Tight Bounds for Dynamic Convex Hull Queries (Again)

Erik D. Demaine Mihai Pătraşcu
MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar St., Cambridge, MA 02139, USA,
{edemaine,mip}@mit.edu

ABSTRACT

The dynamic convex hull problem was recently solved in $O(\lg n)$ time per operation, and this result is best possible in models of computation with bounded branching (e.g., algebraic computation trees). From a data structures point of view, however, such models are considered unrealistic because they hide intrinsic notions of information in the input.

In the standard word-RAM and cell-probe models of computation, we prove that the optimal query time for dynamic convex hulls is, in fact, $\Theta(\frac{\lg n}{\lg \lg n})$, for polylogarithmic update time (and word size). Our lower bound is based on a reduction from the marked-ancestor problem, and is one of the first data structural lower bounds for a nonorthogonal geometric problem. Our upper bounds follow a recent trend of attacking nonorthogonal geometric problems from an information-theoretic perspective that has proved central to advanced data structures. Interestingly, our upper bounds are the first to successfully apply this perspective to dynamic geometric data structures, and require substantially different ideas from previous work.

Categories and Subject Descriptors

E.1 [**Data Structures**]

General Terms

Algorithms, Performance, Theory

Keywords

dynamic convex hull, bounded precision, word RAM

SCG'07, June 6–8, 2007, Gyeongju, South Korea.

1. INTRODUCTION

1.1 Putting the "computational" in computational geometry

A central issue in computational geometry is the discrepancy between the idealized *geometric view* of limited objects with infinite precision, and the realistic *computational view* that everything is represented by (finitely many) bits. The geometric view is inspired by Euclidean geometric constructions from circa 300 BC, effectively modeled by the real RAM and related models for lower bounds (e.g., linear/algebraic decision/computation trees), and often the simplest model in which to design geometric algorithms. The computational view matches the reality of (digital) computers as we know them today and as set forth by Turing in 1936 [26]. Neither view is likely to disappear anytime soon.

A wealth of research attempts to bridge this gap automatically, e.g., by simulating boolean operations over algebraic computations using guaranteed-sufficient finite-precision approximations [7, 18, 19, 28], or by allowing approximate geometric solutions that preserve desired topological features [15, 23]. In general, these approaches attempt to simulate the geometric view on a binary computer while minimizing the sacrifice in time and/or accuracy. Such automatic approaches intrinsically view finite precision as a curse, even though it can also be a benefit: if all intermediate results and outputs must be of finite precision, then so must be the input. In many situations, the required precision of the intermediate results and outputs can be bounded in terms of the input precision, leading to algorithms designed specifically for finite precision [25].

But to go one step further, finite precision can make a problem even easier than its infinite-precision counterpart. This phenomenon has been exploited heavily in nongeometric (one-dimensional) algorithms and data structures, with one of the early examples being the Fast Fourier Transform. For an example closer to home, many geometric problems (e.g., convex hull, Voronoi diagrams, planar point location) require first sorting the input. In the infinite-precision geometric view where inputs can only be manipulated algebraically and then compared, sorting and hence these geometric problems require $\Omega(n \lg n)$ time.

But sorting finite-precision values is far easier. In practice, radix sort is ubiquitous, even for geometric data: it sorts numbers over a polynomial universe in linear time. In theory, there is no superlinear lower bound and the best general algorithm runs in $O(n\sqrt{\lg \lg n})$ expected time [16]. The model of computation for these sorting algorithms (including radix sort) is the *word RAM*, a natural finite-precision analog of the real RAM in which values of the same precision as the input values can be manipulated in a few basic ways in constant time per operation.[1] Once the input is sorted, some geometric problems (e.g., convex hull) become trivial to solve in $O(n)$ additional time.

To simplify the discussion, this paper makes the practical and standard *transdichotomous* assumption that the number of bits of precision, w, is $\Theta(\lg n)$. In fact, our upper bounds will work for *any* precision matching the word size, and the $\Omega(\lg n/\lg\lg n)$ lower bounds work even with precision $O(\lg n)$ and word size $w = \lg^{O(1)} n$.

How does finite precision affect the optimal running times of other geometric problems? This question has been studied mainly in the context of orthogonal problems [21, 11], to which one-dimensional techniques are relatively straightforward to adapt. A recent pair of papers in FOCS 2006 [9, 24] are the first to obtain speedups for an inherently nonorthogonal problem: for (static) planar point location on an input of size n with w-bit precision, they achieve a query time of $\min\left\{\frac{\lg n}{\lg\lg n}, \sqrt{\frac{w}{\lg w}}\right\}$. More recent work [10] from STOC 2007 further improves the case of bulk queries for m points to a total time of $m \cdot 2^{O(\sqrt{\lg\lg n})} + O(n)$. This running time is better than $m \lg^{\varepsilon} n + O(n)$ for all $\varepsilon > 0$. This result can be applied to obtain algorithms with the same time bound (in expectation) for convex hulls, Voronoi diagrams, Delaunay triangulations, bulk nearest-point queries, trapezoidal decomposition, and triangulating polygons with holes.

This early work opens the door for studying other classic, nonorthogonal problems in computational geometry directly in the finite-precision framework that dominates the rest of theoretical computer science. Until this year, it seemed plausible that only orthogonal problems could be solved more quickly than their infinite-precision counterparts. Now we can study the interplay between computation and information (in the sense of information theory, Kolmogorov complexity, and communication complexity) in a truly geometric setting. We stress that the name of the game here is not developing fancy "bit tricks" to exploit word-level parallelism, but rather studying how geometric information such as points and lines can be decomposed in algorithmically useful ways.

1.2 Our results

We make the first advance in the finite-precision framework for a dynamic problem, namely, dynamic planar convex hull. From a computational geometry perspective, this problem is the dynamic generalization of the most basic geometric computation on n points. From a theoretical computer science perspective, dynamic convex hull is equally fundamental because it includes the extremely natural problem of dynamic linear programming.

We prove matching upper and lower bounds on the optimal query time, assuming updates (point insertion and deletion) run in polylogarithmic time. Specifically, the optimal running time of almost all queries considered before is $\Theta\left(\frac{\lg n}{\lg\lg n}\right)$. The sole exception is the gift-wrapping query (walking the hull), which requires only $\Theta(1)$ time. We show how to achieve these optimal bounds for some of the most important queries, namely hull membership and linear programming (extreme-point queries), in $O(\lg n \lg\lg n)$ time per update. More generally (and actually more challengingly), we show how to achieve the optimal query bound for all previously considered queries in $O(\lg^2 n)$ time per update.

In contrast, all previous data structures for dynamic planar convex hull assume infinite precision and are therefore limited to running queries in $\Theta(\lg n)$ time. The original such data structure, of Overmars and van Leeuwen [22], introduced the idea of recursively representing the convex hull, leading to a $\Theta(\lg^2 n)$ update time while supporting all queries. Eighteen years later, Chan [8] had the breakthrough idea of using techniques from decomposable search problems [4], reducing the update time to $\Theta(\lg^{1+\varepsilon} n)$ but limiting the set of supported queries to those that are decomposable (e.g., forbidding line-stabbing queries). This approach was subsequently improved to update times of $\Theta(\lg n \lg\lg n)$ [5] and finally $\Theta(\lg n)$ [6, 17]. The last bound, $\Theta(\lg n)$, is optimal for updates in the infinite-precision real RAM, assuming queries take $O(n^{1-\varepsilon})$ time [6].

1.3 Techniques

Our most sophisticated upper bound starts from the classic dynamic convex hull structure due to Overmars and van Leeuwen [22]. The first idea is to convert the binary tree in this structure into a tree with branching factor $\Theta(\lg^{\varepsilon} n)$, so that its height is $\Theta\left(\frac{\lg n}{\lg\lg n}\right)$. The many years of failed attempts at sublogarithmic planar point location suggest, however, that it is impossible to solve any nontrivial query by spending $\Theta(1)$ time per node in such a tree. For example, determining which child to recurse into for a tangent query boils down to planar point location in a subdivision of complexity $\Theta(\lg^{\varepsilon} n)$, which we do not know how to solve in $o\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)$ time.

Instead, by carefully exploiting the partial information that a query learns about its answer, we show that the time a query spends to determine which child to visit is proportional to the knowledge it learns about the answer. By charging the time cost to the information progress, we can use an amortization argument to show that expensive nodes are rare and thus bound the overall query cost to $O\left(\frac{\lg n}{\lg\lg n}\right)$. This type of insight did not appear in the static problems solved in previous work; thus dynamic geometric problems (which we are the first to consider) seem fundamentally different and particularly interesting.

Our simpler upper bound starts from the decomposable-search approach introduced by Chan [8] and refined by Brodal and Jacob [5]. In this structure, it seems impossible to support the most difficult decomposable query, tangent, in the optimal time bound $\Theta\left(\frac{\lg n}{\lg\lg n}\right)$. Essentially, the trade-off we could make between a node's query cost and the information it reveals relies on an essentially explicit representation of the convex hull as in the Overmars-van Leeuwen structure. Representing the convex hull as the hull of $O\left(\frac{\lg n}{\lg\lg n}\right)$ overlapping convex hulls, as in the Brodal-Jacob structure, restricts us to optimal implementation of linear-programming queries, which can be viewed as tangent queries for points at infinity. So although the update time is better in this case, the techniques required for optimal query bounds actually become less interesting.

Our results therefore illustrate a refined sense of the difficulty of various queries about dynamic planar convex hulls. The challenge with tangent queries is that the input

[1]The basic operations are typically arithmetic $(+, -, \cdot, /)$ and bitwise operations (AND, OR, XOR, SHIFT). Unlike the real RAM, algebraic roots are forbidden.

has two geometric degrees of freedom (the coordinates of the query point); thus we call the query *two-dimensional*. On the other hand, linear programming is essentially *one-dimensional*, the input being defined by a single directional coordinate. This distinction is what makes both linear-programming and tangent queries possible in $\Theta(\frac{\lg n}{\lg\lg n})$ time in the explicit structure but only linear-programming queries possible in the decomposable structure. Our information-theoretic lens therefore highlight an even stronger contrast between the original Overmars-van Leeuwen structure and the more modern structures based on decomposable search: the latter structures are not informationally efficient. It thus remains open to break the $\Theta(\lg^2 n)$ barrier (again) while achieving informationally efficient two-dimensional queries.

Our lower bounds are based on a reduction from the classic marked ancestor problem [1], where queries are known to require $\Omega(\frac{\lg n}{\lg\lg n})$ time when mark/unmark updates run in $\lg^{O(1)} n$ time. The strength of this result is that it holds on the all-powerful cell-probe model [27], which just measures the intrinsic information required to answer a query, without worrying about how the answer would actually be computed from this information. Of course, such lower bounds apply to the word RAM as well.

2. QUERY TAXONOMY

Formally, the *dynamic planar convex hull* problem is to maintain a dynamic set of points, S, subject to insertion and deletion of points and a number of potential queries summarized in Figure 1.[2] We classify queries by the number of (continuous) degrees of freedom in their input:

- Zero-dimensional queries, where the input is the discrete set S:

  1. *Gift wrapping*: Given a vertex of the convex hull, report the two adjacent vertices of the hull. This is the one query that can be supported in $\Theta(1)$ time, and we show how to do so given update time $O(\lg^2 n)$. (As far as we know, this simple result was not observed before, although it is easy to obtain by applying standard tree-threading and persistence techniques to the Overmars-van Leeuwen structure.)
  2. *Hull membership*: Test whether a point is on the convex hull. This query is one of two basic queries for which we prove an $\Omega(\frac{\lg n}{\lg\lg n})$ lower bound, and we show how to achieve this bound given update time $O(\lg n \lg\lg n)$.

- One-dimensional queries, which we can support in $O(\frac{\lg n}{\lg\lg n})$ given update time $O(\lg n \lg\lg n)$:

  1. *Linear programming* (a.k.a. extreme-point queries): Report the extreme point of the set S in a given direction.
  2. *Line decision*: Given a line ℓ, test whether it intersects the convex hull. Although this query might seem two-dimensional, in fact it is a decision version of linear-programming queries: it tests whether the extreme points in the direction perpendicular to ℓ are on opposite sides of ℓ. This query is the second of two basic queries for which we prove an $\Omega(\frac{\lg n}{\lg\lg n})$ lower bound.
  3. *Vertical line stabbing*: Given a vertical line that intersects the convex hull, report the two edges it cuts.
  4. *Containment*: Report whether a point q is contained in the interior of the convex hull. This query is a decision version of vertical line stabbing, because we only need to test that q is between the two edges that intersect the vertical line. This query is more general than hull membership: applying this query to a perturbation (away from the center of mass) determines whether the point is on the hull.

- Two-dimensional queries, which we can support in $O(\frac{\lg n}{\lg\lg n})$ given update time $O(\lg^2 n)$:

  1. *Tangent*: Given a point q outside the convex hull, report the two tangents of the hull that pass through q. This query is more general than linear programming, because linear programming can be reduced to tangents of points at infinity. This query is also more general than containment: we can assume that the point is outside the hull, find its tangents, and then verify that the tangents are correct (by running linear-programming queries perpendicular to the tangents).
  2. *Line stabbing* (a.k.a. bridge finding): Given a line that intersects the convex hull, report the two edges that it cuts.

Our $O(\lg^2 n)$ structure supports all queries efficiently. Our $O(\lg n \lg\lg n)$ structure only supports decomposable-search queries, like all previous structures with $o(\lg^2 n)$ update times. However, there is an additional discrepancy: our latter structure cannot support hull-membership queries or containment queries. Although these problems are essentially zero- and one-dimensional, respectively, these problems are not decomposable, so cannot be handled directly. The only reason that they can be supported by the decomposable structures of [8, 5, 6, 17] is that they are reducible to a set of tangent queries, which are decomposable. Because our decomposable structure cannot support two-dimensional queries like tangent, we do not know how to support these nondecomposable special cases.

3. FAST QUERIES WITH $O(\lg^2 N)$ UPDATES

In this section, we show how to support updates in $O(\lg^2 n)$ time, and queries in $O(\frac{\lg n}{\lg\lg n})$. In this extended abstract, we focus on tangent queries; support for line-stabbing queries is similar. Recall that these two queries are harder than all the others. Our data structure builds on top of the one by Overmars and van Leeuwen [22]. We now quickly sketch this classic data structure, skipping all implementation details which are treated as a black box by our modifications.

The data structure of [22] is a binary tree, in which every node v contains the set of points S_v in a certain vertical slab. Let left(v) and right(v) be the children of v. The node v

[2] We assume for simplicity of exposition that all x coordinates in S are distinct, as are all y coordinates in S.

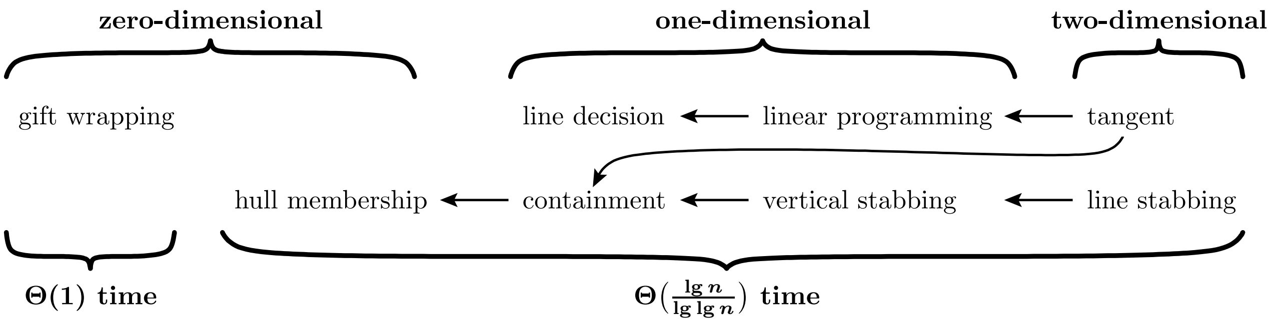

Figure 1: Dynamic planar convex hull queries and their optimal time bounds (assuming polylogarithmic update). The queries in the top row are all decomposable; the queries in the bottom row are not. Arrows indicate reducibility between queries: generalization → specialization. Vertically aligned queries are also dual to each other.

stores a vertical line, splitting S_v into $S_v = S_{\text{left}(v)} \cup S_{\text{right}(v)}$, with $\min\{|S_{\text{left}(v)}|, |S_{\text{right}(v)}|\} = \Omega(|S_v|)$. The children split the sets recursively, down to singleton sets in the leaves. Maintaining this partition is equivalent to maintaining a balanced binary search tree with values stored only in the leaves.

Although Overmars and van Leeuwen developed their structure before the invention of persistence [13], it is easier to see its workings using persistent catenable search trees. Every node v stores a list of the nodes on the convex hull H_v of S_v, represented as a (partially) persistent binary search tree. Then, a query can be answered in logarithmic time based on the hull stored at the root. By standard tree-threading techniques, we can also support gift-wrapping queries in $O(1)$ time.

To maintain this hull dynamically, note that H_v is defined by a convex subchain of $H_{\text{left}(v)}$ and one from $H_{\text{right}(v)}$, plus two new edges (*bridges*) that join them. It is shown in [22] that the bridges can be computed in $O(\lg |S_v|)$ time through binary search. Then, because the binary search trees storing the hulls are persistent and catenable, the information at every node can be recomputed in $O(\lg n)$ time. Thus, updates cost $O(\lg^2 n)$.

3.1 Dealing with Slow Point Location

An obvious idea for improving query time to $O(\frac{\lg n}{\lg\lg n})$ is to store the hulls as (persistent, catenable) B-trees, for some $B = \lg^\varepsilon n$. One would then try to deal with B points in $O(1)$ time (through word packing), so a search could be completed in $O(\log_B n)$ time. It turns out (see below) that the subproblem that must be solved at each node is point location among $O(B)$ segments.

Unfortunately, as mentioned already, this direct approach only works in the one-dimensional context. For point location, it is not known how to handle any superconstant B in $O(1)$ time, and, indeed, this is believed to be impossible. Recent improvements to point location on the RAM [9] provide $O(\frac{\lg B}{\lg\lg B})$ search time at each node. The total query time would be $O(\frac{\lg n}{\lg B} \cdot \frac{\lg B}{\lg\lg B}) = O(\frac{\lg n}{\lg\lg\lg n})$, an exponentially weaker improvement than what we hope for.

To obtain our improvement, one must refine the fusion tree paradigm to *hull fusion*: querying a node of the B-tree (a hull-fusion node) is allowed to take superconstant time, but averaged over all $O(\log_B n)$ nodes that are considered, we spend constant time per node. This follows from an information-progress argument: if querying one node is slow, it is because we have made a lot of progress in understanding the query, and therefore this cannot happen too often.

Our basic tool for point location subqueries is the following fact, which was the central component of the recent data structure by Chan [9]. We sketch the proof for completeness.

Fact 1. [9] *Consider a vertical slab $\{x_L, \dots, x_R\} \times [u]$, and $B \le w$ segments between points (x_L, ℓ_i) and (x_R, r_i), where $\ell_1 \le \dots \le \ell_B$ and $r_1 \le \dots \le r_B$, and ℓ_i, r_i are rational. Assuming that $(\forall) i$, $\ell_{i+1} - \ell_i \ge \frac{\ell_B - \ell_1}{2^{w/B}}$ and $r_{i+1} - r_i \ge \frac{r_B - r_1}{2^{w/B}}$, then in time $O(B)$ we can construct a data structure that supports point location in constant time.*

Proof: First, we map the segments (and later the query) to the universe $[2^{w/B}]^2$, using a projective transformation and rounding. Due to the vertical separation of the segments, the answer to the query can only change by one segment, which can be fixed in constant time. In this universe, we need $O(w/B)$ bits per segment and query, so we can pack all $|T| \le B$ segments in $O(1)$ words, and use parallelism to find the answer in constant time. For full details, see [9]. □

As opposed to [9], which uses this fact to obtain a worst-case running time of $O(\lg m / \lg\lg m)$ for point location among m arbitrary segments, we are interested in the following adaptive bound:

Corollary 2. *Consider a vertical slab $\{x_L, \dots, x_R\} \times [u]$, and $B \le w$ segments between points (x_L, ℓ_i) and (x_R, r_i), where $\ell_1 \le \dots \le \ell_B$ and $r_1 \le \dots \le r_B$, and ℓ_i, r_i are rational. In $O(B^2)$ time, we can construct a data structure, such that a query for a point between segments i and $i+1$ is supported in time $O\left(1 + \frac{B}{w}\left(\lg \frac{\ell_B - \ell_1}{\ell_{i+2} - \ell_{i-1}} + \lg \frac{r_B - r_1}{r_{i+2} - r_{i-1}}\right)\right)$.*

Proof: We select a subset of the segments $T \subset [B]$, by starting with $T = \{1\}$ and applying the following procedure repeatedly. If $i = \max T$, insert into T the lowest $j > i$ such that $\ell_j - \ell_i \ge (\ell_B - \ell_1)/2^{w/B}$ and $r_j - r_i \ge (r_B - r_1)/2^{w/B}$. When we can't insert any more segments, we forcibly insert the topmost segment B, and stop. By Fact 1, we can perform point location among segments in $T \setminus \{B\}$ in constant time, so by one additional we can handle T in constant time. We then recurse between any two consecutive segments in T.

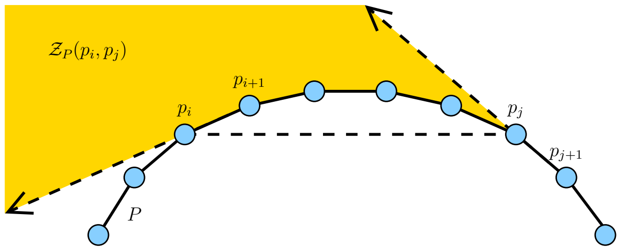

Figure 2: The Zorro $\mathcal{Z}_P(p_i, p_j)$ is the shaded region.

In each step, either the left or right span of the remaining segments decreases by a factor of $2^{w/B}$. After

$$\frac{B}{w}\left(\lg\frac{\ell_B - \ell_1}{\ell_{i+2} - \ell_{i-1}} + \lg\frac{r_B - r_1}{r_{i+2} - r_{i-1}}\right)$$

steps, the subset we are left with cannot include *both* segments $i-1$ and $i+2$, because either the left or right span is too small. In each step, we eliminate at least the bottommost and topmost segment, so we get to the answer in at most 2 additional recursions. □

We can think of $\lg(\ell_B - \ell_1) + \lg(r_B - r_1)$ as the entropy of the search region. If the above data structure takes time t for a query, the entropy decreases by at least $\frac{w}{B}(\Omega(t) - 1)$ bits. Thus, we can hope that the sum of the running times for $\log_B n$ applications of the lemma is bounded by $O(\log_B n + \lg u/\frac{w}{B}) = O(\log_B n + B)$, implying a running time of $O(\lg n/\lg\lg n)$ for, say, $B = \sqrt{\lg n}$. This intuition will indeed prove to be correct, but we need to carefully analyze the geometry of the problem, and show that the information progress is maintained as we query various vertical slabs at various hull-fusion nodes.

3.2 Quantifying Geometric Information

For simplicity, we will only try to determine the right tangent, and assume it lies in the upper convex hull. Left tangents and the lower hull can be handled symmetrically.

We denote an upper convex chain P by its list of vertices from left to right: $P = \langle p_1, p_2, \ldots, p_m\rangle$ where $x(p_i) < x(p_{i+1})$ for all i. Define the *exterior* exterior(P) of an upper convex chain P to be the region bounded by the chain and by the two downward vertical rays emanating from p_1 and p_m that includes points above the chain. In other words, the exterior exterior(P) of P consists of all points left, above, or right of P.

Given an upper convex chain P and indices $1 \le i < j < m$, the *Zorro* $\mathcal{Z}_P(p_i, p_j)$ is the region of points exterior to P, strictly right of the ray from p_{i+1} to p_i, and nonstrictly left of the ray from p_{j+1} to p_j. Thus the Zorro is bounded by these two rays and by the subchain $p_i, p_{i+1}, \ldots, p_j$, as illustrated in Figure 2. Note that the Zorro is an object in the infinite real plane, not on the grid.

The following fact justifies our interest in this definition:

FACT 3. *Let q be a point exterior to an upper convex chain P. Then q is in the Zorro $\mathcal{Z}_P(p_i, p_j)$ if and only if the answer to the right tangent query for q in P is in $\{p_{i+1}, \ldots, p_j\}$.*

Proof: By definition, for each k with $i+1 \le k \le j$, the region of points whose right tangent query answers p_k is a cone emanating from p_k with bounding rays from p_k to p_{k-1} and from p_{k+1} to p_k. This cone is precisely the Zorro $\mathcal{Z}_P(p_{k-1}, p_k)$. Two adjacent such cones, $\mathcal{Z}_P(p_{k-1}, p_k)$ and $\mathcal{Z}_P(p_k, p_{k+1})$, share a bounding ray from p_{k+1} to p_k. Thus their union is $\mathcal{Z}_P(p_{k-1}, p_{k+1})$, so by induction, the union over all k is $\mathcal{Z}_P(p_i, p_j)$. Therefore this Zorro is precisely the region of points whose right tangent query answers one of $p_{i+1}, \ldots, p_j$. □

We also establish a few basic facts that will be useful later:

FACT 4. *Given a point q guaranteed to be exterior to an upper convex chain P, we can test whether q is in the Zorro $\mathcal{Z}_P(p_i, p_j)$ in $O(1)$ time.*

Proof: Though the Zorro's boundary is potentially complicated, if q is known to be outside the polygon, it suffices to test the side of q relative to the lines $p_{i+1}p_i$, p_jp_{j+1}, and p_ip_j (the dashed lines in Figure 2). □

FACT 5. *For any upper convex chain P and any indices $1 \le i < j < k < m$, we have the decomposition: $\mathcal{Z}_P(p_i, p_k) = \mathcal{Z}_P(p_i, p_j) \cup \mathcal{Z}_P(p_j, p_k)$ and $\mathcal{Z}_P(p_i, p_j) \cap \mathcal{Z}_P(p_j, p_k) = \emptyset$.*

Proof: Disjointness follows from Fact 3. The union property follows from taking the union of adjacent Zorro cones as argued in the proof of Fact 3. □

Because Zorros describe the structure of our search problem, we want to define a quantitative measure that allows us to make the information progress argument outlined above. It turns out that information progress is only need (and, actually, only true) for a region of the Zorro. We define the *left slab* of $\mathcal{Z}_P(p_i, p_j)$ as the vertical slab between $x = 0$ and $x = x(p_{i+1})$. The *left vertical extent* $\mathcal{L}(\mathcal{Z}_P(p_i, p_j))$ is the length of the subsegment of the vertical line $x = 0$ intersected by $\mathcal{Z}_P(p_i, p_j)$. The *right vertical extent* $\mathcal{R}(\mathcal{Z}_P(p_i, p_j))$ is the length of the subsegment of the vertical line $x = x(p_{i+1})$ intersected by the Zorro.

DEFINITION 6. *The* entropy *of a Zorro $\mathcal{Z}_P(p_i, p_j)$ is:*

$$H(\mathcal{Z}_P(p_i, p_j)) = \lg \mathcal{L}(\mathcal{Z}_P(p_i, p_j)) + \lg \mathcal{R}(\mathcal{Z}_P(p_i, p_j)).$$

We now establish the following monotonicity property about Zorros "contained" in other Zorros:

FACT 7. *For an upper convex chain P and indices $1 \le i \le i' < j' \le j < m$, we have $\mathcal{Z}_P(p_{i'}, p_{j'}) \subseteq \mathcal{Z}_P(p_i, p_j)$ and $H(\mathcal{Z}_P(p_{i'}, p_{j'})) \le H(\mathcal{Z}_P(p_i, p_j))$.*

Proof: Refer to Figure 3. Fact 5 immediately implies that $\mathcal{Z}_P(p_{i'}, p_{j'}) \subseteq \mathcal{Z}_P(p_i, p_j)$. This geometric containment implies $\mathcal{L}(\mathcal{Z}_P(p_{i'}, p_{j'})) \le \mathcal{L}(\mathcal{Z}_P(p_i, p_j))$, because $\mathcal{Z}_P(p_{i'}, p_{j'}) \cap \{x = 0\} \subseteq \mathcal{Z}_P(p_i, p_j) \cap \{x = 0\}$. The segments at the intersection with $x = x(p_{i+1})$ are similarly contained. Furthermore, $x(p_{i'+1}) \ge x(p_{i+1})$. Because we are working with an upper chain, moving to the right can only decrease vertical extents, so $\mathcal{R}(\mathcal{Z}_P(p_{i'}, p_{j'})) \le \mathcal{R}(\mathcal{Z}_P(p_i, p_j))$. □

Finally, we need to analyze Zorros with respect to a subset of the original chain, because in one step, we plan to analyze only B points out of the hull. The following fact follows from monotonicity of slopes along a convex chain, similarly to previous facts:

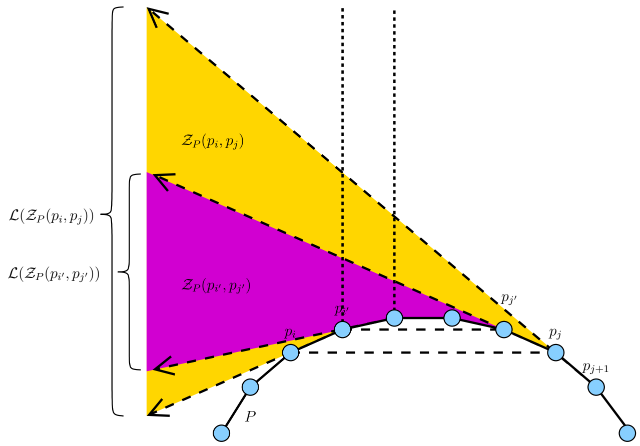

Figure 3: The Zorro $\mathcal{Z}_P(p_{i'}, p_{j'})$ is contained in $\mathcal{Z}_P(p_i, p_j)$.

FACT 8. *For an upper convex chain S and a subsequence $P \subseteq S$ of m vertices, and for indices $1 < i < j < m$, we have:*

$$\mathcal{Z}_P(p_i, p_{j-1}) \cap \text{exterior}(S) \subseteq \mathcal{Z}_S(p_i, p_j) \subseteq \mathcal{Z}_P(p_{i-1}, p_j),$$
$$H(\mathcal{Z}_P(p_i, p_{j-1})) \leq H(\mathcal{Z}_S(p_i, p_j)) \leq H(\mathcal{Z}_P(p_{i-1}, p_j)).$$

3.3 The Data Structure

We first reinterpret the results of Corollary 2 in the language of Zorros. The reason we insist to relate the query time to the entropy of $\mathcal{Z}_P(p_{k-1}, p_{k+2})$, instead of $\mathcal{Z}_P(p_k, p_{k+1})$ is that we will need some slack when switching between Zorros with respect to P, and Zorros with respect to the whole convex hull (see Fact 8).

LEMMA 9. *Given an upper convex chain $P = \langle p_1, p_2, \ldots, p_B \rangle$, in time $B^{O(1)}$ we can build a data structure that answers queries of the following form: given indices $1 < i < j < B-1$, and given a point q guaranteed to be within the Zorro $\mathcal{Z}_P(p_i, p_j)$* and *its left slab, find an index $i < k < j$ such that q is in the Zorro $\mathcal{Z}_P(p_k, p_{k+1})$. The running time of the query is $t = O\big(1 + \frac{B}{w}\big(H(\mathcal{Z}_P(p_i, p_j)) - H(\mathcal{Z}_P(p_{k-1}, p_{k+2}))\big)\big)$.*

Proof: We build the structure for every choice of i and j, incurring an $O(B^2)$-factor penalty in construction time. For some fixed i and j, we need to solve a point location problem in the left slab of $\mathcal{Z}_P(p_i, p_j)$, with the segments given by the intersection of the slab with the rays $p_{i+1}p_i, p_{i+2}p_{i+1}, \ldots, p_{j+1}p_j$. A Zorro $\mathcal{Z}_P(p_k, p_{k+1})$ is actually the wedge between two consecutive rays.

Denote by $\ell_i, \ldots, \ell_j$ the y coordinate of the intersections of these rays with $x = 0$ (the left boundary of the slab). Similarly denote by $r_i, \ldots, r_j$ the intersections with the right boundary of the slab. Note that ℓ_i, r_i are rational.

We will now use the adaptive point location data structure of Corollary 2. If the answer is k, the query time will be asymptotically:

$$\begin{aligned}
t &\approx 1 + \frac{B}{w}\left(\lg \frac{\ell_j - \ell_i}{\ell_{k+2} - \ell_{k-1}} + \lg \frac{r_j - r_i}{r_{k+2} - r_{k-1}}\right) \\
&= 1 + \frac{B}{w}\big(\lg \mathcal{L}(\mathcal{Z}_P(\ell_i, \ell_j)) + \lg \mathcal{R}(\mathcal{Z}_P(\ell_i, \ell_j)) \\
&\qquad - \lg \mathcal{L}(\mathcal{Z}_P(\ell_{k-1}, \ell_{k+2})) - \lg(r_{k+2} - r_{k-1})\big) \\
&\leq 1 + \frac{B}{w}\big(H(\mathcal{Z}_P(\ell_i, \ell_j)) - H(\mathcal{Z}_P(p_{k-1}, p_{k+2}))\big)
\end{aligned}$$

The last inequality follows from $x(p_k) \geq x(p_{i+1})$, using the familiar observation that moving to the right reduces vertical extents. $\square$

The general structure in which we will be performing queries is a B-tree representation of an upper convex chain. For a node v of such a B-tree, let S_v be the set of points in v's subtree, and P_v the set of at most B points stored in the node v. For the sake of queries, each node is augmented with the following information:

- an atomic heap [14] for the x coordinates of the points in P_v.
- the structure of Lemma 9 for the convex chain given by P_v.

3.4 Query invariants

A tangent query proceeds down a root-to-leaf path of the B-tree, spending $O(1)$ time at each node but also applying Lemma 9 at some of the nodes. Therefore the time required by a query is $O(\log_B n)$ plus the total time spent in Lemma 9.

For readability, we will write $\mathcal{Z}_v(a, b)$ for $\mathcal{Z}_{S_v}(a, b)$. At each recursive step of the query, we have $q \in \mathcal{Z}_v(a, b)$ where v is the current node and $a, b \in S_v$. We write $\text{succ}_v(b)$ for the point in S_v which follows b to the right on the upper convex chain. We also assume $\text{succ}_v(b) \in S_v$, to make the Zorro well-defined. In addition, we maintain the invariant that a, b, and $\text{succ}_v(b)$ are nodes of the global upper convex hull C as well. Thus $\mathcal{Z}_v(a, b) = \mathcal{Z}_C(a, b)$.

At the next level of recursion, the Zorro will be some $\mathcal{Z}_{v'}(a', b')$ where v' is a child of v and $a', b', \text{succ}_{v'}(b') \in S_{v'} \cap C$. Furthermore, we will guarantee that $x(a) \leq x(a') < x(b') \leq x(b)$. Hence, by Fact 7, $\mathcal{Z}_{v'}(a', b') = \mathcal{Z}_C(a', b') \subseteq \mathcal{Z}_C(a, b)$, and $H(\mathcal{Z}_C(a', b')) \leq H(\mathcal{Z}_C(a, b))$.

The query may apply Lemma 9 at this recursive step to a Zorro $\mathcal{Z}_{P_v}(p_i, p_j)$, locating q in a Zorro $\mathcal{Z}_{P_v}(p_k, p_{k+1})$. In this case, we guarantee further that

$$\begin{aligned}
\mathcal{Z}_v(a, b) &\supseteq \mathcal{Z}_{P_v}(p_i, p_j) \cap \text{exterior}(C) \\
&\supseteq \mathcal{Z}_{P_v}(p_{k-1}, p_{k+2}) \cap \text{exterior}(C) \supseteq \mathcal{Z}_{v'}(a', b'), \\
H(\mathcal{Z}_v(a, b)) &\geq H(\mathcal{Z}_{P_v}(p_i, p_j)) \\
&\geq H(\mathcal{Z}_{P_v}(p_{k-1}, p_{k+2})) \geq H(\mathcal{Z}_{v'}(a', b')).
\end{aligned}$$

Now we bound the total cost incurred by Lemma 9. By the invariants stated above, $H(Z)$ never increases as we shrink our Zorro Z known to contain q. Furthermore, when we apply Lemma 9, if we spend t time, we guarantee that $H(Z)$ decreases by $\frac{w}{B}(\Omega(t) - 1)$. Hence the total cost incurred by Lemma 9 is at most the maximum total range of $H(\cdot)$, divided by $\frac{w}{b}$. Because the points are on a $u \times u$ grid, any nonzero vertical extent, measured at an x coordinate of the grid, between two lines drawn between grid points, is between $1/u$ and u. Thus, $-2\lg u \leq H(\cdot) \leq 2\lg u$. Because $w \geq \lg u$, the total cost incurred by Lemma 9 is $O(B)$. Therefore the total query time is $O(\log_B n + B) = O(\lg n/\lg\lg n)$, by choosing $B = \lg^{\varepsilon} n$, for any constant $\varepsilon > 0$.

3.5 Querying a hull-fusion node

We now describe how to implement a query using $O(1)$ time at each node plus possibly one application of Lemma 9, while satisfying all of the invariants described above. First we apply the following lemma:

LEMMA 10. *Given a node v with $P_v = \langle p_1, p_2, \ldots, p_m\rangle$, given two points a and b on v's hull where $x(a) < x(b)$, and given a query point $q \in \mathcal{Z}_v(a,b) \cap \text{exterior}(C)$ we can find in $O(1)$ time one of the following outcomes:*

1. *An index $1 \leq k < m-1$ such that q is in the Zorro $\mathcal{Z}_v(p_k, p_{k+1})$.*
2. *Indices $1 < i < j < m-1$ such that $q \in \mathcal{Z}_{P_v}(p_i, p_j)$ where $x(a) \leq x(p_i) < x(p_{j+1}) \leq x(b)$.*

Proof: We round a to the clockwise next bridge point $p_i \in P_v$, and we round b to the counterclockwise next bridge point $p_j \in P_v$. The implementation of this depends on the representation of points, so we defer a discussion of this step until later. We will be able to support this step in constant time.

If $i > j$, then $q \in \mathcal{Z}_v(a,b) \subseteq \mathcal{Z}_v(p_j, p_{j+1})$ (Case 1). Otherwise, $1 < i \leq j < m$. By Fact 5, $\mathcal{Z}_v(a,b) = \mathcal{Z}_v(a,p_i) \cup \mathcal{Z}_v(p_i,p_{j-1}) \cup \mathcal{Z}_v(p_{j-1},p_j) \cup \mathcal{Z}_v(p_j,b)$. In $O(1)$ time, we can determine which of these Zorros contains q. If it is the first Zorro, $q \in \mathcal{Z}_v(a,p_i) \subseteq \mathcal{Z}_v(p_{i-1},p_i)$ (Case 1). If it is the second Zorro, $q \in \mathcal{Z}_v(p_i,p_{j-1})$ (Case 2). If it is the third Zorro, $q \in \mathcal{Z}_v(p_{j-1},p_j)$ (Case 1). If it is the fourth Zorro, $q \in \mathcal{Z}_v(p_j,b) \subseteq \mathcal{Z}_v(p_j,p_{j+1})$ (Case 1). □

Now, if we are in Case 1, say $q \in \mathcal{Z}_v(p_k,p_{k+1})$, then we know to recurse into the recursive subchain between p_k and p_{k+1}, corresponding to some child v'. In this case, we want to recurse with $a' = \max\{a, p_k\}$ and $b' = \min\{b, \text{pred}_{v'}(p_{k+1})\}$ (where max and min are with respect to x coordinates). Thus $x(a) \leq x(a') < x(b') \leq x(b)$, satisfying the guarantee above. Before recursing, however, we check in $O(1)$ time whether $q \in \mathcal{Z}_{v'}(a',b')$; if not, we determine the answer to the right-tangent query to be p_{k+1}.

If we are in Case 2, say $q \in \mathcal{Z}_{P_v}(p_i,p_j)$ where $x(a) \leq x(p_i) < x(p_{j+1}) \leq x(b)$, then there are two subcases. If q is not in the left slab of the Zorro $\mathcal{Z}_{P_v}(p_i,p_j)$, then we perform a successor query $x(q)$ among the x coordinates of the bridge points P_v to find an index i', $i < i' \leq j$, such that $x(p_{i'-1}) \leq x(q) < x(p_{i'})$. Next we test in $O(1)$ time whether q is in the Zorro $\mathcal{Z}_{P_v}(p_{i'},p_j)$. If not, we know to recurse in the recursive chain between $p_{i'-1}$ and $p_{i'}$, and we proceed as in Case 1. Otherwise, we determine that q is in the left slab of $\mathcal{Z}_{P_v}(p_{i'},p_j)$, so we replace i with i' to obtain the other subcase.

So now suppose q is in the left slab of the Zorro $\mathcal{Z}_{P_v}(p_i,p_j)$ where $x(a) \leq x(p_i) < x(p_{j-1}) \leq x(b)$. We can apply Lemma 9 to obtain a Zorro $\mathcal{Z}_{P_v}(p_k,p_{k+1})$ containing q. By Fact 8, $\mathcal{Z}_{P_v}(p_k,p_{k+1}) \cap \text{exterior}(C) \subseteq \mathcal{Z}_v(p_k,p_{k+2})$. By Fact 5, $\mathcal{Z}_v(p_k,p_{k+2}) = \mathcal{Z}_v(p_k,p_{k+1}) \cup \mathcal{Z}_v(p_{k+1},p_{k+2})$. In $O(1)$ time, we can determine which of these three Zorros contains q, and recurse in the corresponding child v' as in Case 1 with Zorro $\mathcal{Z}_{v'}(a',b')$.

Finally we prove the guarantees about Zorro containment and entropy monotonicity. By Fact 8, we have:

$$\begin{aligned} \mathcal{Z}_{P_v}(p_i,p_j) \cap \text{exterior}(C) &\subseteq \mathcal{Z}_v(p_i,p_{j+1}), \\ H(\mathcal{Z}_{P_v}(p_i,p_j)) &\leq H(\mathcal{Z}_v(p_i,p_{j+1})). \end{aligned}$$

Because $x(a) \leq x(p_i) < x(p_{j+1}) \leq x(b)$, $\mathcal{Z}_v(p_i,p_{j+1}) \subseteq \mathcal{Z}_v(a,b)$, and by Fact 7, $H(\mathcal{Z}_v(p_i,p_{j+1})) \leq H(\mathcal{Z}_v(a,b))$. Thus, $H(\mathcal{Z}_{P_v}(p_i,p_j)) \leq H(\mathcal{Z}_v(a,b))$ as desired. On the other hand, by Fact 8:

$$\begin{aligned} \mathcal{Z}_v(p_k,p_{k+2}) &\subseteq \mathcal{Z}_{P_v}(p_{k-1},p_{k+2}), \\ H(\mathcal{Z}_v(p_k,p_{k+2})) &\leq H(\mathcal{Z}_{P_v}(p_{k-1},p_{k+2})). \end{aligned}$$

By Fact 7 and because $x(p_k) \leq x(a') < x(b') \leq x(p_{k+2})$, we have $\mathcal{Z}_{v'}(a',b') \subseteq \mathcal{Z}_v(p_k,p_{k+2}) \subseteq \mathcal{Z}_{P_v}(p_{k-1},p_{k+2})$ and $H(\mathcal{Z}_{v'}(a',b')) \leq H(\mathcal{Z}_v(p_k,p_{k+2})) \leq H(\mathcal{Z}_{P_v}(p_{k-1},p_{k+2}))$ as desired.

3.6 Updates

It remains to describe how we maintain a B-tree with the upper convex hull, as used by the query. A straightforward approach is to only modify the representation of the convex hulls at each node of the Overmars-van Leeuwen structure, storing these as persistent catenable B-trees. Because we do not use parent pointers, we can use standard persistence techniques [13]. Unfortunately, however, a catenable B-tree rebuilds $O(\log_B n)$ nodes per update. Rebuilding a node takes $B^{O(1)}$ time, because we must rebuild the associated data structure of Lemma 9. Finally, the Overmars-van Leeuwen structure performs $O(\lg n)$ splits and joins, so the total update time is $O(\lg^2 n \frac{B}{\lg B}) = O(\lg^{2+\varepsilon} n)$.

To reduce updates to $O(\lg^2 n)$, we abandon persistence, and build the query B-tree in close parallel to the Overmars-van Leeuwen tree. This has a similar flavor to the original approach of Overmars and van Leeuwen [22], which was developed before persistence was known.

Each node of the Overmars-van Leeuwen tree discovers one bridge (because we are only dealing with the upper hull), and two *bridge points* that define it. We compress the bridge points from all nodes on $\lg B - 2$ consecutive levels of the Overmars-van Leeuwen tree into one node of our B-tree. This means a B-tree node will store $2 \cdot (2 \cdot 2^{\lg B - 2} - 1) = B - 2 < B$ points. The depth will be $O(\lg n/\lg B)$. Note, however, that this "B-tree" is not necessarily balanced with respect to the values it stores (the nodes on the hull), but is balanced with respect to the original set of points, closely following the balance of the Overmars-van Leeuwen tree.

An update can only change bridge points on a root-to-leaf path in the Overmars-van Leeuwen tree. This means that only $O(\log_B n)$ nodes of the B-tree are changed, and we can afford to rebuild the associated structures for all of them. This takes time $\log_B n \cdot B^{O(1)} = \lg^{1+\varepsilon} n$, which is a lower-order term.

Finally, we must discuss how points are represented to allow the constant-time operations we have assumed. This is not hard to achieve by maintaining a label with each point. Searching for a point can then be done by comparing against the labels of the bridge points stored in a node of the B-tree. We can use an additional atomic heap for the labels to make this constant time.

4. FAST QUERIES WITH NEAR LOGARITHMIC UPDATE TIME

Brodal and Jacob [5] prove a general reduction from a dynamic convex hull data structure that, on $O(\lg^4 n)$ points, supports queries in $Q(n)$ time and updates in $U(n)$ time, into a dynamic convex hull data structure that, on n points, supports queries in $Q(n) \cdot \frac{\lg n}{\lg \lg n}$, updates in $U(n) \cdot \frac{\lg n}{\lg \lg n}$ and deletes in $O(\lg n \lg \lg n)$ time. The reduction works for decomposable queries. We will show how to build the polylogarithmic structure that supports linear-programming queries in $O(1)$ time and updates in $O((\lg \lg n)^2)$ time, which results in a dynamic convex hull data structure that supports linear-programming queries in $O(\lg n / \lg \lg n)$ time and updates in $O(\lg n \lg \lg n)$ time.

4.1 Data structure

Our data structure maintains the upper convex hull of $k = O(\log^4 n)$ points using four components:

1. A (binary) Overmars-van Leeuwen structure [22], where each node represents the upper convex hull of its descendant points, as the concatenation of a subchain from the convex hull of each child, and two bridges.

2. One atomic heap [14] storing all slopes that appear in the hulls of the nodes of the Overmars-van Leeuwen structure. There are $k' = O(k)$ such slopes. An atomic heap supports insertions, deletions, and predecessor/succesor queries on $\lg^{O(1)} n$ values in $O(1)$ time per operation.

3. A list labeling structure [12, 3] maintaining $O(\sqrt{\lg n})$-bit labels for each such slope such that label order matches slope order. Unlike standard list labeling, our labels must be explicit, without the ability to simultaneously update pieces of several labels via indirection. Fortunately, our label space $2^{O(\sqrt{\lg n})}$ is much larger than our object space $k' = O(\lg^4 n)$. When the label space is polynomially larger than the object space, we can maintain explicit labels in $O(1)$ time per update, e.g., using the root-to-node labels in a weight-balanced search tree structure (BB[α] trees [20] or weight-balanced B-trees [2]); see [3].

The lists representing the convex hull in each node have a nontrivial implementation. First of all, they represent the edges of the hull (in particular, their slopes), rather than the vertices. Second, they are organized as persistent, catenable B-trees with branching factor $B = O(\sqrt{\lg n})$, and thus, height $O(1)$. Each slope is replaced by its label of $O(\sqrt{\lg n})$ bits, which means that a node has $O(\lg n)$ bits of information. We pack each node in one word. We refer to this representation of the convex hulls as *label trees*.

Observe that slopes are sorted on the hull, so a label tree is actually a search tree. Using standard parallel comparisons, we can locate the predecessor/successor labels of a query label in a label tree in constant time.

4.2 Updates

When we insert or delete a point, it affects the hulls of $O(\lg k)$ nodes in the Overmars-van Leeuwen structure. In each of these hulls, we may create $O(1)$ new slopes and/or delete $O(1)$ old slopes. We can compute these $O(\lg k)$ changes in $O(\lg^2 k)$ time using the standard Overmars-van Leeuwen data structure. The atomic heap and the list labeling structures can support these changes in $O(\lg k)$ total time. The labeling structure may update $O(\lg k)$ labels in total, and each label appears in $O(\lg k)$ label trees. Because we can search for a label in a label tree in constant time, we can update the label trees in $O(\lg^2 k)$ total time.

Finally, as we propagate the changes in the node hulls according to Overmars-van Leeuwen, we update the corresponding label trees using persistence, splits, and concatenations. The key property here is that a node can be split at an arbitrary point, or two nodes can be concatenated, in constant time because a node fits in a word. The total update time is therefore $O(\lg^2 k) = O((\lg \lg n)^2)$.

4.3 Linear-programming query

Given a query direction d, we take the slope normal to d. We search for the two adjacent slopes in the global atomic heap. Then we find the label assigned to these slopes in the list labeling structure, and average these two labels together (which is possible if we double the label space). Finally we search for the nearest two labels in the label tree of the root. Thus we obtain the two edges of the overall (root) convex hull whose slopes are nearest to the query slope, so the common endpoint of these two edges is the answer to the query.

5. LOWER BOUND

Our lower bound is based on a reduction from the *marked-ancestor problem* [1]. In this problem, we have a static tree, say, a perfect binary tree. Each node can be either *marked* or *unmarked*. Initially all nodes are unmarked. The two update operations are marking and unmarking a node. The (leaf decision) query is to decide whether a given leaf has a marked ancestor. We can also assume without loss of generality that every leaf has at most one marked ancestor, and trivially that the root is unmarked. Under these conditions, the marked-ancestor problem has the following (tight) lower bound: if updates run in t_u time, queries require $\Omega(\frac{\lg n}{\lg w + \lg t_u})$ time. Assuming $w = \Theta(\lg n)$ and $t_u = \lg^{O(1)} n$, we obtain a lower bound in the following form:

THEOREM 11. *Any data structure for maintaining a convex hull subject to insertion and deletion of points in $\lg^{O(1)} n$ time and either hull-membership or line-decision queries requires $\Omega(\lg n / \lg \lg n)$ amortized time per query in the cell-probe model.*

Proof: We define a mapping from a marked tree to a set of points such that a leaf of the tree has a marked ancestor if and only if the corresponding point is not on the convex hull. Furthermore, we will determine a line incident to each point such that the point is on the convex hull if and only if the corresponding line does not intersect (the interior of) the convex hull. Thus we simultaneously lower-bound both hull-membership and line-decision queries.

The point set consists of three classes of points: "mandatory" points that define the basic nesting structure, one point for each leaf, and one point for each marked node. The construction is contained in an isosceles triangle of height $h_0 = 1$ and base angle $\theta_0 = 45°$.

Our mapping is defined recursively. The construction for a subtree rooted at a node x of depth i is contained in an

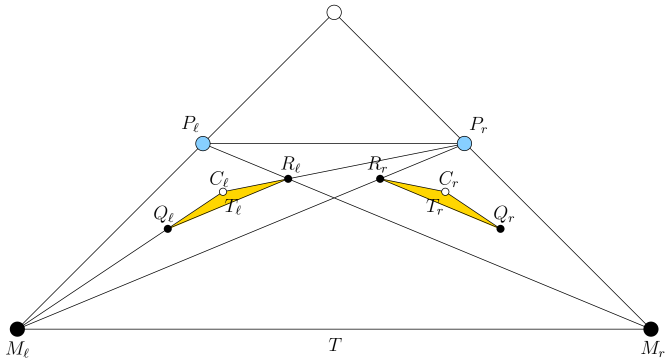

Figure 4: Recursive lower-bound construction. Black points are mandatory; white points do not exist; lightly colored points exist whenever the corresponding node is marked.

isosceles triangle T of height h_i and apex angle θ_i. Given the two constructions for the two children x_l and x_r of x, of depth $i+1$, we describe how to place them into a construction for their parent x at depth i in a given triangle T of height h_i and base angle θ_i. Refer to Figure 4.

First, we identify the two endpoints of the base edge of T as mandatory points. Second, we extend angular bisectors of T at these vertices until they hit the opposite sides, say at points P_ℓ and P_r. These points represent the two children x_ℓ and x_r of x: P_i is present if and only if x_i is marked. Because $P_\ell P_r$ is parallel to $M_\ell M_r$, $\angle M_\ell P_r P_\ell = \angle P_r M_\ell M_r$, which is $\theta/2$; hence, triangle M_ℓ, P_r, P_ℓ is isosceles. Third, we compute the two angular bisectors of this isosceles triangle M_ℓ, P_r, P_ℓ at the vertices M_ℓ and P_r; call their intersection C_ℓ. Fourth, let R_ℓ be the intersection of the angular bisector $P_r C_\ell$ at P_r and the angular bisector $M_r P_\ell$ at M_r. Fifth, let Q_ℓ be the intersection of the angular bisector $M_\ell C_\ell$ at M_ℓ and a line emanating from R_ℓ parallel to $M_\ell P_r$. The left recursive construction is placed in the isosceles triangle $Q_\ell R_\ell C_\ell$, where $Q_\ell R_\ell$ is the base edge so Q_ℓ and R_ℓ become the mandatory points. We repeat Steps 3–5 symmetrically on the right-hand side, and place the right recursive construction in the symmetric isosceles triangle $Q_r R_r C_r$.

We can define an incident line corresponding to each point as follows. For P_ℓ, we choose $P_\ell P_r$. For Q_ℓ, we choose $M_\ell Q_\ell C_\ell$. For R_ℓ, we choose $C_\ell R_\ell P_r$. The choices for R_r and P_r are symmetric.

How do the aspect ratios of the triangles vary with the depth i? Because $Q_\ell R_\ell$ is parallel to $M_\ell P_r$, $\angle C_\ell Q_\ell R_\ell = \angle C_\ell M_\ell P_r$, which is $\theta_i/4$. Hence, $\theta_{i+1} = \theta_i/4$, which solves to $\theta_i = 45°/4^i$.

How do the heights of the triangles vary with the depth i? Let $c = h_0/h_1$ denote the ratio between the heights of T_ℓ and T when T has base angle $\theta_0 = 45°$ and the base is horizontal, as drawn in Figure 4. The ratio h_i/h_{i+1} can be computed by shrinking this figure vertically until the base angle reduces from θ_0 to θ_i. (Note that these length ratios are independent of global scale and rotation.) Specifically, if we shrink vertically by a factor of $\alpha = h_i/h_0$, then the height of T decreases by exactly an α factor, from h_0 to h_i, while the height of T_ℓ decreases by less than an α factor because the segment is not vertical, the vertical extent shrinks by α, and the horizontal extent remains fixed. Hence, $h_{i+1} \geq \alpha h_1 = h_i h_1/h_0$, i.e., $h_i/h_{i+1} \leq h_0/h_1 = c$. In other words, the ratio between consecutive heights is always at most c, so $h_i \geq 1/c^i$.

To encode these points with Cartesian coordinates, our error tolerance must be less than half the minimum feature size, which here is the smallest height $h_{\lg n}$. Because $\lg(1/\theta_i) = \Theta(i) = O(\lg n)$ and $\lg(1/h_i) = \Theta(i) = O(\lg n)$, we can achieve this error tolerance using $O(\lg n)$ bits per coordinate. □

Remark. A simpler lower-bound proof for some queries, including linear-programming, uses duality, where the query becomes vertical line stabbing in a convex envelope. To reduce from marked ancestor, we take an in-order traversal of the nodes in the complete binary tree, and lay them out in this order along a convex semicircle. Marking a node corresponds to drawing the line between the leftmost leaf and the rightmost leaf in its subtree; then any vertical line in the range of that subtree will hit the chord of the line intersected with the disk.

6. CONCLUSIONS

Perhaps the most pressing theoretical question is whether one can obtain a sublogarithmic query time with near-logarithmic update time. As mentioned already, the decomposition techniques of [8, 6], which are the only known method for obtain $o(\lg^2 n)$ update time, seem fundamentally incompatible with information progress arguments.

Another intriguing question is what the right dependence on the precision should be. Going beyond the assumption that $w = \lg^{O(1)} n$, we see that our bounds are not quite tight. The lower bound for all queries is $\Omega(\log_w n)$. The upper bounds from Section 4 are also $O(\log_w n)$. However, for the more complicated queries, our upper bounds are $O(\frac{\lg n}{\lg \lg n})$, which match the lower bound only for $w = \lg^{O(1)} n$. The structure of the point-location algorithms seems to make it hard to improve the upper bound to $O(\log_w n)$. A lower bound separating the two classes of queries would be exciting.

7. REFERENCES

[1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 534–544, 1998.

[2] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[3] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 2002.

[4] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[5] G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log \log n)$ update time. In *Proceedings of the 7th Scandinavian*

Workshop on Algorithm Theory, volume 1851 of *Lecture Notes in Computer Science*, pages 57–70, 2000.

[6] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, Vancouver, Canada, November 2002.

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 341–350, Miami Beach, Florida, 1999.

[8] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 92–99, 1999.

[9] T. M. Chan. Point location in $o(\log n)$ time, Voronoi diagrams in $o(n \log n)$ time, and other transdichotomous results in computational geometry. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 333–342, Berkeley, California, October 2006.

[10] T. M. Chan and M. Pătraşcu. Voronoi diagrams in $n \cdot 2^{O(\sqrt{\lg \lg n})}$ time. In *Proceedings of the 39th ACM Symposium on Theory of Computing*, San Diego, California, June 2007.

[11] M. de Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18(2):256–277, 1995.

[12] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

[13] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[14] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[15] L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. *International Journal of Computatinal Geometry and Applications*, 8(2):157–176, 1998.

[16] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 135–144, Vancouver, Canada, November 2002.

[17] R. Jacob. *Dynamic Planar Convex Hull*. PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, February 2002.

[18] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 351–359, Miami Beach, Florida, 1999.

[19] K. Mehlhorn and S. Schirra. Exact computation with leda_real - theory and geometric applications. In *Symbolic Algebraic Methods and Verification Methods*, pages 163–172. Springer, January 2001.

[20] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[21] M. H. Overmars. Computational geometry on a grid: an overview. In *Theoretical Foundations for Computer Graphics and CAD*, pages 167–184. Springer-Verlag, 1988.

[22] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.

[23] E. Packer. Iterated snap rounding with bounded drift. In *Proceedings of the 22nd Annual Symposium on Computational Geometry*, pages 367–376, Sedona, Arizona, 2006.

[24] M. Pătraşcu. Planar point location in sublogarithmic time. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 325–332, Berkeley, California, October 2006.

[25] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[26] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, August 1936.

[27] A. C. C. Yao. Should tables be sorted? *Journal of the Association for Computing Machinery*, 28(3):615–628, 1981.

[28] C. Yap. Theory of real computation according to EGC. In *Proceedings of the Dagstuhl Seminar on Reliable Implementation of Real Number Algorithms: Theory and Practice*, Lecture Notes in Computer Science, 2006. To appear. http://www.cs.nyu.edu/exact/doc/realtheory.pdf.