Sublinear Explicit Incremental Planar Voronoi Diagrams

ELENA ARSENEVA1,a) JOHN IACONO2,b) GRIGORIOS KOUMOUTSOS2,c) STEFAN LANGERMAN2,d) BORIS ZOLOTOV1,c)

Received: January 8, 2020, Accepted: September 10, 2020

Abstract: A data structure is presented that explicitly maintains the graph of a Voronoi diagram of N point sites in the plane or the dual graph of a convex hull of points in three dimensions while allowing insertions of new sites/points. Our structure supports insertions in $O(N^{3/4})$ expected amortized time, where $\tilde{O}$ suppresses polylogarithmic terms. This is the first result to achieve sublinear time insertions; previously it was shown by Allen et al. that $\Theta(\sqrt{N})$ amortized combinatorial changes per insertion could occur in the Voronoi diagram but a sublinear-time algorithm was only presented for the special case of points in convex position.

Keywords: data structures, voronoi diagrams, computational geometry, combinatorial change, big cell

1. Introduction

Voronoi diagrams and convex hulls are two keystone geometric structures of central importance to computational geometry. We focus the description here on planar Voronoi diagrams of points; the results can be extended to the dual graph of 3D-convex hulls: we describe the way to do so later in the paper, see Section 9. Several algorithms, based on various different techniques, have been developed over the years that compute the Voronoi diagram of a set of $N$ points in optimal time $O(N \log N)$ [4]. Surprisingly however, the problem of maintaining dynamically a Voronoi diagram subject to insertions/deletions of points is not well understood.

1.0.1 Incremental Voronoi Diagrams

In this paper we focus on the problem of maintaining the Voronoi diagram under insertion of new sites. In Allen et al. [1] it was observed that while there could be a linear number of changes to the embedded Voronoi diagram with each site insertion, this is not equivalent to the number of combinatorial changes (i.e., edge insertions and deletions) to the graph of the Voronoi diagram. What more, it was proved that the maximum number of combinatorial changes per site insertion is $\Theta(\sqrt{N})$ amortized. This opened the possibility of maintaining the Voronoi diagram graph under insertions with sublinear update time. Allen et al. [1] achieved this for the restricted case where the sites are in convex position, for which they designed a data structure with $O(\sqrt{N} \log^2 N)$ amortized insertion time. This result relies crucially on the fact that the Voronoi diagram of a set of points in convex position is a tree. Other more restricted special cases have been studied where the number of combinatorial changes is $\Theta(\log N)$ [2], [10].

1.0.2 Our Result

In this work, we provide a data structure that explicitly maintains the graph of a Voronoi diagram of arbitrary point sites in $\mathbb{R}^2$ while allowing insertions of new sites in $O(N^{3/4})$ amortized time, where $\tilde{O}$ suppresses polylogarithmic terms. This is the first data structure supporting insertions in sublinear time for this problem.

We crucially note that we are interested in maintaining explicitly the Voronoi diagram. In particular, we store the diagram as a graph in adjacency list format on which primitive operations, including links and cuts, are performed. This is different than just maintaining a data structure that answers nearest neighbor queries. This case can be solved dynamically in polylogarithmic time by Dynamic Nearest Neighbor (DNN) structures [5], [6], [9]; this relies heavily on the fact that nearest neighbor is a decomposable search problem, whereas maintaining the Voronoi diagram is clearly not. In fact, here we use those DNN structures as subroutines for solving our problem.

We remark that maintaining the Voronoi diagram in sublinear time in the fully dynamic setting (i.e., with both insertions and deletions) is hopeless as the $\Theta(\sqrt{N})$ amortized bound of combinatorial changes for insertions becomes $\Theta(N)$.

1.1 Brief Description of Our Approach

We now give a high-level overview of our approach and the organization of the rest of the paper.

We store the Voronoi diagram as a combinatorial graph, which allows the quick retrieval of any geometric information if needed.

Suppose we wish to insert a new site $s_M$ into the diagram, and let $f$ be the cell of the diagram that contains $s_M$. This cell can be found in polylogarithmic time using DNN structures. It is known [1] that the affected cells that need to be updated, i.e., that undergo combinatorial changes, form a connected region including cell $f$. To update the diagram, we discover all the affected cells by a variation of the breadth-first search starting from $f$.  

---

1 Saint Petersburg State University (SPbU), Universitetskaya nab. 7–9, 199034 St. Petersburg, Russia
2 Université libre de Bruxelles (ULB), Avenue Franklin Roosevelt 50, 1050 Bruxelles, Belgium
a) e.arseneva@spbu.ru
b) john@johniacono.com
c) greg.koumoutsos@gmail.com
d) stefan.langerman@ulb.ac.be
e) boris.a.zolotov@yandex.com

© 2020 Information Processing Society of Japan
1.1.1 Small and Big Cells

The main high-level idea is to divide the cells of the Voronoi diagram into small and big: small are the ones that have at most $N^{1/4}$ vertices and big are the ones that have more. For a Voronoi cell $f$, by the paws of $f$ we denote the Voronoi vertices connected to the boundary of $f$ by one edge. What we do to process small and big cells is different, but is based on the following fact. Given an affected cell $f$ the neighboring cells of $f$ that change can be identified by finding all the paws of $f$ whose Voronoi circles contain the newly inserted site $s_N$.

A small cell is small enough to be processed by simply checking all $O(N^{1/4})$ Voronoi circles of its paws in a brute force way. This takes $O(N^{1/4} \log(N)) = O(N^{1/4})$ time per small cell. Since the amortized number of cells changing is $O(\sqrt{N})$, it takes $O(\sqrt{N} \cdot N^{1/4}) = O(N^{3/4})$ amortized time to perform all updates involving small cells.

For each big cell with $b_i$ neighbors, we store a circular linked list of $\Theta(b_i/N^{1/4})$ data structures, each associated with the consecutive range of $O(N^{1/4})$ of its paws. Each structure stores the Voronoi circles for those paws that are relevant. These Dynamic Circle-Reporting structures (DCRs) are known variants of the DNN structure that support insertion and deletion of circles in polylogarithmic time, and given a query point, report all $k$ circles containing the point in time $O(k)$. Overall, operations involving a big cell require polylogarithmic time in the number of affected neighbors. Since there are at most $O(N^{1/4})$ big cells, the total time to process them is $O(N^{3/4} \log(N)) = O(N^{3/4})$.

Overall, we need $O(N^{3/4})$ amortized time for updating both small and big cells, thus the main result follows.

1.1.2 Note on Previous Work

In a preliminary version of this paper [3] we presented the same result using a randomized data structure and providing bounds on the expected running time; the randomization came solely from the shallow cuttings used in the DNN structure [5]. Since the initial development of this work, Chan presented DNN structures that use shallow cuttings deterministically [6] (implicit in [7]); using this structure all DNN and DCR structures used in this paper can be implemented deterministically. As a result, our overall structure is deterministic.

1.1.3 Organization

The rest of the paper is organized as follows. In Section 2.1 we characterize affected cells that undergo combinatorial changes. In Section 3 we give a detailed description of our data structure. In Section 4 we present the sequence of actions performed with each insertion of a site. In Section 5 we present the procedure to find all the affected cells, and in Sections 6 and 7 we describe the procedure to implement the combinatorial changes and update the data structure accordingly.

2. Preliminaries and Definitions

We begin with standard definitions related to Voronoi diagrams and their basic properties.

Let $S := \{s_1, s_2, \ldots, s_N\}$ be a set of $N$ distinct points in $\mathbb{R}^2$; these points are called sites. Let $\text{dist}(\cdot, \cdot)$ denote the Euclidean distance between two points in $\mathbb{R}^2$. We assume that the sites in $S$ are in general position, that is, no four sites lie on a common circle.

**Definition 1.** The Voronoi diagram of $S$ is the subdivision of $\mathbb{R}^2$ into $N$ cells, called Voronoi cells, one cell for each site in $S$, such that a point $q$ lies in the Voronoi cell of a site $s_i$ if and only if $\text{dist}(q, s_i) < \text{dist}(q, s_j)$ for each $s_j \in S$ with $j \neq i$.

Let $f_i$ denote the Voronoi cell of a site $s_i$. Edges of the Voronoi diagram, called Voronoi edges, are portions of bisectors between two sites which are the common boundary of the corresponding Voronoi cells. Voronoi vertices are points where at least three Voronoi cells meet. The Voronoi circle of a Voronoi vertex $v$ is the circle passing through the sites whose cells are incident to $v$, see Fig. 1 (a). Vertex $v$ is the center of its Voronoi circle.

Since the sites are in the general position, each Voronoi vertex has degree three. Each Voronoi edge is either a segment or a ray and the graph of the Voronoi diagram formed by its edges and vertices is planar and connected.

The next three definitions are specific to our data structure.

**Definition 2.** The size of a Voronoi cell is the number of Voronoi edges constituting its boundary. We denote the size of cell $f$ by $|f|$. **Definition 3.** A Voronoi cell of a Voronoi diagram with $N$ sites is called a big cell if it has size more than $N^{1/4}$. Otherwise it is called small.

**Definition 4.** The paws of Voronoi cell $f$ are the Voronoi vertices

![Fig. 1](a) A Voronoi diagram and (a) a Voronoi circle of vertex $v$ and (b) paw $e$ of Voronoi cell $f$. [Diagram](b)
that are connected to the boundary of \( f \) by an edge and are not themselves on the boundary of \( f \), see Fig. 1 (b). A paw is called relevant if it is not incident to a big cell.

2.1 Combinatorial Changes to the Voronoi Diagram and the Flarb Operation

We now overview the definitions and results from Allen et al. [1] that we need to present our approach. In order to prove the \( \Theta(N^2) \) bound on the number of combinatorial changes caused by insertion of a site, a graph operation called flarb is introduced.

Let \( G \) be a planar 3-regular graph embedded in \( \mathbb{R}^2 \) without edge crossings (edges are not necessarily straight-line). Let \( C \) be a simple closed Jordan curve in \( \mathbb{R}^2 \).

**Definition 5.** Curve \( C \) is called flarbable for \( G \) if:
- the graph induced by vertices inside the interior of \( C \) is connected,
- \( C \) intersects each edge of \( G \) either at a single point or not at all,
- \( C \) passes through no vertex of \( G \), and
- the intersection of \( C \) with each face of \( G \) is path-connected.

For example, curve \( C \) in Fig. 2 (a) is not flarbable since the intersection between its interior (shaded green) and the highlighted face (red) consists of two disconnected parts. In Fig. 2 (b) curve \( C' \) is flarbable.

Given a graph \( G \) and a curve \( C \) flarbable for \( G \), the flarb operation is, informally, removing part of \( G \) that is inside \( C \) and replacing it with \( C \). Formally, the flarb operation for \( G \) and \( C \) is defined as follows (see Fig. 2 (b), Fig. 2 (c)):
- For each edge \( e_i \in G \) that intersects \( C \) let \( u_i \) be its vertex lying inside \( C \) and \( v_i \) its vertex outside \( C \). Create a new vertex \( w_i = C \cap e_i \) and connect it to \( v_i \) along \( e_i \).
- Connect consecutive vertices \( w_i \) along \( C \).
- Delete all the vertices and edges inside \( C \).

Let \( \mathcal{G}(G, C) \) denote the graph obtained by applying the flarb operation to graph \( G \) and curve \( C \).

**Proposition 1.** The following holds for graph \( \mathcal{G}(G, C) \): (a) \( \mathcal{G}(G, C) \) has at most two more vertices than \( G \) does; (b) \( \mathcal{G}(G, C) \) is a 3-regular planar graph; (c) \( \mathcal{G}(G, C) \) has at most one more face than \( G \) does.

**Proof.** Items (a) and (b) are proved in Ref. [1], Lemma 2.2. To prove (c) note that there is one new face bounded by the cycle added along \( C \) while performing the flarb. All the other faces of \( G \) are either deleted, left intact, or cropped by \( C \); these operations obviously do not increase the number of faces. \( \Box \)

**Theorem 2** (Ref. [1]). Let \( G \) be a graph of the Voronoi diagram of a set of \( N-1 \) sites \( s_1 \ldots s_{N-1} \). For any new site \( s_N \) there exists a flarbable curve \( C \) such that the graph of the Voronoi diagram of sites \( s_1 \ldots s_N \) is \( \mathcal{G}(G, C) \).

2.1.1 Cost of the Flarb

We want to analyze the number of structural changes that a graph undergoes when we apply the flarb operation to it. There are two basic combinatorial operations on graphs:
- **Link** is the addition of an edge between two non-adjacent vertices.
- **Cut** is the removal of an existing edge.

Other combinatorial operations, for example insertion of vertex of degree 2, are assumed to have no cost.

**Definition 6.** \( \text{cost}(G, C) \) is the minimum number of links and cuts needed to transform \( G \) into \( \mathcal{G}(G, C) \).

Note that sometimes there are less combinatorial changes needed than the number of edges intersected by \( C \). Consider edges \( e_1, e_2 \) of \( G \) crossed consecutively by \( C \) and edge \( n \) adjacent to them that reappears in \( \mathcal{G}(G, C) \) as a part \( n' \) of \( C \). Then \( n' \) can be obtained without any links or cuts by lifting \( n \) along \( e_1 \) and \( e_2 \) until it coincides with \( n' \) or (which is the same) shrinking \( e_1 \) and \( e_2 \) until their endpoints coincide with their intersections with \( C \) (see Fig. 3). We will call it preserving operation.

**Theorem 3** (Ref. [1]). For a flarbable curve \( C \), it holds that

\[
\text{cost}(G, C) \leq 12|\mathcal{S}(G, C)| + 3|B(G, C)| + O(1).
\]

Where
- \( |B(G, C)| \) is the number of faces of \( G \) wholly contained inside \( C \) (as is such a face on Fig. 2 (b)),
- \( |\mathcal{S}(G, C)| \) is the number of shrinking faces — i.e., the faces whose number of edges decreases when flarb operation is applied (face \( f \) is shrinking on Fig. 2 (b)–2 (c)).

The following upper bound can be used to evaluate the number of combinatorial changes needed to update the graph of a Voronoi diagram when a new site is inserted.
Theorem 4 (Ref. [1]). Consider one insertion of a new site to a Voronoi diagram \( V \).

- The number of cells of \( V \) undergoing combinatorial changes is \( O(N^2) \) amortized in a sequence of insertions;
- There are a constant number of combinatorial changes per cell;
- The cells of \( V \) with combinatorial changes form a connected region.

Further in this paper by a change in cell we always mean a combinatorial change, that is a link or a cut.

3. Description of Data Structure

Our data structure consists of the following parts.

- The graph \( G_N \) of the Voronoi diagram represented by its adjacency list: for each Voronoi vertex \( v \) we store the list of all Voronoi vertices connected to \( v \). Since the sites are in the general position, each list has length 3, therefore we can find and replace its elements in constant time. Thus any link or cut can be performed in constant time as well as insertion or deletion of a vertex of degree 2.
- For each vertex \( v \) its data \( D_v \) is stored. It is a list of the three sites that define the Voronoi circle of \( v \), that is, the sites whose cells are incident to \( v \).
- A dynamic nearest neighbor structure (DNN) [6] for the sites which supports insertion and deletion of sites and nearest neighbor queries in \( \tilde{O}(1) \) amortized time.
- The graph \( \Gamma_N \) of big cells which is simply the dual graph to the subgraph of \( G_N \) formed by big cells. Vertices of \( \Gamma_N \) are big cells themselves and edges connect vertices corresponding to pairs of big cells that are adjacent. Graph \( \Gamma_N \) has \( O(N^2) \) edges, since it is a planar graph of at most \( N^2 \) vertices. For each pair of adjacent big cells \( b_1, b_2 \) we also store two Voronoi vertices they share. We store graph \( \Gamma_N \) as an adjacency list, where for each vertex, its edges are stored in a binary search tree ordered counterclockwise around the corresponding big cell. The vertices of \( \Gamma_N \) are stored in a binary search tree. This allows us to access any edge of \( \Gamma_N \) in \( \tilde{O}(1) \) time.
- For each big cell \( b \) store a circular linked list of \( \Theta(|b| / N^{1/2}) \) data structures each associated with a consecutive range of \( O(N^{1/2}) \) paws of \( B_b \); see Fig. 4. Each structure stores the Voronoi circles of the relevant paws of \( b \) (recall that a paw is relevant if it is not incident to a big cell, see Definition 4).

4. Insertion of a Site

We aim to implement the update of graph \( G_{N-1} \) to become \( G_N \) when a new site \( s_N \) is added to the Voronoi diagram. Our goal is to quickly locate the affected cells that need combinatorial changes, and to avoid processing the other cells. When the cells that need changes are located, we implement these changes using the techniques of Ref. [1].

Let the cell of the new site \( s_N \) be called \( f_N \). We denote the boundary of \( f_N \) by \( C_N \). According to Theorem 2, what we are about to perform is the flarb operation on graph \( G_{N-1} \) of the current Voronoi diagram and curve \( C_N \).

We first use the DNN structure to locate one Voronoi cell, call it \( f_{\text{flarb}} \), that must change — the one whose site is the closest to newly added \( s_N \). We then add \( s_N \) to the DNN. Finally we create the queue with all big cells of \( G_{N-1} \) and cell \( f_{\text{flarb}} \). This whole procedure takes \( \tilde{O}(1) \) time as the list of all big cells is already stored.

We then remove each cell \( f \) from the queue, process it, and add into the queue the small cells neighboring \( f \) with unprocessed changes. We do not have to add big cells neighboring \( f \) as all of them were already in the initial queue and thus will be processed. Figure 6 shows a pseudocode of this procedure.

5. Recognizing Cells with Changes

Let \( f \) be a cell with combinatorial changes. We can identify the neighboring cells of \( f \) that change using the following theorem:

Theorem 5 (Ref. [1]). Let \( g \) be a cell adjacent to \( f \). Let \( v_1, v_2 \) be the vertices of \( g \) that are paws of \( f \). Cell \( g \) needs to undergo combinatorial changes if and only if the Voronoi circle of \( v_1 \) or \( v_2 \) encloses \( s_N \).

See Fig. 5 (a) for an example. Cell \( f \) is a cell with changes, \( n_1 \) and \( n_2 \) are its paws. The Voronoi circle of \( n_1 \) encloses the new site \( s_N \) and the one of \( n_2 \) does not. Therefore cells \( f_1 \) and \( f_2 \) need combinatorial changes as they are incident to vertex \( n_1 \), and cell \( f_3 \) does not need any changes.

We now consider separately the case when cell \( f \) is a big cell...
Fig. 5 Identifying Voronoi cells that need changes. (a) Voronoi circle of vertex $n_1$ encloses $s_N$, and the circle of $n_2$ does not. (b) $v$ is a paw of big cell $f$ returned by a DCR. Highlighted are the cells that are to be added to the queue.

5.2 Cell $f$ is Small
We can look at every paw $n_i$ of $f$ and identify those, whose Voronoi circles enclose $s_N$. This requires $O(N^2)$ time in total. We add to $Q$ small cells adjacent to $f$ that are incident to these paws as they need changes according to Theorem 5.

6. Implementing Combinatorial Changes
In this section we describe how to implement combinatorial changes in a cell $f$ which lies in Changed. We again consider separately the case when $f$ is big (Section 6.1) and the case when $f$ is small (Section 6.2).

6.1 Processing Big Cells
Processing a big cell $f$ consists of the following four steps:

6.1.1 Updating the Vertices
We update a continuous range of $f$’s vertices $v_1$...$v_k$—we need to change their data $D_{v_1}$...$D_{v_k}$ to indicate that these vertices are now incident to the cell of $s_N$ and not the cell of $s$. This can be done in $O(1)$ time using the yard tree $T_f$.

6.1.2 Updating the Graph of the Voronoi Diagram
The first thing to do is a link along $C_N$ creating two vertices: vertex $v_1$ incident to $f$, $f_N$, $f_{Left}$, and vertex $v_2$ incident to $f$, $f_{Right}$ (see Fig. 5 (b)). After this link the part of $f$ inside $C_N$ becomes a part of the new cell $f_N$ — luckily, all vertices of this part are already updated during the previous step.

There may be some big cells adjacent to $f$ that are already processed, creating other parts of the new cell. We have to join these parts together by cutting edges of $f$ that have portions inside $C_N$ and are incident to already processed big cells. Finding these edges in a straightforward way could be slow as $f$ can have a really large number of edges inside $C_N$ and we do not have enough time to look at each of them individually. Luckily, graph $\Gamma_{N-1}$ contains the information about edges shared by big cells. Thus we can in $O(1)$ time find and delete edges incident to both $f$ and already processed big cells inside $C_N$. The edges shared by $f$ and other big cells inside $C_N$ will be deleted when these other big cells will be processed.
6.1.3 Updating the Graph of Big Cells

The two operations we just carried out — split of \( f \) by the new edge \( e_i e_j \) and joining of some cells that are parts of \( f_N \) — can change the set of big cells and add or cut some connections between them. However, \( \Gamma_{N-1} \) can be updated accordingly in \( \tilde{O}(1) \) time when such an operation is executed. It can be done as follows:

While undergoing a split, vertex \( f \) falls apart into two vertices: \( f' \) and \( f'' \) (the latter represents a part of the new cell). They share newly created edge \( e_i e_j \) of the Voronoi diagram. Note that the cells adjacent to \( f'' \) form a continuous range of cells that were adjacent to \( f \).

Thus we need \( \tilde{O}(1) \) time to cut a continuous range from the binary search tree of cells adjacent to \( f \). \( \tilde{O}(1) \) time to add a new edge between \( f' \) and \( f'' \) to their binary search trees and \( \tilde{O}(1) \) time to re-balance the binary search tree of all big cells.

Joining can be also done in \( \tilde{O}(1) \) time. When two cells \( f_1, f_2 \) are joined we remove a node corresponding to \( f_1 \) from binary search tree of \( f_2 \) and vice versa, this takes \( \tilde{O}(1) \) time. We then join the trees of \( f_1 \) and \( f_2 \) also in \( \tilde{O}(1) \) time since cells that were adjacent to \( f_1 \) form a continuous range of cells adjacent to the new one.

6.1.4 Fixing Data Structures

There are two data structures associated with \( f \) that have to be considered:

- \( T_f \) can be updated in \( \tilde{O}(1) \) time the same way we did with the graph of big cells.
- DCRs of relevant paws: when big cells are joined or split, most of DCR-s stay intact. The only DCRs that need to be rebuilt are those whose range contains the endpoints of the edge that is either cut or added. Rebuilding of a DCR takes \( \tilde{O}(N^2) \) time since at most \( \Theta(N^2) \) circles are stored there.

6.2 Processing Small Cells

A small cell is different from a big cell in that we can consider every edge of it, and it will take us \( \tilde{O}(N^2) \) time. We will implement the combinatorial changes in \( f \), and after this we update the DCRs of neighboring big cells.

We can in time \( \tilde{O}(N^2) \) distinguish whether \( f \) has one, two, or more vertices inside the new cell (if they exist). Below we describe these three cases separately.

6.2.1 One Vertex Inside the New Cell

See Fig. 7 (a). Let \( f_i, f_j \) be the cells adjacent to \( f \) that share an edge with \( f \) inside \( C_N \). Let those edges be called \( e_i, e_j \) respectively.

It is certain that neither \( f_i \) nor \( f_j \) have been processed yet: if \( f_i \) is processed then there would be a vertex \( v = C_N \cap e_i \). Then we have to create the face in the graph that is separated from \( f, f_i, f_j \), bounded by \( C_N \), and is a part of the new cell \( f_N \).

To do so, we perform a link operation inside \( f \) along \( C_N \): we create new vertices \( v_1 \) on \( e_i, e_j \) on \( e_i e_j \) and add an edge \( v_1 v_2 \) to \( G_{N-1} \), see Fig. 7 (b). \( v_1 \) is incident to the cells of sites \( s, s_j \) and \( s_N \); \( v_2 \) is incident to the cells of \( s, s_j \) and \( s_N \).

6.2.2 Two Vertices Inside the New Cell

We check whether cells adjacent to \( f \) that share an edge with it inside \( C_N \) have been already processed. If not (see Fig. 8 (a)), we perform a link operation inside \( f \) similarly to the previous paragraph, see Fig. 8 (b).

Otherwise let us denote three faces sharing an edge with cell \( f \) inside curve \( C_N \) by \( f_1, f_2, f_3 \), see Fig. 9 (a).

**Lemma 6.** It is only \( f_2 \) that can have been already processed.

**Proof.** Suppose \( f_1 \) is processed. It must then have an edge along \( C_N \). It implies that there is a vertex where \( C_N \) meets the common edge of \( f_1 \) and \( f \). This vertex becomes the third vertex of \( f \) inside \( C_N \). However, \( f \) has only two such vertices, which is a contradiction. \( \square \)

If \( f_2 \) is processed and is part of \( f_N \) then the data \( \mathcal{D}_N, \mathcal{D}_z \) of its vertices was updated when we were processing it. Therefore \( f \) does not need to undergo any combinatorial changes, the common edge of \( f \) and \( f_N \) can be obtained by preserving operation which was described in Section 2.1, see Fig. 3. Thus no links and cuts are required, see Fig. 9 (b). This completes implementing changes in \( f \).
6.2.3 More Vertices Inside the New Cell

Again we check whether any of the cells adjacent to \( f \) have already been processed. If not, it is enough to perform one link creating two new vertices \( v_1, v_2 \) and to update the data \( D_v \) of all the vertices of cell \( f \) between \( v_1 \) and \( v_2 \); now they are incident to the cell of \( s_N \), see Fig. 10 (a).

If some cells sharing an edge with \( f \) inside \( C_N \) are already processed and represent a part of new cell, then for each processed cell \( f' \) adjacent to \( f \) we also perform a cut removing their common edge \( e \) and then remove vertices incident to this edge that now have degree 2, see Fig. 10 (b).

6.2.4 Updating the DCRs of Big Neighbors of \( f \)

The last step is that for every vertex \( v \) of \( f \) whose list of adjacent cells has changed during update of \( G_{N-1} \) we find all big cells for which \( v \) is a paw (there are at most three such cells, since \( v \) has degree 3), recalculate the Voronoi circle of \( v \), and update Voronoi circle of \( v \) in DCRs of those cells which takes \( O(1) \) time.

7. When Small Cells Become Big

When the size of a cell crosses the threshold of \( N^\frac{1}{4} \), it can be easily identified since we store all the sizes. If a big cell \( b \) is split into a number of cells and one of them is small, or if \( N \) becomes greater than \( |b|^\frac{1}{3} \), we delete all the structures associated with it, including DCRs and the structure \( T_f \). We also remove from \( \Gamma_{N-1} \) the vertex corresponding to \( b \).

The other way around, a new big cell can appear in the diagram when:

\- the new cell \( f_N \) intersects many of the old cells and has more than \( N^\frac{1}{2} \) vertices, or
\- a cell \( f_k \) with \( N^\frac{1}{2} - 1 \) vertices has one vertex inside \( f_k \) and gets one additional vertex while being processed, see Fig. 7.

New cell \( f_N \) inherits the portion of its DCRs from its parts that previously were parts of big cells. The number of circles of paws of previously small cells that need to be added to DCRs can be bounded from above by the size of a small cell times the number of cells that undergo changes — that is,

\[ N^\frac{1}{2} \cdot N^\frac{1}{4} = N^\frac{3}{4}. \]

The structure \( T_{f_k} \) is inherited in part by \( f_k \) from big cells that intersect curve \( C_N \). The number of vertices that have to be added to \( T_{f_k} \) after that is bounded from above by the number of combinatorial changes in current insertion.

The cell \( f_k \) still has size \( |f_k| \leq 2N^\frac{1}{4} \), so yard tree \( T_{f_k} \) can be built in \( O(N^\frac{3}{4}) \): it only takes time polylogarithmic in the size of the cell to add each vertex.

8. Correctness and Time Complexity

**Theorem 7.** Inserting a new site \( s_N \) to our data structure and updating it requires \( \tilde{O}(N^\frac{3}{4}) \) amortized time.

**Proof.** Let \( s \) be the number of small cells that change, and \( b_1, b_2, \ldots, b_{|B|} \) be the big cells. Let \( \ell_i \) be the number of circles returned by the DCR structures of \( b_i \).

All the operations on a small cell take \( \tilde{O}(N^\frac{3}{4}) \) time. For all the big cells together the operations on updating the graph structure and the graph of big cells require \( \tilde{O}(N^\frac{3}{4}) \) total time. The number of DCR-s that have to be rebuilt is bounded from above by the number of changes in the graph.

Finally, the amortized time complexity is

\[ \tilde{O}\left(2N^\frac{3}{4} + \sum_{i=1}^{|B|} \left( \frac{|b_i|}{N^\frac{1}{4}} + \ell_i \right) + N^\frac{3}{4} + sN^\frac{3}{4} \right). \]

Since \( s \) is \( O(N^\frac{3}{4}) \) amortized [1], \( \sum_{i=1}^{|B|} |b_i| \leq N, |B| \leq N^\frac{1}{4} \), and \( \sum_{i=1}^{|B|} \ell_i \leq sN^\frac{3}{4} \), this is simply \( \tilde{O}(N^\frac{3}{4}) \) amortized. \( \square \)

9. Discussion

The problem of maintaining the convex hull of a set of points in \( \mathbb{R}^3 \) subject to point insertion can also be solved using our data structure. Namely, consider the dual problem of maintaining the
intersection of a set of halfspaces. The two blocks of our data structure that are specific for Voronoi diagrams, translate in this setting as follows. To find the first face affected by the insertion (or report that it does not exist) we need to find the vertex extreme in the direction normal to the plane being inserted; if it is affected, then all the three incident faces are affected. We check whether a vertex is affected by determining the above/below relation between this vertex and the plane being inserted. Thus Chan’s structure [6] is again enough for our needs.

There remains a gap between the $O(N^{3/4})$ expected amortized runtime of our structure and the $\Theta(\sqrt{N})$ amortized number of combinatorial changes to the Voronoi diagram. Also, it would be interesting to get output-sensitive bounds, where the update time depends on the number of combinatorial changes. This was achieved by Allen et al. [1] for points in convex position, where their update time is $O(K \log^3 N)$, where $K$ is the number of combinatorial changes. We are unable to show this using our technique, due to the fact that we need to process all $\Theta(N^{3/4})$ big cells, no matter how many of them undergo combinatorial changes.

Acknowledgments This work was completed during the Second Trans-Siberian Workshop on Geometric Data Structures. We thank the organizers and staff of Russian Railways (РЖД) for creating an ideal research environment.

S. L. is the Directeur de recherches du F.R.S.-FNRS. J. I. and G. K. are supported by the Fonds de la Recherche Scientifique-FNRS under Grant no. MISU F 6001 1. E. A. and B. Z. are partially supported by RBF Grant no 20-01-00488 and by “Native towns”, a social investment program of PJSC “Gazprom Neft”. J. I. is supported by NSF grant CCF-1535364.

References

[1] Allen, S.R., Barba, L., Iacono, J. and Langerman, S.: Incremental Voronoi Diagrams, *Discrete & Computational Geometry*, Vol.58, No.4, pp.822–848 (online), DOI: 10.1007/s00454-017-9943-2 (2017).

[2] Aronov, B., Bose, P., Demaine, E.D., Gudmundsson, J., Iacono, J., Langerman, S. and Smid, M.H.M.: Data Structures for Halfplane Proximity Queries and Incremental Voronoi Diagrams, *Algorithmica*, Vol.80, No.11, pp.3316–3334 (online), DOI: 10.1007/s00453-017-0389-y (2018).

[3] Arseneva, E., Iacono, J., Koumoutsos, G., Langerman, S. and Zolotov, B.: Sublinear Explicit Incremental Planar Voronoi Diagrams (Extended Abstract), *Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCG3)* (2019).

[4] Aurenhammer, F. and Klein, R.: Voronoi Diagrams, *Handbook of Computational Geometry*, Sack, J. and Urrutia, J. (Eds.), pp.201–290, North Holland/Elsevier (online), DOI: 10.1016/b978-044482537-7/50006-1 (2000).

[5] Chan, T.M.: A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries, *J. ACM*, Vol.57, No.3, pp.16:1–16:15 (online), DOI: 10.1145/1706591.1706596 (2010).

[6] Chan, T.M.: Dynamic Geometric Data Structures via Shallow Cuttings, *55th International Symposium on Computational Geometry*, SoCG’2019, pp.24:1–24:13 (online), DOI: 10.4230/LIPIcs.SoCG.2019.24 (2019).

[7] Chan, T.M. and Tsakalidis, K.: Optimal Deterministic Algorithms for 2-d and 3-d Shallow Cuttings, *Discrete & Computational Geometry*, Vol.56, No.4, pp.866–881 (online), DOI: 10.1007/s00454-016-9784-4 (2016).

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms*, Massachusetts Institute of Technology, 3rd edition (2009).

[9] Kaplan, H., Mulzer, W., Roditty, L., Seiferth, P. and Sharir, M.: Dynamic Planar Voronoi Diagrams for General Distance Functions and their Algorithmic Applications, *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pp.2495–2504 (online), DOI: 10.1137/1.9781611974782.165 (2017).

[10] Pettie, S.: Applications of Forbidden 0-1 Matrices to Search Tree and Path Compression-Based Data Structures, *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pp.1457–1467 (online), DOI: 10.1137/1.9781611973075.118 (2010).

[11] Sleator, D.D. and Tarjan, R.E.: A Data Structure for Dynamic Trees, *J. Comput. Syst. Sci.*, Vol.26, No.3, pp.362–391 (online), DOI: 10.1016/0022-0000(83)90065-5 (1983).

Elena Arseneva obtained her Ph.D. in Informatics from Universita della Svizzera italiana (USI), Lugano, in 2016. She is now an Assistant Professor at the Mathematics and Computer Science Department of Saint-Petersburg State University, Russia. Her research interests are in Algorithms and Data Structures, Discrete Mathematics, and Computational Geometry.

John Iacono received his Ph.D. from Rutgers University in 2001, and then joined what became the Tandon School of Engineering at New York University. In 2017 he moved to the Algorithms Research Group at the Université libre de Bruxelles in Belgium where he is currently a professor. He is a recipient of a Sloan fellowship and a Fulbright fellowship. His interests are in data structures, algorithms, and computational geometry.

Grigorios Koumoutsos obtained his Ph.D. from Eindhoven University of Technology (TU/e) in 2018. He is now a postdoctoral researcher at the Computer Science Department of ULB. His research interests include Online Algorithms, Data Structures and Computational Geometry.

Stefan Langerman obtained his Ph.D. from Rutgers University in 2001. In 2002 he joined the Computer Science Department at the Université Libre de Bruxelles (ULB) in Belgium where he is currently Associate Professor and Research Director for the F.R.S.-FNRS. His main research interests are in Discrete and Computational Geometry, Algorithms and Data Structures.
Boris Zolotov obtained his Bachelor's degree in Mathematics from St. Petersburg State University in 2019. He is now doing Master studies there at the Department of Mathematics and Computer Science. His main research interests are in Computational Geometry, Data Structures and Algorithms.