where do key comparisons happen in this algorithm? They only happen when a set $B_j \setminus \{p_r\}$ is split into two. In that case each element of that set has to be compared with "pivot" p_r. We want to estimate the expected number of such comparisons. (It makes sense to talk about expectations since our algorithm starts by putting the elements of S into random order.)

Let us fix our attention at an arbitrary key $p \in S$ and ask how often p is involved in a comparison where it is not the "pivot." Let us just concentrate on some round r. If $p \in S_r$, then it certainly did not participate in such a comparison. So assume $p \notin S_r$, which means p must be in some B_j. In that case a comparison involving p only happened if B_j is "new," i.e. did not exist in round $r-1$. But B_j can only be new if one of q_j or q_{j+1} was p_r. Backwards analysis now says that any one of $q_1, \ldots, q_r$ has the same probability of being p_r. Thus with probability at most $2/r$ one of q_j or q_{j+1} was p_r (we say "at most" since the fictitious q_0 and q_{r+1} cannot be p_r). This means that the expected number of comparisons involving p in round r is at most $2/r$, and hence the the expected number of comparisons involving p (not as pivot) over all rounds is at most $2H_n$. Considering every key $p \in S$ in turn immediately yields a $2nH_n$ upper bound for the expected number of comparisons made by our algorithm.

Let us try to tighten this analysis. Call a comparison in our algorithm between a key p and a pivot p_r an *L-comparison* if $p_r < p$; call it an *R-comparison* otherwise. Let p be the k-th smallest key in S and let $L_k = \{q \in S | q \leq p\}$. Let us try to estimate the expected number of L-comparisons for p, i.e. comparisons that involve p as non-pivot and some other member of L_k as pivot, in other words, we want to estimate the number of times that a set B_j that contains p is split and p turns out to be larger than the splitting pivot key. For this purpose it suffices to consider only *L-rounds* of our algorithm, i.e. rounds r where the pivot p_r is in L_k. Clearly there are exactly k such L-rounds. Let us number them from 1 to k. Consider now

L-round i, where $1 \leq i \leq k$. What is the probability P_i that p is involved in an L-comparison, i.e. what is the probability that the set B_j that contains p was just split? Well, out of the i pivots chosen from L_k so far, q_i must have been the last one. By backwards analysis the probability for this event is $1/i$. However, for an L-comparison involving p to occur it must also be the case that p was not one of the i pivots chosen so far. The probability for this event is $(1-i/k)$. It follows that $P_i = (1-i/k)\cdot(1/i)$ and thus the expected number L-comparisons in L-round i that involve p is $1/i-1/k$. Summing over all k L-rounds we get that the expected number of L-comparisons involving p is $H_k - 1$. Summing now over all n keys of S we find that the expected number of all L-comparisons is $\sum_{1\leq k\leq n} H_k - n$, which is $(n+1)H_n - 2n$. By symmetry the expected number of R-comparisons is the same and thus the expected total number of key comparisons made by our QUICKSORT algorithm is $2(n+1)H_n - 4n$.

We should now convince the reader that our version of QUICKSORT has the same running time behaviour as the usual version, so that we can claim that our analysis applies to the ususal version also. The usual version sorts a set S of n keys as follows: Choose a $p \in S$ uniformly at random, and compute the sets $S_< = \{q \in S | q < p\}$ and $S_> = \{q \in S | q > p\}$. Output the result of applying the algorithm recursively to $S_<$, followed by p, followed by the the result of applying the algorithm to $S_>$.

With every run of this algorithm we can associate a binary n-node tree T. It is recursively defined as having p as its root whose left child is the tree associated with the sort of $S_<$ and whose right child is the tree associated with the sort of $S_>$. The number of key comparisons made in a particular run can now be expressed as a function $C(T)$ of the associated tree T, namely the sum of the sizes of all n rooted subtrees, minus n. Thus the expected running time of the ordinary version of QUICKSORT can be expressed as the sum over all possible such trees T of the product of $C(T)$ and the probability that T arises.

We can also associate a binary n-node tree T with every run of our version of QUICKSORT. We do this inductively by associating a tree T_r with every round r of our algorithm. T_r has r interior nodes, namely the keys in S_r, and it has $r+1$ leaves, the sets B_j. Tree T_{r+1} is obtained from T_r by replacing the leaf B_j that contains p_{r+1} by a tree whose root is p_{r+1} and whose left and right children are the newly generated B_j and B_{j+1}, respectively. The final tree T is then T_n stripped of all its leaves. Again the number of key comparisons made in a particular run of our algorithm is expressed exactly by $C(T)$. It follows that the two versions of QUICKSORT have the same running time distribution if every tree T is generated by our version with the same probability as by the usual version. But it is easy to check that indeed both versions produce a particular tree T with probability $\prod_{p \text{ node of } T} 1/(\text{size of subtree rooted at } p)$.

That the expected number of comparisons of QUICKSORT is $2(n+1)H_n - 4n$ is a well known result and has been derived before without using backwards analysis (see e.g. [28]). We will now use backwards analysis to estimate in a reasonably painless way the probability that the running time of QUICKSORT is significantly larger than its expectation (see [47]) for a similar result). For this purpose we will artificially slow down our algorithm as follows: If in round r the new pivot p_r is in B_0, then all keys in B_r are compared with p_r also. Similarly, if

p_r is in B_r, then all keys in B_0 are compared with p_r also. Conceptually we are now performing a cyclical sort of S, where after round r there are r sets B_j, for which index arithmetic is done modulo r. This "cyclical" modification of the algorithm removes all "boundary" effects and makes all keys appear symmetrically the same. We will also slow down our algorithm even further. In each round r the pivot p_r will make two extra comparisons (say, with itself).

We will again partition the comparisons made by our slowed down algorithm into L-comparisons and R-comparisons. Again, an L-comparison is a comparison between a key p and a pivot p_r, where $p_r < p$. We will also consider to be an L-comparison one of the two extra comparisons made by every pivot p_r.

For a key $p \in S$ let L_p be a random variable that counts the number of L-comparisons that p is involved in plus the one L-comparison between p and itself when p is the pivot. Define R_p analogously. Let now $X = \sum_{p \in S}(L_p + R_p)$ be a random variable counting the total number of comparisons made by our slowed down algorithm. We are interested in estimating the probability that X exceeds its expectation by a multiplicative factor of c. This can clearly only happen if at least one of the random variables L_p or R_p exceeds its expectation by a factor of c. Thus we have

$$\Pr(X > c \cdot \mathrm{E[X]}) \leq \sum_{\mathrm{p \in S}} \Pr(\mathrm{L_p} > \mathrm{c} \cdot \mathrm{E[L_p]}) + \sum_{\mathrm{p \in S}} \Pr(\mathrm{R_p} > \mathrm{c} \cdot \mathrm{E[R_p]}) \; .$$

Since our setup is completely symmetric all random variables L_p and R_p have the same distribution. Thus for any $p \in S$ we have

$$\Pr(X > c \cdot \mathrm{E[X]}) \leq 2\mathrm{n} \cdot \Pr(\mathrm{L_p} > \mathrm{c} \cdot \mathrm{E[L_p]}) \; . \tag{1}$$

Let us now fix our attention on some key $p \in S$ and let $Y = L_p$. For $1 \leq r \leq n$ let Y_r be a 0-1 random variable counting the contribution of *round* r to Y. Using the ideas of the previous paragraphs and observing the slow down modifications of our algorithm it is easy to see that Y_r is 1 with probability exactly $1/r$. Thus $\mathrm{E[Y]} = \sum_{1 \leq \mathrm{r} \leq \mathrm{n}} \mathrm{E[Y_r]} = \sum_{1 \leq \mathrm{r} \leq \mathrm{n}} 1/\mathrm{r} = \mathrm{H_n}$ and therefore E[X] the expected number of comparisons in our slowed down algorithm is $2nH_n$.

To estimate $\Pr(Y > c \cdot \mathrm{E[Y]})$ we can now use the well-known Chernoff bound (see [47, 31]), which in one form states that if a random variable Z is the sum of n independent 0-1 random variables and the expectation of Z is E, then for $c \geq 1$

$$\Pr(Z > c \cdot E) \leq e^{-E(1-c+c\log c)} \; .$$

In our case the Y_i's can easily be proven to be independent. Thus we obtain

$$\Pr(Y > c \cdot \mathrm{E[Y]}) \leq \mathrm{e}^{-\mathrm{H_n}(1-\mathrm{c}+\mathrm{c}\log \mathrm{c})} = \mathrm{O}(\mathrm{n}^{-(1+\mathrm{c}(\log \mathrm{c}-1))}) \; .$$

From this and inequality 1 we can now conclude that the probability that the running time of our modified algorithm exceeds its expectation by a multiplicative factor of $c \geq 1$ is $O(n^{-c(\log c - 1)})$. Since this algorithm always performs more comparisons than ordinary QUICKSORT we can conclude that the probability that QUICKSORT makes more than $2cnH_n$ comparisons is also $O(n^{-c(\log c - 1)})$.

6 A Bad Example

Consider the following higher-dimensional triangulation problem: Suppose a set S of n points in $\mathbb{R}^d$ is contained in the interior of a d-simplex D with vertex set $Q = \{q_0, q_1, \ldots, q_d\}$. We are to *triangulate* S, i.e. construct a collection of simplices $\mathcal{T}$, so that $\bigcup\{\Delta \in \mathcal{T}\} = D$, each simplex $\Delta \in \mathcal{T}$ has its vertex set in $S \cup Q$, every $p \in S$ is vertex of some simplex in $\mathcal{T}$, and every two simplices in $\mathcal{T}$ intersect in a common face (which can be the empty set).

Deterministic algorithms for solving this problem in $O(n \log n)$ time have been presented in [4, 26]. Here we consider the following randomized incremental algorithm that was inspired by the QUICKSORT algorithm of the previous section. For the purpose of illustration we will assume that the points in $S \cup Q$ are in non-degenerate position, i.e. no $d+1$ points lie in a common hyperplane.

Our algorithm will at first put the points in S in some random order $p_1, \ldots, p_n$. For $0 \leq r \leq n$ let now $S_r = \{p_i | i \leq r\}$. The algorithm works in n rounds. Upon completion of round r a triangulation $\mathcal{T}_r$ will have been computed, and for each simplex $\Delta \in \mathcal{T}_r$ the set B_Δ will contain the points in $S \setminus S_r$ that lie in the interior of Δ. Initially we have $\mathcal{T}_0 = \{D\}$ and $B_D = S$.

Assuming inductively that the algorithm has correctly completed round $r-1$, our algorithm only needs to do the following in round r: Let Δ be the simplex in $\mathcal{T}_{r-1}$ that has p_r in its interior and let B_Δ be the corresponding subset of $S \setminus S_{r-1}$. The simplex Δ is split into $d+1$ simplices $\Delta_0, \ldots, \Delta_d$, each being a pyramid with a facet of Δ as base and with p_r as apex. The set $B_\Delta \setminus \{p_r\}$ is split accordingly.

In any reasonable implentation the running time of this algorithm will be proportional to the number of times that during the execution the containing simplex changes for a point $p \in S$. It is now tempting to apply backwards analysis to obtain the expectation of this quantity. Fixing attention at a point $p \in S$ and at some round r it seems that the probability that the containing simplex of p changed in round r is at most $(d+1)/r$. After all, either p is contained in S_r, in which case no change occurs, or otherwise p must be in some simplex $\Delta \in \mathcal{T}_r$, which is "new" iff one of its $d+1$ vertices happened to be p_r. By backwards analysis this happens with probability at most $(d+1)/r$. Thus the expected number of containing simplex changes for p in the entire algorithm is at most $(d+1)H_n$, and hence the expected running time of the algorithm is $O((d+1)nH_n)$.

However, this argument is quite wrong. The fallacy lies in the fact that the triangulation $\mathcal{T}_r$ of S_r is not canonical, i.e. it depends on the ordering of the points in S_r. In particular this means that the containing simplex of p is not canonical. Thus it is difficult to argue that p_r is a vertex of that simplex with probability at most $(d+1)/r$.

For dimension $d = 1$ this algorithm indeed does have $O(nH_n)$ expected running time, since any 1-dimensional point set admits exactly one triangulation, and hence $\mathcal{T}_r$ is always canonical. But of course for $d = 1$ the triangulation algorithm of this section is nothing but QUICKSORT. It remains to be seen whether for fixed $d > 1$ the expected running time of this triangulation algorithm is indeed $O(n \log n)$.

7 Linear Programming for Small Dimension

General linear programming has a long history. In this paper we are only interested in the case where the dimension (or number of variables) d is a small constant and m, the number of halfspaces (or constraints), can be quite large. In the last decade deterministic algorithms were developed that solve such linear programs in $O(m)$ worst case time [39, 40, 22, 23, 19]. However those algorithms are mostly of theoretical interest only since they are quite complicated and the dependence of their running times on the dimension d is exponential and has only been shown to be 3^{d^2} at best. More recently Ken Clarkson [20] proposed a randomized algorithm with a remarkable running time of $O(d^2m) + (\log m)O(d)^{d/2+O(1)} + O(d^4\sqrt{m}\log m)$. Here we briefly present another randomized algorithm that was first described in [50]. The main virtues of this algorithm are its simplicity and its amenability to backwards analysis (well — for the purpose of this paper this is a virtue).

Geometrically, linear programming amounts to the following: One is given a set $\mathcal{H}$ of m halfspaces and a vector a in $\mathbb{R}^d$, and one wants to find a vertex v of the polyhedron P formed by the intersection of the halfspaces, so that v maximizes the linear functional specified by a; in other words, v must be contained in the tangent hyperplane of P whose outward normal is a.

The reader might wonder about our specification of linear programming. The required optimum vertex v of P might not exist, either because P is empty or because P is unbounded. Thus we amend our specification. In case of emptiness of P we require this fact to be reported by the algorithm. For the sake of ease of presentation we will ignore for the time being the unboundedness situation and assume that our problem and all subproblems to be encoutered are very well behaved in the sense that if a problem is feasible a unique optimum vertex exists and that this vertex is the intersection of the bounding hyperplanes of exactly d of the given halfspaces. We will show later how those assumptions can be removed.

Here is our algorithm for solving such a linear program given by a set $\mathcal{H}$ of $m \geq d$ halfspaces and a direction a in $\mathbb{R}^d$.

> If $d = 1$, then the problem amounts to finding the smallest (or largest) real number satisfying m inequalities. With m comparisons this number can easily be found or it can be established that a number satisfying all inequalities does not exist.
>
> Now assume $d > 1$. If $|\mathcal{H}| = d$, then by our assumptions the solution is the intersection of the d hyperplanes that bound the halfspaces in $\mathcal{H}$. Thus this optimum vertex can be found in $O(d^3)$ time.
>
> So assume that $|\mathcal{H}| > d$. Choose a halfspace $H \in \mathcal{H}$ uniformly at random. Recursively solve the d-dimensional linear program given by the $m - 1$ halfspaces $\mathcal{H} \setminus \{H\}$ and direction a. This yields an optimum point w. (If such an optimum does not exist, the original problem does not have a solution.)
>
> Now, if w is contained in H, then clearly w is also the solution for the original problem and we are done. If w is not contained in H, then the optimum vertex v

for $\mathcal{H}$, if it exists at all, must be contained in the hyperplane h that bounds H. As a matter of fact v must be the solution of the $(d-1)$-dimensional linear programming problem given by the $m-1$ constraints $\mathcal{H}' = \{I \cap h \mid I \in \mathcal{H} \setminus \{H\}\}$ and direction a', the orthogonal projection of a into h. The solution of that problem is now found recursively.

Let us now analyze the expected running time of this algorithm. The most important issue seems to be to estimate the probability that the $(d-1)$-dimensional problem for $\mathcal{H}'$ needs to be solved. This is exactly where backwards analysis comes into play. Note that this recursive call is necessary iff the optimum vertex v for $\mathcal{H}$ is different from the optimum vertex w for $\mathcal{H} \setminus \{H\}$. But such a difference can only occur if the bounding hyperplane of H is one of the d hyperplanes that define v. But since H is chosen from the m hyperplanes in $\mathcal{H}$ at random this happens with probability d/m.

Now let $T(d,m)$ be the expected running time of our algorithm for solving a linear program with $m \geq d$ halfspaces in $\mathbb{R}^d$. Assuming that testing whether a point v lies in a halfspace H takes $O(d)$ time and that computing the intersection of a halfspace with a hyperplane takes also $O(d)$ time, $T(d,m)$ is defined recursively as follows:

$$T(d,m) = \begin{cases} O(m) & \text{if } d = 1 \\ O(d^3) & \text{if } m = d \\ T(d,m-1) + O(d) + \frac{d}{m}O(dm) + \frac{d}{m}T(d-1,m-1) & \text{otherwise} \end{cases}$$

It is now easy to check that $T(d,m) = O\left(\sum_{1 \leq i \leq d} \frac{i}{(i-1)!} d!m\right)$, which is $O(d!m)$ since the sum converges even without an upper bound for i.

We still have to deal with the various assumptions we made initially. Uniqueness of the optimum vertex can be achieved by standard perturbation techniques, or by requiring the algorithm always to return the optimum vertex that has the lexicographically smallest coordinate representation. Note that in light of the previous section it is crucial for the analysis of our algorithm that the optimum vertex is defined uniquely and canonically.

The assumption that an optimum vertex must always be the intersection of the bounding hyperplanes of exactly d halfspaces can be dropped altogether. Involvement of more than d halfspaces in v makes it less likely that v is different from w and hence cannot increase the running time of our algorithm.

The boundedness assumption appears to be the most difficult to remove. One approach is to enforce it simply by stipulating that we are not interested in all of $\mathbb{R}^d$ but just some "bounding box" B (i.e. we impose explicit lower and upper bounds for the variables; see [50] for details). Another approach involves generalizing the notion of "optimum vertex:" in case of unboundedness define the "optimum" to be the unit vector in the direction of a ray in the feasibility region that maximizes the inner product with the objective direction a. Both approaches require slight changes in the "bottoming-out" part of our procedure. See the next section for an abstract description.

8 Welzl's Minidisk Algorithm

Here we present an algorithm due to Emo Welzl [56] for constructing the smallest enclosing ball for a finite point set $T \subset \mathbb{R}^d$. The algorithm is very similar to the linear programming algorithm of the previous section. However, the idea of recursively solving a problem of *smaller dimension* has to be viewed now as recursively solving a problem with *more equality constraints*.

Below we describe a function `minidisk`(T, C), which takes as input two disjoint finite sets $T, C \subset \mathbb{R}^d$ and which is to return the ball of smallest radius that contains T and has all points of C on its boundary. Of course, for arbitrary sets T and C such a ball need not exist. However, we will assume that the function will only be called with parameters T and C for which the existence of such a ball is guaranteed. In particular note that `minidisk` $(T, \emptyset)$ simply computes the smallest enclosing ball of T, and such a ball of course always exists (for the case $T = \emptyset$ we consider the empty set to be a degenerate ball).

Note that the smallest enclosing ball of a set T will always be determined by at most $d+1$ points of T. For our implementation of `minidisk()` we will assume the existence of a function `primitive_ball` (D), which given any set of D of at most $d+1$ points returns in time $O(d^3)$ the smallest ball that has all of D on its boundary (again assuming the existence of such a ball).

```
minidisk(T, C)
        if T = ∅ then return primitive_ball(C);
        choose some p from T uniformly at random and let T' := T \ {p};
        B' := minidisk(T', C);
        if p ∈ B' then return B'
                  else return minidisk(T', C ∪ {p});
```

The correctness of this function follows form the following two lemmas:

Lemma 8.1 Let T and C be two sets in $\mathbb{R}^d$. The smallest ball that contains T and has all points of C on its boundary is unique (provided it exists).

Proof. Assume B_1 and B_2 are two distinct smallest enclosing balls for T that have C on their boundary. Let c_1 and c_2 be the two centers and let R be the common radius. Then it is easy to check that the smaller ball with center $(c_1+c_2)/2$ and radius r given by $r^2 = R^2 - \langle c_1 - c_2, c_1 - c_2 \rangle$ also contains T and has all points of C on its boundary. □

Lemma 8.2 Let T' and C be two sets in $\mathbb{R}^d$ so that the smallest enclosing ball B' for T' that has all points of C on its boundary exists.

If some point $p \in \mathbb{R}^d$ is not contained in B', then the smallest enclosing ball B for $T \cup \{p\}$ that has all points of C on its boundary also has p on its boundary (provided it exists).

Proof. Assume B does not have p on its boundary, i.e. p lies in the interior of B. But in this case B must also be the smallest enclosing ball for T' hat has C on its boundary. By the previous lemma that ball is unique, i.e. $B = B'$, which would mean $p \in B$ and $p \notin B$, a contradiction. □

For the analysis of the expected running time of `minidisk`(T, C) the most important step is to estimate the probability that a point p chosen randomly from T is not contained in the ball B'. Let B be the smallest enclosing ball of T that has all points of C on its boundary. Lemma 8.2 tells us that $p \notin B'$ implies that p lies on the boundary of B and that $B \neq B'$. In the non-degenerate case where no $d+1$ points of $T \cup C$ are co-spherical it is clear that there are at most $d + 1 - |C|$ choices for p that render B and B' different. It is not too hard to see that this number cannot be larger if there are degeneracies. Thus the probability that a random point p of T is not contained in the ball B' is at most $\delta/|T|$, where $\delta = d + 1 - |C|$.

Now consider a call `minidisk`(T, C), where $T, C \subset \mathbb{R}^d$, $|T| = n$, and $|C| = d + 1 - \delta$. Let $f(n, \delta)$ denote the expected number of recursive invocations `minidisk`(S, D) for which $S \neq \emptyset$, and let $g(n, \delta)$ denote the expected number of recursive invocations for which $S = \emptyset$. Thus f and g satisfy the following recursive relationships:

$$f(n,\delta) \leq \begin{cases} 0 & \text{if } n = 0 \\ 1 + f(n-1,\delta) + \frac{\delta}{n} f(n-1,\delta-1) & \text{otherwise} \end{cases}$$

$$g(n,\delta) \leq \begin{cases} 1 & \text{if } n = 0 \\ g(n-1,\delta) + \frac{\delta}{n} g(n-1,\delta-1) & \text{otherwise} \end{cases}$$

It is an easy inductive exercise to check that now $f(n, \delta) \leq \sum_{1 \leq i \leq \delta} \frac{1}{i!} \delta! n$, which is $O(\delta! n)$, and that $g(n, \delta) \leq \delta!(1 + H_n)^\delta$, which is $O(\delta! \log^\delta n)$. If we assume that an invocation with $S \neq \emptyset$ takes time $O(d)$ and an invocation with $S = \emptyset$ (i.e. a call to `primitive_ball`) takes time $O(d^3)$ we get the following:

Theorem 8.3 For a set T of n points $\mathbb{R}^d$ a call to `minidisk`$(T, \emptyset)$ computes the smallest enclosing ball of T in expected time $O(d(d+1)!n)$.

The type of algorithm presented in this and the previous section can be successfully applied also to other problems, such as computing the smallest enclosing ellipsoid of a point set in $\mathbb{R}^d$, or computing the largest inscribed sphere of a convex polyhedron. It is possible to unify these algorithms by considering the relevant problems as special instances of a suitably axiomatized abstract optimization problem.[3]

Very recently Micha Sharir and Emo Welzl [54] proposed a new axiomatic framework along with a new randomized algorithm for the linear programming type of problems considered here

[3]Previous versions of this paper contained an attempt of such an axiomization. However, that framework turned out to be too weak, and the algorithmic results claimed in those previous versions are fallacious.

and in the previous section. Their method avoids the recursion on the dimension and in the case of linear programming achieves an expected time bound of $O(2^d n)$. The complexity analysis of their new algorithm also exploits aspects of "backwards" analysis. Even more recently the same two authors together with Jiří Matoušek [38] managed to improve the analysis of that algorithm to the remarkable expected time bound of $O((nd+d^3)e^{4\sqrt{d\ln(n+1)}})$.

9 Clarkson's Backwards Analysis of the Conflict Graph Based Convex Hull Algorithm

In their landmark paper on applications of random sampling in computational geometry [18] Clarkson and Shor described a randomized algorithm for constructing convex hulls of point sets in $\mathbb{R}^d$ that has optimal expected running time. Their analysis of the expected running time was based on very general lemmas about random sampling. Recently Ken Clarkson [21] has discovered a new analysis that is completely self-contained and relies heavily on the idea of backwards analysis. In this section we give first a brief description of the convex hull algorithm[4] and then its new analysis.

We want to construct the convex hull of a set S of n points in $\mathbb{R}^d$, with $n > d > 1$. We will assume that S is in non-degenerate position, i.e. no $d+1$ of its points lie in a common hyperplane. Such non-degeneracy can easily be simulated with impunity using standard perturbation techniques [25, pp. 185]. Non-degeneracy ensures that the convex hull of any subset of S is a simplicial polytope.

A few relevant basics about polytopes: Let P be a simplicial d-polytope, let V be the vertex set of P, and let $m = |V|$. It is known that P can have at most $O(m^{\lfloor d/2 \rfloor})$ faces. We call the $(d-1)$-faces of P *facets* and the $(d-2)$-faces *ridges*. Every facet is uniquely identified by the d-tuple of its vertices. Similarly every ridge can be identified by a $(d-1)$-tuple of vertices in V. Since every ridge is contained in precisely two facets one can represent the facial structure of P by its *facet graph* $\mathcal{G}(P)$, which has the facets of P as its nodes and two facets adjacent iff they share a common ridge of P. Note that for simplicial d-polytopes the facet graph is regular of degree d.

Let p be some point in $\mathbb{R}^d$ in non-degenerate position with respect to V. We call a facet F of P *visible* from p iff the hyperplane spanned by F separates P and p. We call F *obscured* otherwise. We call a face G of P visible from p iff it is only contained in facets of P that are visible from p. Obscured faces are defined analogously. We call G a horizon face with respect to x iff it is contained in some visible and some obscured facet.

This terminology allows a convenient characterization of the facial structure of the polytope $P' = conv\,(P \cup \{x\})$ in terms of the faces of P: No visible face of P is a face of P'; all obscured and all horizon faces of P are faces of P'; for each horizon face G of P the pyramid

[4] We actually present a slightly different version than the one in [18] in that we do not dualize and use a slightly different notion of a conflict graph.

conv $(G \cup \{x\})$ is a face of P'; this yields all faces of P'.

This characterization justifies the following method for obtaining P' from P and x. We assume here that the polytopes are represented by their facet graphs. Thus, to be more precise, the procedure outlined below is intended to compute the facet graph $\mathcal{G}(P')$ from x and the facet graph $\mathcal{G}(P)$, and when we talk about facets or ridges we are simultaneoulsy referring to nodes and arcs of a facet graph (which are identified by d-tuples and $(d-1)$-tuples, respectively, of the points in $S \cup \{x\}$).

(i) Determine the set $Vis(x, P)$ of all facets of P that are visible from x. (In case no visible facets exist at all, x must be contained in P already, i.e. $\mathcal{G}(P') = \mathcal{G}(P)$, and nothing further needs to be done.)

(ii) Partition the ridges contained in facets of $Vis(x, P)$ into the set of visible and the set of horizon ridges of P with respect to x, by checking for each ridge whether both containing facets are in $Vis(x, P)$. Delete all visible facets and ridges.

(iii) For each horizon ridge G of P generate the new facet *conv* $(G \cup \{x\})$ of P' (i.e. a new node for the facet graph).

(iv) Generate the new ridges of P' (i.e. the edges between the new nodes of the facet graph).

Step (i) of this algorithm is still rather vaguely specified. We defer the details of how the visibility set $Vis(x, P)$ can actually be obtained. Let us first analyze the cost of this algorithm, but ignoring the cost incurred by step (i).

The cost of step (ii) is clearly proportional to the number of facets in $Vis(x, P)$. But since all those facets are deleted, and every facet can be deleted at most once, we can charge the deletion cost of each facet to its creation, and thus in the amortized sense, step (ii) incurs no cost at all.

Step (iii) has cost proportional to the number of new facets created. These are exactly the facets of P' that contain x. Let us denote their number by $deg(x, P')$.

The number of new ridges created in step (iv) is proportional to the number of new facets, to be precise, their number is $(d-1)deg(x, P')/2$. How can they be found? For every new facet generated in step (iii) the $d-1$ new ridges contained by it can be determined "locally." Radix sorting the $(d-1)$-tuples of vertices (or rather vertex indices) that identify these ridges then allows to match them up and to form the new edges of the facet graph $\mathcal{G}(P')$ in time proportional to $n + deg(x, P')$. When $d < 4$ the radix sort can be avoided: In the case $d = 2$ there are only two new facets and one new ridge, namely x. In the case $d = 3$ one can exploit the planar graph nature of the facet graphs to find the new ridges in time proportional to their number. We omit here the details of how to do this.

We conclude that, ignoring step (i), the total cost of this insertion algorithm is proportional to $deg(x, P')$ in case $d = 2, 3$ and proportional to $n + deg(x, P')$ for $d > 3$.

Consider now the following algorithm for constructing the convex hull of a set S of $n > d$ points in $\mathbb{R}^d$ in non-degenerate position.

1. Put the points of S in a random order $p_1, \ldots, p_n$. For $1 \leq r \leq n$ let S_r denote $\{p_1, \ldots, p_r\}$, and let P_r denote $conv\, S_r$.

2. Form the facet graph $\mathcal{G}(P_{d+1})$. (Note that this graph is simply the complete graph on $d+1$ vertices.)

3. For $d+1 < r \leq n$, using the insertion procedure outlined above, form the facet graph $\mathcal{G}(P_r)$ from $\mathcal{G}(P_{r-1})$.

We want to determine the expected running time of this algorithm. Obviously the most important question is to determine the expected cost of step 3. We know that, ignoring step (i) of the insertion algorithm, the expected cost of the insertion in iteration r is determined by the expectation of $deg(p_r, P_r)$. Now apply backwards analysis. With probability $1/r$ point p_r was the last one in the random permutation of S_r. Thus the expected value of $deg(p_r, P_r)$ is $(1/r) \cdot \sum_{p \in S_r} deg(p, S_r)$, which, since every facet contains exactly d vertices, is $(d/r) \cdot F(P_r)$, where $F(P_r)$ denotes the number of facets of P_r. But P_r has at most r vertices. Thus by the upper bound theorem for polytopes $F(P_r) = O(r^{\lfloor d/2 \rfloor})$, and the expected value of $deg(p_r, P_r)$ is therefore $O(r^{\lfloor d/2 \rfloor - 1})$. We conclude that for $d > 3$ the expected running time of the entire algorithm is $\sum_{d+1<r\leq n} O(r + r^{\lfloor d/2 \rfloor - 1})$, which is $O(n^{\lfloor d/2 \rfloor})$. For $d = 2, 3$ we get that the expected running time of the algorithm is $\sum_{d+1<r\leq n} O(1)$, which is $O(n)$.

But recall that this analysis does not take into account the cost incurred by step (i) of the insertion algorithm. Let us now turn to the details of how that step can be implemented. How can one determine the set $Vis(p_r, P_{r-1})$ of all facets of P_{r-1} that are visible from p_r?

As pointed out in [50] there is a simple solution for this problem that turns out to be reasonably efficient for the case $d > 3$. Since the facets in $Vis(p_r, P_{r-1})$ induce a connected subgraph in the facet graph $\mathcal{G}(P_{r-1})$ it suffices to find just one visible facet. The remaining ones can then be determined by a depth-first search in time proportional to their number. All the visible facets found will be deleted, never to reappear again, and thus we can charge their discovery cost to their creation. In other words, in the amortized sense this depth-first search incurs no cost at all, and we only have to worry about the time necessary to discover one facet of $Vis(p_r, P_{r-1})$. However, this problem is nothing but a linear programming problem with r constraints and in d variables and can thus, as we saw in Section 7, be solved in $O(r)$ expected time. Summing over all n insertions this yields an overall expected cost of $O(n^2)$, which for $d > 3$ is subsumed by the $O(n^{\lfloor d/2 \rfloor})$ expected running time of the remaining parts of the algorithm.

A solution to this "visibility problem" that performs satisfactorily in all dimensions, and not just for $d > 3$, was invented by Clarkson and Shor [18]. In essence, they proposed to maintain at each iteration r of the algorithm the complete visibility set[5] $Vis(p, P_r)$ for each

[5]Clarkson and Shor have the notion of a "conflict graph," which in our case would be a bipartite graph

point $p \in S \setminus S_r$. We will here describe a variant of this approach that was also already considered in [18], where for each point $p \in S \setminus S_r$ only one representative visible facet $VF(p, P_r) \in Vis(p, P_r)$ is maintained (provided such a facet exists at all).

Initially some $VF(p, P_{d+1})$ can be computed for all $p \in S \setminus S_{d+1}$, in $O(n)$ time. For $r > d+1$ how can one compute $VF(p, P_r)$ from $VF(p, P_{r-1})$? For some point $p \in S \setminus S_r$ let $F = VF(p, P_{r-1})$. If F is undefined (because p is contained in P_{r-1}), then also $VF(p, P_r)$ is undefined. If the facet F is also a facet of P_r, then one can choose $VF(p, P_r) = F$. We only have to actually do something if the facet F of P_{r-1} is not a facet of P_r any more, i.e. $F \in Vis(p_r, P_{r-1})$ and is thus one of the facets that gets deleted in step (ii) of the insertion algorithm. In order to discover all $p \in S \setminus S_r$ for which we actually have to do something we need to maintain for each facet the set of points p for which the facet is the representative visible facet.

To find the replacement p-visible facet of P_r we now start at F a depth-first search in the facet graph $\mathcal{G}(P_{r-1})$ to discover all facets in $Vis(p_r, P_{r-1}) \cap Vis(p, P_{r-1})$ in time proportional to their number. Let D be the set of horizon ridges (with respect to p_r) contained in those facets. For each ridge $G \in D$ now check if its containing facet of P_{r-1} that is not in $Vis(p_r, P_{r-1})$ (which is therefore a facet of P_r) is visible from p, and check if the "new" facet $conv(G \cup \{p_r\})$ of P_r is visible from p. If no p-visible facet is found this way, then we make $VF(p, P_r)$ undefined, otherwise we set $VF(p, P_r)$ to one of those p-visible facets. The correctness of this approach is a consequence of the fact that for a polytope P and any set $X \subset \mathbb{R}^d \setminus P$ the facets of P that are visible from all points in X induce a connected subgraph of the facet graph $\mathcal{G}(P)$.

The cost of finding the new representative p-visible facet on P_r is thus proportional to the size of $Vis(p_r, P_{r-1}) \cap Vis(p, P_{r-1})$, or in other words, the number of visibilities between facets and p that cease to exist with the insertion of p_r. Therefore, in order to estimate the cost of maintaining representative visible facets for all p over the entire algorithm, we need to determine the expected number of visibilities between facets and points that cease to exist in the course of the algorithm. Obviously this is the same as the expected number of visibilities that come into existence. We will now estimate the latter quantity.

Let $R = \{p_1, \ldots, p_r\}$. Since the p_i's are in a random order R is now a random subset of S of size r. What is the expected number of visibilites between facets of $conv\ R$ and points in $S \setminus R$ that came into existence when the last point of R was inserted? Let's do it backwards! Which visibilities would disappear if a random point q of R was removed? Exactly those that involved a facet that contained q. Since every facet is determined by exactly d points of R the probability that any particular facet contains q is d/r. It follows that the expected number of visibilities that disappear when a random point of R is removed (or visibilites created when q is inserted) is

$$\frac{d}{r} \sum_{q \in S \setminus R} |Vis(q, conv\ R)| \ .$$

whose nodes are the facets of P_r and the points in $S \setminus S_r$ and that has an arc joining facet F with point q iff F is visible from q.

Let $vis(q,R)$ denote $|Vis(q, conv\, R)|$. For $A \subset S$ let now $f(A)$ denote the number of facets of $conv\, A$, and for $a \in A$ let $deg(a,A)$ now denote the number of facets of $conv\, A$ that contain a. Here comes an ingenious observation due to Ken Clarkson [21]:

$$vis(q,R) = f(R) - f(R \cup \{q\}) + deg(q, R \cup \{q\}) \;, \tag{2}$$

since the facets of $conv\, R$ that are not visible from q are exactly the facets of $conv\,(R \cup \{q\})$ that do not contain q.

It follows that C_r, the expected number of visibilities created when the r-th point of S is inserted, is

$$C_r = \frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} \frac{d}{r} \sum_{q \in S \setminus R} vis(q,R) = \frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} \frac{d}{r} \sum_{q \in S \setminus R} \Big(f(R) - f(R \cup \{q\}) + deg(q, R \cup \{q\}) \Big) \;.$$

Now let $f_r = \frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} f(R)$ denote the expected value of $f(R)$. Note that f_r actually also depends on the set S. We will estimate each of the three main summands of the sum above separately.

$$\frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} \frac{d}{r} \sum_{q \in S \setminus R} f(R) = \frac{d}{r}(n-r) f_r$$

$$\begin{aligned}
\frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} \frac{d}{r} \sum_{q \in S \setminus R} f(R \cup \{q\}) &= \frac{1}{\binom{n}{r}} \sum_{\substack{R' \subset S \\ |R'|=r+1}} \frac{d}{r} \sum_{q \in R'} f(R') \\
&= \frac{1}{\binom{n}{r+1}} \sum_{\substack{R' \subset S \\ |R'|=r+1}} \frac{\binom{n}{r+1}}{\binom{n}{r}} \frac{d}{r}(r+1) f(R') \\
&= \frac{1}{\binom{n}{r+1}} \sum_{\substack{R' \subset S \\ |R'|=r+1}} \frac{d}{r}(n-r) f(R') = \frac{d}{r}(n-r) f_{r+1} \\
&= \frac{d}{r+1}(n-(r+1)) f_{r+1} + \frac{dn}{r(r+1)} f_{r+1}
\end{aligned}$$

$$\begin{aligned}
\frac{1}{\binom{n}{r}} \sum_{\substack{R \subset S \\ |R|=r}} \frac{d}{r} \sum_{q \in S \setminus R} deg(q, R \cup \{q\}) &= \frac{1}{\binom{n}{r+1}} \sum_{\substack{R' \subset S \\ |R'|=r+1}} \frac{\binom{n}{r+1}}{\binom{n}{r}} \frac{d}{r} \sum_{q \in R'} deg(q, R') \\
&= \frac{1}{\binom{n}{r+1}} \sum_{\substack{R' \subset S \\ |R'|=r+1}} \frac{n-r}{r+1} \cdot \frac{d}{r}(d \cdot f(R')) \\
&= \frac{d^2}{r(r+1)}(n-r) f_{r+1}
\end{aligned}$$

Thus the total expected number of visibilities created in the entire course of the algorithm is

$$\sum_{d+1\le r<n} C_r = \sum_{d+1\le r<n} \left(\frac{d}{r}(n-r)f_r - \frac{d}{r+1}(n-(r+1))f_{r+1} - \frac{dn}{r(r+1)}f_{r+1} + \frac{d^2}{r(r+1)}(n-r)f_{r+1} \right) .$$

This is a telescoping sum, and therefore we get

$$\sum_{d+1\le r<n} C_r = \frac{d}{d+1}(n-d-1)f_{d+1} + d(d-1)n \sum_{d+1\le r<n} \frac{f_r}{r(r+1)} - d^2 \sum_{d+1\le r<n} \frac{f_{r+1}}{r+1} .$$

But since any d-polytope with r vertices has $O(r^{\lfloor d/2 \rfloor})$ facets it is certainly the case that $f_r = O(r^{\lfloor d/2 \rfloor})$. Thus it is easy to see that for $d > 3$ the expected total number of visibilities created is $O(n^{\lfloor d/2 \rfloor})$, whereas for $d = 2, 3$ this number is $O(nH_n)$, which is $O(n \log n)$.

We can therefore conclude that the randomized incremental algorithm for constructing the convex hull of n points in $\mathbb{R}^d$ has an expected running time of $O(n \log n)$ for $d = 2, 3$ and $O(n^{\lfloor d/2 \rfloor})$ for $d > 3$, which was the best we could hope for. Note that this analysis gives amazingly tight bounds for the expected number of visibilities for the case $d = 2, 3$.

10 Odds and Ends

It should be pointed out that the type of analysis presented in the previous section is not particular to the convex hull problem but can be applied to randomized incremental construction in the formal framework of Clarkson and Shor [18]. In their terminology the generalization of the crucial insight (2) is the observation that the regions defined by R that do not conflict with object q are exactly those regions defined by $R \cup \{q\}$ that do not involve q. Note that most of the problems and algorithms presented in this paper actually fall into Clarkson and Shor's framework — maybe this paper should have been made much shorter.

There are a number of problems and algorithms that should have been included in this survey, but were not because of time constraints. Maybe the most serious gap concerns geometric searching, in particular planar subdivsion searching. A search structure is constructed using a randomized algorithm; query times are then random variables with respect to the "coin flips" made during the construction. Backwards analysis works very well for determining the expectation of those query times; see for instance [51]. In that paper a little twist is also added to this approach, which yields a rather straightforward randomized method for triangulating a simple polygon in nearly optimal $O(n \log^* n)$ expected time.

In section 5 we gave some analysis of how tightly the running time of QUICKSORT is concentrated around its expectation. What about tail estimates for the running times of the other algorithms presented in this paper? What is the probability that the actual running time on a problem of size n exceeds the expectation by a multiplicative factor of c? For the polygon triangulation algorithm of secion 2 one can use a general result of Mehlhorn [41] to show that this probability is at most $\frac{1}{e}(\frac{e}{c})^c$. For Mulmuley's algorithm of section 3 Matoušek and Seidel

[37] have recently shown a tail estimate of $O(n^{-c})$, provided K, the number of intersecting segment pairs, is not too small relative to n. For the linear programming algorithm of section 7 a bound of $O(c^{-d!})$ is given in [50], where d is the dimension. A similar bound applies to the algorithm of section 8. To my knowledge no non-trivial tail estimate is known for the convex hull algorithm of section 9.

11 Acknowledgements

Many people have wittingly and unwittingly contributed to this paper in one form or another. I would like to give credit to all the participants of the various DIMACS workshops on computational geometry in 1989/90. In particular I would like to thank Ken Clarkson, Emo Welzl, Günter Rote, Peter Shor, Kurt Mehlhorn, Ricky Pollack, Leo Guibas, Micha Sharir, Herbert Edelsbrunner, Ketan Mulmuley, and Otfried Schwarzkopf. Finally, I am grateful to János Pach for his seemingly infinite patience.

Thanks to Laszlo Kozma for seeing to it that finally, in 2011, this version of the paper actually contains figures.

References

[1] P.K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. *"Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs."* Proc. 6th ACM Symp. on Computational Geometry (1990), pp 203–210.

[2] A. Aggarwal, L.J. Guibas, J. Saxe, and P.W. Shor. *"A Linear Time Algorithm for Computing the Voronoi Diagram of a Convex Polygon."* Proc. 19th ACM Symp. on Theory of Computing (1987) pp 39–47.

[3] F. Aurenhammer. *"Voronoi Diagrams — A Survey."* To appear in ACM Computing Surveys.

[4] D. Avis and H. ElGindy. *"Triangulating Simplicial Point Sets in Space."* Proc. 2nd ACM Symp. on Computational Geometry (1986) pp 133–141.

[5] J.L. Bentley and T.A. Ottmann. ‘‘*Algorithms for Reporting and Counting Geometric Intersections."* IEEE Transactions on Computers 28 (1979) pp 643–647.

[6] J.L. Bentley and M.I. Shamos. *"Divide-and-Conquer for Linear Expected Time."* Information Processing Letters 7 (1978) pp 87–91.

[7] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. *"Applications of Random Sampling to On-line Algorithms in Computational Geometry."* INRIA Tech. Report 1285 (1990).

[8] J.D. Boissonnat, O. Devillers, and M. Teillaud. *"A Randomized Incremental Algorithm for Constructing Higher Order Voronoi Diagrams."* to appear in Algorithmica.

[9] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. *"On-line Algorithms with Good Expected Behaviours."* Manuscript (1991).

[10] B. Chazelle. *"Reporting and Counting Segment Intersections."* J. Computer System Science 32 (1986) pp 156–182.

[11] B. Chazelle and H. Edelsbrunner. *"An Optimal Algorithm for Intersecting Line Segments in the Plane."* Proc. 29th IEEE Symp. on Foundations of Computer Science (1988) pp 590–600.

[12] B. Chazelle, H. Edelsbrunner, L.J. Guibas, and M. Sharir. *"Computing a Face in an Arrangement of Line Segments."* Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms (1991) pp 441–448.

[13] B. Chazelle, L.J. Guibas, and D.T. Lee. *"The Power of Geometric Duality."* BIT 25 (1985) pp 76–90.

[14] P. Chew. *"Building Voronoi Diagrams for Convex Polygons in Linear Expected Time."* Manuscript (1986).

[15] K.L. Clarkson. *"A Probabilistic Algorithm for the Post Office Problem."* Proc. 17th ACM Symp. on Theory of Computing (1985), pp 175–184.

[16] K.L. Clarkson. *"New Applications of Random Sampling in Computational Geometry."* Discrete & Computational Geometry 2 (1987), pp 195–222.

[17] K.L. Clarkson and P.W. Shor. *"Algorithms for Diametral Pairs and Convex Hulls that are Optimal, Randomized, and Incremental."* Proc. 4th ACM Symp. on Computational Geometry (1988), pp 12–17.

[18] K.L. Clarkson and P.W. Shor. *Applications of Random Sampling in Computational Geometry, II."* Discrete & Computational Geometry 4 (1989), pp 387–421.

[19] K.L. Clarkson. *"Linear Programming in $O(n3^{d^2})$ Time."* Inf. Proc. Letters 22 (1986) pp 21–24.

[20] K.L. Clarkson. *"A Las Vegas Algorithm for Linear and Integer Programming when the Dimension is Small."* Manuscript; a preliminary version appeared in Proc. 29th IEEE Symp. on Foundations of Computer Science (1988) pp 452–456.

[21] K.L. Clarkson. *Personal Communication*, September 10 (1990).

[22] M.E. Dyer. *"Linear Algorithms for Two and Three-Variable Linear Programs."* SIAM J. on Computing 13 (1984) pp 31–45.

[23] M.E. Dyer. *"On a Multidimensional Search Technique and its Applications to the Euclidean One-Centre Problem."* SIAM J. on Computing 15 (1986) pp 725–738.

[24] M.E. Dyer and A.M. Frieze *"A Randomized Algorithm for Fixed-Dimensional Linear Programming."* Mathematical Programming 44 (1989) pp 203–212.

[25] H. Edelsbrunner. **Algorithms in Combinatorial Geometry.** Springer Verlag (1987).

[26] H. Edelsbrunner, F.P. Preparata, and D.B. West. *"Tetrahedrizing Point Sets in Three Dimensions."* Tech. Rep. UIUCDCS-R-86-1310, Univ. of Illinois, Dept. Computer Science (1986).

[27] H. Edelsbrunner, J. O'Rourke, and R. Seidel. *"Constructing Arrangements of Hyperplanes and Applications."* SIAM J. on Computing 15 (1986), pp 341–363.

[28] G.H. Gonnet. **Handbook of Algorithms and Data Structures.** Addison-Wesley (1984).

[29] R.L. Graham. *"An efficient algorithm for determining the convex hull of a finite planar set."* Inform. Proc. Lett., 1 (1972), pp 132–133.

[30] L.J. Guibas, D.E. Knuth, and M. Sharir. *"Randomized Incremental Construction of Delaunay and Voronoi Diagrams."* Proc. ICALP (1990).

[31] T. Hagerup and C. Rüb. *"A Guided Tour of Chernoff Bounds."* Inform. Proc. Letters 33 (1989/90), pp 305–308.

[32] D. Haussler and E. Welzl. *"Epsilon-Nets and Simplex Range Queries."* Discrete & Computational Geometry 2 (1987), pp 127–151.

[33] C.A.R. Hoare. *"Quicksort."* Computer Journal 5.1 (1962), pp 10–15.

[34] R.M. Karp. *"An Introduction to Randomized Algorithms."* To appear in Discrete Applied Mathematics.

[35] D.G. Kirkpatrick, R. Seidel. *"The Ultimate Planar Convex Hull Algorithm?"* SIAM J. on Comput., Vol. 15, No. 1 (1986), pp 287–299.

[36] R. Klein. **Concrete and Abstract Voronoi Diagrams.** Springer Verlag, Lecture Notes in Computer Science 400 (1989).

[37] J. Matoušek and R. Seidel. *"On Tail Estimates for Mulmuley's Segment Intersection Algorithm."* In preparation.

[38] J. Matoušek, M. Sharir, and E. Welzl. *"A Subexponential Bound for Linear Programming."* To appear in Proc. of 8th ACM Symp. on Computational Geometry (1992).

[39] N. Megiddo. *"Linear-Time Algorithms for Linear Programming in* $\mathbb{R}^3$ *and Related Problems."* SIAM J. on Computing 12 (1983) pp 759–776.

[40] N. Megiddo. *"Linear Programming in Linear Time when the Dimension is Fixed."* Journal of the ACM 31 (1984) pp 114–127.

[41] K. Mehlhorn. *Personal Communication*, October (1990).

[42] K. Mulmuley. *"A Fast Planar Partition Algorithm: Part I."* Proc. 29th IEEE Symp. on Foundations of Computer Science (1988), pp 580–589.

[43] K. Mulmuley. *"A Fast Planar Partition Algorithm: Part II."* Proc. 5th ACM Symp. on Computational Geometry (1989), pp 33–43.

[44] K. Mulmuley. *"On Obstructions in Relation to a Fixed Viewpoint."* Proc. 30th IEEE Symp. on Foundations of Computer Science (1989), pp 592–597.

[45] F.P. Preparata and M.I. Shamos. **Computational Geometry – An Introduction.** Springer Verlag (1985).

[46] M.O. Rabin. *"Probabilistic Algorithms."* In J.F. Traub, editor, **Algorithms and Complexity, Recent Results and New Dierections.** Academic Press, New York (1976), pp 21–39.

[47] P. Raghavan. *"Lecture Notes on Randomized Algorithms."* IBM T.J. Watson Research Center Computer Science Report RC 15430 (1990).

[48] G. Rote. *Personal Communication*, October 15 (1990).

[49] R.Sedgewick. **Quicksort.** Garland, New York (1978).

[50] R. Seidel. *"Linear Programming and Convex Hulls Made Easy."* Proc. 6th ACM Symp. on Computational Geometry (1990), pp 211–215.

[51] R. Seidel. *"A Simple and Fast Incremental Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons."* To appear in COMPUTATIONAL GEOMETRY: Theory and Applications (1991).

[52] M.I. Shamos. **Computational Geometry.** Ph.D. thesis, Dept. of Computer Science, Yale Univ. (1978).

[53] M.I. Shamos and D. Hoey. *"Closest Point Problems."* Proc. 16th IEEE Symp. on Foundations of Computer Science (1975) pp 151–162.

[54] M. Sharir and E. Welzl. *"A Combinatorial Bound for Linear Programming and Related Problems."* Proc. of 9th Symp. on theoretical Aspects of Computer Science (STACS 1992).

[55] J.S. Vitter and Ph. Flajolet. *"Average-Case Analysis of Algorithms and Data Structures."* In J. van Leeuwen, editor, **Handbook of Theoretical Computer Science: Algorithms and Complexity.** Elsevier (1990), pp 431–524.

[56] E. Welzl. *"Smallest Enclosing Disks (Balls and Ellipsoids)."* In H. Maurer, editor, **New Results and New Trends in Computer Science.** Springer Lecture Notes in Computer Science 555 (1991), pp 359–370.