Incremental Voronoi Diagrams

Landon Chu* Emily Zhang† Midori Zhou‡

December 14, 2019

Abstract

In this project, we survey and discuss incremental Voronoi diagrams and their applications to halfplane proximity queries. We examine how two different abstract representations of the incremental Voronoi diagram, both making use of *grappa trees*, are updated through *flarb operations* to reflect the insertion of a new site into the Voronoi diagram. We study the amortized combinatorial cost of flarb operations and describe an algorithm for performing the flarb when sites are added in convex position.

1 Introduction

Voronoi diagrams are a very useful data structure in computational geometry for solving a variety of problems regarding the proximity of points in a plane. They are widely used in fields ranging from informatics, to civics and planning, to the natural sciences. Voronoi diagrams are formally defined as follows. Given a finite set of sites S, every site $p_i \in S$ is surrounded by a convex polygon $V(i)$, called the Voronoi region, with the property that p_i is the closest of the sites in S to any point in $V(i)$. Because every $V(i)$ is an intersection of halfplanes, every $V(i)$ is convex. All of the Voronoi regions divide the plane up into a convex net called the Voronoi diagram of S. Once the Voronoi diagram is constructed, it can handle point location queries in the cells of the Voronoi diagram.

1.1 Static Voronoi Diagrams

Voronoi diagrams were first introduced in 1975 [8] along with an optimal $O(n \log n)$ time construction algorithm that used a divide-and-conquer approach. In 1987, Fortune's Algorithm, a sweepline technique for constructing Voronoi diagrams, was introduced [6]. This technique was far simpler to implement than the divide-and-conquer approach, whose merge step involved complicated details. The sweepline technique is also able to compute the Voronoi diagram of weighted point sites.

The progress made in Voronoi diagrams produced $O(n \log n)$ algorithms for several problems which only had $\Omega(n^2)$ algorithms prior to the $O(n \log n)$ Voronoi diagram introduced in 1975. These problems include, but are not limited to, nearest or farthest neighbor, Euclidean minimum spanning tree, minimum weight triangulation, largest empty circle, and convex hull [8].

*lschu@mit.edu
†eyzhang@mit.edu
‡xmzhou@mit.edu

The farthest point Voronoi diagram, where each Voronoi region $V(i)$ consists of the points that are farther from p_i than any other site, is a variation of the classical nearest point Voronoi diagram. Later in this paper, we will discuss techniques for modifying both nearest and farthest point Voronoi diagrams. In [1], a linear time algorithm for computing the Voronoi diagram of a convex polygon was introduced. We will discuss how such a diagram can be maintained when sites are added incrementally [2].

1.2 Incremental Voronoi Diagrams

In many applications, sites are incrementally inserted into the set S of sites. Allen et al. [2] showed that the amortized number of structural changes (edge insertions and removals) needed to update a Voronoi diagram of S to reflect an inserted site is $O(\sqrt{n})$.

Aronov et al. [3] showed an amortized $O(\log n)$ upper bound on the number of structural changes needed to update an incremental farthest point Voronoi diagram in the case where the points are inserted along their convex hull in counterclockwise order. Their proof is existential, and they do not provide an algorithm to find the amortized $O(\log n)$ links and cuts that ought to be performed when a site is inserted.

Allen et al. presented an $O(K \log^7 n)$ algorithm which maintains the Voronoi diagram of points in convex position as new points are inserted, where K is the number of structural changes. By the Allen et al. bound, the runtime of this algorithm in the case of general insertion order is $O(\sqrt{n} \log^7 n)$. By the Aronov et al. bound, the runtime of this algorithm in the case where sites are inserted along the convex hull in counterclockwise order is $O(\log^8 n)$.

2 Grappa Trees

In order to store the representation of the Voronoi diagrams while efficiently supporting incremental operations, both papers make use of *grappa trees* [3], a data structure based on the worst-case formulation of Sleator and Tarjan's link-cut trees [9]. Like link-cut trees, grappa trees are a dynamic graph data structure representing a forest of trees, based on the key idea of representing trees in terms of vertex-disjoint *preferred paths.* In grappa trees, each path is a maximal vertex-disjoint path stored in a biased binary tree; paths are ordered by tree depth and connected to each other by non-path edges.

Unlike link-cut trees, however, grappa trees are meant to represent forests of binary trees for their usage in representing Voronoi diagrams, which allows for several key modifications. Every original node in a represented tree can be forced to have degree 3 by adding a superroot above the root and adding extra left and/or right children to original vertices as required. For a direct representation of a Voronoi diagram, these external vertices can represent points at infinity. The topology of grappa trees allows them to represent 3-regular Voronoi diagrams for sets of points in convex position, which will not have cycles in their Voronoi edges. Additionally, grappa trees support maintaining "left" and "right" marks on each edge of the represented trees, enabling several new operations beyond those offered by link-cut trees.

Grappa trees support the following operations in worst-case $O(\log n)$ time:

MAKE-TREE(v): Construct a new grappa tree T with a single internal vertex v. Note that two external vertices and a superroot are also attached to v with edges with null labels.
LINK(u, v): Given an external vertex v in T_v and a superroot u in T_u, connect the parent of v to the child of u to and delete extra external nodes to merge T_u and T_v into a new tree T.
CUT(e): Delete the existing edge $e = (u, v)$ in tree T, splitting into two trees T_1 and T_2, one containing u and one containing v.
EVERT(v): Make external node v the superroot of the tree, reversing the orientation of every edge along the path from the superroot to v.
LEFT-MARK(T, v, m_l): Set the left mark of every edge on the path from the superroot to v in T to the new mark m_l.
RIGHT-MARK(T, v, m_r): Set the right mark of every edge on the path from the superroot to v in T to the new mark m_r.
ORACLE-SEARCH(T, O_e): Search for the edge e in tree T in the following way: given two incident edges f and f' and their left and right marks, the provided oracle $O_e(f, f', m_f^l, m_f^r, m_{f'}^l, m_{f'}^r)$ determines in constant time whether e is in the component of $T - f$ that contains f', or in the rest of the tree. The oracle-search returns the found edge e along with its left and right marks.

Denote as T the expanded rooted binary tree which is being represented; its grappa tree structure will be denoted R. Internal vertices in the biased binary trees represent edges, and leaves represent vertices of T; due to the structure of T, we can also consider each leaf to represent the unique non-path child edge of its vertex. Then, R is a tree with vertices representing path and non-path edges of T. By explicitly storing both R and T, which is possible due to the bounded-degree assumption, it is possible to maintain and propagate markings across all operations with only a constant-factor cost. Since, as with link-cut trees, paths and biasing can be chosen for R to have height $O(\log n)$, the marking and oracle-search operations can also be implemented with $O(\log n)$ worst-case runtimes [3].

3 TREE-FLARB

After we add a new site to an existing Voronoi diagram, we need to reconstruct the current graph by doing a series of links and cuts. There are various techniques of reconstructing a Voronoi diagram such as randomization in [7]. We will focus on two deterministic operations, TREE-FLARB and CYCLE-FLARB (Section 5), that model insertions of new sites in two different abstract representations of Voronoi diagrams. For each operation, we motivate and describe the algorithm; then we analyze the amortized cost of insertions.

In this section, we investigate a special case of incremental Voronoi diagrams, where all the sites are added in convex position in counterclockwise order. In this case, we use a TREE-FLARB operation to model the insertion of a new site. We want to show that the amortized cost (number of structural changes) of each TREE-FLARB operation is $O(\log n)$. Before we introduce the TREE-FLARB operation, we construct an expanded binary tree T for a Voronoi diagram D.

Definition 1. *In an expanded binary tree $G(V, E)$, $v \in V$ is internal if and only if it is not a leaf or the superroot. $(v_1, v_2) \in E$ is internal if and only if v_i and v_j are both internal.*

Definition 2. *An anchored subtree is a connected subgraph which consists of the root and a subset of the internal nodes.*

We construct the expanded binary tree T in the following way: for every vertex v_i in D, we create an internal node n_i in T; consequently, for every finite edge e_i in D we create an internal edge in T. For every infinite edge in D coming from vertex v_i, there is an external edge from node n_i to an external node n_i' in T. We make the most recently inserted vertex the root of T, with an external edge going to the superroot. We sort based on the counterclockwise order around a vertex, which corresponds to an internal node in T. [3]

The goal of a TREE-FLARB operation is to transform an *anchored subtree* into a right-leaning path from the root. Whenever a new point is inserted, a new Voronoi vertex forms, which becomes the new root of the tree. We then use the TREE-FLARB operation once on S to restructure the tree to represent the new Voronoi diagram.

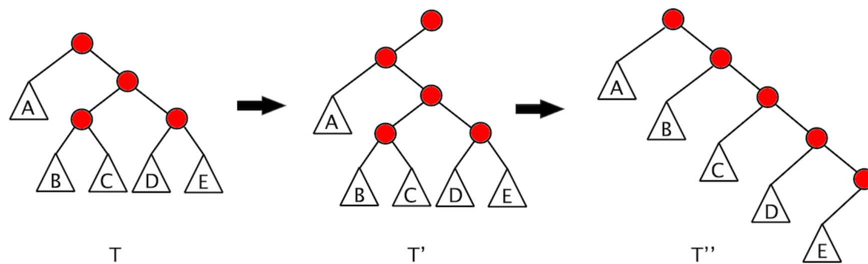

Figure 1: T is the tree that represents the original Voronoi diagram, and the *anchored subtree* S are the nodes in red. In T', a new root is added. T'' is the result of performing a TREE-FLARB on T'.

Definition 3. *A Zig rotation is a right rotation.*

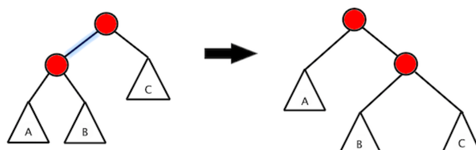

Figure 2: An example of a *Zig* rotation - the highlighted edge is a light edge

Definition 4. *A Zag rotation involves restructuring into a right-leaning subtree an anchored subtree where there is a path from the root that goes left one edge, right k edges, and left one edge again.*

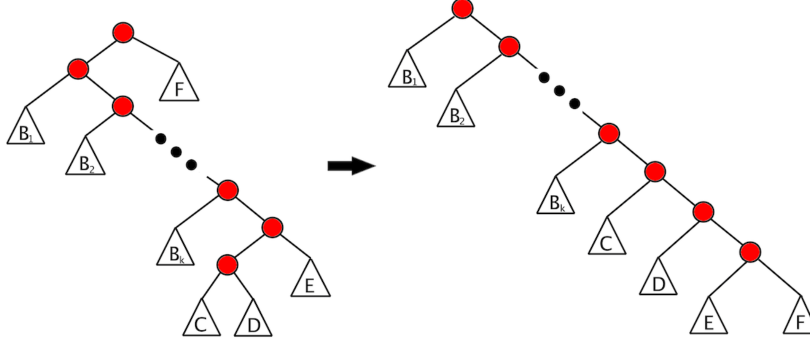

Figure 3: An example of a *Zag* rotation

Definition 5. *A Stretch operation makes an anchored subtree right-leaning by concatenating the right-leaning paths in the anchored subtree along the right "spine".*

We hope to prove that the amortized cost of the entire TREE-FLARB operation is $O(\log n)$ structural changes. We use the potential function $\Phi = c \sum_{n_i \in T} \lg(\frac{weight(subtree_{left}(n_i))}{weight(subtree_{right}(n_i))})$, where $weight(\mathcal{T})$ is the total number of nodes in subtree $\mathcal{T}$, and c is a constant.

Definition 6. *A heavy path from a node is a path where every edge on the path recursively goes to the subtree with a larger weight. A light edge is an edge that connects two heavy paths.*

The TREE-FLARB operation begins with adding a root to the the *anchored subtree*; this takes a constant number of pointer changes, and the potential changes by at most $O(\log n)$ because only the new root contributes to the change in potential.

We do a *Zig* rotation whenever we have a light left edge in the anchored subtree S. We do a *Zag* rotation whenever we have a path in S that goes left one edge, right k edges and left one edge again.

Observation 1. *After completing all possible Zig and Zag rotations on the graph, we observe that the graph must have a right-leaning path consisting of the root and all light edges. For every vertex on the path, if it has a left subtree, the edge to its left child is heavy and its left subtree is a right-leaning path.*

Now we perform the final *Stretch* operation to make the internal nodes of this tree a right-leaning path, like T'' in Fig. 1.

Lemma 1. *The amortized cost of a Zig rotation on a light edge is 0.*

Proof. The actual cost of this operation is a constant number of pointer changes.

We will use the notations in Fig.2 to analyze the change in potential. Since we are rotating on a light edge, C has a larger weight than the total weight of A and B. For the original root, its left subtree becomes lighter and its right subtree remains the same, so its potential decreases. For the new root, its right subtree remains the same but its left subtree is at least twice its weight before, so its potential drops by at least c. We can pick a constant c that offsets the number of pointer changes to make the amortized cost 0. □

Lemma 2. *The amortized cost of a Zag rotation is 0.*

Proof. We do a constant number pointer changes through moving a constant number of subtrees and 3 rotations; hence, the actual cost is constant.

To analyze the change in potential, we first consider the nodes in the middle branch that is right-leaning. Their left subtrees remain the same but their right subtrees become heavier in the new tree, so their potential decreases. For the rest of the nodes, their potential drops by at least c [3], so their contribution to the overall change in potential can offset the number of pointer changes by picking an appropriate constant c for the potential function; hence the amortized cost is 0. □

Lemma 3. *After exhausting all possible Zig and Zag rotations, the number of pointer changes required for a Stretch operation is amortized $O(\log n)$.*

Proof. The actual cost of the concatenation is $O(\log n)$ pointer changes: due to Observation 1, the weight on the right subtree decreases by half after a heavy left edge, so the number of concatenations is $O(\log n)$. Since the left subtree gets moved into the right subtree, the potential decreases. Therefore, the total amortized cost of the last step is $O(\log n)$. □

Combining Lemma 1, 2, and 3, we conclude that the amortized number of structural changes for each TREE-FLARB operation is $O(\log n)$.

4 Incremental Voronoi Diagram via Grappa Trees and TREE-FLARB

In this section, we prove the existence of a data structure supporting the the construction of an incremental Voronoi diagram when points are added in convex position in counterclockwise order. Furthermore, we also show that we can build an oracle for the grappa tree's ORACLE-SEARCH function for halfplane proximity queries.

Note that this proof is solely existential as it does not provide an algorithm for finding the structural changes to be made.

Theorem 1. *There exists a data structure that supports insertions of new points in convex position in counterclockwise order in a Voronoi Diagram in $O(\log n)$ structural changes, as well as halfplane proximity queries in amortized $O(\log n)$ time.*

Proof. We present the grappa tree that represents an incremental Voronoi diagram as in section 3, where each node in the grappa tree represents a Voronoi vertex, corresponding by dual to a Delaunay triangle, and internal and external edges represent finite edges infinite edges of the Voronoi diagram, respectively. We also use left and right marks on each edge in the grappa tree to respectively label the Voronoi regions to the left and right of the edge. By comparison to the standard incremental algorithm for computing a Delaunay triangulation [5, Section 9.3], we notice that the changes made upon inserting a new point for the Delaunay triangulation correspond directly to the changes made by a TREE-FLARB operation with respect to the dual Voronoi diagram, giving correctness of maintaining the Voronoi diagram via TREE-FLARB [3].

Whenever we make a pointer change in the flarb operation, the marks on the edges incident to a new region need to be updated. We update their right marks by calling the RIGHT-MARK function on the rightmost node in T, which recursively changes the right marks in the right "spine" of the

tree. For the left marks, we call the LEFT-MARK function on the linked root when we call the LINK function. Since we make at most $O(n \log n)$ updates on all the markings in the tree in total when n vertices are inserted into the tree, the amortized number of changes per insertion is $O(\log n)$.

Now we construct the query oracle to be used by the ORACLE-SEARCH operation for supporting nearest- or farthest-neighbor queries in the data structure. The oracle takes in two incident edges (u, v), (v, w) and the left and right marks of both edges, and returns which side of the edge (u, v) has the answer to the query. Let p_i, p_j, p_k be the 3 points that make up the Delaunay triangle for v, then one of the marks of (v, w) is p_i or p_j, and the other one is p_k. We take the intersection of the the lines bisecting the sides of the Delaunay triangulation to find the Voronoi vertex corresponding to v, then we draw two rays from this vertex opposite to p_i and p_j that split the plane into two parts. Finally, we can find which of the two parts contain the query point in constant time by using the two marks on the edges. This completes the description of the constant-time query oracle.

We prove in Section 3 that a TREE-FLARB operation requires amortized $O(\log n)$ structural changes, and we flarb once for every new insertion. Given a constant time oracle, we know that ORACLE-SEARCH in grappa trees takes $O(\log n)$ time. □

5 CYCLE-FLARB

In this section, we introduce the CYCLE-FLARB operation, which models the insertion (in any order) of new sites in Voronoi diagrams.

The Voronoi diagram for any set of points is a 3-regular planar graph after using the line at infinity to join the endpoints of unbounded edges in clockwise order. Additionally, inserting a new point into the plane means adding a new face to the Voronoi diagram. This structural change can be implemented by a CYCLE-FLARB operation where the curve C is the boundary of the new face.

5.1 Definitions and Notation

Definition 7. *A planar graph G is flarbable on simple closed curve C, if*
1. the graph induced by the set of vertices of G that lie in the interior of C is connected,
2. C intersects each edge of G at most once,
3. C does not pass through any vertex of G.

Definition 8. *For a 3-regular planar graph G that is flarbable on simple closed curve C, a* CYCLE-FLARB *operation $F(G, C)$ modifies G as follows:*
1. For each edge (u, v) with one vertex u lying in the interior of C and one vertex v lying in the exterior of C, create a new vertex w at $C \cap (u, v)$ and connect w to v along (u, v),
2. For every two newly created vertices that are on edges intersected by C successively, create a new edge between them along C,
3. Finally, delete all vertices lying in the interior of C and all their incident edges.

Let G be a planar graph flarbable on simple closed curve C. Let f be a face of G. We have the following definitions:
1. $|f|$ is the number of edges on the boundary of f.
2. f' is the modified version of face f after $F(G, C)$.
3. A face f is augmented if $|f'| > |f|$, shrinking if $|f'| < |f|$, and preserved if $|f'| = |f|$.
4. $P(G, C)$, $A(G, C)$, $S(G, C)$ are the set of preserved, augmented, and shrinking faces crossed by

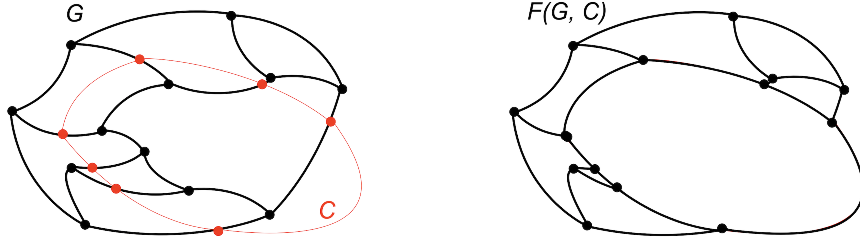

Figure 4: CYCLE-FLARB example; note that a flarb may induce very few structural changes in the graph.

curve C, respectively.
5. Let E_C be the set of *fleeq edges*: edges of G crossed by C.
6. Let $B(G,C)$ be the set of faces of graph G completely contained in the interior of curve C.

5.2 Cost of a CYCLE-FLARB

We analyze the minimum number of structural changes that G undergoes when a CYCLE-FLARB is performed; we call this the cost of the flarb. A structural change is either a link or a cut.

Lemma 4. $COST(F(G,C)) \leq 12|S(G,C)| + 3|B(G,C)| + O(1)$.

Proof. Consider G_C, the graph induced by the vertices in the interior of C taken together with the fleeq edges and their endpoints. All vertices in G_C have degree 3 except for the $|E_C|$ vertices which lie outside of curve C which have degree 1. It follows from some work with Euler's formula that G_C has at most $2|E_C| + 3|B(G,C)|$ edges.

Note that if two preserved faces share a non-fleeq edge e in G_C, then the four neighbors of the endpoints of e lie outside of C (because the number of edges bounding the faces must be preserved after the flarb). Because G_C is connected, e and its neighbors would make up the entire G_C graph, so the bound holds trivially.

Every preserved face contributes to at least 3 edges being preserved (2 fleeq edges and a 3rd edge entirely in G_C). One of those fleeq edges might be shared between two different preserved faces, so each preserved face contributes to at least 2 unique preserved edges of G_C, and at most $2|E_C| + 3|B(G,C)| - 2|P(G,C)|$ edges are cut.

We must reintroduce an edge for every non-preserved fleeq edge and every non-preserved face. Thus, to complete the flarb, we require at most $2(|E_C| - |P(G,C)|)$ link operations.

$$\begin{aligned} COST(F(G,C)) &\leq 4|E_C| + 3|B(G,C)| - 4|P(G,C)| \\ &\leq 4(|A(G,C)| + |S(G,C)| + |P(G,C)|) + 3|B(G,C)| - 4|P(G,C)| \\ &= 4(|A(G,C)| + |S(G,C)|) + 3|B(G,C)| \\ &\leq 12|S(G,C)| + 3|B(G,C)| \end{aligned}$$

The last inequality follows from the fact that in the nontrivial case, every augmented face must neighbor at least one shrinking face. □

5.3 Cost of a Sequence of CYCLE-FLARB Operations

Let $G = (V, E)$ be a 3-regular planar graph, with set F of faces. To analyze the amortized cost of each operation in a sequence of flarb operations on graph G, we use potential function

$$\Phi(G) = \lambda \sum_{f \in F} \mu(f)$$

where $\mu(f) = \min\{\lceil \sqrt{|V|} \rceil, |f|\}$, and λ is some sufficiently large constant.

Note that when $|f|$ and $|f'|$ are both larger than $\sqrt{|V|}$, $\mu(f)$ is unchanged; we'll call such faces f *large faces*. This allows us to focus only on smaller faces in our analysis.

By considering the number of edges in the connected subgraph defined by all the edges of G enclosed or intersected by C that bound some face that γ crosses through, we arrive at the following lemma.

Lemma 5. *Given G flarbable on curve C and sub-curve $\gamma \subseteq C$, let $f_1, \dots, f_k$ be the sequence of faces crossed by γ, and let $f'_1, \dots, f'_k$ be their corresponding modified faces after $F(G, C)$. Then*

$$\sum_{i=1}^{k} (|f_i| - |f'_i|) \geq \frac{|S(G, \gamma)|}{2}$$

Notation: Let $\mathcal{G}^0 = G$ and $\mathcal{G}^i = F(\mathcal{G}^{i-1}, C_{i-1})$ where $\mathcal{G}^i$ is flarbable on simple closed curve C_i for all nonnegative integers i.

Theorem 2. *For a 3-regular planar graph $G = (V, E)$ and some sequence of disjoint cycles $C_0, \dots, C_N$ such that $\mathcal{G}^i$ is flarbable on C_i,*

$$COST(F(\mathcal{G}^{i-1}, C_{i-1})) + \Phi(\mathcal{G}^i) - \Phi(\mathcal{G}^{i-1}) \leq O(\sqrt{|V_i|})$$

Proof. We split curve C_i into smaller curves $\gamma_1, \dots, \gamma_h$ so that each $\gamma_i \subseteq C$ is a maximal curve that doesn't intersect the interior of a face with more than $\sqrt{|V_i|}$ edges. Let s_j be the number of shrinking faces crossed by γ_j.

From section 5.2, we got an upper bound on $COST(F(C_i, \mathcal{G}^i))$

$$\begin{aligned} COST(F(C_i, \mathcal{G}^i)) &\leq 12|S(G_i, C_i)| + 3|B(G_i, C_i)| + O(1) && (1) \\ &\leq 12\sqrt{|V_i|} + 12 \sum_{j=1}^{h} s_j + 3|B(G_i, C_i)| + O(1) && (2) \end{aligned}$$

Let f_n be the new face created by the flarb. Let $A(C_i)$ be the set of faces crossed by curve C_i. Recall that when $|f|$ and $|f'|$ are both larger than $\sqrt{|V|}$; we'll call these faces *large faces*. Let L_i be the set of large faces.

Next, we upper bound $\Phi(\mathcal{G}^i) - \Phi(\mathcal{G}^{i-1})$.

$$\Phi(\mathcal{G}^i) - \Phi(\mathcal{G}^{i-1}) = \mu(f_n) + \lambda \sum_{f \in A(C_i)} (\mu(f') - \mu(f)) - \lambda \sum_{f \in B(G^{i-1}, C_{i-1})} \mu(f) \tag{3}$$

$$\leq \mu(f_n) + \lambda \sum_{j=1}^{h} \left(\sum_{f \in A(\gamma_j)} (\mu(f') - \mu(f)) \right) + \lambda \sum_{f \in L_i} (\mu(f') - \mu(f)) - 3\lambda |B(G^{i-1}, C_{i-1})| \tag{4}$$

$$\leq \sqrt{|V_i|} + \lambda \sum_{j=1}^{h} \left(\sum_{f \in A(\gamma_j)} (\mu(f') - \mu(f)) \right) + \lambda |L_i| - 3\lambda |B(G^{i-1}, C_{i-1})| \tag{5}$$

$$\leq \sqrt{|V_i|} - \frac{\lambda}{2} \sum_{j=1}^{h} s_j + \lambda |L_i| - 3\lambda |B(G^{i-1}, C_{i-1})| \tag{6}$$

Inequality (4): We broke up the first summation by independently considering the large faces in L_i and the small faces crossed by some subcurve. Additionally, step 3 of CYCLE-FLARB deletes all of the faces, each with at least 3 edges, in the interior of the cycle.
Inequality (5): Each face can gain at most one edge, so $\mu(f') - \mu(f) \leq 1$. By definition, $\mu(f_n) \leq \sqrt{|V_i|}$.
Inequality (6): We apply Lemma 5 to the first summation.

Replacing $|L_i|$ with $\sqrt{|V_i|}$ in equation 6 and taking that together with equation (2) along with a sufficiently large λ ($\lambda \geq 24$), we arrive at the statement of the theorem. □

Theorem 1 gives us the result that the amortized number of structural changes needed to update a Voronoi diagram is $O(\sqrt{n})$. We note that this is also a lower bound as it is possible to construct a graph where the amortized cost of a flarb is $\Theta(\sqrt{n})$ [2].

6 Performing CYCLE-FLARB for Sites Added in Convex Position

We consider the problem of maintaining the incremental Voronoi diagram for sites being added in convex position. We use grappa trees to explicitly store the Voronoi diagram, and maintain the structure through flarb operations on the insertion of new sites. Each time a new site is inserted, it defines a flarbable curve C, corresponding to the boundary of the new Voronoi cell; conducting this flarb operation on the Voronoi diagram yields the necessary changes to the structure of the graph for the new diagram.

Recall that a grappa tree represents an expanded binary tree T, and also maintains left and right marks for each edge of T. Here, T represents the topology of the 3-regular planar graph of the Voronoi diagram, where external vertices denote points at infinity; left and right marks on an edge e of T are used to denote the two faces of the Voronoi diagram adjacent to e. These markers allow the grappa tree to store the geometric representation of the Voronoi diagram, as without them the tree only stores the topology.

6.1 Performing the Flarb

Let S be a set of sites in the plane in convex position, and $V(S)$ be the grappa tree representing the Voronoi diagram on S as above. Given a new site q in the plane such that $S \cup \{q\}$ is in convex

position, let S' denote $S \cup \{q\}$, and let $V(S')$ be the Voronoi diagram on S' whose representation we'd like to obtain. Let $\partial CELL(q, S')$ denote the boundary of Voronoi cell of q in $V(S')$, the cell we'd like to add to our existing diagram via a flarb operation. Note that $\partial CELL(q, S')$ is a flarbable curve C on the Halin graph view of $V(S)$ (connecting external vertices in a cycle). Also, this curve will contain one of the external vertices of $V(S)$, namely, the point at infinity of the bisecting edge between the two neighbors of q along the convex hull of S'. The basic process of computing this flarb operation to transform from $V(S)$ to $V(S')$ is outlined as follows:

1. Identify the external vertex p of $V(S)$ that is inside C; call $\textsc{Evert}(p)$ on $V(S)$ so that $V(S)$ is rooted at p.

2. Consider the heavy-path decomposition of $V(S)$. After step 1, the root of each path is the endpoint closest to p. Compute the set R_C of the heavy-path roots which are inside C. Any heavy edge in $V(S)$ that intersects C must lie on one of the heavy paths of one of these roots.

3. Since C is flarbable, the portion of $V(S)$ contained in C is connected; thus, for each heavy path h_r whose root r is in R_C, the portion of the path contained in C is connected. We can then identify the last vertex contained in C for each of these paths; equivalently, we can identify the unique fleeq-edge e_r in each path crossing C, if there exists one.

4. Find and mark the non-preserved edges for deletion. Recall that, with respect to the flarb, a preserved edge is an edge that bounds a preserved face. Thus, a preserved edge is an edge that reappears, or a fleeq-edge adjacent to an edge that reappears. Denote as $V_q(S)$ the subtree induced by the edges of $V(S)$ intersecting C; all non-preserved edges will be in $V_q(S)$. We can find all of these non-preserved edges by a method outlined in Section 6.1.3; we then mark each of them as *shadow edges* for removal. Denote as σ the number of shadow/non-preserved edges; we'd now like to remove the shadow edges and reconnect the graph in the right order.

5. Consider each connected component that would be formed when all shadow edges are removed from $V_q(S)$. By constructing an Eulerian tour on the subtree given by $V_q(S)$ we can determine the order in which these components will need to be reconnected after actually removing the shadow edges; we will do this efficiently by turning connected components into supernodes to compress the tree, to ensure we can obtain the tour in time proportional to σ. This compressed tree can be constructed in $O(\sigma \log \sigma)$ time.

6. Remove the shadow edges, resulting in a (compressed) forest of connected components. Some vertices will be left isolated; delete the isolated internal vertices. The node p at infinity will become isolated, and we will replace it by two new vertices p_1, p_2 (which will correspond to two new points at infinity in the final diagram). For each of the components lying entirely outside C, create a new anchor node as the parent of the component. Construct a path from p_1 to p_2 through the connected components (through their supernodes and anchor nodes) in the Eulerian-tour-order of their leaves along new edges. Then, decompress the supernodes and reattach the components using the new edges.

Having concluded modifications to $V_q(S)$, these changes to the whole tree $V(S)$ yield the new tree $V(S')$ corresponding to the Voronoi diagram on S', as desired.

Some more details on the processes of the steps follow:

6.1.1 Step 2: Computing R_C

Computing R_C can be done in $O(|R_C| \log^6 n)$ time by making use of a data structure presented by Chan [4] which answers extreme-point queries about convex hulls, and through some clever geometric transformations to reframe this as such a convex-hull problem [2].

6.1.2 Step 3: Identifying fleeq-edges in paths

Recall that for a Voronoi diagram (with enough sites), each vertex v is the center of a *definer circle* which passes through at least three sites, the *definers* of v. For a root $r \in R_C$, we can identify the fleeq-edge e_r (if it exists) in its corresponding heavy path h_r in $O(\log n)$ time using an oracle-search. To do so, we construct an oracle which, given adjacent edges f and f' which share vertex v, determines in $O(1)$ time which "side" of the tree e_r is on by checking the regions/sites specified by the face-markers of f and f' and checking whether q lies in the definer circle of v.

6.1.3 Step 4: Identifying non-preserved edges

Since we store face-markers on edges, we can check geometrically in constant time whether a given vertex is contained in C by referring to the face-markers on its incident edges and checking its definers. Using this, we can clearly check whether any given edge (v, w) is a fleeq-edge. We can also check whether an edge is one that reappears by checking if its two adjacent edges are fleeq-edges; thus, we can test in constant time whether a given edge is preserved. To find non-preserved edges in $V_q(S)$, we first check all edges (w, u) , where w is the last vertex of h_r inside C (the vertex of the fleeq-edge e_r closer to r). Any non-preserved node gets marked *shadow*, as mentioned. Next, we find all *bent edges* on heavy paths, which are edges (u, v) such that the light edges of v and u are opposite-sided (left and right, or vice versa). Note that a bent edge can't be preserved; we identify and mark them efficiently by modifying the biased binary trees in the grappa tree. After running these checks, the edges marked as shadow will be exactly the non-preserved edges [2]. Additionally, we can show the following:

Lemma 6. $|R_C| = O(\sigma \log n)$

Proof. Consider each root $r \in R_C$ and its parent p_r; the edge (r, p_r) is a light edge, and p_r is part of a distinct heavy path h_t for some root $t \in R_C$. Construct a set of *dependency paths* as follows: add a *dependency pointer* from r to t if p_r is the first vertex in the unique fleeq-edge of h_t, which hooks the vertices in R_C into a collection of paths.

Then, for each dependency path, if we look at the sink r of the dependency path, we note that the edge (r, p_r) cannot be preserved, since p_r is neither a fleeq-edge nor incident to a fleeq-edge. Each dependency path has such a (distinct) non-preserved edge. Also, each dependency path has length $O(\log n)$, and the vertex set of the dependency paths together is R_C. Then, σ is at least the number of dependency paths, which is $\Omega(|R_C|/\log n)$, so $|R_C| = O(\sigma \log n)$.

6.2 CYCLE-FLARB Analysis

The flarb operation for inserting the new site q can be implemented in $O(K \log^7 n)$ time, where K is the cost of the flarb; computing the flarb takes $O(|R_C| \log^6 n + \sigma \log n)$, and by the above lemma, $|R_C| = O(\sigma \log n)$. The work for the flarb uses $\Theta(\sigma)$ links and cuts. To construct an incremental

Voronoi diagram for n sites in convex position, we have $K = O(\sqrt{n})$ for sites inserted in arbitrary order [2], or $K = O(\log n)$ for sites added in counterclockwise order [3]. $\square$

7 Discussion and Conclusion

In [3], Aronov et al. presented several methods of preprocessing n sites in convex position in the plane in order to answer halfplane proximity queries, including the grappa-tree based Voronoi construction discussed above. Interesting to note is that the incremental nature of the Voronoi construction appears to be purely a side-effect of the construction, since the intended application only requires a static diagram on the n points. In contrast, Allen et al. [2] set out with the specific goal of constructing incremental Voronoi diagrams; hence, they present their own flarb operation along with analysis and bounds which are relevant to *general* incremental Voronoi diagram construction; this CYCLE-FLARB comes from the natural view of adding a new site to a Voronoi diagram, which requires reallocating space in the plane for the new cell corresponding to the site, defining a flarbable curve. Their algorithmic construction of an incremental Voronoi diagram then makes use of grappa trees from [3], which can only represent Voronoi diagrams on points in convex position, though there is no restriction on insertion order.

In conclusion, we presented two abstract representations of incremental Voronoi diagrams. One handles points added in convex position in counterclockwise order in amortized $O(\log n)$ structural changes. The other handles general insertion order in amortized $O(\sqrt{n})$ structural changes and $O(K \log^7 n)$ time, where K is the number of structural changes.

8 Acknowledgments

We would like to thank Professor Karger and Professor Madry for teaching this Advanced Algorithms course. Further we'd like to thank Cenk Baykal, Joshua Brunner, and Sam Park for their assistance throughout the semester. Finally, we would like to thank Daniel Sleator and Robert Tarjan for their enlightening and prolific work in data structures.

References

[1] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the voronoi diagram of a convex polygon. *Discrete & Computational Geometry*, 4(6):591–604, 1989.

[2] S. R. Allen, L. Barba, J. Iacono, and S. Langerman. Incremental voronoi diagrams. *Discrete & computational geometry*, 58(4):822–848, 2017.

[3] 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. *Algorithmica*, 80(11):3316–3334, 2018.

[4] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *Journal of the ACM (JACM)*, 57(3):16, 2010.

[5] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[6] S. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4):153, 1987.

[7] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2495–2504. SIAM, 2017.

[8] M. I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162. IEEE, 1975.

[9] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.