Check for updates

Dynamic point location in arrangements of hyperplanes

Ketan Mulmuley[1,*] and Sandeep Sen[2]

1: Computer Science Dept, The University of Chicago
2: AT&T Bell Laboratories, Murray Hill, NJ 07974

Abstract

We present algorithms for maintaining data structures supporting fast point location queries in arrangements of hyperplanes with dimension less than or equal to four. This data structure allows for deletion and insertion of hyperplanes. Our algorithms use random bits in the construction of the data-structure but do not make any assumptions about the update sequence or the hyperplanes in the input. In two dimensions, we are able to obtain $O(\log n)$ query time, $O(n \log n)$ update time and $O(n^2)$ space bound. In dimensions three and four we obtain near-optimal bounds for space, query time and also update time, where all these bounds hold with high-probability. (The probability is with respect to randomization in the data structure.) Our algorithm is simple and its extension to arbitrary dimensions is closely tied to efficient local point-searching in triangulated convex polytopes. Moreover, our approach has a versatile quality which is likely to have further applications to other dynamic algorithms.

1 Introduction

Maintaining data structures that allow periodic updates has received much attention in the past and in recent years. Typical operations include insertion and deletion of elements from a given universe like points, segments etc. and at any given stage we may have to answer queries about the present set of elements. One of the challenging goals in designing data-structure for such dynamic environment is to be able to match the query time with that of the static case (one in which the set of elements remain fixed but each instance of query could be different). At the same time it is also critical that one does not expend too much space for the data-structure and also keep the update time minimal. Balanced binary trees supporting dictionary operations is perhaps the most commonly used dynamic data structure and it also matches the asymptotic performance of searching in an ordered set. In order to compete with the static case, the dynamic data structures typically need to be more sophisticated and sometimes turn out to be prohibitively difficult to implement. Examples of some sophisticated dynamic data structures include data structures for planar point location [6, 9, 12, 13].

A more recent line of attack for designing dynamic data structures has been the use of randomization. The term randomized algorithms in this paper will refer to algorithms that do not assume any distribution of the input but use random bits to make choices at different stages of the algorithm for any input. Skip Lists ([14]) and Randomized Search Trees ([1]) are examples of dynamic data structures proposed recently and use randomization. Their performance bounds compare very favorably with their deterministic counterparts (that is the balanced binary trees) and are much simpler to implement. The obvious trade-off is that the performance bounds are guaranteed with certain probabilities which in spite of being less than 1 are usually acceptable for most applications. In particular, if one can guarantee performance bounds with probability $1 - 1/n^\alpha$ for a large enough $\alpha > 1$, where n is the input size, then even for moderate values of n this is very close to 1. Bounds of these form are often referred to in the literature as *high probability* bounds. These are stronger than bounds on the mean behavior, which cannot predict the probability of deviation from the *expected* behavior. The following notation will be used in the paper. We

*Supported by NSF grant CCR 8906799 and Packard Fellowship

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-426-0/91/0006/0132 $1.50

say a randomized algorithm has resource (like time, space, etc.) bound $\tilde{O}(g(n))$ if there is a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $cg(n)$ with probability $\geq 1 - 1/n^{\alpha}$, for $\alpha \geq 1$; α depends on c. By choosing c large enough, one can get a large enough value for α.

The static point location problem for arrangements of hyperplanes has been satisfactorily solved [4] by making use of randomization and then subsequently derandomizing it efficiently ([3]). In this paper, we further investigate the use of randomization for searching in arrangements of hyperplanes in a dynamic environment. In dimension two, we are able to obtain a very simple algorithm which guarantees $\tilde{O}(\log n)$ query time, $\tilde{O}(n^2 \log n)$ space, and $\tilde{O}(n \log n)$ update time. The expected space requirement is $O(n^2)$. In dimension three and four too, we obtain near optimal bounds (up to a polylogarithmic factor). Our data structure for dynamic point location in arrangements of lines is extremely simple in comparison with the sophisticated data structures in [6, 9, 12, 13]. No efficient dynamic algorithms were earlier known for this problem in dimensions higher than two. A possible extension of our algorithm to arbitrary dimensions is closely tied to efficient local point-searching in triangulated convex polytopes. In dimension higher than four we can give a very simple algorithm which achieves $\tilde{O}(\sqrt{n}polylogn)$ query time and $\tilde{O}(n^{d-1}polylogn)$ *amortized* update time; it is omitted in this abstract. Random sampling results in ([10, 4, 15]) have contributed significantly towards our arrangement searching algorithms.

Notation: In this paper, we shall use $|\ |$ to denote the size operation. Thus if N is a set, $|N|$ denotes its size, if f is a convex polytope, $|f|$ is the number of its all subfaces, and so on.

2 The Basic Algorithm

In this section we present a high-level, dimension independent description of our basic approach. Some steps of our algorithm are dependent on the dimension. We shall present the implementation of these steps in later sections where we shall instantiate our basic algorithm in various dimensions.

We begin by describing a procedure for building a point-location data-structure in the static case and subsequently argue that its extension to dynamic situation is straightforward. The static algorithm is reminiscent of an algorithm due to Clarkson [4] turned upside-down. Given a set N of hyperplanes in R^d, we shall denote the induced arrangement by $G(N)$. The d-cells of $G(N)$ can have an unbounded number of facets and this turns out to be problematic. Hence, we shall work with a certain triangulation $H(N)$ of $G(N)$ that is obtained by decomposing each d-cell of $G(N)$ into simplices or, in general, cells with bounded number of facets. We shall leave the exact nature of $H(N)$ completely abstract at this point, except that it will assumed to satisfy the following condition: Each d-cell f of $G(N)$ is decomposed into $O(|f|)$ simplices, or in general cells, each of which is "defined" by a bounded number of hyperplanes. As an abuse of notation, we shall refer to the d-cells of $H(N)$ as d-simplices, even though, strictly speaking, they need not be simplices.

The following basic algorithm builds a point location structure $\tilde{H}(N)$ that can be used to locate the d-simplex of $H(N)$ containing any query point $p \in R^d$.

Let $N = N_1$. $\tilde{H}(N) = \tilde{H}(N_1)$ is defined recursively as follows:

1. Build the triangulation $H(N_1)$.

2. For each hyperplane in N_1, toss an unbiased coin. Let N_2 be the set of hyperplanes in N_1 for which the toss turned out to be head. Build $\tilde{H}(N_2)$ recursively.

3. Associate with each d-simplex Δ of $H(N_2)$ a list $L(\Delta)$ of hyperplanes in $N_1 \setminus N_2$ that intersect Δ and conversely with each hyperplane in $N_1 \setminus N_2$, we associate a list of d-simplices in $H(N_2)$ that it intersects. We also say that the hyperplanes in $L(\Delta)$ conflict with Δ and $L(\Delta)$ is called its conflict list.

4. Build a data structure $Descent(2, 1)$ that provides a "descending link" between $\tilde{H}(N_2)$ and $H(N)$. This structure is used in point location queries, in a manner to be described soon. At this stage, we shall leave the nature of this descent structure completely abstract.

An important fact regarding our point location structure is that, for every $l > 1$, N_l is a random sample of N_{l-1} of roughly half the size. Hence, the random sampling results in [4, 10] imply that, with very high probability, for every d-simplex Δ of $H(N_l)$ and every $l > 1$, $|L(\Delta)| = \tilde{O}(\log n)$. This immediately implies that the space requirement of this data structure is $\tilde{O}(n^d \log n)$, where n is the size of N. We shall denote the size of N_l by n_l.

To locate a point p in $H(N) = H(N_1)$, we recursively locate the d-simplex Δ_2 in $H(N_2)$ containing p. We assume that we are given a *descent oracle* so that given Δ_2, $L(\Delta_2)$ and the Descent structure $Descent(2,1)$, the d-simplex Δ_1 in $H(N_1)$ containing p can be located quickly, i.e. in time proportional to the size of $L(\Delta_2)$, up to a polylog factor. An analogous condition is assumed to hold at every recursive level. If Δ_i denotes the d-simplex of $H(N_i)$ containing p then we already know that, for all i, $|L(\Delta_i)|$ is $\tilde{O}(\log n)$. As the number of levels in $\tilde{H}(N)$ is easily seen to be $\tilde{O}(\log n)$, this implies $\tilde{O}(polylog(n))$ bound on the query time. Of course, we have proven this bound for a *fixed* query point. But as we shall see later, this easily translates into a poly-logarithmic bound for any query point, because there will be only polynomially many distinct search paths in our data structure. To get a tighter bound on the query time, such as $\tilde{O}(\log n)$ bound in dimension two, we need to use refined random sampling results that will be proven later in this paper.

To make our data structure dynamic we adopt the following scheme. Our procedures for addition and deletion of a hyperplane will be such that, at any given time, the state of our data structure will be independent of the actual sequence of updates that built it. Thus if N were to denote the set of currently existing hyperplanes that have added but not deleted so far, then $\tilde{H}(N)$ will be as if it were built by the above static procedure applied to N. This will ensure that the random sampling results that are crucial to analyze our static data structure carry over, more or less unaffected, to the dynamic setting.

Let us now see how to add a new hyperplane h to $\tilde{H}(N)$. We first toss an unbiased coin successively until we get a tail. Let j be the number of heads obtained before getting a tail. We shall simply "add" h to levels 1 through $j+1$. For $1 \leq l \leq j+1$, let $\bar{N}_l$ denote $N_l \cup \{h\}$. Addition of h to the l-th level is carried out in three steps.

1. Update $H(N_l)$ to $H(\bar{N}_l)$.
2. Construct conflict lists of the new d-simplices in $H(\bar{N}_l)$.
3. Update $Descent(l+1, l)$.

The third step is dependent on the exact nature of the descent structures. Hence, we shall only elaborate the first two steps.

The zone of a hyperplane (in d dimension) is defined as the following.

Let h_0 be a hyperplane in an arrangement $G(H)$. A $k-$face f for $0 \leq k \leq d-1$ is said to be visible from h_0 if there is a line segment s that connects f and h_0 such that the interior of s is contained in h_0 or in a cell of $\mathcal{A}(H)$. The *zone* of h_0 is the set of $k-$faces that are visible from h_0.
Define $Zone(N_l, h)$, the *Zone* of h in the arrangement $G(N_l)$. The Zone Theorem in [8] states that

Theorem 1 (Zone theorem) *The maximum cardinality of $Zone(N_l, h)$ is $O(n_l^{d-1})$, where n_l is the size N_l, and moreover, $Zone(N_l, h)$ can also be determined in $O(n_l^{d-1})$ time.*

Let f be any d-cell in $Zone(N_l, h)$. We remove all d-simplices in the old triangulation of f. Next we split f along h into two d-cells f_1 and f_2 and triangulate f_1 and f_2 all over. All triangulation schemes to be considered in this paper are simple enough so that triangulation of f_1 and f_2 can be carried out in $O(|f_1| + |f_2|) = O(|f|)$ time.

We also need to construct conflict lists of all d-simplices in the triangulation of f_1 and f_2. Let h' be a hyperplane in $N_l - N_{l-1}$ that intersects f. From the old conflict information, we can figure out all 1-faces (edges) of f intersecting h'. Hence, by a straightforward search in the new triangulations of f_1 and f_2, we can determine all d-simplices within f_1 and f_2 that intersect h' in time proportional to their number. Because the size of every conflict list, new or old, is $\tilde{O}(\log n)$, with high probability, it follows that the total cost of updating the conflict lists is $\tilde{O}(\sum_f |f| \log n) = \tilde{O}(n_l^{d-1} \log n)$, where f ranges over all d-cells of $G(N_l)$ intersecting h.

To summarize:

Lemma 1 *The cost inserting a new hyperplane in $\tilde{H}(N)$ is $\tilde{O}(n^{d-1} \log n)$, ignoring the cost of updating the descent structures.*

Deletion is the exact reversal of addition, that is, the cost of deletion is no more than inserting the hyperplane immediately afterwards. Hence, we shall merely state:

Lemma 2 *The cost deleting any hyperplane from $\tilde{H}(N)$ is $\tilde{O}(n^{d-1} \log n)$, ignoring the cost of updating the descent structures.*

In higher dimensions, the descent oracle becomes a critical bottle-neck. Our broad objective in dimension d is to obtain a polylogarithmic search time and $O(n^{d-1} \log^a n)$ update time, for some fixed constant a, where n denotes the number of hyperplanes

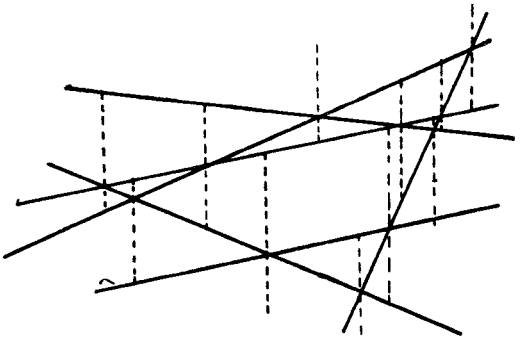

Figure 1: A trapezoidal decomposition.

currently in the data structure. We are able to obtain an efficient implementation of the descent oracle up to dimension four. An efficient (polylogarithmic time) implementation of this step to arbitrary dimensions would extend our results likewise. In the next section, we instantiate the basic algorithm given here in dimension two. Dimension three and four will be dealt with in Section 4.

3 Two dimensional arrangements

Let N be a set of n lines in R^2 and let $G(N)$ denote the induced arrangement. The convex regions of $G(N)$ need not have a bounded number of sides. Hence, using a well known scheme, we decompose each convex region of $G(N)$ into vertical trapezoids. ¿From each vertex of the convex region (polygon) extend a vertical ray directed towards the interior until it meets an edge of the polygon (See Fig.1). This partitions the convex polygon into trapezoids. (If required these trapezoids can be triangulated by drawing a diagonal. But this is not necessary, since each trapezoid is obviously "defined" by a bounded number of lines.) When the above procedure is repeated for all convex regions of $G(N)$ we get the triangulation $H(N)$ that we shall use in our basic algorithm.

The only thing that remains to be specified in the definition of our search structure $\tilde{H}(N)$ is the nature of the descent structures. The descent structure $Descent(l, l-1)$ between two successive levels l and $l-1$ will be defined as simply the superposition of the triangulation $H(N_l)$ and the arrangement$G(N_{l-1})$ (not its triangulation). We shall also denote this superposition by $H(N_l) \oplus G(N_{l-1})$.

Equipped with this descent structure, it is really easy to descend from level l to level $l-1$ during point location. Let Δ_l be the trapezoid in $H(N_l)$ containing the query point q. Let $\Delta_l \cap G(N_{l-1})$ denote the restriction of the arrangement $G(N_{l-1})$ to Δ_l, which is available to us from $Descent(l, l-1)$. We can easily locate the trapezoid within $\Delta_l \cap G(N_{l-1})$ containing q in $O(|L(\Delta)|)$ time. One easy way to do this is as follows. We determine the first line h in N_{l-1} that intersects the vertical ray from q directed upwards. Let q' be the point of intersection. Obviously h is either the line bounding the upper side of Δ_l or it belongs to $L(\Delta_l)$. Next we locate q' in $\Delta_l \cap G(N_{l-1})$ by simply walking along h within $G(N_{l-1})$. Once we know where q is located in $H(N_l) \oplus G(N_{l-1})$, in additional $O(|L(\Delta_l)|)$ time, we can figure out the trapezoid Δ_{l-1} of $H(N_{l-1})$ containing q.

Thus we can descend from level l to level $l-1$ in $O(|L(\Delta_l)|)$ time. With high probability, $|L(\Delta_l)| = O(\log n)$, for all l, and the number of levels is $O(\log n)$. It follows that that the time required to locate a fixed point q is $\tilde{O}(\log^2 n)$. The following theorem shows that the query time is, in fact, $\tilde{O}(\log n)$.

Theorem 2 *For a fixed query point q, $\sum_i |L(\Delta_i)|$ is $\tilde{O}(\log n)$, where Δ_i is the trapezoid containing q in $H(N_i)$.*

Proof: Let $NB(s)$ denote the random variable that is equal to the number of tails obtained before obtaining s heads in succession of binomial trials with a fair coin. $NB(s)$ is the familiar Negative Binomial distribution. When $s = 1$, it is the geometric distribution. We shall show that, for all i, $|L(\Delta_i)| = O(NB(a))$, for some fixed constant a. Because the coin tosses at each level, used in the definition of data structure, are independent from the coin tosses used in the preceding levels, it then follows that, for any fixed constant c, $\sum_{i \leq c \log n} |L(\Delta_i)| = O(NB(ca \log n)) = \tilde{O}(\log n)$, using Chernoff bound [2] for negative binomial distributions. As the number of levels is $\tilde{O}(\log n)$, this will prove the theorem.

So fix a level i. Also fix the set N_i of lines occurring in the i-th level of the data structure. The set N_{i+1} is determined by flipping a fair coin for each line in N_i and retaining those lines for which the toss was head. We shall prove that:

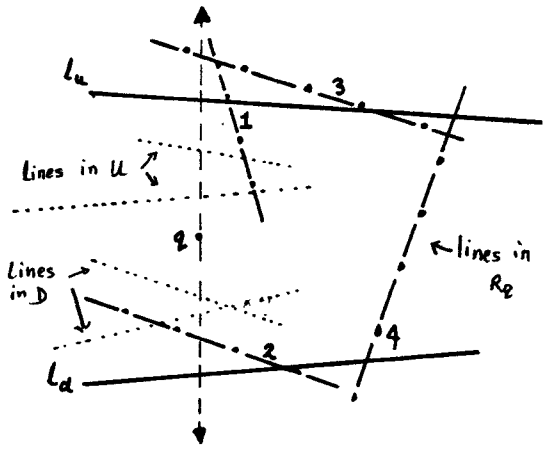

Figure 2:

Lemma 3 *There is an imaginary, online ordering $h_1, h_2, \cdots$ of all lines in N_i such that the set of lines "defining" or intersecting the trapezoid Δ_{i+1} always occurs as an initial subsequence of $h_1, h_2, \cdots$. By online ordering we mean that h_{k+1} can be chosen on the basis of the known coin toss results for $h_1, \cdots, h_k$. Note that Δ_{i+1} is not known to us a priori, because it depends on the results of coin tosses for the lines in N_i.*

As the number of lines defining any trapezoid is at most four, it follows from the lemma that $|L(\Delta_{i+1})|$ is $O(NB(4))$.

Proof of the lemma: Consider the ordered set V_u of lines (in the increasing Y direction) in N_i intersecting the vertical line extending upward from query point q. See fig.2.

Initially we shall toss coins for these lines in V_u, in the increasing Y direction away from q, until we obtain a head, and then (temporarily) stop. Let l_u be the line for which we obtained head. Let $U \subseteq V_u$ denote the set of lines before l_u for which we obtained tails. Clearly, $l_u \in N_{i+1}$, whereas no line in U belongs to N_{i+1}. Thus l_u is obviously going to be bounding the top of the trapezoid Δ_{i+1}, which we do not know completely as yet. Moreover, all lines in U obviously conflict with Δ_{i+1}.

Now we resume our coin tossing, in a symmetric manner, for the lines in N_i intersecting the vertical line extending downward from q, until we obtain a head, and then we again stop temporarily. Let D be the set lines for which we obtained tails and let l_d be the line for which we obtained head. Obviously, l_d is going to be bounding the bottom of the trapezoid Δ_{i+1}, which we know partially by now.

Now discard (hypothetically) the lines in U and D and consider the intersections of the remaining lines with l_u and l_d. Let R_q be the set of remaining lines that intersect either l_u or l_d to the right of the vertical line through q. We order R_q as follows. Given two lines l_1 and l_2 in R_q, we say that $l_1 \ll l_2$, if the y-coordinate of either $l_1 \cap l_u$ or $l_1 \cap l_d$ is less than the y-coordinates of both $l_2 \cap l_u$ and $l_2 \cap l_d$. Fig.2 shows ordering of R_q. Now we resume tossing coins for the lines in R_q in the increasing order, until we obtain head. Let l_r be the line for which we obtained head. It is then clear that l_r defines the right side of Δ_{i+1} in the sense that the right side of Δ_{i+1} will be extending from the intersection of either l_u or l_d with l_r. Moreover, all lines for which we obtained tails, will conflict with Δ_{i+1}.

Now discard (hypothetically) the lines in R_q too. Let L_q be the set of remaining lines intersecting either l_u or l_d to the left of the vertical line through q. We order L_q in a symmetric fashion, and resume tossing coins for the lines in L_q in the increasing order (away from q) until we get head and then temporarily stop. Let l_l be the line for which we obtained head. It is clear that it "defines" the left side of the trapezoid Δ_{i+1}, and all lines for which we obtained tails conflict with Δ_{i+1}.

At this point the trapezoid Δ_{i+1} containing q in the $(i+1)$-st level has been completely determined. Indeed l_u, l_d, l_r, l_l are the lines defining Δ_{i+1} and the lines for which we obtained tails so far are precisely the lines in conflict with Δ_{i+1}. (We did not take into account the exceptional cases such as when Δ_{i+1} is unbounded or when it is, in fact, a triangle. But a slight modification to the argument will cover these cases too.)

We can now toss coins for the remaining lines in any order whatsoever. It follows that the above online sequence of tosses has the desired property. □

In the above theorem, we showed that the query time is $\tilde{O}(\log n)$ for a fixed query point. We further note that there are only polynomially many distinct combinatorial search paths for a given data structure. By combinatorially distinct, we imply different sequence of triangles in the search path. More precisely, let $\bar{G}(N)$ be the refinement of $G(N)$ obtained by passing infinite vertical lines through all intersections among the lines in N. Then, for a fixed region R in $\bar{G}(N)$, it is easy to see that the search path in $\tilde{H}(N)$ remains the same if the query point

lies anywhere in R. This implies that the cost of locating any point is $\tilde{O}(\log n)$.

The space requirement of our data structure is $\tilde{O}(n^2 \log n)$, because with high probability $|L(\Delta)|$ is $O(\log n)$, for every trapezoid $\Delta \in H(N_l)$ and every l. Using the results in [5], it can be shown that the "average" conflict size of a trapezoid in any level is $O(1)$, because N_l is a random sample of N_{l-1} of roughly half the size. This implies that the expected space required by our data structure is $O(n^2)$.

Now let us estimate the cost of adding or deleting a line. We shall only consider addition, because deletion is the exact reversal of addition. By Lemma 1, we only need to worry about the cost of updating the descent structures $Descent(l, l-1)$, $1 < l \leq j+1$, where j is the number of successive heads obtained. We shall only consider the case $1 \leq l \leq j$, the remaining case (when the line is not chosen in the sample) being straightforward from the *Zone* theorem. Fix l. Fig.3a shows the old descent structure $H(N_l) \oplus G(N_{l-1})$. The lines in N_l are shown dark, whereas the lines in N_{l-1} are shown light. The new line h being added is shown dashed. The vertical sides of the trapezoids are shown dotted. We have only shown the restriction of $H(N_l) \oplus G(N_{l-1})$ to $Zone(N_l, h)$, because all the changes take place in this zone.

Our goal is to update the old descent structure $H(N_l) \oplus G(N_{l-1})$ to the new descent structure $H(\bar{N}_l) \oplus G(\bar{N}_{l-1})$, where $\bar{N}_l = N_l \cup \{h\}$ and $\bar{N}_{l-1} = N_{l-1} \cup \{h\}$. This is done as follows.

1. The vertical sides of the trapezoids in $H(N_l)$ intersecting h are split. See fig.3b.

2. We add vertical segments through the intersections of h with the lines in N_l, one at a time, in the the increasing order of their x-coordinates. Fig.3c shows addition of the vertical segment through an intersection v on h, assuming that vertical segments through all intersections to the left of v have already been added. We insert this vertical segment, starting at v and traveling in the upward and downward direction, by successively traversing the faces in $H(\bar{N}_l) \oplus G(\bar{N}_{l-1})$ (or more precisely, its part computed so far). During any face traversal, we visit only those vertices of that face that lie to the left of the vertical segment being inserted. This ensures that when the insertion of all vertical segments is complete, any given junction of $H(\bar{N}_l) \oplus G(\bar{N}_{l-1})$ could have been visited in only $O(1)$ face traversals.

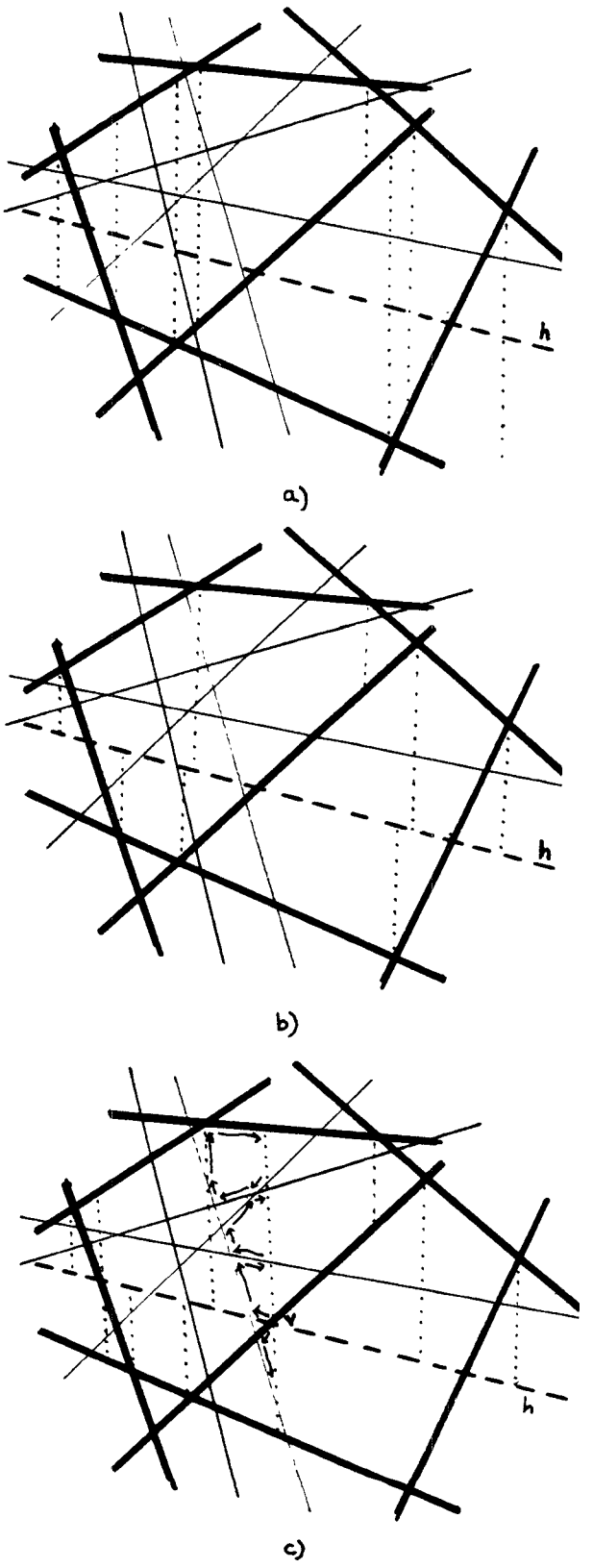

Figure 3: Updating the descent structure

It is easy to see that the time taken by the whole procedure is

$$O(n_{l-1} + \sum_{\Delta} |L(\Delta)| + m_l),$$

where n_{l-1} is the size of N_{l-1}, Δ ranges over all destroyed and newly created trapezoids in $H(N_{l-1})$ and $H(\bar{N}_{l-1})$ respectively, and m_l denotes the total number of intersections among the lines in N_{l-1} that lie within $Zone(N_l, h)$. By Zone Theorem and the fact that the conflict size of every trapezoid is $\tilde{O}(\log n)$, it follows that the second term in the above bound it $\tilde{O}(n_{l-1} \log n)$. We shall now show that m_l is also $\tilde{O}(n_{l-1} \log n)$.

Lemma 4 *The total number of vertices of $G(N_{l-1})$ lying in $Zone(N_l, h)$ can be bound by $\tilde{O}(n_{l-1} \log n)$.*

Proof (sketch): We account for the intersections by charging it to the intersecting lines (so the actual number of intersections is half this number). Consider a line of $L \in N_{l-1}$ that is not in N_l and look at the ordered list of intersections with lines of N_{l-1}. For L, we denote the number of lines that it intersects in a trapezoid Δ_j (in $H(N_l)$) by $I_{L,j}$. *To make the proof technically precise, we need to apply the kind of arguments about an on-line ordering used in Lemma 3, but we leave out the details here.* Clearly $I_{L,j}$ is upper-bounded by a geometric random variable with parameter 1/2 and moreover $I_{L,j}$'s are independent over different trapezoids (two lines intersect in only one point and all the n_{l-1} lines are totally ordered with respect to their intersections with L). It can be shown again from Chernoff bounds that , $\sum_{j_1}^{j_k} I_{L,j_i} = \tilde{O}(k)$ for $k \geq c \log n$ for some constant $c > 1$.

Now consider all the trapezoids in the zone of h and let t_i denote the number of trapezoids that line $L_i \in N_{l-1} \setminus N_l$ intersects. We know that $\sum_i t_i = \tilde{O}(n_l \log n)$. Let $\mathcal{I}$ denote the set of indices of lines L_i that intersect less than $c \log n$ zone-cells. Clearly the total number of intersections that they contribute to is $\tilde{O}(|\mathcal{I}| \log n)$. For $i \in N_{l-1} \setminus \mathcal{I}$, let m_i be the number of intersections L_i contributes to. Let V_L denote the set of vertices in the zone of h. Then we consider the equivalence class of lines that intersect the same zone-cells. The number of such equivalence classes is bound by the *semispaces* of V_h which is $O(n_l^2)$ since $V_L = O(n_l)$. This implies that for all L_i, i not in $\mathcal{I}$, $m_i = \tilde{O}(t_i)$ (t_i is at least the number of zone-cells that L_i intersects). Hence the total number of intersections can be bound by

$$O(\sum_{i \in N_{l-1} \setminus \mathcal{I}} t_i + |\mathcal{I}| \log n)$$

with high probability which is $\tilde{O}(n_{l-1} \log n)$.

This bound is for a fixed zone of h. The number of combinatorially different zones can again be bound by the number of semispaces of the vertices of the arrangement which is $O(n^4)$. The lemma follows. □

There exists a simpler argument to bound the total cost of step 2 by observing that the actual cost of step 2 for any vertical segment is bounded by the size of the *zone* of $\tilde{O}(\log n)$ lines within the left trapezoid that is bordered by this vertical segment; see fig.3c. Over the $O(n_l)$ such segments this adds up to $\tilde{O}(n_l \log n)$. However we feel that the previous lemma could be of independent interest which also has a natural analogue in higher dimensions.

To summarize: the cost of updating $Descent(l, l-1)$ is $\tilde{O}(n_{l-1} \log n)$. Summing over all levels, it follows that the total cost of updating the descent structures is $\tilde{O}(\log n \sum_l n_l) = \tilde{O}(n \log n)$.

We summarize our main result as follows:

Theorem 3 *Let A be an arrangement of n lines in a plane. There exists a dynamic point location data structure with expected space $O(n^2)$ and query time $\tilde{O}(\log n)$ which also allows for insertion/deletion of lines in time $\tilde{O}(n \log n)$. The space bound is also $\tilde{O}(n^2 \log n)$.*

Remark: Using the best known deterministic schemes for dynamic point location [6, 13], one can achieve $O(\log^2 n)$ and $O(n \log n)$ bounds for search and update times respectively. These are considerably more involved procedures.

4 Extension to higher dimensions

In this section, we specify the descent structures for dimension three and four. To pinpoint the bottleneck in higher dimensions, we imagine giving a dynamic point location algorithm in arbitrary dimension d, assuming that this has already been done for dimension less than d. The base case, $d = 2$, has already been described in Section 3.

Let N be a set of hyperplanes in R^d, and let $G(N)$, as before, denote the induced arrangement. The triangulation $H(N)$ of $G(N)$ that we shall use is defined as follows. We triangulate the j-faces of $G(N)$, $j \leq d$, by induction on j. If $j = 2$, we use the scheme in Section 3 to decompose the 2-faces, which are convex polygons, into trapezoids by passing segments through their vertices that are parallel to, say, $\{x_1 = 0\}$ hyperplane. Otherwise, let f be

any j-face of $G(N)$, $j > 2$. Let v denote the vertex of f with the smallest x_d coordinate; it is possible that v lies at "infinity". By our inductive hypothesis, all facets of f have been triangulated. So we simply extend the "simplices" on the boundary of f to cones with apex at v. This gives us a simple triangulation of f. When all j-faces of $G(N)$ are triangulated in this fashion, we get the triangulation $H(N)$ that we sought. A d-cell of $H(N)$ will be called a d-simplex, though, strictly speaking, it is not a simplex. But it is defined by a bounded number of hyperplanes.

Now let us turn to the descent oracle. We assume that $Descent(i, i-1)$ contains a recursively defined dynamic point location structure for the lower dimensional arrangement $G(N_{i-1}) \cap Q$, for each hyperplane $Q \in N_{i-1}$. We shall denote this structure by $\tilde{H}(N_{i-1}, Q)$. Its maintenance is done by recursively applying our lower dimensional point location algorithm. In addition, $Descent(i, i-1)$ will contain a certain *static* point location structure $\tilde{f}$ for every d-cell f of $G(N_{i-1})$, so that given any point $p \in f$, we can locate in poly-log time the d-simplex $\Delta \in H(N_{i-1})$ containing p in the triangulation of f. We shall come to the construction of $\tilde{f}$ later. For a moment, let us assume that we can associate such a point location structure with every d-cell of $G(N_{i-1})$.

Now the descent from level i to level $i-1$ during point location is easy to carry out. Given a simplex $\Delta = \Delta_i \in H(N_i)$ containing the query point p, we can locate the simplex Δ_{i-1} in $H(N_{i-1})$ containing p as follows. Examine the conflict list $L(\Delta)$ consisting of the hyperplanes in $N_{i-1} - N_i$ intersecting Δ. If $L(\Delta)$ is empty, we are lucky. Any vertex v of Δ will tell us the d-cell f of $G(N_{i-1})$ containing p. Now we simply use $\tilde{f}$ to locate the d-simplex in $H(N_{i-1})$ containing p. If $L(\Delta)$ is not empty, we proceed as follows. Let v be any fixed vertex of Δ. Let (p, v) denote the ray starting at p directed towards v. If (p, v) does not intersect any hyperplane in $L(\Delta)$, then v again tells us the d-cell of $G(N_{i-1})$ containing p, and we proceed as before. Otherwise, let Q be the first hyperplane in $L(\Delta)$ that (p, v) hits, and let q be the point of intersection. By using the dynamic point location structure associated with Q in $Descent(i, i-1)$, we locate q in $G(N_{i-1}, Q) = G(N_{i-1}) \cap Q$. This tells us the d-cell $f \in G(N_{i-1})$ containing p, and we again proceed as before.

So what remains now is to keep a point location structure $\tilde{f}$ associated with every d-cell $f \in G(N_i)$, for all i. A careful examination of the basic algorithm in Section 3.1 will reveal that $\tilde{f}$ only needs to be a static point location structure. Indeed, every time a new d-cell f gets created at any level, we can afford to construct $\tilde{f}$ from scratch, by mercy of the Zone Theorem. The following theorem now tells us that such structure $\tilde{f}$ can be constructed fast, if $d = 3, 4$.

Theorem 4 *Let C be any convex polytope in R^d, $d \leq 4$. We assume that all vertices of C have distinct x_d coordinates, using the usual perturbation arguments. Assume that we are given the facial structure of C. Let $|C|$ denote the size of C, i.e., to say the total number of its vertices. Let C' denote the triangulation of C defined above. Then one can construct a point location structure in $O(|C|)$ time and space, for $d = 3$, and $O(|C| \log n)$ time and space, for $d = 4$, so that given any point p lying within C, one can locate the d-simplex in C' containing p in $O(\log n)$ time, for $d = 3$, and $O(\log^2 n)$ time, for $d = 4$.*

Proof: When $d = 3$, one can use the point location structure of [7], for example. We shall only consider the most interesting case $d = 4$. Let v denote the x_d-minimum on C; if v lies at infinity only a simple modification to the argument is needed. Let $P(v)$ be any fixed hyperplane through v that supports C. Let $P \neq P(v)$ be any hyperplane parallel to $P(v)$ that intersects C. Let Lp denote the linear projection through v onto P. More precisely, any point $q \in R^4$ is mapped to the point of intersection $Lp(q)$ between P and the line through v and q. Let $Lp(C')$ denote the projection of C' onto P. The triangulation C' of C is such that point location in C' is reducible to point location in $Lp(C')$: to locate the d-simplex in C' containing a given point $q \in C$, we simply locate $Lp(q)$ in $Lp(C')$. But $D = L(C')$ is a convex partition of the three dimensional hyperplane P. Hence, we shall be done by the following theorem. □

Theorem 5 *Let D be any convex partition of R^3. Let m denote the size of D. Assume that the facial structure of D is available to us. One can construct a point location structure in $O(m \log m)$ time so that, given any point $q \in R^3$, we can locate q in D in $O(\log^2 m)$ time.*

Proof (sketch): Our basic idea is to extend the planar point location algorithm of Sarnak and Tarjan [16] to dimension three in a very natural way. Fix the coordinates x, y, z in R^3. Let $v_1, v_2, \cdots, v_m$ be the ordered list of the vertices of D, ordered according to their z-coordinates. As a convention,

let $v_0 = (0, 0, -\infty)$ and $v_{m+1} = (0, 0, \infty)$. Let $D_i, 0 < i \leq m$ denote the intersection of D with the open slab $\{z(v_i) < z < z(v_{i+1})\}$. For a moment, assume that these slabs are given to us in some form. Then it is very easy to locate q. One locates v_i such that $z(v_i) < z(q) < z(v_{i+1})$. Then one locates q in D_i. Of course, one can not construct the slabs D_i explicitly, because the total size of these slabs can be much larger than the size of D. Let $\bar{D}_i$ denote the intersection of D_i with any plane of the form $\{z = a\}$, where $z(v_i) < a < z(v_{i+1})$. It is easy to see that the facial structure of D_i is "isomorphic" to $\bar{D}_i$, for any such choice of the hyperplane. Each $\bar{D}_i$ is a two-dimensional convex partition. If only, we had a point location structure for each $\bar{D}_i$ available to us in some form, we could, with some care, reduce point location in D_i to point location in $\bar{D}_i$. Of course, for the same reason as before, we can not construct the point location structure for each $\bar{D}_i$ explicitly. However, notice that the successive planar partitions $\bar{D}_i$ and $\bar{D}_{i+1}$ do not differ that much. Hence, it should be possible to obtain a point location structure for $\bar{D}_{i+1}$ by simply remembering its "difference" with respect to the point location structure for $\bar{D}_i$. This is not possible in arbitrary dimension. But, because each $\bar{D}_i$ is a planar partition, one can use the dynamic point location algorithm of [6, 13] as follows. Let us sweep D with a plane $P_t = \{z = t\}$ moving in the z direction, starting at $z = -\infty$ At every time t, during the sweep, we maintain a dynamic point location structure for the *frontier* $D_t = D \cap P_t$, which is a planar partition. When P_t sweeps past a junction v_i, we update the point location structure for D_i in the vicinity of v_i to obtain a point location structure for D_{i+1}. Using the update operations in [6] this can be done in $O(s(i) \log m)$, where $s(i)$ denotes the structural change in the frontier at v_i. However, we must somehow make sure that the point location structure for D_i is still accessible to us—in some implicit form. This is achieved by using a *persistent* [16] form of the data structures in [6]. This will give us, at the end of the sweep, one global data structure that contains a point location structure for each $\bar{D}_i$ as an implicit substructure, so that one can perform point location in each $\bar{D}_i$ in $O(\log^2 m)$ time. □

We have now completely specified the descent structures, for $d = 3$ and 4. Let us bound the query time. For the basis case, $d = 2$, we already know from Theorem 3, that point location takes $\tilde{O}(\log n)$ time. As the conflict size of every d-simplex in our data structure is $\tilde{O}(\log n)$, Using Theorem 4 it is easily seen that the descent from level i to level $i-1$, using the procedure given in the beginning of the section takes $\tilde{O}(\log^2 n)$ time, for $d = 3$. For $d = 4$, the bound is seen to be $O(\log^3 n)$, by using one more call to the recursive argument. From Theorem 3 and Theorem 4, and the fact that the conflict size of every d-simplex is $\tilde{O}(\log n)$, it also follows that the space requirement of our data structure is $\tilde{O}(n^3 \log n)$, for $d = 3$, and $\tilde{O}(n^4 \log n)$, for $d = 4$. The expected space bound for $d = 3$ is $O(n^3)$. This follows because N_i is a random sample of N_{i-1} of roughly half the size, hence the "average" conflict size [5] of any simplex in the i-th level is $O(1)$, for all i.

Let us now see how to add a new hyperplane h to our data structure $\tilde{H}(N)$. By Lemma 1 we only need to worry about updating the descent structures. $Descent(i, i-1)$ is updated as follows:

1. Update the lower dimensional point location structures associated with the arrangements $G(N_{i-1}, Q) = G(N_{i-1}) \cap Q$, for every $Q \in N_{i-1}$.

2. Remove the static point location structures associated with the d-cells in $G(N_{i-1})$ intersecting h.

3. Construct a completely new static point location structure for each d-cell of $G(\bar{N}_{i-1})$ adjacent to h, where $\bar{N}_{i-1} = N_{i-1} \cup \{h\}$.

For $d = 3$, it easily follows from recursive considerations and Theorems 3 and 4 that the cost of updating $Descent(i, i-1)$ is $\tilde{O}(n_{i-1}^2 \log n + \sum_f |f|)$, where f ranges over all 3-cells of $G(N_{i-1})$ intersecting h. By Zone Theorem, the second term is $O(n_{i-1}^2)$. Thus the total cost of updating the descent structures, for $d = 3$, is $\tilde{O}(\sum_i n_i^2 \log n) = \tilde{O}(n^2 \log n)$. In dimension four, it similarly follows that the total cost of updating the descent structures is $\tilde{O}(n^3 \log n)$.

Deletion is the exact reversal of addition, and hence will not be discussed any further. To summarize:

Theorem 6 *The query time of our algorithm is $\tilde{O}(\log^2 n)$, in dimension three, and $\tilde{O}(\log^3 n)$, in dimension four. The cost of update is $\tilde{O}(n^2 \log n)$, in dimension three, and $\tilde{O}(n^3 \log n)$ in dimension four. The space bounds in dimensions three and four are $\tilde{O}(n^3 \log n)$ and $\tilde{O}(n^4 \log n)$ respectively. The expected space bound in dimension three is $O(n^3)$.*

5 Conclusion

In this paper we have presented a very simple scheme for maintaining a dynamic point location structure for arrangements of hyperplanes, with $d \leq 4$. Its extension to arbitrary dimension depends critically on static point location in triangulated convex polytopes, which is a fundamental problem in its own right.

We have not discussed variations of our basic scheme and further simplifications at the cost of poly-logarithmic blow-up in update time. For example, in Section 4, $Descent(i, i-1)$ associated recursively defined lower dimensional dynamic point location structures with each hyperplane in N_{i-1}. It is possible avoid this recursion, by making use of the "superposition" technique in Section 3. Another aspect that will be discussed in the full version of the paper is parallelization of the update procedure. We shall present efficient polylogarithmic time parallel algorithm for the update procedure.

One of the main contributions of this paper has been adaptation of the skip-list methodology [14] for dynamic algorithms. We believe that this is a versatile and powerful tool which is likely to have further applications [11].

References

[1] C. Aragon and R. Seidel, "Randomized Search Trees," *Proc. of the 30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 540-545.

[2] Chernoff, H., "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations," Annals of Math. Statistics 23, 1952, pp. 493-507.

[3] Chazelle B., Friedman J., "A deterministic view of random sampling and its in geometry", Proc. of the FOCS 88.

[4] K.L. Clarkson, "New applications of random sampling in computational geometry," Discrete and Computational Geometry, 1987, pp. 195-222.

[5] K.L. Clarkson and P. Shor, "Applications of Random Sampling in Computational Geometry, II," Discrete and Computational Geometry, 4, 1989, pp. 387-421.

[6] S. Cheng, R. Janardan, "New results on dynamic planar point location", *Proc. of the FOCS, 1990.*

[7] D. Dobkin and D. Kirkpatrick, "A linear time algorithm for determining the separation of convex polyhedra," "Journal of Algorithms, 6(3), 1985, pp. 381-392.

[8] H. Edelsbrunner, J. O'Rourke, R. Seidel, "Constructing arrangements of lines and hyperplanes with applications", *SIAM J. Computing, 15*, 1986, pp. 341-363.

[9] O. Fries, K. Mehlhorn, S. Naeher, "Dynamization of geometric data structures", *Proc. of the first ACM Symp. on Comp. Geom., 1985, pp. 168-176.*

[10] D. Haussler and E. Welzl, "ϵ-nets and Simplex range queries," Discrete and Computational Geometry, 2(2), 1987, pp. 127-152.

[11] K. Mulmuley, "Randomized multidimensional search trees: dynamic sampling", *in this volume.*

[12] M. Overmars, "The design of dynamic data structures", *Lecture notes in computer science, Springer-Verlag, 1983.*

[13] F. Preparata, R. Tamassia, "Fully dynamic point location in a monotone subdivision", *SIAM J. of Comp., 18 (1989), pp. 811-830.*

[14] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," Communications of the ACM, Volume 33 Number 6, June 1990, pp. 668-676.

[15] J.H. Reif and S. Sen, "Optimal randomized parallel algorithms for computational geometry," Proc. of the 16th International conference on Parallel Processing, 1987, *full version to appear in Algorithmica.*

[16] Sarnak N., Tarjan R., "Planar point location using persistent search trees", *Comm. ACM 29(1986), 669-679.*