Data Structures for Halfplane Proximity Queries and Incremental Voronoi Diagrams

Boris Aronov[1*], Prosenjit Bose[2**], Erik D. Demaine[3***], Joachim Gudmundsson[4], John Iacono[1***], Stefan Langerman[5†], and Michiel Smid[2]

[1] Department of CIS, Polytechnic University, Brooklyn, NY, USA
[2] School of Computer Science, Carleton University, Ottawa, ON, Canada
[3] Computer Science and Artificial Intelligence Lab, MIT, Cambridge, MA, USA
[4] National ICT Australia, Sydney, Australia
[5] Départment d' Informatique, Université Libre de Bruxelles, Brussels, Belgium

Abstract. We consider preprocessing a set S of n points in the plane that are in convex position into a data structure supporting queries of the following form: given a point q and a directed line ℓ in the plane, report the point of S that is farthest from (or, alternatively, nearest to) the point q subject to being to the left of line ℓ. We present two data structures for this problem. The first data structure uses $O(n^{1+\varepsilon})$ space and preprocessing time, and answers queries in $O(2^{1/\varepsilon} \log n)$ time. The second data structure uses $O(n \log^3 n)$ space and polynomial preprocessing time, and answers queries in $O(\log n)$ time. These are the first solutions to the problem with $O(\log n)$ query time and $o(n^2)$ space.

In the process of developing the second data structure, we develop a new representation of nearest-point and farthest-point Voronoi diagrams of points in convex position. This representation supports insertion of new points in counterclockwise order using only $O(\log n)$ amortized pointer changes, subject to supporting $O(\log n)$-time point-location queries, even though every such update may make $\Theta(n)$ combinatorial changes to the Voronoi diagram. This data structure is the first demonstration that deterministically and incrementally constructed Voronoi diagrams can be maintained in $o(n)$ pointer changes per operation while keeping $O(\log n)$-time point-location queries.

1 Introduction

Line simplification is an important problem in the area of digital cartography [6,9,13]. Given a polygonal chain P, the goal is to compute a simpler polygonal chain Q that provides a good approximation to P. Many variants of this problem arise depending on how one defines *simpler* and how one defines *good*

* Research supported in part by NSF grant ITR-0081964 and by a grant from US-Israel Binational Science Foundation.
** Research supported in part by NSERC.
*** Research supported in part by NSF grants CCF-0430849 and OISE-0334653.
† Chercheur qualifié du FNRS

approximation. Almost all of the known methods of approximation compute distances between P and Q. Therefore, preprocessing P in order to quickly answer distance queries is a common subproblem to most line simplification algorithms.

Of particular relevance to our work is a line simplification algorithm proposed by Daescu et al. [7]. Given a polygonal chain $P = (p_1, p_2, \ldots, p_n)$, they show how to compute a subchain $P' = (p_{i_1}, p_{i_2}, \ldots, p_{i_m})$, with $i_1 = 1$ and $i_m = n$, such that each segment $[p_{i_j} p_{i_{j+1}}]$ of P' is a good approximation of the subchain of P from p_{i_j} to $p_{i_{j+1}}$. The amount of error is determined by the point of the subchain that is farthest from the line segment $[p_{i_j} p_{i_{j+1}}]$. To compute this approximation efficiently, the key subproblem they solve is the following:

Problem 1 (Halfplane Farthest-Point Queries). *Preprocess n points $p_1, p_2, \ldots, p_n$ in convex position in the plane into a data structure supporting the following query: given a point q and a directed line ℓ in the plane, report the point p_i that is farthest from q subject to being to the left of line ℓ.*

Daescu et al. [7] show that, with $O(n \log n)$ preprocessing time and space, these queries can be answered in $O(\log^2 n)$ time. On the other hand, a naïve approach achieves $O(\log n)$ query time by using $O(n^3)$ preprocessing time and $O(n^3)$ space. The open question they posed is whether $O(\log n)$ query time can be obtained with a data structure using subcubic and ideally subquadratic space.

In this paper, we solve this problem with two data structures. The first, relatively simple data structure uses $O(n^{1+\varepsilon})$ preprocessing time and space, and answers queries in $O(2^{1/\varepsilon} \log n)$ time. The second, more sophisticated data structure uses $O(n \log^3 n)$ space and polynomial preprocessing time, and answers queries in $O(\log n)$ time. Both of our data structures apply equally well to halfplane farthest-point queries, described above, as well as the opposite problem of halfplane nearest-point queries—together, *halfplane proximity queries.*

Dynamic Voronoi diagrams. An independent contribution of the second data structure is that it provides a new efficient representation for maintaining the nearest-point or farthest-point Voronoi diagram of a dynamic set of points. So far, point location in dynamic planar Voronoi diagrams has proved difficult because the complexity of the changes to the Voronoi diagram or Delaunay triangulation for an insertion can be linear at any one step. The randomized incremental construction avoids this worst-case behavior through randomization. However, for the deterministic insertion of points, the linear worst-case behavior cannot be avoided, even if the points being incrementally added are in convex position, and are added in order (say, counterclockwise). For this specific case, we give a representation of a (nearest-point or farthest-point) Voronoi diagram that supports $O(\log n)$-time point location in the diagram while requiring only $O(\log n)$ amortized pointer changes in the structure for each update. So as not to oversell this result, we note that we do not have an efficient method of determining which pointers to change (it takes $\Theta(n)$ time per change), so the significance of this representation is that it serves as a proof of the existence of an encoding of Voronoi diagrams that can be modified with few changes to the encodings while still supporting point location queries. However, we believe that our combinatorial observations about Voronoi diagrams will help lead to efficient dynamic Voronoi diagrams with fast queries.

Currently, the best incremental data structure supporting nearest-neighbor queries (one interpretation of "dynamic Voronoi diagrams") supports queries and insertions in $O(\log^2 n / \log\log n)$. This result uses techniques for decomposable search problems described by Overmars [14]; see [5]. Recently, Chan [4] developed a randomized data structure supporting nearest-neighbor queries in $O(\log^2 n)$ time, insertions in $O(\log^3 n)$ expected amortized time, and deletions in $O(\log^6 n)$ expected amortized time.

2 A Simple Data Structure

Theorem 2. *There is a data structure for halfplane proximity queries on a static set of n points in convex position that achieves $O(2^{1/\varepsilon} \log n)$ query time using $O(n^{1+\varepsilon})$ space and preprocessing.*

Our proof is based on starting from the naïve $O(n^3)$-space data structure mentioned in the introduction, and then repeatedly apply a space-reducing transformation. We assume that either all queries are halfplane farthest-point queries or all queries are halfplane nearest-point queries; otherwise, we can simply build two data structures, one for each type of query.

Both the starting data structure and the reduction use Voronoi diagrams as their basic primitive. More precisely, we use the farthest-site Voronoi diagram for the case of halfplane farthest-point queries, and the nearest-site Voronoi diagram for the case of halfplane nearest-point queries. When the points are in convex position and given in counterclockwise order, Aggarwal et al. [1] showed that either Voronoi diagram can be constructed in linear time. Answering point-location queries in either Voronoi diagram of points in convex position can be done in $O(\log n)$ time using $O(n)$ preprocessing and space [11].

The proof of this and other results can be found in the full paper [2]:

Lemma 3. *There is a static data structure for halfplane proximity queries on a static set of n points in convex position, called* Okey*, that achieves $O(\log n)$ query time using $O(n^3)$ space and preprocessing.*

Transform 4. *Given any static data structure $\mathcal{D}$ for halfplane proximity queries on a static set of n points in convex position that achieves $Q(n)$ query time using $M(n)$ space and preprocessing, and for any parameter $m \le n$, there is a static data structure for halfplane proximity queries on a static set of n points in convex position, called $\mathcal{D}$-Dokey, that achieves $2\,Q(n) + O(\log n)$ query time using $\lceil n/m \rceil\, M(m) + O(n^2/m)$ space and preprocessing.*

By starting with the data structure Okey of Lemma 3, and repeatedly applying the Dokey transformation of Transformation 4, we obtain the structure Okey-Dokey-Dokey-Dokey-..., or Okey-Dokey^k, which leads to the following:

Corollary 5. *For every integer $k \ge 1$, Okey-Dokey^{k-1} is a data structure for halfplane proximity queries on a static set of n points in convex position that achieves $O(2^k \log n)$ query time using $O(n^{(2k+1)/(2k-1)})$ space and preprocessing.*

3 Grappa Trees

Our faster data structure for halfplane proximity queries requires the manipulation of binary trees with a fixed topology determined by a Voronoi diagram. To support efficient manipulation of such trees, we introduce a data structure called *grappa trees*. This data structure is a modification of Sleator and Tarjan's link-cut trees [16] that supports some unusual additional operations.

Definition 6. *Grappa trees solve the following data-structural problem: maintain a forest of rooted binary trees with specified topology subject to*

$T =$ Make-Tree(v): *Create a new tree T with vertex v (not in another tree).*

$T =$ Link(v, w, d, m_ℓ, m_r): *Given a vertex v in some tree T_v and the root w of a different tree T_w, add an edge (v, w) to make w a child of v, merging T_v and T_w into a new tree T. The value $d \in \{\ell, r\}$ specifies whether w becomes a left or a right child of v; such a child should not have existed previously. The new edge (v, w) is assigned a* left mark *of m_ℓ and a* right mark *of m_r.*

$(T_1, T_2) =$ Cut(v, w): *Delete the existing edge (v, w), causing the tree T containing it to split into two trees, T_1 and T_2. Here one endpoint of (v, w) becomes the root of the tree T_i that does not contain the root of T.*

Mark-Right-Spine(T, m): *Set the right mark of every edge on the right spine of tree T (i.e., the edge from the root of T to its right child, and recursively such edges in the right subtree of T) to the new mark m, overwriting the previous right marks of these edges.*

$(e, m^*_\ell, m^*_r) =$ Oracle-Search(T, O_e): *Search for the edge e in tree T. The data structure can find e only via* oracle queries*: given two incident edges (u, v) and (v, w) in T, the oracle $O_e(u, v, w, m_\ell, m_r, m'_\ell, m'_r)$ determines in constant time which of the subtrees of $T - v$ contains x.*[1] *(Note that edges (u, v) and (v, w) are considered to exist in $T - v$, even though one of their endpoints has been removed.) The data structure provides the oracle with the left mark m_ℓ and the right mark m_r of (u, v), as well as the left mark m'_ℓ and the right mark m'_r of (v, w), and at the end, it returns the left mark m^*_ℓ and the right mark m^*_r of the found edge e.*

Theorem 7. *There exists an $O(n)$-space constant-in-degree pointer-machine data structure that maintains a forest of grappa trees and supports each operation in $O(\log n)$ worst-case time per operation, where n is the total size of the trees affected by the operation.*

4 Rightification of a Tree: Flarbs

The fixed-topology binary search tree maintained by our faster data structure for halfplane proximity queries changes in a particular way as we add sites to a Voronoi diagram. We delay the specific connection for now, and instead define

[1] Given the number of arguments, it is tempting to refer to the oracle as $O(A, B, D, G, I, L, S)$, but we will resist that temptation.

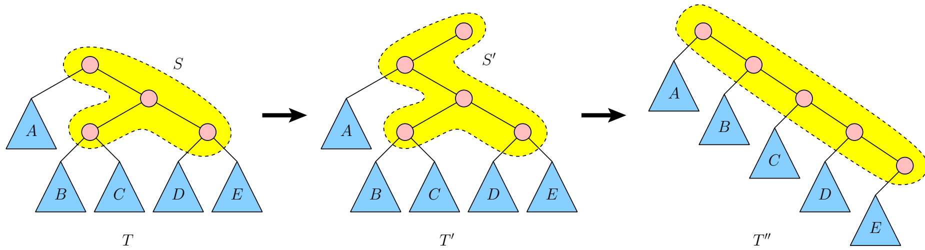

Fig. 1. An example of a flarb. The anchored subtree is highlighted.

the way in which the tree changes: a tree restructuring operation called a "flarb". Then we bound the work required to implement a sequence of n flarbs by showing that the total number of pointers changes (i.e., the total number of parent/left-child and parent/right-child relationships that change) is $O(n \log n)$. Thus, for the remainder of this section, we use the term *cost* to refer to (a constant factor times) the number of pointer changes required to implement a tree-restructuring operation, not the actual running time of the implementation. This bound on cost will enable us to implement a sequence of n flarbs via $O(n \log n)$ link and cut operations, for a total of $O(n \log^2 n)$ time.

The flarb operation is parameterized by an "anchored subtree" which it transforms into a "rightmost path". An *anchored subtree* S of a binary search tree T is a connected subgraph S of T that includes the root of T. A *right-leaning path* in a binary search tree T is a path monotonically descending through the tree levels, always proceeding from a node to its right child. A *rightmost path* in T is a right-leaning path that starts at the root of T.

The *flarb* operation[2] of an anchored subtree S of a binary search tree T is a transformation of T defined as follows; refer to Figure 1. First, we create a new root node r with no right child and whose left child subtree is the previous instance of T; call the resulting binary search tree T'. We extend the anchored subtree S of T to an anchored subtree S' of T' by adding r to S. Now we rearrange S' into a rightmost path on the same set of nodes, while maintaining the binary search tree order (in-order traversal) of all nodes. The resulting binary search tree T'' is the result of flarbing S in T.

Theorem 8. *A sequence of n flarb operations, starting from an empty tree, can be implemented at a cost of $O(\log n)$ amortized pointer changes per flarb.*

Proof. We use the potential method of amortized analysis, with a potential function inspired by the analysis of splay trees [17]. For any node x in a tree T, let $w(x)$ be the *modified weight* of the subtree rooted at x, which is the number of nodes in the subtree plus the number of null pointers in the tree. In

[2] "Flarb" is a clever abbreviation of a long technical term whose meaning we cannot reveal for reasons we cannot comment on at the moment, perhaps simply due to lack of space or of the aforementioned purported meaning. Note that this notion of flarb is different from that of [3].

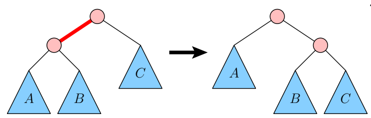

Fig. 2. A zig: The thick edge belongs to the anchored subtree S' and is light.

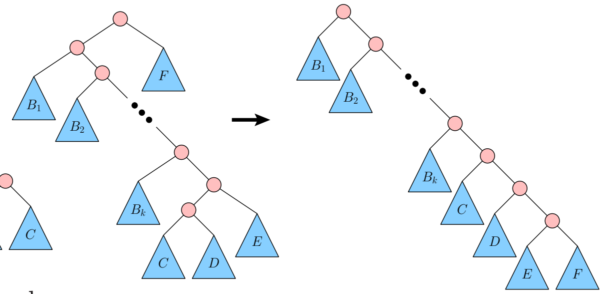

Fig. 3. A zag.

other words, we add dummy nodes as leaves in place of each null pointer in T, for the purpose of computing subtree size. Define $\varphi(x) = \lg \frac{w(left(x))}{w(right(x))}$. Clearly $|\varphi(x)| \leq \lg(2n-1)$, because the smallest possible subtree contains no real nodes and one dummy node, and the largest possible subtree contains $n-1$ real nodes and n dummy nodes. The potential of a tree T with n nodes is $\Phi(T) = \sum_x \varphi(x)$, with the sum taken over the (real) nodes x in T. Therefore, $|\Phi(T)| = O(n \log n)$.

For the purposes of the analysis, we use the following heavy-path decomposition of the tree. The *heavy path* from a node continues recursively to its child with the largest subtree, and the *heavy-path decomposition* is the natural decomposition of the tree into maximal heavy paths. Edges on heavy paths are called *heavy edges*, while all other edges are called *light edges*.

To analyze a flarb in a binary search tree T, we decompose the transformation into a sequence of several steps, and analyze each step separately.

First, the addition of the new root node r can be performed by changing a constant number of pointers in the tree. To implement rightification, we first execute several simplifying steps of two types, called "zig" and "zag",[3] in no particular order. A *zig* is executed whenever a light left edge is part of the anchored subtree S'; see Figure 2. The zig operation simply involves a right rotation on the edge in question. A *zag* is performed whenever there exists, within the anchored subtree S', a path that goes left one edge, right zero or more edges, and then left again one edge; see Figure 3. The zag operation performs a constant number of pointer changes to re-arrange the path in question into a right-leaning path. The full paper [2] shows that each zig or zag has zero amortized cost.

After all possible zigs and zags have been exhausted, we claim that the anchored subtree S' must have the form shown in Figure 4. Indeed, any tree that has no light left edge and no right-leaning path delimited by two left edges must have this form. In particular, because the rightmost path in this tree must be light, its length is at most $\lg(2n+1)$.

[3] Unlike most terminology in this paper, these terms are used for no particular reason. Cf. footnote 2.

The final *stretch* operation, which completes the flarb, simply converts this tree into a rightmost path by effectively concatenating the subsidiary right-leaning paths, incorporating them into the main path. Only $O(\log n)$ actual pointer changes are required. The potential does not increase because left subtrees of every node shrink and right subtrees grow, if they change at all. Thus, the amortized cost of the stretch is $O(\log n)$. $\square$

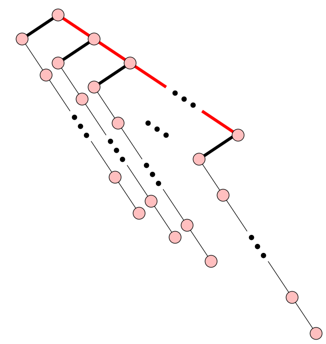

Fig. 4. S' before the final stretch. Thick light edges are light, and thick black edges are heavy.

5 Transformations

We focus on the farthest-point case, but the proofs apply to nearest-point too.

Transform 9. *Given a grappa tree data structure supporting each operation in* $O(\log n)$ *worst-case time, and given a data structure to incrementally maintain a tree created by* n *flarbs with* $O(\log n)$ *amortized pointer changes per flarb, we can construct an* $O(n \log^2 n)$*-space data structure that supports* $O(\log n)$*-time farthest-point queries on any prefix of a sequence of points in convex position in counterclockwise order.*

Proof. We construct an incremental data structure that supports $O(\log n)$-time farthest-point queries on the current sequence of points, $\langle p_1, p_2, \ldots, p_n \rangle$, and supports appending a new point p_{n+1} to the sequence provided that this change maintains the invariant that the vertices remain in convex position and in counterclockwise order. Thus the insertion order equals the index order and equals the counterclockwise traversal order of a convex polygon. The data structure runs on a pointer machine in which each node has bounded in-degree. Thus we can apply the partial-persistence transform of [10] and obtain the ability to support farthest-point queries on any prefix of the inserted points in $O(\log n)$ time. The space is proportional to the number of pointer changes during insertions.

We consider the ordered tree T formed by the finite segments of the farthest-point Voronoi diagram, ignoring their precise geometry; see Figure 5. More precisely, the *farthest-point Voronoi diagram* [15, Section 6.3] divides the plane into n cells by classifying each point q in the plane according to which of $p_1, p_2, \ldots, p_n$ is the farthest from q. The *farthest-point Delaunay triangulation* [12] is the dual of the farthest-point Voronoi diagram, i.e., it triangulates the convex polygon with vertices $p_1, p_2, \ldots, p_n$ by connecting two vertices whenever the corresponding Voronoi cells share an edge. We consider the dual tree T of this farthest-point Delaunay triangulation of the convex polygon, i.e., the dual graph excluding the infinite region exterior to the convex polygon. Each edge in this tree corresponds to (a nongeometric representation of) a finite edge of the farthest-point Voronoi diagram, which is the bisector of two of the points p_i and p_j that are adjacent in the Delaunay triangulation. Each node in the tree represents a vertex in the

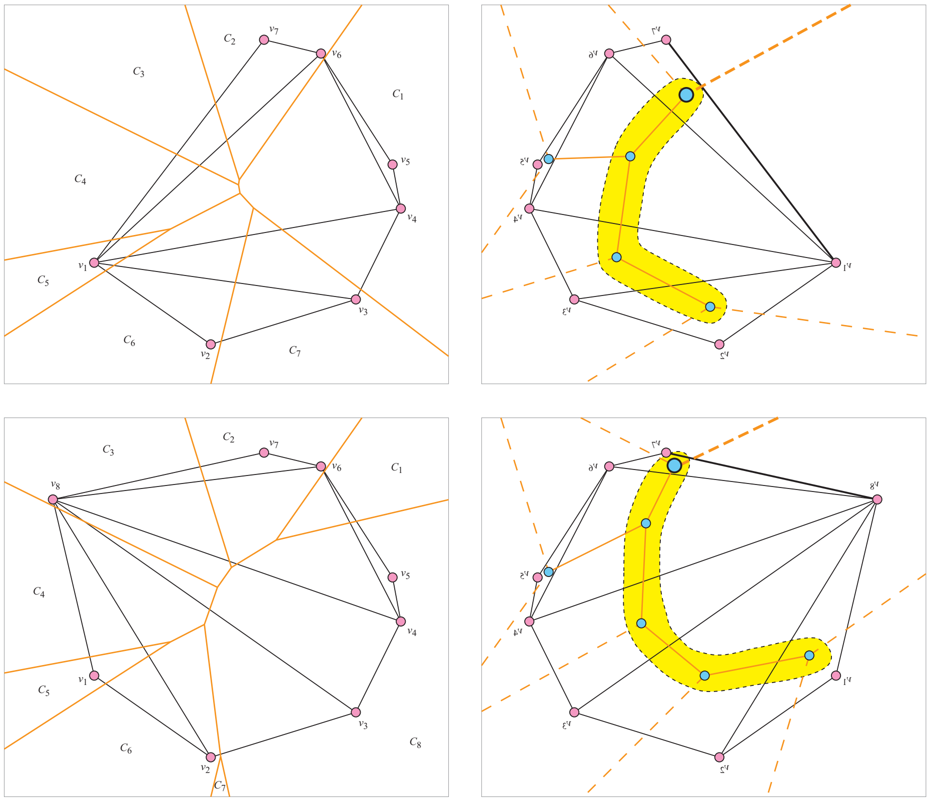

Fig. 5. Adding vertex v_8 in counterclockwise order. Top: Before. Bottom: After. Left: Farthest-point Voronoi diagram and its dual, the Delaunay triangulation. Right: Delaunay triangulation and its dual, the tree T with attached infinite rays drawn as dashed lines, drawn in mirror image so that geometric left versus right matches the order in the Voronoi diagram. The root vertex of T and its parent edge are emboldened.

farthest-point Voronoi diagram, or equivalently a triangle in the farthest-point Delaunay triangulation, and therefore has degree $d \leq 3$, where any degree deficit corresponds to $3 - d$ infinite rays in the farthest-point Voronoi diagram not represented in the tree T.

We can view the tree T as a binary search tree as follows. First, we root the tree at the node corresponding to the unique triangle in the Delaunay triangulation bounded by the edge connecting the first inserted point p_1 and the most recently inserted point p_n. We view the infinite ray emanating from the Voronoi vertex as the "parent edge" of this root node, defining the notion of *left child* versus *right child* of a node according to the counterclockwise order around the Voronoi vertex. (Note that this order is the opposite of the order defined by the triangulation, so in Figure 5 (right), we draw T in mirror image so that its geometric notions of left and right match that of the Voronoi diagram.) Second, we assign keys to nodes consistent with the in-order traversal. For each tree node corresponding to a Delaunay triangle with vertices p_i, p_j, p_k, where $i < j < k$, we assign a key of j. In other words, we assign the median of the three vertex labels of the Delaunay triangle to be the key of the corresponding tree node.

One way to view this key assignment is as follows. If we imagine adding an infinite rays in place of each absent child in the tree, and add an infinite ray in place of the absent parent of the root (the dashed lines in Figure 5, right), matching the counterclockwise order around the Voronoi vertex, then we decompose the plane into regions corresponding to Voronoi regions, each of which corresponds to a single point p_i. All of the nodes bounding p_i's region correspond to triangles incident to p_i. We assign the key i to the unique such node in T that is closest to the root of T, or equivalently the least common ancestor of such nodes, which is the inflection point between two descendant paths that bound the region. Two exceptions are $i = 1$ and $i = n$: the vertices incident to p_1 are those on the left spine of T, and the vertices incident to p_n are those on the right spine of T.

In this view, we also define the *left mark* of an edge to be the label of the region to the left of the edge, and similarly for the *right mark*. Thus, the two marks of an edge define the two points p_i and p_j whose bisector line contains the Voronoi edge. If an edge is the left edge of its parent node, then the edge's right mark is simply the key of that parent, because the right edge of the parent creates an inflection point at the parent. Similarly, if an edge is the right edge of its parent node, then the edge's left mark is the key of that parent. Intuitively, in either case, if we walk up from the edge on its "underside", then we immediately find a local maximum in the region. On the other hand, in either case, the other mark of the edge is the key of the parent node of the deepest ancestor edge that has the opposite orientation (left versus right): this bending point is the first inflection point we encounter as we walk up the tree on the "top side" of the edge. We use a grappa tree to represent T and the left and right marks of edges.

Next we consider the effect of inserting a new point p_{n+1}. As in the standard incremental algorithm for Delaunay construction [8, Section 9.3], we view the changes to the farthest-point Delaunay triangulation as first adding a triangle p_1, p_n, p_{n+1} and then flipping a sequence of edges to restore the farthest-point Delaunay property. The key property of the edge-flipping process is that all flipped edges end up incident to the newly inserted point p_{n+1}. Therefore these changes can be interpreted in the tree as adding a new root node, whose left child is the previous root, and then choosing a collection of nodes to move to the right path of the new root. This collection of nodes induces a connected subtree because the triangles involved in the flips form a connected set. (In particular, the flipping algorithm considers the neighbors of a triangle for flipping only if the triangle was already involved in a flip.) Thus, the changes correspond exactly to a flarb, with the flexibility of the flarb operation encompassing the various possibilities of which edges get flipped to maintain the farthest-point Delaunay property. Another way to view the addition of p_{n+1} is directly in the

Voronoi diagram. The point p_{n+1} will capture the region R_{n+1} for which p_{n+1} is the farthest neighbor. The region R_{n+1} is a convex polygon. Outside R_{n+1}, the Voronoi diagram is unchanged, so all edges of the new Voronoi diagram are either bisectors of the same two points as before, or are edges of R_{n+1}. In T after the flarb, R_{n+1} corresponds to the right spine.

Each pointer change during a flarb operation can be implemented with one cut and one link operation. Therefore the grappa tree implements the $O(n \log n)$ total pointer updates from flarb operations in $O(n \log^2 n)$ total pointer updates. It remains to update the marks on the edges. By the incremental Voronoi/Delaunay view above, the only edges for which these marks might change are the edges incident to the new region R_{n+1}, i.e., the edges on the right spine. We update the right marks on all of these edges by calling Mark-Right-Spine$(T, n+1)$. The left mark of each edge on the right spine is simply the key of the parent node of the edge. During the execution of the flarb, various right paths were cut and pasted together with cuts and links to form the final right spine. The edges on the final right spine that were originally part of a right path in T already had a left mark equal to the key of their parent node. Any other edges on the final right spine were just added via links, so their left marks can be set accordingly by specifying the right m_ℓ argument to Link. Thus, the total number of pointer updates remains $O(n \log^2 n)$. This concludes the space bound of the data structure.

To support farthest-point queries, it suffices to build an oracle for the grappa tree's Oracle-Search. Specifically, given two incident edges (u, v) and (v, w), the oracle must determine which subtree of $T - v$ has the answer to the farthest-point query. Using the two marks on the two edges, two of which must be identical, we can determine the three vertices p_i, p_j, and p_k of the Delaunay triangle corresponding to vertex v in T. The vertex of the Voronoi diagram corresponding to v lies at the intersection of the three perpendicular bisectors between these three vertices of the Delaunay triangle. We draw three rays from this Voronoi vertex to each of the three corners of the Delaunay triangle. These three rays divide the plane into three sectors, and the Voronoi regions corresponding to the nodes in each subtree of $T - v$ lie entirely in one of these sectors, with exactly one subtree per sector. In constant time, we can decide which of the three sectors contains the query point q. The farthest-point Voronoi region containing the query point q is guaranteed to be incident to the corresponding subtree, and therefore we obtain a suitable answer for the oracle query. At the end, Oracle-Search will narrow the search to a specific edge of T, meaning that the query point q is in one of the two Voronoi regions incident to the corresponding Voronoi edge. In constant time, using the two labels on that edge of the tree, we can determine which side of the bisector contains q, and therefore which Voronoi region contains q, i.e., which point p_i is farthest from q. □

Transform 10. *Given an $O(n \log^2 n)$-space data structure that supports $O(\log n)$-time farthest-point queries on any prefix of a sequence of n points ordered in convex position in counterclockwise order, we can construct an $O(n \log^3 n)$-space data structure that supports $O(\log n)$-time farthest-point-left-of-line queries on n points in convex position.*

Combining Theorems 7 and 8 with Transforms 9 and 10, we obtain:

Corollary 11. *There is an $O(n \log^3 n)$-space data structure that supports $O(\log n)$-time halfplane proximity queries on n points in convex position.*

Corollary 12. *There is an $O(n)$-space data structure for maintaining a nearest-point or farthest-point Voronoi diagram of a sequence of points in convex position in counterclockwise order. The data structure supports inserting a new point at the end of the sequence, subject to preserving the invariants of convex position and counterclockwise order, in $O(\log n)$ amortized pointer changes per insertion; and supports point-location queries in $O(\log n)$ worst-case time.*

6 Open Problems and Conjectures

Several intriguing open problems remain open. One obvious question is whether the $O(n \log^3 n)$ space of our second data structure can be improved while keeping the optimal $O(\log n)$ query time. One specific conjecture in this direction is this:

Conjecture 13. *A sequence of n flarb operations, starting from an empty tree, can be implemented at a cost of $O(1)$ amortized pointer changes per flarb.*

We have no reason to believe that our $O(\log n)$ amortized bound is tight. Reducing the bound to $O(1)$ amortized would shave off a $O(\log n)$ factor from our space and preprocessing time. More importantly, it would increase our understanding of dynamic Voronoi diagrams, reducing the $O(\log n)$ amortized update time in Corollary 12 to $O(1)$ amortized. The potential function we use is inherently logarithmic; a completely new idea is needed here for further progress.

On the issue of improving our understanding of dynamic Voronoi diagrams, we pose the following problem:

Open Problem 14. *Is there a data structure for maintaining a Voronoi diagram of a set of points in convex position that allows point to be inserted in $\log^{O(1)} n$ time while supporting $O(\log n)$ point location queries?*

Here we relax the condition that the points be inserted in counterclockwise order, but maintain the restriction that they be in convex position. Although our potential function does not give the result, it is possible that a slight variation of it does.

Finally, it would be interesting to improve the construction time in our second data structure, in particular so that it completely subsumes the first data structure:

Open Problem 15. *Can the pointer changes caused by a flarb be found and implemented in $o(n)$ time, preferably $\log^{O(1)} n$ time?*

We have not been able to fully transform our combinatorial observations about the number of pointer changes into an efficient algorithm, because we lack efficient methods for finding which pointers change. Solving this question would improve our construction time by almost a linear factor, and would provide a reasonably efficient dynamic Voronoi data structure for inserting points in convex position in counterclockwise order.

Acknowledgments. This work was initiated at the Schloss Dagstuhl Seminar 04091 on Data Structures, organized by Susanne Albers, Robert Sedgewick, and Dorothea Wagner, and held February 22–27, 2004 in Germany. This work continued at the Korean Workshop on Computational Geometry and Geometric Networks, organized by Hee-Kap Ahn, Christian Knauer, Chan-Su Shin, Alexander Wolff, and René van Oostrum, and held July 25–30, 2004 at Schloss Dagstuhl in Germany; and at the 2nd Bertinoro Workshop on Algorithms and Data Structures, organized by Andrew Goldberg and Giuseppe Italiano, and held May 29–June 4, 2005 in Italy. We thank the organizers and institutions hosting both workshops for providing a productive research atmosphere. We also thank Alexander Wolff for introducing the problem to us.

References

1. A. Aggarwal, L. J. Guibas, J. B. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.*, 4(6):591–604, 1989.
2. B. Aronov, P. Bose, E. D. Demaine, J. Gudmundsson, J. Iacono, S. Langerman, and M. Smid. Data structures for halfplane proximity queries and incremental voronoi diagrams. arXiv:cs.CG/0512093, `http://arXiv.org/abs/cs.CG/0512093`
3. T. Calling. The adventures of Flarb Demingo! `http://www.thecalling.co.za/flarb_pictures.htm`, 2005. See also the fan site, `http://www2.fanscape.com/thecalling/streetteam/flarb.html`.
4. T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proc. 17th ACM-SIAM Sympos. Discrete Algorithms*, 2006.
5. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, 1992.
6. R. G. Cromley. *Digital Cartography*. Prentice Hall, August 1991.
7. O. Daescu, N. Mi, C.-S. Shin, and A. Wolff. Farthest-point queries with geometric and combinatorial constraints. *Computat. Geom. Theory Appl.*, 2006. To appear.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computat. Geom. Theory Appl.*. Springer, second edition, 1999.
9. B. D. Dent. *Cartography: Thematic Map Design*. William C Brown Pub, fifth edition, July 1998.
10. 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.
11. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
12. D. Eppstein. The farthest point Delaunay triangulation minimizes angles. *Computat. Geom. Theory Appl.*, 1(3):143–148, March 1992.
13. R. B. McMaster and K. S. Shea. *Generalization in Digital Cartography*. Association of American Cartographers, Washington D.C., 1992.
14. M. H. Overmars. *The Design of Dynamic Data Structures*, LNCS 156, 1983.
15. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.
16. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.
17. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.