A Dynamic Data Structure for 3-d Convex Hulls and 2-d Nearest Neighbor Queries*

Timothy M. Chan†

September 15, 2009

Abstract

We present a fully dynamic randomized data structure that can answer queries about the convex hull of a set of n points in three dimensions, where insertions take $O(\log^3 n)$ expected amortized time, deletions take $O(\log^6 n)$ expected amortized time, and extreme-point queries take $O(\log^2 n)$ worst-case time. This is the first method that guarantees polylogarithmic update and query cost for arbitrary sequences of insertions and deletions, and improves the previous $O(n^\varepsilon)$-time method by Agarwal and Matoušek a decade ago. As a consequence, we obtain similar results for nearest neighbor queries in two dimensions and improved results for numerous fundamental geometric problems (such as levels in three dimensions and dynamic Euclidean minimum spanning trees in the plane).

1 Introduction

One of the classic problems in geometric data structures is *nearest neighbor search* (often dubbed the "post office" problem): preprocess a set P of n point sites so that we can quickly report the closest site to any given query point. In two dimensions, an optimal solution with $O(n \log n)$ preprocessing time and $O(\log n)$ query time has long been known and represents one of the major early achievements in computational geometry [4, 32]. However, the *dynamic* version in which we are allowed to insert and delete sites yields a much different story and is surprisingly still open: while the one-dimensional problem can be solved with $O(\log n)$ update time and $O(\log n)$ query time by standard balanced search trees, no method with the same performance is known in two dimensions.

We briefly review previous work on this dynamic 2-d problem. A straightforward sublinear method with roughly $O(\sqrt{n})$ update and query time was known early on [3]. At FOCS'92, Agarwal and Matoušek [2] presented the first improvements. In the 2-d case, they gave two methods, one with $O(\log n)$ query time and $O(n^\varepsilon)$ amortized update time, and another with $O(n^\varepsilon)$ query time and $O(\log^2 n)$ amortized update time. Here, ε is an arbitrary but fixed positive constant. There has been no further general improvement since (up to now); in particular, no existing method achieves polylogarithmic query and update time simultaneously.

Various special cases have been studied, though. When *only insertions* are allowed, a standard technique [3] yields an $O(\log^2 n)$ amortized bound for updates and queries. The same result extends

*A preliminary version of this paper has appeared in *Proc. 17th ACM-SIAM Sympos. Discrete Algorithms*, pages 1196–1202, 2006.

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). This work has been supported by NSERC.

to the *off-line* case (where the update sequence may contain both insertions and deletions but we are given the entire sequence in advance) and the *semi-online* case (where during the insertion of an element, we are given the time when that element will be deleted) [17].

Several researchers—Clarkson, Mehlhorn, and Seidel [14], Devillers, Meiser, and Teillaud [15], Schwarzkopf [34], and Mulmuley [30]—explored the case where the update sequence is *random* in a certain sense (for example, the insertion order is equally likely to be any permutation of the elements, and each deletion is equally likely to be applied to any of the current elements). For 2-d nearest neighbor search, updates can be performed in optimal $O(\log n)$ expected time, while queries can be answered in $O(\log^2 n)$ time [30]. These average-case analyses suggest that good running times may not be too difficult to achieve in practice. However, these analyses reveal nothing about the running time for a specific update sequence, or for an update sequence performed during the execution of an algorithm (which is hardly random). This is disappointing, as one of the motivations for designing data structures, after all, is that they can be used within algorithms to solve static problems.

In this paper, we present the first general data structure for dynamic 2-d nearest neighbor search that has polylogarithmic update and query cost for any (on-line) sequence of updates. The expected amortized insertion time is $O(\log^3 n)$, the expected amortized deletion time is $O(\log^6 n)$, and the worst-case query time is $O(\log^2 n)$; the data structure can be built in $O(n \log^2 n)$ expected time. As usual, expectation is with respect to the random choices made within the algorithm. Our algorithm uses randomization in only a mild way: an adversary may even have access to the random choices made by earlier operations as he/she devises an update sequence. Our algorithm is simpler than Agarwal and Matoušek's [2].

Our result (like Agarwal and Matoušek's) is actually derived for a more general problem:

> Maintain a dynamic set P of points in 3-d so that we can quickly answer *extreme-point* queries for the *convex hull* of P, i.e., finding a hull vertex that is extreme along a given direction.

By a well-known lifting transformation [4], 2-d nearest neighbor queries reduce to such 3-d queries. Other kinds of queries about the 3-d convex hull can also be answered in polylogarithmic time. (See Section 6 for more details.) We comment that for the analogous dynamic 2-d convex hull problem, a polylogarithmic method was known since the early 1980s; Overmars and van Leeuwen's original $O(\log^2 n)$ time bound [31] was later improved to $O(\log^{1+\varepsilon} n)$ by this author [8] and eventually to $O(\log n)$ by Brodal and Jacob [5] for extreme-point queries (for other kinds of queries, such as intersecting the convex hull with a given line, the best time bound is currently $O(\log^{3/2} n)$ [8]). The techniques we use for the 3-d problem turn out to be quite different from those used for the 2-d problem.

It is hard to overstate the importance of convex hulls in computational geometry. Among the many consequences of our result include improved dynamic data structures for ray shooting queries in a 3-d intersection of halfspaces, 3-d linear programming, 2-d smallest enclosing circle, 2-d diameter, 2-d bichromatic closest pair, 2-d Euclidean minimum spanning trees, and 2-d incremental width. Our result also automatically leads to improved (static) algorithms for computing the convex layers of a 3-d point set and the k-level of a 3-d set of planes. (See Section 7 for more information.)

We briefly point out the main difficulties in designing a data structure for dynamic 3-d convex hulls. Insertions can be treated by standard techniques [3], so we can focus on deletions. Even in the simplest case where all points are in convex position, we cannot just maintain the 3-d convex hull explicitly: although a random vertex has constant average degree (since the total degree is

linear), we have no control over which vertex will be deleted next and it may have high degree. Thus, the convex hull could experience large structural change in every deletion. One idea is to initially remove all "bad" vertices of degree higher than a certain constant (since there are only a fraction of such vertices) and handle them in a separate subset (since we can answer an extreme-point query by querying each subset separately). This idea needs refinement, however, since the low-degree vertices themselves may have high degrees in their own convex hull. We end up using a multi-layer approach and a quantity different from the degree (namely, the number of "conflict lists" that the vertex participates in). Although the description of the final method is not long (we even provide pseudocode), it does require a number of carefully orchestrated ideas, drawing inspiration from several previous approaches, for example, one of Agarwal and Matoušek's dynamic data structures [2], the author's static data structure for halfspace range reporting [7], and Eppstein's dynamic closest pair technique [19]. (See Section 5 for more discussion.)

2 Preliminaries

As noted earlier, it suffices to consider the dynamic 3-d convex hull problem, because 2-d nearest neighbor queries reduce to 3-d extreme-point queries. It suffices to consider the upper hull, since the lower hull can be treated symmetrically and we can query the lower and upper hulls separately. The dual version [4, 30, 32] of the problem is more convenient to study and is equivalent to *vertical ray shooting in a lower envelope of planes*:

> Maintain a dynamic set H of n planes in 3-d so that given a vertical line q, we can quickly report the lowest plane at q, denoted by $\text{ANS}(H, q)$.

We assume that the input planes are in general position, by standard perturbation techniques.

Our method is self-contained with the exception of the use of an optimal data structure for the static problem, and one well-known geometric tool—cuttings [12, 30]. Given a set H of n planes and a region γ, a $(1/r)$-*cutting of* γ is a collection of disjoint open cells (tetrahedra) such that the union of the closure of the cells contains γ and each cell intersects at most $\lceil n/r \rceil$ planes of H. The *conflict list* of a cell Δ refers to the subset of all planes of H intersecting Δ and is denoted by H_Δ.

The $(\leq k)$-*level* refers to the closure of the region of all points q such that at most k planes of H are strictly below q. The following lemma states that a cutting of small size exists covering a large number of levels.

Lemma 2.1 *For any set H of n planes and parameter r, there exists an $O(1/r)$-cutting of the $(\leq n/r)$-level of H into $O(r)$ cells. We can construct this cutting, along with the conflict lists of all cells, in $O(n \log n)$ expected time. The cells are (unbounded) tetrahedra and are* vertical *in the sense that they contain the point $(0, 0, -\infty)$.*

The above is a special case of Matoušek's "shallow cutting lemma" [25]. The proof was by sampling: roughly speaking, we take a random sample $R \subseteq H$ of size about r, form a canonical triangulation of the arrangement of R, and return the cells that intersect the $(\leq n/r)$-level. The conflict list of each cell would then have size close to $O(n/r)$; extra steps are required to make it truly $O(n/r)$. The $O(n \log n)$ expected running time was obtained by an algorithm of Ramos [33]. That vertical cells are sufficient for this particular case of the shallow cutting lemma was observed by the author [10, Lemma 3.1].

Our solution to the dynamic lower envelope problem is described in two sections. Section 3 contains a deletion-only data structure that solves the problem only "partially" for some queries; this section is where Lemma 2.1 comes into play. (Incidentally, Lemma 2.1 is the only place that requires randomization.) Section 4 then uses this intermediate solution to obtain the fully dynamic data structure; this section is essentially geometry-free.

3 A Partial Data Structure That Supports Deletions

We first describe an incomplete data structure that supports deletions for a set S of n planes. The data structure is incomplete in the sense that it can only handle queries when the answers belong to a certain subset $S_{\text{live}} \subseteq S$ of "live" members. Initially, we guarantee that at least half of the planes are in S_{live}. Subsequently, each deletion can "kill" off only a small (polylogarithmic amortized) number of additional planes from S_{live}. A static subset S_{static} is used to handle queries.

The construction of the data structure consists of logarithmically many layers, where in each layer we take a cutting of a certain size and remove "bad" planes that cause too many conflicts. The precise pseudocode is given below. (All logarithms are in base 2, and c and c' are appropriate integer constants.)

construct(S):

1. $S_0 = S$
2. for $i = 1, \ldots, \lceil \log n \rceil$ do {
3. $\quad T_i$ = a $(1/2^i)$-cutting of the $(\leq \lceil |S_{i-1}|/c2^i \rceil)$-level of S_{i-1} into $c'2^i$ vertical cells
4. $\quad S_i = S_{i-1} - \{$all planes that intersect more than $4c'\lceil \log n \rceil$ cells of $T_i\}$
5. $\quad$ for each $\Delta \in T_i$ do
6. $\quad\quad$ compute the conflict list $(S_i)_\Delta$ and set $j_\Delta = 0$

   }
7. $S_{\text{live}} = S_{\text{static}} = S_{\lceil \log n \rceil}$

Each deletion of a plane involves examining the conflict lists that the plane participates in. When a fraction of the members of a conflict list has been deleted, we kill off all remaining members of the list. (The motive behind this unusual strategy will be revealed later, in the proof of Lemma 3.1(c).) Fix a constant $c'' > 4c$. The procedure below is to be executed regardless of whether the plane h to be deleted is live or dead.

delete(S, h):

1. remove h from S, and remove h from S_{live} (if it is in S_{live})
2. for each i and each $\Delta \in T_i$ with $h \in (S_i)_\Delta$ do {
3. $\quad j_\Delta = j_\Delta + 1$
4. $\quad$ if $j_\Delta = \lceil |(S_i)_\Delta|/c'' \rceil$ then remove all planes in $(S_i)_\Delta$ from S_{live}

   }

We establish the key properties of this data structure:

Lemma 3.1

(a) construct() *takes* $O(n \log^2 n)$ *expected time and puts at least* $n/2$ *planes in* S_{live}.

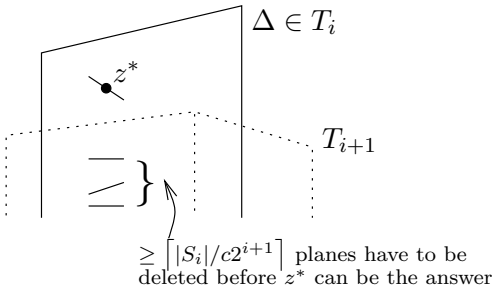

Figure 1: Proof of Lemma 3.1(c).

(b) *Subsequently,* n_D delete() *operations take* $O(n_D \log^2 n)$ *time and remove at most* $O(n_D \log^2 n)$ *planes*[1] *from* S_{live}.

(c) *For any vertical line* q, *if* $\text{ANS}(S, q) \in S_{\text{live}}$, *then* $\text{ANS}(S, q) = \text{ANS}(S_{\text{static}}, q)$.

Proof:

(a) Because the total number of intersecting pairs of cells of T_i and planes of S_i is at most $c'2^i \cdot \lceil n/2^i \rceil \leq 2c'n$, line 4 removes at most $n/(2\lceil \log n \rceil)$ planes per iteration. Thus, the total number of planes removed is at most $n/2$. The time bound follows by applying Lemma 2.1 in each iteration.

(b) For each i with $h \in S_i$, h intersects at most $4c'\lceil \log n \rceil$ cells $\Delta \in T_i$. Thus, the number of increments of the j_Δ's performed by each delete is $O(\log^2 n)$. Charge each increment c'' units. Since $|(S_i)_\Delta|$ units need to be accumulated before the $\leq |(S_i)_\Delta|$ removals in line 4 can occur, the total number of removals is bounded by the total number of charges $O(n_D \log^2 n)$. The time bound also follows (assuming pointers between planes and conflict lists have been set up).

(c) Let $h^* = \text{ANS}(S, q) \in S_{\text{live}}$ and let $z^* = q \cap h^*$. If z^* lies inside some $\Delta \in T_{\lceil \log n \rceil}$, then since $|(S_{\text{static}})_\Delta| \leq 1$, $h^* = \text{ANS}(S_{\text{static}}, q)$ and we are done. Otherwise, let i be such that z^* lies inside some $\Delta \in T_i$ but outside all cells of T_{i+1} (as in Figure 1). Note that $S_{\text{live}} \subseteq S_{\text{static}} \subseteq S_i$. Since T_{i+1} contains the $(\leq \lceil |S_i|/c2^{i+1} \rceil)$-level of S_i, the number of planes of S_i that are strictly below z^* is at least $\lceil |S_i|/c2^{i+1} \rceil \geq \lceil n/c2^{i+2} \rceil \geq \lceil |(S_i)_\Delta|/4c \rceil$. In order for z^* to be the answer, all these planes must have been deleted from S. However, because of line 4, all planes in $(S_i)_\Delta$, including h^*, would have been removed from S_{live}: a contradiction. □

4 The Fully Dynamic Data Structure

We are now ready to present our data structure that supports both insertions and deletions for a dynamic set H of planes. We maintain a collection of $O(\log n)$ subsets S, each of which is stored in the data structure from Section 3. At all times, we maintain the invariant that the subsets S_{live} are disjoint with $\bigcup_S S_{\text{live}} = H$. However, the subsets S themselves (or S_{static} for that matter) are not

[1] In other words, the amortized number of planes removed is $O(\log^2 n)$. A slight variant of the algorithm, described in the proceedings version of this paper, can actually achieve a worst-case instead of an amortized bound.

necessarily disjoint, because of the presence of dead planes. For each subset S in the collection, we store the lower envelope of S_{static}.

The initial data structure can be built by invoking the construction procedure from Section 3 logarithmically many times until every plane is live in some subset, as required by the invariant:

preprocess(S):
1. if $S \neq \emptyset$ then add subset S, call construct(S), and preprocess($S - S_{\text{live}}$)

To insert a new plane to H, we simply add a new subset of size 1 to the collection. To ensure that the number of subsets is logarithmic, we "clean up" by repeatedly merging subsets of similar sizes (and eliminating dead planes encountered along the way). For a subset S, we define its *depth* to be $\lfloor \log |S_{\text{live}}| \rfloor$. Set $b = 16$.

insert(h):
1. add new subset $S = \{h\}$ with $S_{\text{live}} = S_{\text{static}} = S$
2. while there exist b subsets $S^{(1)}, \ldots, S^{(b)}$ at the same depth do
3. remove subsets $S^{(1)}, \ldots, S^{(b)}$ and call preprocess($S^{(1)}_{\text{live}} \cup \cdots \cup S^{(b)}_{\text{live}}$)

After the clean-up in lines 2–3 above, there can be at most $b - 1$ subsets at each depth, for a total of $O(\log n)$ subsets, as claimed.

To delete a plane h from H, we invoke the deletion procedure from Section 3 to each subset S containing h. (There may be more than one subset, as multiple dead copies of h may exist.) Some planes become dead in S as a result of this procedure. The key idea is to reinsert these newly deceased planes back to the data structure (ending up in other subsets of the collection), thereby restoring the invariant.

delete(h):
1. $A = \emptyset$
2. for each subset S containing h do
3. call delete(S, h), and add all planes just removed from S_{live} to A
4. for each $g \in A - \{h\}$ do call insert(g)

The query algorithm is simple, as vertical ray shooting is "decomposable": we can answer a query by querying each subset in the collection separately.

query(q):
1. $B = \emptyset$
2. for each subset S with $\text{ANS}(S_{\text{static}}, q) \in S_{\text{live}}$ do
3. add $\text{ANS}(S_{\text{static}}, q)$ to B
4. return $\text{ANS}(B, q)$

The description of our data structure is complete. We now provide an amortized analysis:

Theorem 4.1

(a) *Any sequence of n_I* insert() *and n_D* delete() *operations on an initial set of size n_0 takes total expected update time $O(n_0 \log^2 n + n_I \log^3 n + n_D \log^6 n)$, where n is the maximum size of the set at any time.*

(b) query() *answers vertical ray shooting queries correctly in* $O(\log^2 n)$ *time.*

Proof:

(a) Define the *potential* of the data structure to be $\sum_S |S_{\text{live}}| \log |S_{\text{live}}|$. At any moment, the potential is upper-bounded by $n \log n$ due to disjointness of the subsets S_{live}.

The cost of preprocessing a set of n_0 planes is, by Lemma 3.1(a), $O(\sum_{j=0}^{\infty}(n_0/2^j)\log^2(n_0/2^j)) = O(n_0 \log^2 n_0)$.

Regarding insertion, consider one iteration of lines 2–3. Say the sets $S^{(i)}$ are at depth k. The cost of this step is $O(p \log^2 p)$, like above, with $p = |S^{(1)}_{\text{live}}| + \cdots + |S^{(b)}_{\text{live}}|$. Note that $p \geq 2^{k+4}$ for our choice of $b = 16$. The increase in the potential during this step is at least

$$\begin{aligned} \sum_{j=1}^{\infty} \frac{p}{2^j} \log \frac{p}{2^j} - \sum_{i=1}^{b} |S^{(i)}_{\text{live}}| \log |S^{(i)}_{\text{live}}| &\geq p \log p \sum_{j=1}^{\infty} \frac{1}{2^j} - p \sum_{j=1}^{\infty} \frac{j}{2^j} - p(k+1) \\ &\geq p(k+4) - 2p - p(k+1) \;=\; p. \end{aligned}$$

We can thus upper-bound the insertion cost by the potential increase multiplied by a $\log^2 n$ factor.

Regarding deletion, consider lines 1–3. By Lemma 3.1(b), the size of all the A sets generated is at most $O(\log^2 n)$ times the number of calls to delete(S, h). Since this number is $O(n_D \log n)$ over the entire update sequence, the total size of the A's, as well as the cost of the calls to delete(S, h), is $O(n_D \log^3 n)$. However, the potential decreases. Noting that in general $(x+y)\log(x+y) - x \log x \;=\; y \log(x+y) + x \log(1 + y/x) \;\leq\; y \log(x+y) + x \cdot O(y/x) \;=\; y \log(x+y) + O(y)$, we see that the decrease in potential during this step is at most

$$|S_{\text{live}}| \log |S_{\text{live}}| - (|S_{\text{live}}| - |A|) \log(|S_{\text{live}}| - |A|) \;\leq\; O(|A| \log |S_{\text{live}}|) \;\leq\; O(|A| \log n).$$

Over the entire update sequence, the total potential decrease is thus $O(n_D \log^4 n)$. The running time of line 4 is bounded by the potential increase times $\log^2 n$, as already shown above.

Therefore, the total cost of the entire update sequence is bounded by $O(n_D \log^3 n +$ (total potential increase) $\log^2 n)$, where

$$\begin{aligned} \text{total potential increase} &= (\text{final potential} - \text{initial potential}) + \text{total potential decrease} \\ &= O(n \log n - n_0 \log n_0 + n_D \log^4 n) \;=\; O(n_I \log n + n_D \log^4 n). \end{aligned}$$

We thus obtain the desired time bound.

(b) Correctness is easy to see: by the invariant, $\text{ANS}(H, q)$ is in S_{live} for some subset S, and by Lemma 3.1(c), $\text{ANS}(H, q) = \text{ANS}(S, q) = \text{ANS}(S_{\text{static}}, q)$. The running time follows since for each of the $O(\log n)$ subsets S, $\text{ANS}(S, q)$ takes $O(\log n)$ time by known static data structures for vertical ray shooting in 3-d lower envelopes (a static planar point location problem [4, 32]). □

Note the slightly unconventional definition of n above; it is easy to make n the current size of H, by applying a standard amortization trick [29] (namely, whenever the value of n has doubled or halved, rebuild the whole data structure from scratch).

5 Remarks

We note how various ideas used in our data structure are related to previous approaches:

- The idea of taking $O(\log n)$ layers of samples/cuttings is akin to skip lists. More specifically, our construction in Section 3 resembles a static data structure of the author [7] for 3-d halfspace range reporting. There, conflict lists of triangulations of samples also play a vital role, but without the removal of bad planes (nor the subsequent removal of planes from conflict lists).

- The idea of removing a fraction of bad planes and treating them in separate disjoint subsets stems from one of Agarwal and Matoušek's original dynamic data structures for lower envelopes [2]. Agarwal and Matoušek also used cuttings (of $O(n^\varepsilon)$ size) but in a top-down recursive fashion, which inevitably incurred some loss of efficiency. Our multi-layer usage of cuttings of different sizes avoids this problem.

- The idea of decomposing a dynamic set into disjoint subsets and handling insertions by repeated merging is standard [3] and is sometimes called the "logarithmic method". Our data structure in Section 4 is more delicate, however, as we let deletions trigger multiple reinsertions (and also let dead elements remain in each subset). Traces of this nontrivial form of the logarithmic method can be seen in Eppstein's technique [19, 20] for dynamic closest-pair-type problems, and Section 4 is inspired by his technique. (In Eppstein's closest-pair solution, the partial data structure involved is the "conga line"; there, a deletion triggers only one reinsertion per subset.)

6 More Queries

Our data structure can handle other kinds of queries besides vertical ray shooting in the lower envelope of planes.

Nonvertical ray shooting. Given a *nonvertical* line q, we can find the topmost (or bottommost) point of q that lies on the lower envelope of H (if such a point exists). As can be easily seen, our query algorithm works immediately for this more general form of ray shooting, with no change at all in the description and correctness proof (except for the use of an optimal static data structure for nonvertical ray shooting, as provided by Dobkin and Kirkpatrick [16]). We can confirm that the query line q does intersect the lower envelope, by checking whether the returned point on q lies on the lower envelope (which reduces to answering a vertical ray shooting query at the point). The query time remains $O(\log^2 n)$.

The same result holds for nonvertical ray shooting in the intersection of a dynamic set of halfspaces in 3-d, by querying the lower halfspaces and upper halfspaces separately.

Back in primal space, ray shooting queries correspond to *gift-wrapping* queries: given a line q (not intersecting the convex hull), find the two planes through q that are tangent to the convex hull.

Linear programming queries. For a dynamic set of halfspaces in 3-d, a linear programming query asks for a point inside the intersection that is extreme along a given direction. (See [6, 18] for earlier results on the semi-online case.) Because these queries are not decomposable, we cannot directly apply our query algorithm.

Matoušek [26] showed how linear programming queries reduce to ray shooting queries via a multidimensional parametric search, without any change to the data structure. Specifically, suppose that there is a parallel query algorithm for vertical ray shooting that uses $\pi(n)$ processors and runs in $\tau(n)$ time. Let $t_k(n)$ be the time required to solve a "k-dimensional" linear programming query, i.e., finding a point inside the intersection that is restricted within a given k-flat and extreme along a given direction. Matoušek obtained $t_k(n) = O(t_{k-1}(n)\tau(n)\log\pi(n) + (\tau(n)\log\pi(n))^{k+1})$.

Our vertical ray shooting algorithm is clearly parallelizable with $\tau(n) = O(\log n)$ and $\pi(n) = O(\log n)$. Our nonvertical ray shooting algorithm provides the base case $t_1(n) = O(\log^2 n)$. We can thus solve general linear programming queries in $t_3(n) = O(\log^4 n \log^4 \log n)$ time. (The query time can probably be improved with more work.)

Back in primal space, we can use linear programming queries to *intersect the convex hull with a line* within the same time.

k-lowest-planes queries and halfspace range reporting. Next, we consider the following generalization of vertical ray shooting: given a vertical line q and an integer k, report the k lowest planes at q. (See [7, 33] for earlier results on the static case.) By repeated vertical shooting and deletions, we can answer such a query in $O(k\log^6 n)$ expected amortized time. We show that the time bound can be lowered by a direct modification of the algorithm. Specifically, we use the following extension of Lemma 3.1(c), assuming $c'' > 8c$:

Lemma 6.1 *For any vertical line q, if h^* is among the k lowest planes of S at q and $h^* \in S_{\rm live}$, then h^* is among the $8ck$ lowest planes of $S_{\rm static}$ at q.*

Proof: Let $z^* = q \cap h^*$. If z^* lies inside some $\Delta \in T_{\lceil\log(n/8ck)\rceil}$, then since $|(S_{\rm static})_\Delta| \le 8ck$, the conclusion follows. Otherwise, let $i < \log(n/8ck)$ be such that z^* lies inside some $\Delta \in T_i$ but outside all cells of T_{i+1} (as in Figure 1). Note that $k \le n/c2^{i+3}$ and $S_{\rm live} \subseteq S_{\rm static} \subseteq S_i$. Let K be the k lowest planes of S at q. Since T_{i+1} contains the $(\le \lceil |S_i|/c2^{i+1}\rceil)$-level of S_i, the number of planes of $S_i - K$ that are strictly below z^* is at least $\lceil |S_i|/c2^{i+1}\rceil - |K| \ge \lceil n/c2^{i+2}\rceil - k \ge \lceil n/c2^{i+3}\rceil \ge \lceil |(S_i)_\Delta|/8c\rceil$. In order for h^* to be in K, all these planes must have been deleted from S. However, because of line 4, all planes in $(S_i)_\Delta$, including h^*, would have been removed from $S_{\rm live}$: a contradiction. □

By the lemma, to answer a query, it suffices to examine the $8ck$ lowest planes in $S_{\rm static}$ over every subset S in the collection. With known static data structures for k-lowest-planes queries [7], these planes can be generated in $O(\log n + k)$ time per subset, for a total of $O((\log n + k)\log n)$. We can skip over planes not in $S_{\rm live}$, and return the k lowest among all $O(k\log n)$ live planes generated. The overall query time is $O(\log^2 n + k\log n)$.

The problem of reporting all planes below a query point reduces to the above queries and can also be solved in $O(\log^2 n + k\log n)$ time, where k denotes the output size of the query. This follows from a standard trick of "guessing" the output size (e.g., see [7, proof of Corollary 2.5]).

Back in primal space, the problem is equivalent to *halfspace range reporting* (finding all points inside a query halfspace) for a dynamic 3-d point set. By the lifting transformation, we can also find the k *nearest/farthest neighbors* to a query point, or report all points inside/outside a query circle, for a dynamic 2-d point set, in the same amount of time.

7 Applications

We list some of the many consequences of our new data structure:

- We can maintain the *smallest enclosing circle* of a dynamic 2-d point set in expected amortized time $O(\log^6 n)$, improving the previous $O(n^\varepsilon)$ bound [2]. This problem lifts to convex programming over a halfspace intersection, which is similar enough to linear programming queries and can be handled by the same multidimensional parametric search technique [26].

- We can maintain the *diameter* as well as the *bichromatic closest/farthest pair* of a dynamic 2-d point set in $O(\log^8 n)$ expected amortized time (instead of $O(n^\varepsilon)$ [2]). This follows from Eppstein's technique [19]: any dynamic data structure for nearest/farthest neighbor queries with $O(t(n))$ update and query time can be converted into a dynamic data structure for closest/farthest pair with $O(t(n)\log^2 n)$ amortized time.

- We can maintain the *Euclidean minimum spanning tree* (EMST) of a dynamic 2-d point set in $O(\log^{10} n)$ expected amortized time. This is a consequence of a known reduction to bichromatic closest pair by Eppstein [19]: a dynamic data structure for bichromatic closest pair with $t_{\text{bcp}}(n)$ amortized time and a dynamic data structure for minimum spanning trees for graphs with $t_{\text{mst}}(n)$ amortized time can be combined to yield a dynamic EMST algorithm with $O((t_{\text{bcp}}(n)+t_{\text{mst}}(n))\log^d n)$ amortized time. Here, $d = 2$ and, by known graph results [23], $t_{\text{mst}}(n) = O(\log^4 n)$.

- We can maintain the *all-nearest-neighbors graph* of a dynamic 2-d point set (where we place a directed edge from each point to its nearest neighbor) in $O(\log^6 n)$ expected amortized time. This follows from a reduction of the problem to vertical ray shooting for a dynamic lower envelope noted in [9].

- We can maintain the *width* of a 2-d point set under *insertions* (without deletions) in $O(\log^7 n)$ expected amortized time (instead of $O(n^\varepsilon)$). This follows from a method of Eppstein [21], which requires an amortized $O(\log n)$ number of updates and vertical ray shooting queries for a dynamic 3-d halfspace intersection.

- We can obtain a randomized algorithm for the 3-d *convex layers* (or "convex hull peeling depth") problem that runs in $O(n\log^6 n)$ expected time, improving Agarwal and Matoušek's $O(n^{1+\varepsilon})$ bound [2]. This follows from the standard gift-wrapping algorithm that constructs the layers with $O(n)$ deletions and queries. Compare our result with Chazelle's $O(n\log n)$ algorithm for the 2-d convex layers problem back in 1985 [11].

- We can obtain a randomized algorithm for constructing the *k-level* for a 3-d arrangement of n planes that runs in $O(n\log^2 n + f\log^6 n)$ expected time, improving the previous $O(n^{1+\varepsilon}f)$ bound [2], where f denotes the output size. This follows as the k-level can be constructed by performing $O(f)$ insertions, deletions, and nonvertical ray shooting queries.

  According to the current combinatorial bound on the k-level in 3-d [36], $f = O(nk^{3/2})$ always. The author [7] showed how to speed up any $O(T(n))$-time k-level algorithm to run in $O(n\log n + (n/k)T(k))$ expected time. By plugging in $T(n) = O(n^{5/2}\log^6 n)$, we now have an $O(n\log n + nk^{3/2}\log^6 k)$-time randomized algorithm.

- We can report all *local minima* of the first k levels for a 3-d arrangement of n planes in $O(n \log^2 n + k^3 \log^6 n)$ expected time. This follows from a method of Matoušek [27], which requires $O(k^3)$ insertions, deletions, and linear programming queries.

8 Modifications

A more practical version? Our method is easy to implement except for the cutting step (Lemma 2.1). Unfortunately, Ramos' cutting algorithm [33] is fairly complicated. We point on an alternative that is simpler and uses lower envelopes of samples as its only tool. The algorithm becomes slower by polylogarithmic factors but achieves high-probability bounds:

Lemma 8.1 *Let H be a set of n planes, let $r \leq n$ and $N \geq n$, and let c_0 be any constant. Take $c_0 \log N$ independent random samples of H, each of size $r/2$, and consider the vertical decomposition of the region underneath the lower envelope of each sample. With probability at least $1 - 1/N^{c_0-3}$, the $O(r \log N)$ cells of these $O(\log N)$ decompositions cover the $(\leq n/r)$-level of H, and the conflict list of each cell has size $O((n/r) \log N)$. We can construct the cells, along with their conflict lists, in $O(n \log n \log N)$ time.*

Proof: That each cell has size $O((n/r) \log N)$ with high probability is well-known; e.g., see [13]. To see that the cells cover the $(\leq n/r)$-level with high probability, fix a vertex v in the arrangement of H with $k \leq n/r$ planes below it. The probability that v is above the lower envelope of one sample is at most $(r/2)(k/n) \leq 1/2$. The probability that v is above all lower envelopes is at most $(1/2)^{c_0 \log N} = 1/N^{c_0}$. There are $\binom{n}{3}$ vertices in the arrangement. □

Note that in Section 3, it is not important that cells in T_i are disjoint. We can thus use the above lemma in place of Lemma 2.1 to compute each T_i and adjust parameters by logarithmic factors in the algorithm and the analysis accordingly. This version of the data structure achieves $O(\text{polylog } N)$ update and query time w.h.p. (i.e., with probability at least $1-1/N^{c_0}$ for an arbitrarily large constant c_0); we set N to be the total number of planes.

The resulting query algorithm is Monte Carlo, however: in the above lemma, although we can easily confirm that all conflict lists have small sizes (and if not, resample), we do not have an efficient way to test whether the cells indeed cover the $(\leq n/r)$-level. Still, we can obtain a Las Vegas version of the query algorithm with additional ideas as follows.

Given the query line q, suppose that the Monte Carlo algorithm returns the plane h. Let $z = q \cap h$. For each subset S in the collection, we want to confirm that no plane in S_{live} is strictly below z. We first find the largest i such that z lies inside some $\Delta \in T_i$. This can be done in polylogarithmic time, by planar point location queries for each lower envelope. If $i = \lceil \log n \rceil$, we can confirm directly that none of the $O(\log N)$ planes in $(S_{\text{static}})_\Delta$ is in S_{live} and strictly below z. Otherwise, we confirm that $j_\Delta \geq \lceil |(S_i)_\Delta|/4c \rceil$; by the same argument as in the proof of Lemma 3.1(c), we know that this condition is true if the cells in T_{i+1} indeed cover the $(\leq \lceil |S_i|/c2^{i+1} \rceil)$-level of S_i, i.e., this is true w.h.p. Since this condition implies that none of the planes in $(S_i)_\Delta$ is in S_{live}, this confirmation ensures correctness. Since we know that confirmation is successful w.h.p., whenever it fails, we can switch to a brute-force search. The resulting query algorithm runs in polylogarithmic time w.h.p. and always returns the correct answer.

An $O(n \log\log n)$-space version. Our method requires $\Omega(n \log n)$ space, since the total size of the conflict lists stored in one partial data structure is $\Omega(n)$ for each T_i. We now describe a more complicated variant of the data structure that uses less space, based on a suggestion by Peyman Afshani (personal communication).

In the partial data structure in Section 3, the idea is to store just the size of the conflict lists $(S_i)_\Delta$, not the lists themselves. This requires only $O(\sum_i |T_i|) = O(n)$ space. In line 2 of the delete(S, h) procedure, for each i with $h \in S_i$, we need to find all cells $\Delta \in T_i$ intersecting h. Since Δ intersects h iff one of its (three) vertices lies above h, this can be accomplished by finding all vertices of T_i that lie above h—a halfspace range reporting query. In line 4 of delete(S, h), we need to re-generate the conflict list $(S_{\text{static}})_\Delta$ and remove its planes from S_{live} (note that planes in $(S_i)_\Delta - (S_{\text{static}})_\Delta$ need not be removed, as they are not in S_{live}). Since h intersects Δ iff h lies below one of its (three) vertices, we can compute $(S_{\text{static}})_\Delta$ by halfspace range reporting queries on S_{static} in dual space. By using the current best static method for halfspace range reporting [7, 33], the space requirement is $O(n \log\log n)$.

We need one more change, to ensure that $|S_{\text{live}}| = \Theta(n)$: in the partial data structure, as soon as $|S_{\text{live}}| \leq n/4$, remove everything from S_{live} and destroy the structure. Lemma 3.1(b) still holds (since this condition can occur only after $\Omega(n/\log^2 n)$ deletions, in which case we can afford $O(n)$ more removals). Since one partial data structure now requires $O(|S_{\text{live}}| \log\log |S_{\text{live}}|)$ space, by disjointness of the S_{live}'s, the whole data structure requires $O(n \log\log n)$ space.

Dynamic lower envelopes of curves in 2-d. Our method can handle nonvertical ray shooting queries in a lower envelope of x-monotone curves in 2-d, assuming that each pair of curves intersects at most a constant s number of times. This is interesting, as most known dynamic data structures for 2-d lower envelopes of lines (i.e., 2-d convex hulls), including the classical method by Overmars and van Leeuwen [31], do not extend to the case of curves.

All we need is a generalization of Lemma 2.1:

Lemma 8.2 *For any set H of n curves in the plane and parameter r, there exists an $O(1/r)$-cutting of the $(\leq n/r)$-level of H into $O(r\beta(n))$ cells. We can construct this cutting, along with the conflict lists of all cells, in $O(n\beta(n)\log n)$ expected time. The cells are all x-monotone regions unbounded from below (but not necessarily of constant complexity). Here, $\beta(\cdot)$ is an inverse-Ackermann-like function (depending on s).*

Proof: Pick a random integer $k \in [n/r, 2n/r)$ and compute the k-level L of H. Draw downward vertical rays at every (n/r)-th vertex of L. Return the cells formed by partitioning the region underneath L with these rays. (The idea of using levels to construct cuttings is not new; e.g., see [24].)

It is known [35] that the $(\leq 2n/r)$-level has $O(n(n/r)\beta(n))$ vertices in total. Thus, L has an expected $O(n\beta(n))$ number of vertices. (If the actual number exceeds this bound by a constant factor, we can repick k.) The number of cells is therefore $O(n\beta(n)/(n/r)) = O(r\beta(n))$.

For each cell Δ, a curve intersects Δ iff it intersects one of the two bounding rays or passes through a vertex of L between the two rays. The number of such curves is therefore at most $2n/r + 2n/r + n/r = O(n/r)$.

We can construct the k-level in $O(n\beta(n)\log n)$ expected time by an algorithm of Har-Peled [22]. By sweeping the vertices of L from left to right, we can keep track of the subset of curves intersecting each downward ray and therefore generate the conflict lists within the same time bound. □

Note that in Section 3, it is not important that cells in T_i have constant complexity. We can thus use the above lemma to compute each T_i's and adjust parameters by inverse-Ackermann-like factors in the algorithm and the analysis accordingly. Static vertical ray shooting for curves can still be done in logarithmic time by binary search. The resulting data structure has $O(\beta(n)\log^3 n)$ amortized insertion time, $O(\beta(n)^2\log^6 n)$ amortized deletion time, and $O(\log^2 n)$ query time.

9 Open Problems

The most obvious open problem is to reduce the number of logarithmic factors in our results. It seems plausible that one logarithmic factor could be removed from the insertion and deletion time, by improving the $O(n\log^2 n)$ construction time bound in Section 3: Ramos [33] actually showed how to compute $O(\log n)$ cuttings of different sizes in $O(n\log n)$ total expected time; what prevents us from immediately applying this result is that we remove bad planes each time we move on to the next cutting. In any case, reducing the update time all the way to $O(\log n)$ appears extremely difficult; even for the insertion-only case, an optimal method is currently not known.

It might be possible to de-amortize our time bounds by known tricks [29]. Obtaining a deterministic polylogarithmic method remains open.

In a recent development [1], a more space-efficient data structure has been found for the static 3-d halfspace range reporting problem. By this result and the ideas from Section 8, the space bound for our dynamic 3-d data structure can be further improved from $O(n\log\log n)$ to $O(n)$.

Finally, we point out that Agarwal and Matoušek [2] actually described their dynamic data structures for lower envelopes of hyperplanes in any fixed dimension. Bounds get much worse as soon as the dimension exceeds 3, even for static data structures [25, 28]. Still, one may ask whether some of the n^ε factors in Agarwal and Matoušek's higher-dimensional bounds could be replaced by polylogarithmic factors. Our method does not seem to yield new results here.

Acknowledgement. I thank Peyman Afshani for useful discussions on how to improve the space bound.

References

[1] P. Afshani and T. M. Chan. Optimal halfspace range reporting in three dimensions. In *Proc. 20th ACM-SIAM Sympos. Discrete Algorithms*, pages 180–186, 2009.

[2] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13:325–345, 1995.

[3] J. Bentley and J. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2000.

[5] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002.

[6] T. M. Chan. Deterministic algorithms for 2-d convex programming and 3-d online linear programming. *J. Algorithms*, 27:147–166, 1998.

[7] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM J. Comput.*, 30:561–575, 2000.

[8] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[9] T. M. Chan. Semi-online maintenance of geometric optima and measures. *SIAM J. Comput.*, 32:700-716, 2003.

[10] T. M. Chan. Low-dimensional linear programming with violations. *SIAM J. Comput.*, 34:879–893, 2005.

[11] B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31:509–517, 1985.

[12] B. Chazelle. Cuttings. In *Handbook of Data Structures and Applications* (D. P. Mehta and S. Sahni, eds.), CRC Press, pages 25.1–25.10, 2005.

[13] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[14] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3:185–121, 1993.

[15] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. *Comput. Geom. Theory Appl.*, 2:55–80, 1992.

[16] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

[17] D. P. Dobkin and S. Suri. Maintenance of geometric extrema. *J. ACM*, 38:275–298, 1991.

[18] D. Eppstein. Dynamic three-dimensional linear programming. *ORSA J. Comput.*, 4:360–368, 1992.

[19] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:111–122, 1995.

[20] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Experimental Algorithmics*, 5:1–23, 2000.

[21] D. Eppstein. Incremental and decremental maintenance of planar width. *J. Algorithms*, 37:570–577, 2000.

[22] S. Har-Peled. Taking a walk in a planar arrangement. *SIAM J. Comput.*, 30:1341–1367, 2000.

[23] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[24] J. Matoušek. Construction of ε-nets. *Discrete Comput. Geom.*, 5:427–448, 1990.

[25] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2:169–186, 1992.

[26] J. Matoušek. Linear optimization queries. *J. Algorithms*, 14:432–448, 1993. Also with O. Schwarzkopf in *Proc. 8th ACM Sympos. Comput. Geom.*, pages 16–25, 1992.

[27] J. Matoušek. On geometric optimization with few violated constraints. *Discrete Comput. Geom.*, 14:365–384, 1995.

[28] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete Comput. Geom.*, 10:215–232, 1993.

[29] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Heidelberg, 1984.

[30] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

[31] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.*, 23:166–204, 1981.

[32] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[33] E. Ramos. On range reporting, ray shooting, and k-level construction. In *Proc. 15th ACM Sympos. Comput. Geom.*, pages 390–399, 1999.

[34] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd IEEE Sympos. Found. Comput. Sci.*, pages 197–206, 1991.

[35] M. Sharir. On k-sets in arrangements of curves and surfaces. *Discrete Comput. Geom.*, 6:593–613, 1991.

[36] M. Sharir, S. Smorodinsky, and G. Tardos. An improved bound for k-sets in three dimensions. *Discrete Comput. Geom.*, 26:195–204, 2001.