Planar Distance Oracles with Better Time-Space Tradeoffs*

Yaowei Long  
IIS, Tsinghua University

Seth Pettie  
University of Michigan

Abstract

In a recent breakthrough, Charalampos Charalampopoulos, Gawrychowski, Mozes, and Weimann [9] showed that exact distance queries on planar graphs could be answered in $n^{o(1)}$ time by a data structure occupying $n^{1+o(1)}$ space, i.e., up to $o(1)$ terms, optimal exponents in time $0$ and space $1$ can be achieved simultaneously. Their distance query algorithm is recursive: it makes successive calls to a point-location algorithm for planar Voronoi diagrams, which involves many recursive distance queries. The depth of this recursion is non-constant and the branching factor logarithmic, leading to $(\log n)^{o(1)} = n^{o(1)}$ query times.

In this paper we present a new way to do point-location in planar Voronoi diagrams, which leads to a new exact distance oracle. At the two extremes of our space-time tradeoff curve we can achieve either

$$n^{1+o(1)} \text{ space} \quad \text{and} \quad \log^{2+o(1)} n \text{ query time}, \quad \text{or}$$

$$n \log^{2+o(1)} n \text{ space} \quad \text{and} \quad n^{o(1)} \text{ query time}.$$ 

All previous oracles with $\tilde{O}(1)$ query time occupy space $n^{1+\Omega(1)}$, and all previous oracles with space $\tilde{O}(n)$ answer queries in $n^{\Omega(1)}$ time.

*This work was supported by NSF grants CCF-1637546 and CCF-1815316, and a grant from IIS, Tsinghua University.
1 Introduction

A distance oracle is a data structure that answers distance queries (or approximate distance queries) w.r.t. some underlying graph or metric space. On general graphs there are many well known distance oracles that pit space against multiplicative approximation \[O(1)\], space against mixed multiplicative/additive approximation \[O(\kappa)\], and, in sparse graphs, space against query time \[O(\kappa \log \log n)\]. Refer to Sommer [35] for a survey on distance oracles.

Whereas approximation seems to be a necessary ingredient to achieve any reasonable space/query time on general graphs, structured graph classes may admit exact distance oracles with attractive time-space tradeoffs. In this paper we continue a long line of work [3, 15, 10, 28, 32, 33, 7, 32, 11, 20, 9] focused on exact distance oracles for weighted, directed planar graphs.

History. Between 1996-2012, work of Arikati et al. [3], Djidjev [15], Chen and Xu [10], Fukcharoenphol and Rao [18], Klein [28], Wulff-Nilsen [42], Nussbaum [33], Cabello [7], and Mozes and Sommer [32] achieved space \(\tilde{O}(S)\) and query time \(\tilde{O}(n/\sqrt{S})\), for various ranges of \(S\) that ultimately covered the full range \([n,n^2]\).

In 2017, Cabello [8] introduced planar Voronoi diagrams as a tool for solving metric problems in planar graphs, such as diameter and sum-of-distances. This idea was incorporated into new planar distance oracles, leading to \(\tilde{O}(n^{5/2}/S^{3/2})\) query time \([14]\) for \(S \in [n^{3/2}, n^{5/3}]\) and \(\tilde{O}(n^{3/2}/S)\) query time \([20]\) for \(S \in [n, n^{1/2}]\). Finally, in a major breakthrough Charalampopoulos, Gawrychowski, Mozes, and Weimann [9] demonstrated that up to \(n^{o(1)}\) factors, there is no tradeoff between space and query time, i.e., space \(n^{1+o(1)}\) and query time \(n^{o(1)}\) can be achieved simultaneously. In more detail, they proved that space \(O(n^{4/3}/\log n)\) allows for query time \(O(\log^2 n)\), space \(\tilde{O}(n^{1+\epsilon})\) allows for query time \(O(\log n)^{1/\epsilon - 1}\), and space \(O(n \log^{2+1/\epsilon} n)\) allows for query time \(O(n^{2\epsilon})\).

The Charalampopoulos et al. structure is based on a hierarchical \(r\)-decomposition of the graph, \(r = (n, n^{(m-1)/m}, \ldots, n^{1/m})\). (See Section 2.) Given \(u, v\), it iteratively finds the last boundary vertex \(u_i\) on the shortest \(u-v\) path that lies on the boundary of the level-\(i\) region containing \(u\). Given \(u_{i-1}\), finding \(u_i\) amounts to solving a point location problem on an external Voronoi diagram, i.e., a Voronoi diagram of the complement of a region in the hierarchy. Each point location query is solved via a kind of binary search, and each step of the binary search involves 3 recursive distance queries that begin at a “higher” level in the hierarchy. This leads to a tradeoff between space \(\tilde{O}(n^{1+1/m})\) and query time \(O(\log n)^{m-1}\).

See Table 1 for a summary of the space-time tradeoffs exact and approximate planar distance oracles.

New Results. In this paper we develop a more direct and more efficient way to do point location in external Voronoi diagrams. It uses a new persistent data structure for maintaining sets of non-crossing systems of chords, which are paths that begin and end at the boundary vertices of a region, but are internally vertex disjoint from the region. By applying this point location method in the framework of Charalampopoulos et al. [9], we obtain a better time-space tradeoff, which is most noticeable at the “extremes” when \(O(n)\) space or \(O(1)\) query time is prioritized.

Theorem 1.1. Let \(G\) be an \(n\)-vertex weighted planar digraph with no negative cycles, and let \(\kappa, m \geq 1\) be parameters. A distance oracle occupying space \(O(mn^{1+1/m+1/\kappa})\) can be constructed in \(\tilde{O}(n^{3/2+1/m} + n^{1+1/m+1/\kappa})\) time that answers exact distance queries in \(O(2^m \kappa \log^2 n \log \log n)\) time. At the two extremes of the space-time tradeoff curve, we can construct oracles in \(n^{3/2+o(1)}\) time with either

- \(n^{1+o(1)}\) space and \(\log^{2+o(1)} n\) query time, or

- \(n \log^{2+o(1)} n\) space and \(n^{o(1)}\) query time.

Our new point-location routine suffices to get the query time down to \(O(\log^3 n)\). In order to reduce it further to \(O(\log^{2+o(1)} n)\), we develop a new dynamic tree data structure based on Euler-Tour trees [23] with \(O(\kappa n^{1/\kappa})\) update time and \(O(\kappa)\) query time. This allows us to generate MSSP (multiple-source shortest paths) structures with a similar space-query tradeoff, specifically, \(O(\kappa n^{1+1/\kappa})\) space and \(O(\kappa \log \log n)\) query time. Our MSSP construction follows Klein [28] (see also [20]), but uses our new dynamic tree in lieu of
| Reference | Space | Query Time |
|-----------|-------|------------|
| Arikati, Chen, Chew, Das, Smid & Zaroliagis 1996 | $S \in [n^{3/2}, n^2]$ | $O\left(\frac{n^2}{S}\right)$ |
| Djidjev 1996 | $S \in [n, n^2]$ | $O\left(\frac{n^2}{S}\right)$ |
|  | $S \in [n^{4/3}, n^{3/2}]$ | $O\left(\frac{n}{\sqrt{S}} \log n\right)$ |
| Chen & Xu 2000 | $S \in [n^{4/3}, n^2]$ | $O\left(\frac{n}{\sqrt{S}} \log \left(\frac{n}{S}\right)\right)$ |
| Fakcharoenphol & Rao 2006 | $O(n \log n)$ | $O(\sqrt{n} \log^2 n)$ |
| Wulff-Nilsen 2010 | $O(n^2 \log^4 \log n \log n)$ | $O(1)$ |
| Nussbaum 2011 | $O(n)$ | $O(n^{1/2+\epsilon})$ |
|  | $S \in [n^{4/3}, n^2]$ | $O\left(\frac{n}{\sqrt{S}}\right)$ |
| Cabello 2012 | $S \in [n^{4/3} \log^{1/3} n, n^2]$ | $O\left(\frac{n}{\sqrt{S}} \log^{3/2} n\right)$ |
| Mozes & Sommer 2012 | $S \in [n \log \log n, n^2]$ | $O\left(\frac{n}{\sqrt{S}} \log^2 n \log^{3/2} \log n\right)$ |
|  | $O(n)$ | $O(n^{1/2+\epsilon})$ |
| Cohen-Addad, Dahlgaard & Wulff-Nilsen 2017 | $S \in [n^{3/2}, n^{5/3}]$ | $O\left(\frac{n^{3/2}}{S^{3/2}} \log n\right)$ |
| Gawrychowski, Mozes, Weimann & Wulff-Nilsen 2018 | $\tilde{O}(S)$ for $S \in [n, n^{3/2}]$ | $\tilde{O}\left(\frac{n^{3/2}}{S}\right)$ |
| Charalampopoulos, Gawrychowski, Mozes & Weimann 2019 | $O(n^{4/3} \sqrt{\log n})$ | $O(\log^2 n)$ |
|  | $n^{1+o(1)}$ | $n^{o(1)}$ |
| new 2020 | $n^{1+o(1)}$ | $\log^{2+o(1)} n$ |
|  | $n \log^{2+o(1)} n$ | $n^{o(1)}$ |

| (1 + $\epsilon$)-Approx. Oracles | Space | Query Time |
|-------------------------------|-------|------------|
| Thorup 2001 | $O(ne^{-1} \log^2 n)$ | $O(\log \log n + \epsilon^{-1})$ |
|  | $O(ne^{-1} \log n)$ | $O(\epsilon^{-1})$ (Undir.) |
| Klein 2002 | $O(n(\log n + \epsilon^{-1} \log \epsilon^{-1}))$ | $O(\epsilon^{-1})$ (Undir.) |
| Kawarabayashi, Klein, & Sommer 2011 | $O(n)$ | $O(\epsilon^{-2} \log^2 n)$ (Undir.) |
| Kawarabayashi, Sommer, & Thorup 2013 | $\tilde{O}(n \log n)$ | $\tilde{O}(\epsilon^{-1})$ (Undir., Unweight.) |
|  | $\tilde{O}(n)$ | $\tilde{O}(\epsilon^{-1})$ (Undir.) |

Table 1: Space-query time tradeoffs for exact and approximate planar distance oracles. $\tilde{O}$ hides $\log(\epsilon^{-1} \log n)$ factors.
Sleator and Tarjan’s Link-Cut trees [37], and uses persistent arrays [12] in lieu of [16] to make the data structure persistent.

**Organization.** In Section 2 we review background on planar embeddings, planar separators, multiple-source shortest paths, and weighted Voronoi diagrams. In Section 3 we introduce key parts of the data structure and describe the query algorithm, assuming a certain point location problem can be solved. Section 4 introduces several more components of the data structure, and shows how they can be applied to solve this particular point location problem in near-logarithmic time. The space and query-time claims of Theorem 1.1 are proved in Appendix C. Appendix A gives the MSSP construction based on Euler Tour trees. Appendix B explains how to remove a simplifying assumption made throughout the paper, that the boundary vertices of every region in the $\vec{r}$-decomposition lie on a single hole, which is bounded by a simple cycle.

2 Preliminaries

2.1 The Graph and Its Embedding

A weighted planar graph $G = (V, E, \ell)$ is represented by an abstract embedding: for each $v \in V(G)$ we list the edges incident to $v$ according to a clockwise order around $v$. We assume the graph has no negative weight cycles and further assume the following, without loss of generality.

- All the edge-weights can be made non-negative ($\ell : E \to \mathbb{R}_{\geq 0}$) [24]. Furthermore, via randomized or deterministic perturbation [17], we can assume there are no zero weight edges, and that shortest paths are unique in any subgraph of $G$.
- The graph is connected and triangulated. Assign all artificial edges weight $n \cdot \max_{e \in E(G)} \{\ell(e)\}$ so as not to affect any finite distances.
- If $(u, v) \in E(G)$ then $(v, u) \in E(G)$ as well. (In the circular ordering around $v$, they are represented as a single element \{u,v\}.)

Suppose $P = (v_0, v_1, \ldots, v_k)$ is a path oriented from $v_0$ to $v_k$, and $e = (v_i, u)$ is an edge not on $P$, $i \in [1, k−1]$. Then $e$ is to the right of $P$ if $e$ appears between $(v_i, v_{i+1})$ and $(v_{i−1}, v_i)$ in the clockwise order around $v_i$, and left of $P$ otherwise.

2.2 Separators and Divisions

Lipton and Tarjan [30] proved that every planar graph contains a separator of $O(\sqrt{n})$ vertices that, once removed, breaks the graph into components of at most $2/3$ the size. Miller [31] showed that every triangulated planar graph has a $O(\sqrt{n})$-size separator that consists of a simple cycle. Frederickson [19] defined a division to be a set of edge-induced subgraphs whose union is $G$. A vertex in more than one region is a boundary vertex; the boundary of a region $R$ is denoted $\partial R$. Edges along the boundary between two regions appear in both regions. The $r$-divisions of [19] have $\Theta(n/r)$ regions each with $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices.

We use a linear-time algorithm of Klein, Mozes, and Sommer [29] for computing a hierarchical $\vec{r}$-division, where $\vec{r} = (r_m, \ldots, r_1)$ and $n = r_m > \cdots > r_1 = \Omega(1)$. Such an $\vec{r}$-division has the following properties:

- (Division & Hierarchy) For each $i$, $R_i$ is the set of regions in an $r_i$-division of $G$, where $R_m = \{G\}$ consists of the graph itself. For each $i < i' \leq m$ and $R_i \in R_i$, there is a unique $R_i' \in R_{i'}$ such that $E(R_i) \subseteq E(R_i')$. The $\vec{r}$-division is therefore represented as a rooted tree of regions.
- (Boundaries and Holes) The $O(\sqrt{n/r})$ boundary vertices of any $R_i \in R_i$ lie on a constant number of faces of $R_i$, called holes, each bounded by a cycle (not necessarily simple).
We supplement the \( r \)-division with a zeroth level. The layer-0 \( R_0 = \{ \{ v \} \mid v \in V(G) \} \) consists of singleton sets, and each \( \{ v \} \) is attached as a (leaf) child of an arbitrary \( R \in R_1 \) for which \( v \in R \).

Suppose \( f \) is one of the \( O(1) \) holes of region \( R \) and \( C_f \) the cycle around \( f \). The cycle \( C_f \) partitions \( E(G) \) into two parts. Let \( R^{f,\text{out}} \) be the graph induced by the part disjoint from \( R \), together with \( C_f \), i.e., \( C_f \) appears in both \( R \) and \( R^{f,\text{out}} \). To keep the description of the algorithm as simple as possible, we will assume that \( \partial R \) lies on a single simple cycle (hole) \( f_R \), and let \( R^{\text{out}} \) be short for \( R^{f,\text{out}} \). The modifications necessary to deal with multiple holes and non-simple boundary cycles are explained in Appendix A.

### 2.3 Multi-source Shortest Paths

Suppose \( H \) is a weighted planar graph with a distinguished face \( f \) on vertices \( S \). Klein’s MSSP algorithm takes \( O(|H| \log |H|) \) time and produces an \( O(|H| \log |H|) \)-size data structure such that given \( s \in S \) and \( v \in V(H) \), returns \( \text{dist}_H(s,v) \) in \( O(|H|) \) time. Klein’s algorithm can be viewed as continuously moving the source vertex around the boundary face \( f \), recording all changes to the SSSP tree in a dynamic tree data structure \cite{37}. It is shown \cite{28} that each edge in \( H \) enters and leaves the SSSP tree exactly once, meaning the number of changes is \( O(|H|) \). Each change to the tree is effected in \( O(|H|) \) time \cite{37}, and the generic persistence method of \cite{10} allows for querying any state of the SSSP tree. The important point is that the total space is linear in the number of updates to the structure \( O(|H|) \) times the update time \( O(|H|) \). As observed in \cite{20}, this structure can also answer other useful queries in \( O(|H|) \) time. Lemma 2.1 is similar to \cite{28,20} except that we use a dynamic tree data structure based on Euler Tour trees \cite{23} rather than Link-Cut trees \cite{37}, which allows for a more flexible tradeoff between update and query time. Because our data structure does not satisfy the criteria of Driscoll et al.’s \cite{10} persistence method for pointer-based data structures, we use the folklore implementation of persistent arrays\footnote{Dietz \cite{12} credits this method to an oral presentation of Dietzfelbinger et al. \cite{13}, which highlighted it as an application of dynamic perfect hashing.} to make any RAM data structure persistent, with doubly-logarithmic slowdown in the query time. See Appendix A for a proof of Lemma 2.1.

**Lemma 2.1.** (Cf. Klein \cite{28}, Gawrychowski et al. \cite{20}) Let \( H \) be a planar graph, \( S \) be the vertices on some distinguished face \( f \), and \( \kappa \geq 1 \) be a parameter. An \( O(|H|^{1+1/\kappa}) \)-space data structure can be computed in \( O(|H|) \) time that answers the following queries in \( O(|H|) \) time.

- Given \( s \in S, v \in V(H) \), return \( \text{dist}_H(s,v) \).

- Given \( s \in S, u, v \in V(H) \), return \((x, e_u, e_v)\), where \( x \) is the least common ancestor of \( u \) and \( v \) in the SSSP tree rooted at \( s \) and \( e_x \) is the edge on the path from \( x \) to \( z \) (if \( x \neq z \), \( z \in \{u, v\} \)).
2.4 Additively Weighted Voronoi Diagrams

Let $H$ be a weighted planar graph, $f$ a distinguished face whose vertices $S$ are called sites, and $\omega : S \to \mathbb{R}_{\geq 0}$ be a weight function on sites. We augment $H$ with large-weight edges so that it is triangulated, except for $f$. For $s \in S, v \in V(H)$, define

$$d^\omega(s, v) \overset{\text{def}}{=} \omega(s) + \text{dist}_H(s, v).$$

The Voronoi diagram $\text{VD}[H, S, \omega]$ is a partition of $V(H)$ into Voronoi cells, where for $s \in S$,

$$\text{Vor}(s) \overset{\text{def}}{=} \{ v \in V(H) \mid \forall s' \neq s. (d^\omega(s, v), -\omega(s)) < (d^\omega(s', v), -\omega(s'))\}$$

In other words, $\text{Vor}(s)$ is the set of vertices that are closer to $s$ than any other site, breaking ties in favor of larger $\omega$-values. We usually work with the dual representation of a Voronoi diagram. It is constructed as follows.

- Define $\hat{S}$ to be the set of sites with nonempty Voronoi cells, i.e., $\hat{S} = \{s \in S \mid s \in \text{Vor}(s)\}$. The case $|\hat{S}| = 1$ is trivial, so assume $|\hat{S}| \geq 2$.
- Add large-weight dummy edges to $H$ so that $\hat{S}$ appear on the boundary of a single face $\hat{f}$, but is otherwise triangulated. Observe that this has no effect on the Voronoi cells.
- An edge is bichromatic if its endpoints are in different cells. In particular, the edges bounding $\hat{f}$ are entirely bichromatic. Define $\text{VD}_0^\ast$ to be the (undirected) subgraph of $H^\ast$ consisting of the duals of bichromatic edges.
- Obtain $\text{VD}_1^\ast$ from $\text{VD}_0^\ast$ by repeatedly contracting edges incident to a degree-2 vertex, terminating when there are no degree-2 vertices, or when it becomes a self-loop. Observe that in $\text{VD}_1^\ast$, $\hat{f}^\ast$ has degree $|\hat{S}|$ and all other vertices have degree 3; moreover, the faces of $\text{VD}_1^\ast$ are in one-to-one correspondence with the Voronoi cells.
- We obtain $\text{VD}^\ast = \text{VD}^\ast[H, S, \omega]$ by splitting $\hat{f}^\ast$ into $|\hat{S}|$ degree-1 vertices, each taking an edge formerly incident to $\hat{f}^\ast$. It was proved in [20] Lemma 4.1] that $\text{VD}^\ast$ is a single tree.
- We store with $\text{VD}^\ast$ supplementary information useful for point location. Each degree-3 vertex $g^\ast$ in $\text{VD}^\ast$ corresponds a trichromatic face $g$ whose three vertices, say $y_0, y_1, y_2$, belong to different Voronoi cells. We store in $\text{VD}^\ast$ the sites $s_0, s_1, s_2 \in S$ such that $y_i \in \text{Vor}(s_i)$. We also store a centroid decomposition of $\text{VD}^\ast$. A centroid of a tree $T$ is a vertex $c$ that partitions the edge set of $T$ into disjoint subtrees $T_1, \ldots, T_{\text{deg}(c)}$, each containing at most $(|E(T)| + 1)/2$ edges, and each containing $c$ as a leaf. The decomposition is a tree rooted at $c$, whose subtrees are the centroid decompositions of $T_1, \ldots, T_{\text{deg}(c)}$. The recursion bottoms out when $T$ consists of a single edge, which is represented as a single (leaf) node in the centroid decomposition.

The most important query on Voronoi diagrams is point location.

**Lemma 2.2.** (Gawrychowski et al. [20]) The PointLocate$(\text{VD}^\ast[H, S, \omega], v)$ function is given the dual representation of a Voronoi diagram $\text{VD}^\ast[H, S, \omega]$ and a vertex $v \in V(H)$ and reports the $s \in S$ for which $v \in \text{Vor}(s)$. Given access to an MSSP data structure for $H$ with source-set $S$ and query time $\tau$, we can answer PointLocate$(\text{VD}^\ast[H, S, \omega], v)$ queries in $O(\tau \cdot \log |H|)$ time.

The challenge in our data structure (as in [9]) is to do point location when our space budget precludes storing all the relevant MSSP structures. Nonetheless, we do make use of PointLocate when the MSSP data structures are available.

---

2 The latter case only occurs when $|\hat{S}| = 2$.
3 If we skipped the step of forming the face $\hat{f}$ on the site-set $\hat{S}$ and triangulating the rest, $\text{VD}^\ast$ would still be acyclic, but perhaps disconnected. See [20] [9].
4 I.e., internal nodes correspond to vertices of $T$; leaf nodes correspond to edges of $T$. 
Figure 2: (a) The original $H$ is a triangulated grid, with $f$ being the exterior face. The boundary vertices $\hat{S}$ with non-empty Voronoi cells are marked with colored halos. Edges are added so that $\hat{S}$ are on the exterior face $\hat{f}$. The vertices of $VD^*$ are the duals of trichromatic faces, and those derived by splitting $f^*$ into $|\hat{S}|$ vertices. The edges of $VD^*$ correspond to paths of duals of bichromatic edges. (b) The dual representation $VD^*$. (c) A centroid decomposition of $VD^*$. 
3 The Distance Oracle

As in [9], the distance oracle is based on an \( r \)-decomposition, \( r = (r_m, \ldots, r_1) \), where \( r_i = n^{1/m} \) and \( m \) is a parameter. Suppose we want to compute \( \text{dist}_G(u, v) \). Let \( R_0 = \{ u \} \) be the artificial level-0 region containing \( u \) and \( R_t \in \mathcal{R}_t \) be the level-\( t \)-ancestor of \( R_0 \). (Throughout the paper, we will use “\( R_t \)” to refer specifically to the level-\( t \) ancestor of \( R_0 = \{ u \} \), as well as to a generic region at level-\( t \). Surprisingly, this will cause no confusion.) Let \( t \) be the smallest index for which \( v \notin R_t \) but \( v \in R_{t+1} \). Define \( u_i \) to be the last vertex on \( \partial R_i \) encountered on the shortest path from \( u \) to \( v \). The main task of the distance query algorithm is to compute the sequence \((u = u_0, \ldots, u_t)\). Suppose that we know the identity of \( u_i \) and \( t > i \). Finding \( u_{i+1} \) now amounts to a point location problem in \( VD[\mathcal{R}^\text{out}_{i+1}, \partial R_{i+1}, \omega] \), where \( \omega(s) \) is the distance from \( u_i \) to \( s \in \partial R_{i+1} \). However, we cannot apply the fast \text{PointLocate} \ routine because we cannot afford to store an MSSP structure for every \( (\mathcal{R}^\text{out}_{i+1}, \partial R_{i+1}) \), since \( |\mathcal{R}^\text{out}_{i+1}| = \Omega(|G|) \). Our point location routine narrows down the number of possibilities for \( u_{i+1} \) to at most two candidates in \( O(\kappa \log^{2+o(1)} n) \) time, then decides between them using two recursive distance queries, but starting at a higher level in the hierarchy. There are about \( 2^m \) recursive calls in total, leading to a \( O(2^{m\kappa \log^{2+o(1)} n}) \) query time.

The data structure is composed of several parts. Parts (A) and (B) are explained below, while parts (C)-(E) will be revealed in Section 4.2.

(A) (MSSP Structures) For each \( i \in [0, m - 1] \) and each region \( R_i \in \mathcal{R}_i \) with parent \( R_{i+1} \in \mathcal{R}_{i+1} \), we store an MSSP data structure (Lemma 2.1) for the graph \( \mathcal{R}^\text{out}_i \), and source set \( \partial R_i \). However, the structure only answers queries for \( s \in \partial R_i \) and \( u, v \in \mathcal{R}^\text{out}_i \cap \partial R_{i+1} \). Rather than represent the full SSSP tree from each root on \( s \in \partial R_i \), the MSSP data structure only stores the tree induced by \( \mathcal{R}^\text{out}_i \cap \partial R_{i+1} \), i.e., the parent of any vertex \( v \in \mathcal{R}^\text{out}_i \cap \partial R_{i+1} \) is its nearest ancestor \( v' \) in the SSSP tree such that \( v' \in \mathcal{R}^\text{out}_i \cap \partial R_{i+1} \). If \((v', v)\) is a “shortcut” edge corresponding to a path in \( \mathcal{R}^\text{out}_{i+1} \), it has weight \( \text{dist}_{\mathcal{R}^\text{out}_i}(v', v) \).

We fix a \( \kappa \) and let the update time in the dynamic tree data structure be \( O(\kappa n^{1/\kappa}) \) time. Thus, the space for this structure is \( O(|\mathcal{R}^\text{out}_i \cap \partial R_{i+1}| + |\partial R_i| \cdot |\mathcal{R}_{i+1}| \cdot \kappa n^{1/\kappa}) = O(r_{i+1} \cdot \kappa n^{1/\kappa}) \) since each edge in \( \mathcal{R}^\text{out}_i \cap \partial R_{i+1} \) is swapped into and out of the SSSP tree once [28], and the number of shortcut edges on \( \partial R_{i+1} \) swapped into and out of the SSSP is at most \( |\partial R_{i+1}| \) for each of the \( |\partial R_i| \) sources. Over all \( i \) and \( \Theta(n/r_i) \) choices of \( R_i \), the space is \( O(m n^{1+1/m+1/\kappa}) \) since \( r_{i+1}/r_i = n^{1/m} \).

(B) (Voronoi Diagrams) For each \( i \in [0, m - 1] \) and \( R_i \in \mathcal{R}_i \) with parent \( R_{i+1} \in \mathcal{R}_{i+1} \), and each \( q \in \partial R_i \), define \( \text{VD}^\text{out}_i(q, R_{i+1}) \) to be \( \text{VD}^\star[\mathcal{R}^\text{out}_{i+1}, \partial R_{i+1}, \omega] \), with \( \omega(s) = \text{dist}_G(q, s) \). The space to store the dual diagram and its centroid decomposition is \( O(|\partial R_{i+1}|) = O(\sqrt{r_{i+1}}) \). Over all choices for \( i, R_i \), and \( q \), the space is \( O(m n^{1+1/(2m)}) \) since \( \sqrt{r_{i+1}/r_i} = n^{1/(2m)} \).

Due to our tie-breaking rule in the definition of \( \text{Vor}(\cdot) \), locating \( u_{i+1} \) \((t \geq i + 1)\) is tantamount to performing a point location on a Voronoi diagram in part (B) of the data structure.

Lemma 3.1. Suppose that \( q \in \partial R_i \) and \( v \notin \partial R_{i+1} \). Consider the Voronoi diagram associated with \( \text{VD}^\text{out}_i(q, R_{i+1}) \) with sites \( \partial R_{i+1} \) and additive weights defined by distances from \( q \) in \( G \). Then \( v \in \text{Vor}(s) \) if and only if \( s \) is the last \( \partial R_{i+1}-\text{vertex} \) on the shortest path from \( q \) to \( v \) in \( G \), and \( d^\star(s, v) = \text{dist}_G(q, v) \).

Proof. By definition, \( d^\star(s, v) \) is the length of the shortest path from \( q \) to \( v \) that passes through \( s \) and whose \( s-v \) suffix does not leave \( \mathcal{R}^\text{out}_{i+1} \). Thus, \( d^\star(s, v) \geq \text{dist}_G(q, v) \) for every \( s \), and \( d^\star(s, v) = \text{dist}_G(q, v) \) for some \( s \). Because of our assumption that all edges are strictly positive, and our tie-breaking rule for preferring larger \( \omega \)-values in the definition of \( \text{Vor}(\cdot) \), if \( v \in \text{Vor}(s) \) then \( s \) must be the last \( \partial R_{i+1}-\text{vertex} \) on the shortest \( q-v \) path. \( \square \)

3.1 The Query Algorithm

A distance query is given \( u, v \in V(G) \). We begin by identifying the level-0 region \( R_0 = \{ u \} \in \mathcal{R}_0 \) and call the function \( \text{Dist}(u, v, R_0) \). In general, the function \( \text{Dist}(u_i, v, R_i) \) takes as arguments a region \( R_i \), a source
vertex $u_i$ on the boundary $\partial R_i$, and a target vertex $v \notin R_i$. It returns a value $d$ such that

$$
\text{dist}_G(u_i, v) \leq d \leq \text{dist}_{R_i}^\text{out}(u_i, v).
$$

Note that $R_i^\text{out} = G$, so the initial call to this function correctly computes $\text{dist}_G(u, v)$. When $v$ is “close” to $u_i$ ($v \in R_i^\text{out} \cap R_{i+1}$) it computes $\text{dist}_{R_i}^\text{out}(u_i, v)$ without recursion, using part (A) of the data structure. When $v \in R_i^\text{out}$ it performs point location using the function $\text{CentroidSearch}$, which culminates in up to two recursive calls to $\text{Dist}$ on the level-$(i+1)$ region $R_{i+1}$. Thus, the correctness of $\text{Dist}$ hinges on whether $\text{CentroidSearch}$ correctly computes distances when $v \in R_i^\text{out}$.

**Algorithm 1 Dist($u_i, v, R_i$)**

**Input:** A region $R_i$, a source $u_i \in \partial R_i$ and a destination $v \in R_i^\text{out}$.

**Output:** A value $d$ such that $\text{dist}_G(u_i, v) \leq d \leq \text{dist}_{R_i}^\text{out}(u_i, v)$.

1. if $v \in R_i^\text{out} \cap R_{i+1}$ then
   
   return $d \leftarrow \text{dist}_{R_{i+1}}^\text{out}(u_i, v)$ \hfill \text{▷ I.e., } i = t

2. end if

3. $f^* \leftarrow \text{root of the centroid decomposition of } VD_i^\text{out}(u_i, R_i)$

4. return $d \leftarrow \text{CentroidSearch}(VD_i^\text{out}(u_i, R_i), v, f^*)$

The procedure $\text{CentroidSearch}$ is given $u_i \in \partial R_i$, $v \in R_i^\text{out}$, $VD_i^\text{out} = VD_i^\text{out}(u_i, R_{i+1})$ and a node $f^*$ on the centroid decomposition of $VD_i^\text{out}$. It ultimately computes $u_{i+1} \in \partial R_{i+1}$ for which $v \in \text{Vor}(u_{i+1})$ and returns

$$
\omega(u_{i+1}) + \text{Dist}(u_{i+1}, v, R_{i+1}) \leq \text{dist}_G(u_i, u_{i+1}) + \text{dist}_{R_{i+1}}^\text{out}(u_{i+1}, v) = \text{dist}_G(u_i, v),
$$

Line 5 or 9 of $\text{CentroidSearch}$

Defn. of $\omega$; guarantee of $\text{Dist}$ (Eqn. (1))

Lemma 3.1

The algorithm is recursive, and bottoms out in one of two base cases (Line 5 or Line 9). The first way the recursion can end is if we reach the bottom of the centroid decomposition. If $f^*$ is a leaf of the decomposition, it corresponds to an edge in $VD_i^\text{out}$ separating the Voronoi cells of two sites, say $s_1$ and $s_2$. At this point we know that either $u_{i+1} = s_1$ or $u_{i+1} = s_2$, and determine which case is true with two recursive calls to $\text{Dist}(s_j, v, R_{i+1})$, $j \in \{1, 2\}$ (Lines 2–5). In general, $f^*$ is dual to a trichromatic face $f$ composed of three vertices $y_0, y_1, y_2$ in clockwise order, which are, respectively, in distinct Voronoi cells of $s_0, s_1, s_2$. The three shortest $s_j$-$y_j$ paths and $f$ partition the vertices of $R_{i+1}$ into six parts, namely the shortest $s_j$-$y_j$ paths themselves, and the interiors of the regions bounded by $\partial R_{i+1}$, two of the $s_j$-$y_j$ paths and an edge of $f$. See Figure 3. The $\text{Navigation}$ function returns a pair (flag, $a^*$) that identifies which part $v$ is in. If flag = terminal then $a^* \in \{s_0, s_1, s_2\}$ is interpreted as a site, indicating that $v$ lies on the shortest path from $a^*$ to its $f$-vertex. In this case we return $\omega(a^*) + \text{Dist}(a^*, v, R_{i+1}) = \text{dist}_G(u_i, v)$ with just one call to $\text{Dist}$. If flag = nonterminal then $a^*$ is the correct child of $f^*$ in the centroid decomposition. In particular, $f^*$ is incident to three edges $e_0^f, e_1^f, e_2^f$ dual to $\{y_0, y_2\}, \{y_1, y_0\}, \{y_2, y_1\}$. The children of $f^*$ in the centroid decomposition are $f_0^*, f_1^*, f_2^*$, with $f_j^*$ ancestral to $e_j^f$. We have $a^* = f_j^*$ if $v$ lies to the right of the chord $(s_j, \ldots, y_j, y_{j-1}, \ldots, s_{j-1})$ in $R_{i+1}^\text{out}$. For example, in Figure 3 $v$ lies to the right of the $(s_0, \ldots, y_0, y_2, \ldots, s_2)$ path. In this case we continue the search recursively from $a^* = f_0^*$. 

8
Figure 3: Here $f^*$ is a degree-3 vertex in $\text{VD}_\text{out}^*(u_i, R_{i+1})$, corresponding to a trichromatic face $f$ on vertices $y_0, y_1, y_2$, which are in the Voronoi cells of $s_0, s_1, s_2$ on the boundary $\partial R_{i+1}^\text{out}$. The shortest $s_j-y_j$ paths partition $V(R_{i+1}^\text{out})$ into six parts: the three shortest paths and the three regions bounded by them and $f$. Let $e_0^*, e_1^*, e_2^*$ be the edges in $\text{VD}_\text{out}^*$ dual to $\{y_0, y_2\}, \{y_1, y_0\}, \{y_2, y_1\}$. In the centroid decomposition $e_0^*, e_1^*, e_2^*$ are in separate subtrees of $f^*$. Let $f_j^*$ be the child of $f^*$ ancestral to $e_j^*$, which is either $e_j^*$ itself, or a trichromatic face to the right of the “chord” $(s_j, \ldots, y_j, y_{j-1}, \ldots, s_{j-1})$. \textbf{CentroidSearch} locates the site whose Voronoi cell contains $v$ via recursion. It calls \textbf{Navigation}, a function that finds which of the 6 parts contains $v$. If $v$ lies on an $s_j-y_j$ path the \textbf{CentroidSearch} recursion terminates; otherwise it recurses on the correct child $f_j^*$ of $f^*$.
Furthermore, and Lemma 3.1 that and the base case on Lines 8–9 also works correctly.

Thus, the main challenge is to design an efficient Navigation function, i.e., to solve the restricted point location problem in $R^*_i$ depicted in Figure 3. Whereas Charalampopoulos et al. [9] solve this problem using several more recursive calls to Dist, we give a new method to do this point location directly, in $O(k \log^{1+o(1)} n)$ time per call to Navigation.

4 The Navigation Oracle

The input to Navigation is the same as CentroidSearch, except that $f^*$ is guaranteed to correspond to a trichromatic face $f$. Define $y_j, s_j, e_j, f_j, j \in \{0, 1, 2\}$ as in the discussion of CentroidSearch. The Navigation
function determines the location of \( v \) relative to \( f \) and the shortest \( s_j-y_j \) paths. It delegates nearly all the actual computation to two functions: \textbf{SitePathIndicator}, which returns a boolean indicating whether \( v \) is on the shortest \( s_j-y_j \) path, and \textbf{ChordIndicator}, which indicates whether \( v \) lies strictly to the right of the oriented chord \( (s_j,\ldots,y_j,y_{j-1},\ldots,s_{j-1}) \). If so, we return the centroid child \( f^*_j \) of \( f^* \) in this region. Three calls each to \textbf{SitePathIndicator} and \textbf{ChordIndicator} suffice to determine the location of \( v \).

\begin{algorithm}
\caption{Navigation}(\text{VD}\textsubscript{out}\textsuperscript{*}(u_i,R_{i+1}),v,f^*)
\begin{algorithmic}[1]
\Statex \textbf{Input}: The dual representation \text{VD}\textsubscript{out}\textsuperscript{*}(u_i,R_{i+1}) of a Voronoi diagram, a vertex \( v \in R\textsuperscript{out}\textsuperscript{*}_{i+1} \), and a centroid \( f^* \) in the centroid decomposition. The face \( f \) is on \( y_0,y_1,y_2 \), which are in the Voronoi cells of \( s_0,s_1,s_2 \), and \( f^*_j \) is the child of \( f^* \) containing the edge dual to \( \{y_j,y_{j-1}\} \).
\Statex \textbf{Output}: (terminal, \( s_j \)) if \( v \) is on the shortest \( s_j-y_j \) path, or (nonterminal, \( f^*_j \)) where \( f^*_j \) is the child of \( f^* \) ancestral to an edge bounding \( v \)'s Voronoi cell.
\State \( s_0,s_1,s_2 \leftarrow \text{sites corresponding to } f^* \)
\For {\( j = 0,1,2 \)}
\If {\textbf{SitePathIndicator}(\text{VD}\textsubscript{out}\textsuperscript{*}(u_i,R_{i+1}),v,f^*,j) \ returns \textbf{True}}
\State \textbf{return} (\text{terminal}, \( s_j \))
\EndIf
\EndFor
\For {\( j = 0,1,2 \)}
\If {\textbf{ChordIndicator}(\text{VD}\textsubscript{out}\textsuperscript{*}(u_i,R_{i+1}),v,f^*,j) \ returns \textbf{True}}
\State \textbf{return} (\text{nonterminal}, \( f^*_j \))
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

In Section\ref{sec:algorithm} we formally introduce the notion of \textit{chords} used informally above, as well as some related concepts like \textit{laminar} sets of chords and \textit{maximal} chords. In Section\ref{sec:datastructure} we introduce parts \( (C\cdot{}E) \) of the data structure used to support \textbf{Navigation}. The functions \textbf{SitePathIndicator} and \textbf{ChordIndicator} are presented in Sections\ref{sec:sitepath} and\ref{sec:chordindicator}.

\subsection{Chords and Pieces}
We begin by defining the key concepts of our point location method: \textit{chords}, \textit{laminar chord sets}, \textit{pieces}, and the \textit{occludes} relation.

\textbf{Definition 4.1. (Chords)} Fix an \( R \) in the \( \mathcal{r} \)-decomposition and two vertices \( c_0,c_1 \in \partial R \). An oriented simple path \( c_0\mathcal{C}_1 \) is a \textit{chord} of \( R\textsuperscript{out} \) if it is contained in \( R\textsuperscript{out} \) and is internally vertex-disjoint from \( \partial R \). When the orientation is irrelevant we write it as \( \overrightarrow{c_0c_1} \).

\textbf{Definition 4.2. (Laminar Chord Sets)} A set of chords \( C \) for \( R\textsuperscript{out} \) is \textit{laminar} (non-crossing) if for any two such chords \( C = \overrightarrow{c_0c_1}, C' = \overrightarrow{c_2c_3} \), if there exists a \( v \in (C \cap C') - \partial R \) then the subpaths from \( c_0 \) to \( v \) and from \( c_2 \) to \( v \) are identical; in particular \( c_0 = c_2 \).

The orientation of chords does not always coincide with a natural orientation of paths defined by the algorithm. For example, in Figure\ref{fig:example} the oriented chord \( \overrightarrow{s_0s_2} = (s_0,\ldots,y_0,y_2,\ldots,s_2) \) is composed of three parts: a shortest \( s_0-y_0 \) path (whose natural orientation coincides with that of \( \overrightarrow{s_0s_2} \)), the edge \( \{y_0,y_2\} \) (which has no natural orientation in this context), and the shortest \( s_2-y_2 \) path (whose natural orientation is the reverse of its orientation in \( \overrightarrow{s_0s_2} \)). The orientation serves two purposes. In Definition\ref{def:chords} we can speak unambiguously about the parts of \( R\textsuperscript{out} \) to the \textit{right} and \textit{left} of \( \overrightarrow{s_0s_2} \). In Definition\ref{def:laminar} the role of the orientation is to ensure that the partition of \( R\textsuperscript{out} \) into \textit{pieces} induced by \( C \) can be represented by a \textit{tree}, as we show in Lemma\ref{lem:tree}.

\textbf{Definition 4.3. (Pieces)} A laminar chord set \( C \) for \( R\textsuperscript{out} \) partitions the faces of \( R\textsuperscript{out} \) into pieces, excluding the face on \( \partial R \). Two faces \( f,g \) are in the same piece iff \( f^* \) and \( g^* \) are connected by a path in \( (R\textsuperscript{out})^* \) that
avoids to duals of edges in $C$ and edges along the boundary cycle on $\partial R$. A piece is regarded as the subgraph induced by its faces, i.e., it includes their constituent vertices and edges. Two pieces $P_1, P_2$ are adjacent if there is an edge $e$ on the boundary of $P_1$ and $P_2$ and $e$ is in a unique chord of $C$. See Figure 4.

**Lemma 4.1.** Suppose $C$ is a laminar chord set for $R^{out}$, $P = P(C)$ is the corresponding piece set and $E$ are pairs of adjacent pieces. Then $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ is a tree, called the piece tree induced by $C$.

**Proof.** The claim is clearly true when $C$ contains zero or one chords, so we will try to reduce the general case to this case via a peeling argument. We will find a piece $P$ with degree 1 in $\mathcal{T}$, remove it and the chord bounding it, and conclude by induction that the truncated instance is a tree. Reattaching $P$ implies $\mathcal{T}$ is a tree.

Let $C = c_0c_1 \in C$ be a chord such that no edge of any other chord appears strictly to one side of $C$, say to the right of $C$. Let $P$ be the piece to the right of $C$. (In Figure 4 the chords bounding $P_1, P_2, P_{11}, P_{12}$ would be eligible to be $C$.) Let $C = (c_0 = v_0, v_1, v_2, \ldots, v_k = c_1)$ and $v_j$ be such that the edges of the suffix $(v_j, \ldots, v_k)$ are on no other chord, meaning the vertices $\{v_j, \ldots, v_k\}$ are on no other chord. Let $g_j$ be the face to the left of $(v_j, v_{j+1})$. It follows that there is a path from $g_j$ to $g_{k-1}$ in $(R^{out})^*$ that avoids the duals of all edges in $C$ and along $\partial R$. All pieces adjacent to $P$ contain some face among $\{g_j, \ldots, g_{k-1}\}$, but these are in a single piece, hence $P$ corresponds to a degree-1 vertex in $\mathcal{T}$. Let $P$ be bounded by $C$ and an interval $B$ of the boundary cycle on $\partial R$. Obtain the “new” $R^{out}$ by cutting along $C$ and removing $P$, the new $\partial R$ by substituting $C$ for $B$, and the new chord-set $C$ by removing $C$ and trimming any chords that shared a non-empty prefix with $C$. By induction the resulting piece-adjacency graph is a tree; reattaching $P$ as a degree-1 vertex shows $\mathcal{T}$ is a tree.

**Definition 4.4. (Occludes Relation)** Fix $R^{out}$, chord $C$, and two faces $f, g$, neither of which is the hole defined by $\partial R$. If $f$ and $g$ are on opposite sides of $C$, we say that from vantage $f$, $C$ occludes $g$. Let $C$ be a set of chords. We say $C \in C$ is maximal in $C$ with respect to a vantage $f$ if there is no $C' \subseteq C$ such that $C'$ occludes a strict superset of the faces that $C$ occludes. (Note that the orientation of chords is irrelevant to the occludes relation.)
It follows from Definition 4.4 that if $C$ is laminar, the set of maximal chords with respect to $f$ are exactly those chords intersecting the boundary of $f$’s piece in $\mathcal{P}(C)$.

We can also speak unambiguously about a chord $C$ occluding a vertex or edge not on $C$, from a certain vantage. Specifically, we can say that from some vantage, $C$ occludes an interval of the boundary cycle on $\partial R$, say according to a clockwise traversal around the hole on $\partial R$ in $R_{\text{out}}^\omega$. This will be used in the ChordIndicator procedure of Section 4.4.2.

4.2 Data Structures for Navigation

Parts (C)–(E) of the data structure are used to implement the SitePathIndicator and ChordIndicator functions.

(C) (More Voronoi Diagrams) For each $i$, each $R_i \in \mathcal{R}_i$, and each $q \in \partial R_i$, we store $\text{VD}^*_\text{out}(q, R_i)$, which is $\text{VD}^*[R_i^\text{out}, \partial R_i, \omega]$, where $\omega(s) = \text{dist}(q, s)$. The total space for these diagrams is $O(n)$ and dominated by part (B).

(D) (Chord Trees; Piece Trees) For each $i$, each $R_i \in \mathcal{R}_i$, and source $q \in \partial R_i$, we store the SSSP tree from $q$ induced by $\partial R_i$ as a chord tree $T_q^{R_i}$. In particular, the parent of $x \in \partial R_i$ in $T_q^{R_i}$ is the nearest ancestor in the SSSP tree from $q$ that lies on $\partial R_i$. Every edge of $T_q^{R_i}$ is designated a chord if the corresponding path is contained in $R_i^\text{out}$ but not in $R_i$, or a non-chord otherwise. Define $C_q^{R_i}$ to be the set of all chords in $T_q^{R_i}$, oriented away from $q$; this is clearly a laminar set since shortest paths are unique and all prefixes of shortest paths are shortest paths. Define $\mathcal{P}_q^{R_i}$ to be the corresponding partition of $R_i^\text{out}$ into pieces, and $T_q^{R_i}$ the corresponding piece tree. Define $T_q^{R_i}[x]$ to be the path from $q$ to $x$ in $T_q^{R_i}$, $C_q^{R_i}[x]$ the corresponding chord-set, and $\mathcal{P}_q^{R_i}[x]$ the corresponding piece-set.

The data structure answers the following queries

MaximalChord($R_i, q, x, P, P'$): We are given $R_i$, $q, x \in \partial R_i$, a piece $P \in \mathcal{P}_q^{R_i}$, and possibly another piece $P' \in \mathcal{P}_q^{R_i}$ (which may be Null). If $P'$ is Null, return any maximal chord in $C_q^{R_i}[x]$ from vantage $P$. If $P'$ is not Null, return the maximal chord $C_q^{R_i}[x]$ (if any) that occludes $P'$ from vantage $P$.

AdjacentPiece($R_i, q, e$): Here $e$ is an edge on the boundary cycle on $\partial R_i$. Return the unique piece in $\mathcal{P}_q^{R_i}$ with $e$ on its boundary.

(E) (Site Tables; Side Tables) Fix an $i$ and a diagram $\text{VD}^*_\text{out} = \text{VD}^*_\text{out}(u', R_i)$ from part (B) or (C). Let $f^*$ be any node in the centroid decomposition of $\text{VD}^*_\text{out}$, with $y_j, s_j, j \in \{0, 1, 2\}$ defined as usual, and let $R_{u'} \in \mathcal{R}_{u'}$ be the ancestor of $R_i$; $i' \geq i$. Fix $j \in \{0, 1, 2\}$ and $i' > i$. Define $q$ and $x$ to be the first and last vertices on the shortest $s_j'y_j$ path that lie on $\partial R_{u'}$. We store $(q, x)$ and dist($G(u', x)$).

We also store whether $R_{u'}^\text{out}$ lies to the left or right of the site-centroid-site chord $s_j'y_jy_j^{-1}s_j'\overline{y_j}$ in $R_i^\text{out}$, or Null if the relationship cannot be determined, i.e., if the chord crosses $\partial R_{u'}$. These tables increase the space of (B) and (C) by a negligible $O(m)$ factor.

Part (D) of the data structure is the only one that is non-trivial to store compactly. Our strategy is as follows. We fix $R_i$ and $q \in \partial R_i$ and build a dynamic data structure for these operations relative to a dynamic subset $\hat{C} \subseteq C_q^{R_i}$ subject to the insertion and deletion of chords in $O(\log |\partial R_i|)$ time. By inserting/deleting $O(|\partial R_i|)$ chords in the correct order, we can arrange that $\hat{C} = C_q^{R_i}[x]$ at some point in time, for every $x \in \partial R_i$. Using the generic persistence technique for RAM data structures (see [12]) we can answer MaximalChord queries relative to $C_q^{R_i}[x]$ in $O(\log |\partial R_i| \log \log |\partial R_i|)$ time.

\[\text{This is one place where we use the assumption that all boundary holes are simple cycles.}\]

\[\text{This is another place where we use the assumption that holes are bounded by simple cycles.}\]

\[\text{Our data structure works in the pointer machine model, but it has unbounded in-degrees so the theorem of Driscoll et al. [10] cannot be applied directly. It is probably possible to improve the bound to } O(\log |\partial R_i|) \text{ but this is not a bottleneck in our algorithm.}\]
Lemma 4.2. Part (D) of the data structure can be stored in $O(mn \log n)$ total space and answer $\text{MaximalChord}$ queries in $O(\log n \log \log n)$ time and $\text{AdjacentPiece}$ queries in $O(1)$ time.

Proof. We first address $\text{MaximalChord}$. Let $T = T_{\rho_i}^{R_i}$ be the piece tree. The edges of $T$ are in 1-1 correspondence with the chords of $C = C_{\rho_i}^{R_i}$ and if $P, P' \in \mathcal{P} = \mathcal{P}_{\rho_i}^{R_i}$ are two pieces, the path from $P$ to $P'$ in $T$ crosses exactly those chords that occlude $P'$ from vantage $P$ (and vice versa). We will argue that to implement $\text{MaximalChord}$ it suffices to design an efficient dynamic data structure for the following problem; initially all edges are unmarked.

$\text{Mark}(e)$ Mark an edge $e \in E(T)$.

$\text{Unmark}(e)$ Unmark $e$.

$\text{LastMarked}(P', P)$ Return the last marked edge on the path from $P'$ to $P$, or $\text{Null}$ if all are unmarked.

By doing a depth-first traversal of the chord tree $T_{\rho_i}^{R_i}$, marking/unmarking chords as they are encountered, the set $\{e \in E(T) \mid e$ is marked$\}$ will be equal to $C_{\rho_i}^{R_i}[x]$ precisely when $x$ is first encountered in DFS. To answer a $\text{MaximalChord}(R_i, q, x, P, P')$ query we interact with the state of the data structure when the marked set is $C_{\rho_i}^{R_i}[x]$. If $P'$ is not null we return $\text{LastMarked}(P', P)$. Otherwise we pick an arbitrary (marked) chord $C \in C_{\rho_i}^{R_i}[x]$, get the adjacent pieces $P'_1, P'_2$ on either side of $C$, then query $\text{LastMarked}(P'_1, P)$ and $\text{LastMarked}(P'_2, P)$. At least one of these queries will return a chord and that chord is maximal w.r.t. vantage $P$. (Note that $C$ must separate $P$ from either $P'_1$ or $P'_2$.)

We now argue how all three operations can be implemented in $O(\log n)$ worst case time. The ideas are standard, so we do not go into great detail. Root $T$ arbitrarily and subdivide every edge; the resulting tree is also called $T$. Every node in $T$ knows its depth. The vertices corresponding to subdivided edges may carry marks. In order to answer $\text{LastMarked}$ queries it suffices to be able to find least common ancestors, and, given nodes $P_d, P_a$, where $P_a$ is an ancestor of $P_d$, to find the first and last marked node on the path from $P_d$ to $P_a$. Decompose the vertices of $T$ using a heavy path decomposition. Each vertex points to the path that it is in. Each path in the decomposition is a data structure that maintains an ordered set of its marked nodes, a pointer to the most ancestral marked node in the path, and a pointer to the parent, in $T$, of the root of the path. It is straightforward to find LCAs in this structure in $O(\log n)$ time.

Suppose we want to find the first and last marked node on the path from $P_d$ to $P_a$, an ancestor of $P_d$. Let $Z_0, Z_1, \ldots, Z_t, t = O(\log n)$ be the heavy paths ancestral to $P_d$ such that $P_d \in Z_0, P_a \in Z_t$. Let $v_j \in Z_j$ be the nearest ancestor to $P_d$. We can find $j^*$ such that $Z_{j^*}$ contains the first marked node on the $P_d$-$P_a$ path in $\ell = O(\log n)$ time by comparing $v_j$ against the most ancestral marked node in $Z_j$, $j = 0, 1, \ldots, t$. We can then find the first marked node by finding the marked predecessor of $v_{j^*}$ in $Z_{j^*}$, in $O(\log n)$ time. Finding the last marked node on the path from $P_d$ to $P_a$ is similar. $\text{Mark}$ and $\text{Unmark}$ are implemented by keeping a balanced binary search tree over the marked nodes in each heavy path.

For fixed $R_i, q \in \partial R_i$ there are $O(|\partial R_i|)$ $\text{Mark}$ and $\text{Unmark}$ operations, each taking $O(\log n)$ time. Over all choices of $i, R_i$, and $q$ the total update time is $O(mn \log n)$. After applying generic persistence for RAM data structures (see [12]) the space becomes $O(mn \log n)$ and the query time for $\text{LastMarked}$ becomes $O(\log n \log \log n)$.

Turning to $\text{AdjacentPiece}(R_i, q, e)$, there are $|\partial R_i|^2$ choices of $(q, e)$. Hence all answers can be precomputed in a lookup table in $O(mn)$ space.

\[\square\]

4.3 The SitePathIndicator Function

The $\text{SitePathIndicator}$ function is relatively simple. We are given $\text{VD}^*_\text{out}(u_i, R_{i+1}), v \in \mathcal{P}_{\text{out}}^{R_{i+1}},$ a centroid $f^* \in \mathcal{P}_{\text{out}}^{R_{i+1}}, f$ being a trichromatic face on $y_0, y_1, y_2$, which are, respectively, in the Voronoi cells of $s_0, s_1, s_2 \in \partial R_{i+1}$, and an index $j \in \{0, 1, 2\}$. We would like to know if $v$ is on the shortest $s_j$-to-$y_j$ path. Recall that $t$ is such that $v \notin R_t$ but $v \in R_{t+1}$.

\[^8\text{Of course, } O(1) \text{ time is also possible [22, 4] but this is not the bottleneck in the algorithm.}\]
Figure 5: (a) If \( z = x \) and \( y_j \) is not in \( R_{t+1} \), \( x' \) is the last boundary vertex of \( \partial R_{t+1} \) on the \( s_j-y_j \) path. (b) If \( z = x \) and \( y_j \) is in \( R_{out} \cap R_{t+1} \) then \( x' = y_j \). (Not depicted: if \( y_j \in R_t \) then \( x' = x \).) We test whether \( v \) is on the shortest \( x-x' \) path. If \( z \neq x \) then \( z' \) is well defined and the position of \( y_j \) is immaterial; we test whether \( v \) is on the shortest \( z-z' \) path (depicted in (a)).

Using the lookup tables in part (E) of the data structure, we find the first and last vertices (\( q \) and \( x \)) of \( \partial R_t \) on the \( s_j-y_j \) path. If \( q,x \) do not exist then \( v \) is certainly not on the \( s_j-y_j \) path (Line 4). We may assume \( z \) lies on the \( q-x \) path. If \( z = x \) then there are three cases to consider, depending on whether the destination \( y_j \) of the path is in \( R_{out} \cap R_{t+1} \), or in \( R_{out} \), or in \( R_t \). If \( y_j \in R_{out} \cap R_{t+1} \) we let \( x' = y_j \); if \( y_j \in R_{out} \) we let \( x' \) be the last vertex of \( \partial R_{t+1} \) encountered on the shortest \( s_j-y_j \) path (part (E)); and if \( y_j \in R_t \) we let \( x' = x \). In all cases, \( x' \) is the last vertex of the shortest \( s_j-y_j \) path that is contained in relevant subgraph \( R_{out} \cap R_{t+1} \). (Figure 5(a,b) illustrates the first two possibilities for \( x' \).) Now \( v \) is on the \( s_j-y_j \) path iff it is on the \( x-x' \) shortest path, which can be answered using part (A) of the data structure (Lines 19, 21). (Figure 5(b) illustrates one way for \( v \) to appear on the \( x-x' \) path.) In the remaining case \( z \) is on the shortest \( q-x \) path but is not \( x \), meaning the child \( z' \) of \( z \) on \( T^R_q[x] \) is well defined. If \( zz' \) is a chord (corresponding to a path in \( R_{out} \)) then \( v \) is on the shortest \( s_j-y_j \) path iff it is on the shortest \( z-z' \) path in \( R_{out} \), which, once again, can be answered with part (A) of the data structure (Lines 26, 28). See Figure 5(a) for an illustration of this case.

Remark 1. Strictly speaking we cannot apply Lemma 2.2 (Gawrychowski et al. [20]) since we do not have an MSSP structure for all of \( R_{out} \). Part (A) only handles distance/LCA queries when the query vertices are in \( R_{out} \cap R_{t+1} \). It is easy to make Gawrychowski et al.’s algorithm work using parts (A) and (E) of the data structure. See the discussion at the end of Section 4.4.3.
Algorithm 4 SitePathIndicator(VD* out(u_i, R_{i+1}), v, f^*, j)

Input: The dual representation VD* out(u_i, R_{i+1}) of a Voronoi diagram, a vertex v \in R_{i+1}^\text{out}, and an s_j-to-y_j site-centroid shortest path \((s_j, y_j)\) with respect to f^* in VD*.

Output: True if v is on s_j-to-y_j shortest path, or False otherwise.

1. \(R_t \leftarrow \) the ancestor of \(R_i\) s.t. \(v \notin R_t, v \in R_{t+1}\).
2. \((q, x) \leftarrow\) first and last \(\partial R_t\) vertices on the shortest \(s_j-y_j\) path. \(\triangleright\) Part (E) of the data structure
3. if \(q, x\) are Null then
   4. return False
5. end if
6. \(z \leftarrow\) PointLocate(VD* out(q, R_t), v) \(\triangleright\) Uses parts (A,C,E) of the data structure
7. if \(z\) is not on \(T^{R_t}_q[x]\) then
   8. return False
9. end if
10. if \(z = x\) then
11. if \(y_j\) is in \(R_t^\text{out} \cap R_{t+1}\) then
12. \(x' \leftarrow y_j\)
13. else if \(y_j \notin R_{t+1}\) then
14. \(x' \leftarrow\) last \(\partial R_{t+1}\) vertex on \(s_j-y_j\) path. \(\triangleright\) Part (E)
15. else
16. \(x' \leftarrow x\) \(\triangleright\) I.e., \(y_j \notin R_t^\text{out}\)
17. end if
18. if \(v\) is on the shortest \(x-x'\) path then
19. return True \(\triangleright\) Part (A)
20. else
21. return False
22. end if
23. end if
24. \(z' \leftarrow\) the child of \(z\) on \(T^{R_t}_q[x]\) \(\triangleright\) Part (D)
25. if \(z'z_j\) is a chord in \(C^{R_t}_q[x]\) and \(v\) is on the shortest \(z-z'\) path in \(R_t^\text{out}\) then \(\triangleright\) Part (A)
26. return True
27. end if
28. return False

4.4 The ChordIndicator Function

The ChordIndicator function is given VD* out(u_i, R_{i+1}), v \in R_{i+1}^\text{out}, a centroid f^*, with \(\{y_j, s_j\}\) defined as usual, and an index \(j \in \{0, 1, 2\}\). The goal is to report whether v lies to right of the oriented site-centroid-site chord

\[
C^* = s_jy_jy_{j-1}s_{j-1}^{-1},
\]

which is composed of the shortest \(s_j-y_j\) and \(s_{j-1}-y_{j-1}\) paths, and the single edge \(\{y_j, y_{j-1}\}\). It is guaranteed that \(v\) does not lie on \(C^*\), as this case is already handled by the SitePathIndicator function.

Figure 6 illustrates why this point location problem is so difficult. Since we know \(v \in R_{t+1}\) but not in \(R_t\), we can narrow our attention to \(R_t^\text{out} \cap R_{t+1}\). However the projection of \(C^*\) onto \(R_t^\text{out}\) can touch the boundary \(\partial R_t\) an arbitrary number of times. Define C to be the set of oriented chords of \(R_t^\text{out}\) obtained by projecting \(C^*\) onto \(R_t^\text{out}\).

Luckily C has some structure. Let \((q_j, x_j)\) and \((q_{j-1}, x_{j-1})\) be the first and last \(\partial R_t\) vertices on the shortest \(s_j-y_j\) and \(s_{j-1}-y_{j-1}\) paths, respectively. (One or both of these pairs may not exist.) The chords of C are in one-to-one correspondence with the chords of \(C_1 \cup C_2 \cup C_3\), defined below, but as we will see, sometimes with their orientation reversed.
Figure 6: (a) The projection of a site-centroid-site chord $C^* = \overrightarrow{s_jy_jy_{j-1}s_{j-1}}$ of $R^\text{out}_{i+1}$ onto $R^\text{out}_t$ yields a set $C$ of chords of $R^\text{out}_t$, partitioned into three classes. Let $q_j, x_j$ and $q_{j-1}, x_{j-1}$ be the first and last $\partial R_t$-vertices on the $s_j$-$y_j$ and $s_{j-1}$-$y_{j-1}$ paths. (b) $C_1$: all chords in $T^{R_t}_{q_j}[x_j]$. (c) $C_2$: all chords in $T^{R_t}_{q_{j-1}}[x_{j-1}]$. Their orientation is the reverse of their counterparts in $C^*$. (d) $C_3$: the single chord $\overrightarrow{x_jy_jy_{j-1}x_{j-1}}$. 
\(C_1\): By definition \(C_1 = C^{R_t}_{q_1}[x_1]\) contains all the chords on the path from \(q_j\) to \(x_j\), stored in part (D) of the data structure. Moreover, the orientation of \(C_1\) agrees with the orientation of \(C^*\). The blue chords of Figure (10 a) are isolated as \(C_1\) in Figure (10 b).

\(C_2\): By definition \(C_2 = C^{R_t}_{q_{j-1}}[x_{j-1}]\) contains all the chords on the path from \(q_{j-1}\) to \(x_{j-1}\). The red chords of \(C\) in Figure (10 a) are represented by chords \(C_2\), but with reversed orientation. Figure (10 c) depicts \(C_2\).

\(C_3\): This is the singleton set containing the oriented chord \(x_{j-1}x_j\) consisting of the shortest \(x_j\)-y and \(x_{j-1}\)-y paths and the edge \(\{y_j, y_{j-1}\}\).

The chord-set \(C\) partitions \(R^*_t\) into a piece-set \(P\), with one such piece \(P \in P\) containing \(v\). (Remember that \(v\) is not on \(C^*\).) We can also consider the piece-sets \(P_1, P_2, P_3\) generated by \(C_1, C_2, C_3\). Let \(P_1 \in P_1, P_2 \in P_2, P_3 \in P_3\) be the pieces containing \(v\). Since, ignoring orientation, \(C = C_1 \cup C_2 \cup C_3\), it must be that \(P = P_1 \cup P_2 \cap P_3\). In order to determine whether \(v\) is to the right of \(C^*\), it suffices to find some chord \(C \in C\) bounding \(P\) and ask whether \(v\) is to the right of \(C\). Thus, \(C\) must also be on the boundary of one of \(P_1, P_2\), or \(P_3\).

The high-level strategy of \textit{ChordIndicator} is as follows. First, we will find some piece \(P_1' \in P^{R_t}_q\) that is contained in \(P_1\) using the procedure \textit{PieceSearch} described below, in Section 4.4.1. The chords of \(C_1\) bounding \(P_1\) are precisely the maximal chords in \(C_1\) from vantage \(P_1'\). Using \textit{MaximalChord} (part (D)) we will find a candidate chord \(C_1 \in C_1\), and one edge \(e\) on the boundary cycle of \(\partial R_t\) occluded by \(C_1\) from vantage \(P_1'\). Turning to \(C_2\), we use \textit{AdjacentPiece} to find the piece \(P_2 \in P^{R_t}_{q_{j-1}}\) adjacent to \(e\). Then, using \textit{PieceSearch} and \textit{MaximalChord} again, we find a \(P_2' \in P^{R_t}_{q_{j-1}}\) contained in \(P_2\) and the maximal chord \(C_2\) occluding \(P_e\) from vantage \(P_2'\). Let \(C_3\) be the singleton chord in \(C_3\). We determine the “best” chord \(C_t \in \{C_1, C_2, C_3\}\), decide whether \(v\) lies to the right of \(C_t\), and return this answer if \(t \in \{1, 3\}\) or reverse it if \(t = 2\). Recall that chords in \(C_2\) have the opposite orientation as their counterparts in \(C_1\).

\textbf{PieceSearch} is presented in Section 4.4.1 and \textbf{ChordIndicator} in Section 4.4.2.

\subsection*{4.4.1 \textbf{PieceSearch}}

We are given a region \(R_t\), a vertex \(v \in R^*_t \cap R_{t+1}\), and two vertices \(q, x \in \partial R_t\). We must locate any piece \(P' \in P^{R_t}_q\) that is contained in the unique piece \(P \in P^{R_t}_q[x]\) containing \(v\). The first thing we do is find the last \(\partial R_t\) vertex \(z\) on the shortest path from \(q\) to \(v\), which can be found with a call to \textbf{PointLocate} on \(VD^*_q(q, R_t)\). (This uses parts (A,C,E) of the data structure.) The shortest path from \(z\) to \(v\) cannot cross any chord in \(C^*_q[x]\) (since they are part of a shortest path), but it can coincide with a prefix of some chord in \(C^*_q[x]\). Thus, if no chord of \(C^*_q[x]\) is incident to \(z\), then we are free to return any piece containing \(z\). (There may be multiple options if \(z\) is an endpoint of a chord in \(C^*_q[x]\). This case is depicted in Figure 7.) When \(z = z_0\), we know that \(v \in P_1 \cup \cdots \cup P_3\) and return any piece containing \(z\).) In general \(z\) may be incident to up to two chords \(C_1, C_2 \in C^*_q[x]\). (This occurs when the shortest \(q-x\) path touches \(\partial R_t\) at \(z\) without leaving \(R^*_t\).) In this case we determine which side of \(C_1\) and \(C_2\) \(v\) is on (using parts (A) and (E) of the data structure; see Lemma 4.3 in Section 4.4.3 for details) and return the appropriate piece adjacent to \(C_1\) or \(C_2\). This case is depicted in Figure 7 with \(z = z_1\); the three possible answers coincide with \(v \in \{v_1, v_2, v_3\}\).

\subsection*{4.4.2 \textbf{ChordIndicator}}

Let us walk through the \textbf{ChordIndicator} function. If \(C^* = s_j y_j y_{j-1} s_{j-1}\) does not touch the interior of \(R^*_t\) then the left-right relationship between \(C^*\) and \(v \notin R_t\) is known, and stored in part (E) of the data structure. If this is the case the answer is returned immediately, at Line 3. A relatively simple case is when \(C_1\) and \(C_2\) are empty, and \(C = C_3\) consists of just one chord \(C_3 = x_j y_j y_{j-1} x_{j-1}\). We determine whether \(v\) is to the right or left of \(C_3\) and return this answer (Line 8). (Lemma 4.3 in Section 4.4.3 explains how to test whether \(v\) is to one side of a chord.) Thus, without loss of generality we can assume \(C_1 \neq \emptyset\) and \(C_2\) may or may not be empty.

Recall that \(P_1\) is \(v\)'s piece in \(P^{R_t}_{q_j}[x_j]\). Using \textbf{PieceSearch} we find a piece \(P_1' \subseteq P_1\) in the more refined partition \(P^{R_t}_{q_1}\) and find a \textbf{MaximalChord} \(C_1 \in C_1\) from vantage \(P_1'\), and hence from vantage \(v\) as well. We
Algorithm 5 $\text{PieceSearch}(R_t, q, x, v)$

**Input:** A region $R_t$, two vertices $q, x \in \partial R_t$, and a vertex $v$ not on the $q$-to-$x$ shortest path in $G$.

**Output:** A piece $P' \in \mathcal{P}_{R_t}$, which is a subpiece of the unique piece $P \in \mathcal{P}_{R_t}[x]$ containing $v$.

1. $z \leftarrow \textbf{PointLocate}(\text{VD}^\ast_{\text{out}}(q, R_t), v)$ \hspace{1cm} \triangleright \text{Uses parts (A,C,E) of the data structure}
2. if $z$ is not the endpoint of any chord in $\mathcal{C}_{R_t}[x]$ then
3. \hspace{1cm} \textbf{return} any piece in $\mathcal{P}_{R_t}$ containing $z$.
4. \hspace{1cm} \textbf{end if}
5. $C_1, C_2 \leftarrow$ two chords in $\mathcal{C}_q$ adjacent to $z$ ($C_2$ may be $\textbf{Null}$)
6. Determine whether $v$ is to the left or right of $C_1$ and $C_2$. \hspace{1cm} \triangleright \text{Part (A); see Lemma 4.3}
7. \hspace{1cm} \textbf{return} a piece adjacent to $C_1$ or $C_2$ that respects the queries of Line 6.

Figure 7: Solid chords are in $\mathcal{C}_q[R_t][x]$. Dashed chords are in $\mathcal{C}_q[R_t]$ but not in $\mathcal{C}_q[R_t][x]$. When $z = z_0, v = v_0$, the piece in $\mathcal{P}_q[R_t][x]$ containing $v$ is the union of $P_5 - P_3$. **PieceSearch** reports any piece containing $z_0$. When $z = z_1, v \in \{v_1, v_2, v_3\}$, $z$ is incident to two chords $C_1, C_2$. **PieceSearch** decides which side of $C_1, C_2$ $v$ is on (see Lemma 4.3), and returns the appropriate piece adjacent to $C_1$ or $C_2$. 
regard $\partial R_t$ as circularly ordered according to a clockwise walk around the hole on $\partial R_t$ in $R_t^{\text{out}}$. The chord $C_1$ occludes an interval $I_1$ of $\partial R_t$ from vantage $v$. If $C_1$ is not one of the chords bounding $P$, then $C_3$ or some $C_2 \in C_3$ must occlude a superset of $I_2$, so we will attempt to find such a $C_2$, as follows.

Let $e$ be the first edge on the boundary cycle occluded by $C_1$, i.e., $e$ joins the first two elements of $I_1$. Using $\text{AdjacentPiece}$ we find the unique piece $P_e \in P_{\partial R_t}$ with $e$ on its boundary. Using $\text{PieceSearch}$ again we find $P'_2 \in P_{\partial R_t}^{R_t}$ contained in $P_2$, and using $\text{MaximalChord}$ again, we find the maximal chord $C_2 \in C_2$ that occludes $P_e$ from vantage $P'_2$, and hence from vantage $v$ as well. Observe that since all chords in $C_2$ are vertex-disjoint from $C_1$, if $C_2 \neq \text{Null}$ then $C_2$ must occlude a strictly larger interval $I_2 \supset I_1$ of $\partial R_t$. (If $C_2$ is $\text{Null}$ then $I_2 = \emptyset$.) It may be that $C_1$ and $C_2$ are both not on the boundary of $P$, but the only way that could occur is if $C_3 \in C_3$ occludes a superset of $I_1$ and $I_2$ on the boundary $\partial R_t$. We check whether $v$ lies to the right or left of $C_3$ and let $I_3$ be the interval of $\partial R_t$ occluded by $C_3$ from vantage $v$. If $I_3$ does not cover $e$, then we cannot conclude that $C_3$ is superior than $C_1/C_2$. Thus, we find the chord $C_3 \in \{C_1, C_2, C_3\}$ that covers $e$ and maximizes $|I_3|$. $C_3$ must be on the boundary of $P$, so the left-right relationship between $v$ and $C^*$ is exactly the same as the left-right relationship between $v$ and $C_3$, if $\ell \in \{1, 3\}$, and the reverse of this relationship if $\ell = 2$ since chords in $C_2$ have the opposite orientation as their subpath counterparts in $C^*$.

Figure 8 illustrates how $\ell$ could take on all three values.

### 4.4.3 Side Queries

Lemma 4.3 explains how we test whether $v$ is to the right or left of a chord, which is used in both $\text{PieceSearch}$ and $\text{ChordIndicator}$.

**Lemma 4.3.** For any $C \in C_1 \cup C_2 \cup C_3$ and $v$ not on $C$, we can test whether $v$ lies to the right or left of $C$ in $O(\alpha \log \log n)$ time, using parts (A) and (E) of the data structure.

**Proof.** There are several cases.

**Case 1.** Suppose that $C = \overline{c_0c_1} \in C_1 \cup C_2$ corresponds to the shortest path from $c_0$ to $c_1$ in $R_t^{\text{out}}$, $c_0, c_1 \in \partial R_t$. Let $c'_0, c'_1$ be pendant vertices attached to $c_0, c_1$, embedded inside the face of $R_t^{\text{out}}$ bounded by $\partial R_t$. The shortest $c'_0-v$ paths and $c'_0-c'_1$ paths branch at some point. We ask the MSSP structure (part (A)) for the least common ancestor, $w$, of $v$ and $c'_1$ in the shortcutted SSSP tree rooted at $c'_0$. This query also returns...
Algorithm 6 ChordIndicator(VD\textsubscript{out}(u_i, R_{i+1}), v, f^*, j)

\textbf{Input}: The dual representation VD\textsubscript{out} = VD\textsubscript{out}(u_i, R_{i+1}) of a Voronoi diagram, a centroid \( f^* \) in VD\textsubscript{out} with face \( f \) on vertices \( y_0, y_1, y_2 \), which are in the Voronoi cells of \( s_0, s_1, s_2 \), an index \( j \in \{0, 1, 2\} \), and a vertex \( v \in R^\text{out}_{i+1} \) that does not lie on the site-centroid-site chord \( C^* = \overrightarrow{s_j y_j y_{j-1}} \).

\textbf{Output}: True if \( v \) lies to the right of \( C^* \), and False otherwise.

\begin{enumerate}
\item \( R_t \leftarrow \) the ancestor of \( R_i \) s.t. \( v \notin R_t, v \in R_{t+1} \). \( C \) is the projection of \( C^* \) onto \( R^\text{out}_t \).
\item if the left/right relationship between \( R^\text{out}_t \) and \( C^* = \overrightarrow{s_j y_j y_{j-1}} \) is known then
\item return stored True/False answer. \hspace{1cm} \triangleright \text{Part (E)}
\end{enumerate}

\begin{enumerate}
\item end if
\item \( (q_j, x_j) \leftarrow \) first and last \( \partial R_t \)-vertices on shortest \( s_j \)-\( y_j \) path. \hspace{1cm} \triangleright \text{Part (E)}
\item \( (q_{j-1}, x_{j-1}) \leftarrow \) first and last \( \partial R_t \)-vertices on shortest \( s_{j-1} \)-\( y_{j-1} \) path. \hspace{1cm} \triangleright \text{Part (E)}
\item if \( C_1 = C_2 = \emptyset \) then
\item return True if \( v \) is to the right of the \( C_3 \)-chord \( \overrightarrow{x_j y_j y_{j-1} x_{j-1}} \), or False otherwise. \hspace{1cm} \triangleright \text{W.l.o.g., continue under the assumption that } C_1 \neq \emptyset.
\end{enumerate}

\begin{enumerate}
\item end if
\item \( P'_1 \leftarrow \text{PieceSearch}(R_t, q_j, x_j, v) \) \hspace{1cm} \triangleright \text{Uses parts (A,C)}
\item \( C_1 \leftarrow \text{MaximalChord}(R_t, q_j, x_j, P'_1, \perp) \) \hspace{1cm} \triangleright \text{Part (D)}
\item \( I_1 \leftarrow \) the clockwise interval of hole \( \partial R_t \) occluded by \( C_1 \) from vantage \( v \).
\item \( e \leftarrow \) edge joining first two elements of \( I_1 \).
\item \( P_e \leftarrow \text{AdjacentPiece}(R_t, q_{j-1}, e) \) \hspace{1cm} \triangleright \text{Part (D)}
\item \( P'_2 \leftarrow \text{PieceSearch}(R_t, q_{j-1}, x_{j-1}, v) \) \hspace{1cm} \triangleright \text{Uses parts (A,C)}
\item \( C_2 \leftarrow \text{MaximalChord}(R_t, q_{j-1}, x_{j-1}, P'_2, P_e) \) \hspace{1cm} \triangleright \text{Part (D); may return Null}
\item \( I_2 \leftarrow \) the clockwise interval of hole \( \partial R_t \) occluded by \( C_2 \) from vantage \( v \). \hspace{1cm} \triangleright \text{Null if } C_2 = \text{Null}
\item \( C_3 \leftarrow \) single chord in \( C_3 \), if any. \hspace{1cm} \triangleright \text{May be Null}
\item \( I_3 \leftarrow \) the clockwise interval of hole \( \partial R_t \) occluded by \( C_3 \) from vantage \( v \). \hspace{1cm} \triangleright \text{Null if } C_3 = \text{Null}
\item \( \ell \leftarrow \text{index such that } I_\ell \text{ covers } e, \text{ and } |I_\ell| \text{ is maximum.} \)
\item if \( \ell \) is to the right of \( C_\ell \) and \( \ell \in \{1, 3\} \) or \( \ell \) is to the left of \( C_\ell \) and \( \ell = 2 \) then
\item return True
\item end if
\item end if
\item return False
the two tree edges $e_v, e_{c'_1}$ leading to $v$ and $c'_1$, respectively. Let $e_w$ be the edge connecting $w$ to its parent. If the clockwise order around $w$ is $e_w, e_{c'_1}, e_v$ then $v$ lies to the right of $c_0c_1'$; otherwise it lies to the left. Note that if the shortest $c'_0-c'_1$ and $c'_0-v$ paths in $G$ branch at a point in $R^\text{out}_{t+1}$, then $w$ will be the nearest ancestor of the branchpoint on $\partial R_{t+1}$ and one or both of $e_v, e_{c'_1}$ may be “shortcut” edges in the MSSP structure. See Figure 9(a) for a depiction of this case.

![Figure 9](image-url)

**Figure 9**: (a) The chord $C \in C_1 \cup C_2$ corresponds to a shortest path, which may pass through $R^\text{out}_{t+1}$, in which case it is represented in the MSSP structure with shortcut edges (solid, angular edges). (b) The chord $C = \overrightarrow{x_jy_jy_{j-1}x_{j-1}}$ is in $C_3$, and $f$ lies in $R^\text{out}_t \cap R_{t+1}$. This is handled similarly to (a). (c) Here $f$ lies in $R^\text{out}_{t+1}$, $\hat{x}_j, \hat{x}_{j-1}$ are the last $\partial R_{t+1}$ vertices on the $s_j-y_j$ and $s_{j-1}-y_{j-1}$ paths. If the shortest $x'_j-\hat{x}_j$ and $x'_j-v$ paths branch, we can answer the query as in (b). If $x'_j-\hat{x}_j$ is a prefix of $x'_j-v$, $e_v = (\hat{x}_j, \hat{v})$, and $\hat{v} \in \partial R_{t+1}$, then we can use the clockwise order of $\hat{x}_j, \hat{v}, \hat{x}_{j-1}$ around the hole on $\partial R_{t+1}$ to determine whether $v$ lies to the right of $C$. (Not depicted: the case when $\hat{v} \notin \partial R_{t+1}$.)

**Case 2.** Now suppose $C = \overrightarrow{x_jy_jy_{j-1}x_{j-1}}$ is the one chord in $C_3$. Consider the following distance function $\hat{d}$ for vertices in $z \in R^\text{out}_t$:

$$
\hat{d}(z) = \min \left\{ \text{dist}_G(u_i, x_j) + \text{dist}_G(x_j, z), \ \text{dist}_G(u_i, x_{j-1}) + \text{dist}_G(x_{j-1}, z) \right\}.
$$

The purpose of adding $c'_0, c'_1$ is to make sure all three edges $e_v, e_{c'_1}, e_{c'_1}$ exist. The vertices $c'_0, c'_1$ are not represented in the MSSP structure. The edges $(c'_0, c_0)$ and $(c_1, c'_1)$ can be simulated by inserting them between the two boundary edges on $\partial R_t$ adjacent to $c_0$ and $c_1$, respectively.
Observe that the terms involving \(u_i\) are stored in part (E) and, if \(z \in R^\text{out}_t \cap R_{t+1}\), the other terms can be queried in \(O(\kappa \log \log n)\) time using part (A). It follows that the shortest path forest w.r.t. \(d\) has two trees, rooted at \(x_j\) and \(x_{j-1}\). Using part (A) of the data structure we compute \(\hat{d}(v)\), which reveals the \(j^* \in \{j, j - 1\}\) such that \(v\) is in \(x_{j^*}\)'s tree. At this point we break into two cases, depending on whether \(f\) is in \(R^\text{out}_t \cap R_{t+1}\), or in \(R^\text{out}_t \cap R_{t+1}\). We assume \(j^* = j\) without loss of generality and depict only this case in Figure 9(b,c).

**Case 2a.** Suppose that \(f\) is in \(R^\text{out}_t \cap R_{t+1}\). Let \(y^*_j\) be a pendant vertex attached to \(x_j\) embedded inside \(f\) and let \(x^*_j\) be a pendant attached to \(x_j\) embedded in the face on \(\partial R_t\). The shortest \(x^*_j-y^*_j\) and \(x^*_j-v\) paths diverge at some point. We query the MSSP structure (part (A)) to get the least common ancestor \(w\) of \(y^*_j\) and \(v\) and the three edges \(e_{y^*_j}, e_v, e_w\) around \(w\), then determine the left/right relationship as in Case 1. (If \(j^* = j - 1\) then we would reverse the answer due to the reversed orientation of the \(x_{j-1}-y_{j-1}\) subpath w.r.t. \(C\).) Once again, some of \(e_{y^*_j}, e_v, e_w\) may be shortcut edges between \(\partial R_{t+1}\)-vertices or artificial pendant edges. See Figure 9(b). 

**Case 2b.** Now suppose \(f\) lies in \(R^\text{out}_t \cap R_{t+1}\). We get from part (E) the last vertices \(\hat{x}_j, \hat{x}_{j-1} \in \partial R_{t+1}\) that lie on the \(s_j-y_j\) and \(s_{j-1}-y_{j-1}\) shortest paths. We ask the MSSP structure of part (A) for the least common ancestor \(w\) of \(\hat{x}_j\) and \(v\) in the shortcutted SSSP tree rooted at \(x^*_j\), and also get the three incident edges \(e_{\hat{x}_j}, e_v, e_w\). The edges \(e_v\) and \(e_w\) exist and are different, but \(e_{\hat{x}_j}\) may not exist if \(w = \hat{x}_j\), i.e., if \(v\) is a descendant of \(\hat{x}_j\). If all three edges \(\{e_{\hat{x}_j}, e_v, e_w\}\) exist we can determine whether \(v\) lies to the right of \(C\) as in Case 1 or 2a.

**Case 2b(i).** Suppose \(w = \hat{x}_j\) and \(e_{\hat{x}_j}\) does not exist. Let \(e_v = (\hat{x}_j, \hat{v})\). If \(\hat{v} \in \partial R_{t+1}\) then \(e_v\) represents a path that is completely contained in \(R^\text{out}_{t+1}\). Thus, if we walk clockwise around the hole of \(R^\text{out}_{t+1}\) on \(\partial R_{t+1}\) and encounter \(\hat{x}_j, \hat{v}, \hat{x}_{j-1}\) in that order then \(v\) lies to the right of \(C\), and if we encounter them in the reverse order then \(v\) lies to the left of \(C\). See Figure 9(c).

**Case 2b(ii).** Finally, suppose \(\hat{v} \notin \partial R_{t+1}\) and \(e_v = (\hat{x}_j, \hat{v})\) is a normal edge in \(G\). Redefine \(e_{\hat{x}_j}\) to be the first edge on the path from \(\hat{x}_j\) to \(y^*_j\). Now we can determine if \(v\) is to the right of \(C\) by looking at the clockwise order of \(e_w, e_v, e_{\hat{x}_j}\) around \(\hat{x}_j\). 

As pointed out in Remark 1, Lemma 2.1 does not immediately imply that Line 6 of SitePathIndicator and Line 1 of PieceSearch can be implemented efficiently. Gawrychowski et al.’s 20 implementation of PointLocate requires MSSP access to \(R^\text{out}_t\), whereas part (A) only lets us query vertices in \(R^\text{out}_t \cap R_{t+1}\). Gawrychowski et al.’s algorithm is identical to CentroidSearch, except that Navigation is done directly with MSSP structures. Suppose we are currently at \(f^*\) in the centroid decomposition, with \(y_j, s_j\) defined as usual. Gawrychowski’s algorithm finds \(j\) minimizing \(\omega(s_j) + \text{dist}_{R^\text{out}_t}(s_j, v)\) using three distance queries to the MSSP structure, then decides whether the \(s_j'-v\) shortest path is a prefix of the \(s_j-y_j\) shortest path, and if not, which direction it branches in.\(^{11}\) If \(f\) is in \(R^\text{out}_t \cap R_{t+1}\) we can proceed exactly as in Gawrychowski et al. 20. If not, we retrieve from part (E) the last vertex \(\hat{x}\) of \(\partial R_{t+1}\) on the \(s_j-y_j\) shortest path, use \(\hat{x}\) in lieu of \(y^*_j\) for the LCA queries, and tell whether the \(s_j'-v\) path branches to the right exactly as in Lemma 4.3 Case 2b.

**5 Analysis**

This section constitutes a proof of the claims of Theorem 1.1 concerning space complexity and query time; refer to Appendix C for an efficient construction algorithm. 

---

\(^{10}\)We could store \(e_{\hat{x}_j}\) in part (E) of the data structure but that is not necessary. If \(e_0, e_1\) are the edges adjacent to \(\hat{x}_j\) on the boundary cycle of \(\partial R_{t+1}\), then we can use any member of \(\{e_0, e_1\}\setminus\{e_w\}\) as a proxy for \(e_{\hat{x}_j}\).

\(^{11}\)\(s_j', y_j'\) being pendant vertices attached to \(s_j, y_j\) as in Lemma 4.3.
Combining Lemmas 2.1 and 2.2 (see Section 4.4.3), \textbf{PointLocate} runs in $O(\kappa \log n \log \log n)$ time. Together with \textbf{Lemma 4.3} it follows that \textbf{PieceSearch} also takes $O(\kappa \log n \log \log n)$ time. \textbf{SitePathIndicator} uses \textbf{PointLocate}, the MSSP structure, and $O(1)$-time tree operations on $T^R_i$ and the $\bar{r}$-hierarchy like least common ancestors and level ancestors \cite{22, 4, 5, 21}. Thus \textbf{SitePathIndicator} also takes $O(\kappa \log n \log \log n)$ time. The calls to \textbf{MaximalChord} and \textbf{AdjacentPiece in ChordIndicator} take $O(\log n \log \log n)$ time by \textbf{Lemma 4.1} and testing which side of a chord $v$ lies on takes $O(\kappa \log \log n)$ time by \textbf{Lemma 4.3}. The bottleneck in \textbf{ChordIndicator} is still \textbf{PieceSearch}, which takes $O(\kappa \log n \log \log n)$ time. The only non-trivial parts of \textbf{Navigation} are calls to \textbf{SitePathIndicator} and \textbf{ChordIndicator}, so it, too, takes $O(\kappa \log n \log \log n)$ time.

An initial call to \textbf{CentroidSearch} (Line 5 of \textbf{Dist}) generates at most $\log n$ recursive calls to \textbf{CentroidSearch}, culminating in the last recursive call making 1 or 2 calls to \textbf{Dist} with the \textquote{\textendash\textendash} parameter incremented. Excluding the cost of recursive calls to \textbf{Dist}, the cost of \textbf{CentroidSearch} is dominated by calls to \textbf{Navigation}, i.e., an initial call to \textbf{CentroidSearch} costs $\log n \cdot O(\kappa \log \log n) = O(\kappa \log^2 n \log \log n)$ time. Let $T(i)$ be the cost of a call to $\textbf{Dist}(u_i, v, R_i)$. We have

$$T(m - 1) = O(\kappa \log \log n)$$

$$T(i) = 2T(i + 1) + O(\kappa \log^2 n \log \log n)$$

It follows that the time to answer a distance query is $T(0) = O(2^m \cdot \kappa \log^2 n \log \log n)$.

The space complexity of each part of the data structure is as follows. \textbf{A} is $O(\kappa mn^{1+1/m+1/\kappa})$ by \textbf{Lemma 2.1} and the fact that $r_{i+1}/r_i = n^{1/m}$. \textbf{B} is $O(\kappa mn^{1+1/(2m)})$ since $\sqrt{r_{i+1}/r_i} = n^{1/(2m)}$. \textbf{C} is $O(mn)$ since $\sum_{i} n/r_i \cdot \sqrt{r_i} = O(mn)$. \textbf{D} is $O(mn \log n)$ by \textbf{Lemma 4.2} and \textbf{E} is $O(m)$ times the space cost of (B) and (C), namely $O(m^2n^{1+1/(2m)})$. The bottleneck is (A).

We now explain how $m, \kappa$ can be selected to achieve the extreme space and query complexities claimed \textbf{Theorem 1.1}. To optimize for query time, pick $\kappa = m$ to be any function of $n$ that is $\omega(1)$ and $o(\log \log n)$. Then the query time is

$$O(2^m \kappa \log^2 n \log \log n) = \log^{2+o(1)} n$$

and the space is

$$O(mkn^{1+1/m+1/\kappa}) = n^{1+o(1)}.$$

To optimize for space, choose $\kappa = \log n$ and $m$ to be a function that is $\omega(\log n / \log \log n)$ and $o(\log n)$. Then the space is

$$O(mkn^{1+1/m+1/\kappa}) = o\left(n^{1+1/m} \log^2 n\right) = n \cdot 2^{o(\log \log n)} \cdot \log^2 n = n \log^{2+o(1)} n,$$

and the query time

$$O(2^m \kappa \log^2 n \log \log n) = 2^{o(\log n)} \log^3 n \log \log n = n^{o(1)}.$$

### 5.1 Speeding Up the Query Time

Observe that the space of (B) is asymptotically smaller than the space of (A). Replace (B) with (B’)

(B’) \textbf{(Voronoi Diagrams)} Fix $i$, a region $R_i \in \mathcal{R}_i$ with ancestors $R_{i+1} \in \mathcal{R}_{i+1}$ and $R_{i+4} \in \mathcal{R}_{i+4}$. For each $q \in \partial R_i$ store

$$VD^*_\text{out}(q, R_{i+1}) = VD^*[R^\text{out}_{i+1}, \partial R_{i+1}, \omega]$$

$$VD^*_\text{farout}(q, R_{i+4}) = VD^*[R^\text{out}_{i+4}, \partial R_{i+4}, \omega]$$

only if $i < m - 4$

with $\omega(s) = \text{dist}_G(q, s)$ in both cases. Over all regions $R_i$, the space for storing all $VD^*_\text{out}$’s is $\tilde{O}(n^{1+1/(2m)})$ since $\sqrt{r_{i+1}/r_i} = n^{1/(2m)}$ and the space for $VD^*_\text{farout}$’s is $\tilde{O}(n^{1+2/m})$ since $\sqrt{r_{i+4}/r_i} = n^{2/m}$.  

24
Now the space for (A) is $\tilde{O}(n^{1+1/m+1/\kappa}) = \tilde{O}(n^{1+2/m})$ is balanced with (B'). In the Dist function we now consider three possibilities. If $v \in R_{i+1}$ we use part (A) to solve the problem without recursion. If $v \not\in R_{i+1}$ but $v \in R_{i+4}$ we proceed as usual, calling CentroidSearch(VD$_{out}^*(u_i, R_{i+1}), v, \cdot$), and if $v \not\in R_{i+4}$ we call CentroidSearch(VD$_{farout}^*(u_i, R_{i+4}), v, \cdot$). Observe that the depth of the Dist-recursion is now at most $t/4 + O(1) < m/4 + O(1)$, giving us a query time of $O(m^2/m\log 2 n \log \log n)$ with space $\tilde{O}(n^{1+2/m})$.

6 Conclusion

In this paper we have proven that it is possible to simultaneously achieve optimal space or query time, up to a $\log^{2+o(1)} n$ factor, and near-optimality in the other complexity measure, up to an $n^{o(1)}$ factor. The main open question in this area is whether there exists an exact distance oracle with $\tilde{O}(n)$ space and $\tilde{O}(1)$ query time. This will likely require new insights into the structure of shortest paths, which could lead, for example, to storing correlated versions of Voronoi diagrams more efficiently, or avoiding the binary branching recursion in our query algorithm.

Acknowledgements. We thank Danny Sleator and Bob Tarjan for discussing update/query time tradeoffs for dynamic trees.

References

[1] Ittai Abraham and Cyril Gavoille. On approximate distance labels and routing schemes with affine stretch. In Proceedings of the 25th International Symposium on Distributed Computing (DISC), volume 6950 of Lecture Notes in Computer Science, pages 404–415, 2011.

[2] Rachit Agarwal. The space-stretch-time tradeoff in distance oracles. In Proceedings of the 22nd European Symposium on Algorithms (ESA), volume 8737 of Lecture Notes in Computer Science, pages 49–60, 2014.

[3] Srinivasa Rao Arikati, Danny Z. Chen, L. Paul Chew, Gautam Das, Michiel H. M. Smid, and Christos D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In Proceedings 4th Annual European Symposium on Algorithms (ESA), volume 1136 of Lecture Notes in Computer Science, pages 514–528, 1996.

[4] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN), volume 1776 of Lecture Notes in Computer Science, pages 88–94. Springer, 2000.

[5] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. Theor. Comput. Sci., 321(1):5–12, 2004.

[6] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. ACM Transactions on Algorithms, 11(3):1–29, 2015.

[7] Sergio Cabello. Many distances in planar graphs. Algorithmica, 62(1-2):361–381, 2012.

[8] Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. ACM Trans. Algorithms, 15(2):21:1–21:38, 2019.

[9] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. In Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC), pages 138–151, 2019.

[10] Danny Z. Chen and Jinhui Xu. Shortest path queries in planar graphs. In Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC), pages 469–478, 2000.
[11] Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. In Proceedings 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 962–973, 2017.

[12] Paul F. Dietz. Fully persistent arrays. In Proceedings of the First Workshop on Algorithms and Data Structures (WADS), volume 382 of Lecture Notes in Computer Science, pages 67–74, 1989.

[13] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 524–531, 1988.

[14] Edsger W Dijkstra. A note on two problems in connexion with graphs. Numerische mathematik, 1(1):269–271, 1959.

[15] Hristo Djidjev. On-line algorithms for shortest path problems on planar digraphs. In Proceedings of the 22nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG), volume 1197 of Lecture Notes in Computer Science, pages 151–165, 1996.

[16] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. J. Comput. Syst. Sci., 38(1):86–124, 1989.

[17] Jeff Erickson, Kyle Fox, and Luvsandondov Lkhamsuren. Holiest minimum-cost paths and flows in surface graphs. In Proceedings of the 50th Annual ACM Symposium on Theory of Computing (STOC), pages 1319–1332, 2018.

[18] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci., 72(5):868–889, 2006.

[19] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput., 16(6):1004–1022, 1987.

[20] Pawel Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 515–529, 2018.

[21] Torben Hagerup. Still simpler static level ancestors. CoRR, abs/2005.11188, 2020.

[22] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. SIAM J. Comput., 13(2):338–355, 1984.

[23] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM, 46(4):502–516, 1999.

[24] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. J. ACM, 24(1):1–13, 1977.

[25] Ken-ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In Proceedings of the 38th Int’l Colloquium on Automata, Languages and Programming (ICALP), volume 6755 of Lecture Notes in Computer Science, pages 135–146, 2011.

[26] Ken-ichi Kawarabayashi, Christian Sommer, and Mikkel Thorup. More compact oracles for approximate distances in undirected planar graphs. In Proceedings of the 24th ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 550–563, 2013.

[27] Philip Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 820–827, 2002.
A MSSP via Euler Tour Trees (Proof of Lemma 2.1)

Let us recall the setup. We have a planar graph $H$ with a distinguished face $f$, and wish to answer $\text{dist}_H(s,v)$ queries w.r.t. any $s$ on $f$ and $v \in V(H)$, and LCA queries w.r.t. any $s$ on $f$ and $u,v \in V(H)$. Klein \cite{28} proved that if we move the source vertex $s$ around $f$ and record all the changes to the SSSP tree, every edge in $E(H)$ can be swapped into and out of the SSSP at most once, i.e., there are $O(|H|)$ updates in total. Thus, if we maintain the SSSP tree as the source travels around $f$ in a dynamic data structure with update time $U$ and query time $T$ (for distance and LCA queries), the universal persistence method for RAM data structures (see \cite{12}) yields an MSSP data structure with space $O(|H|U)$ and query time $O(T \log \log |H|)$. Thus, to establish Lemma 2.1 it suffices to design a dynamic data structure for the following:
Figure 10: The effect of $\text{Swap}(d, i, \cdot)$ on the Euler Tour. The interval $((b, d), (d, e), \cdots, (e, d), (d, b))$ is spliced out and inserted between $(h, i)$ and $(i, j)$, and the elements $(b, d), (d, b)$ are renamed $(i, d), (d, i)$.

$\text{InitTree}(s^*, T)$: Initialize a directed spanning tree $T$ from root $s^*$. Edges have real-valued lengths.

$\text{Swap}(v, p, l)$: Let $p'$ be the parent of $v$; $p$ is not a descendant of $v$. Update $T \leftarrow T - \{(p', v)\} \cup \{(p, v)\}$, where $(p', v)$ has length $l$.

$\text{Dist}(v)$: Return $\text{dist}_T(s^*, v)$.

$\text{LCA}(u, v)$: Return the LCA $y$ of $u$ and $v$ and the first edges $e_u, e_v$ on the paths from $y$ to $u$ and from $y$ to $v$, respectively.

Here $s^*$ will be a fixed root vertex embedded in $f$ with a single, weight-zero, out-edge to the current root on $f$. Changes to the SSSP tree are effected with $O(|H|)$ $\text{Swap}$ operations. Klein [28] and Gawrychowski [20] use Sleator and Tarjan’s Link-Cut trees [37], which support $\text{Swap}$, $\text{Dist}$, and $\text{LCA}$ (among other operations) in $O(\log |T|)$ time. We will use a souped-up version of Henzinger and King’s [23] Euler Tour trees. Let $\text{ET}(T)$ be an Euler tour of $T$ starting and ending at $s^*$. The elements of $\text{ET}(T)$ are edges, and each edge of $T$ appears twice in $\text{ET}(T)$, once in each direction. Each edge in $T$ points to its two occurrences in $\text{ET}(T)$.

Suppose $T_{\text{ante}}$ is the tree before a $\text{Swap}$ operation and $T_{\text{post}}$ the tree afterward. It is easy to see that $\text{ET}(T_{\text{post}})$ can be derived from $\text{ET}(T_{\text{ante}})$ by $O(1)$ splits and concatenates, and renaming the two elements corresponding to the swapped edge. See Figure 10. We will argue that the dynamic tree operations $\text{Swap}$, $\text{Dist}$, $\text{LCA}$ can be implemented using the following list operations.

$\text{InitList}(L)$: Initialize a list $L$ of weighted elements.

$\text{Split}(e_0)$: Element $e_0$ appears in some list $L$. Split $L$ immediately after element $e_0$, resulting in two lists.

$\text{Concatenate}(L_0, L_1)$: Concatenate $L_0$ and $L_1$, resulting in one list.

$\text{Add}(e_0, e_1, \delta)$: Here $e_0, e_1$ are elements of the same list $L$. Add $\delta \in \mathbb{R}$ to the weight of all elements in $L$ between $e_0$ and $e_1$ inclusive.

$\text{Weight}(e_0)$: Return the weight of $e_0$.

$\text{RangeMin}(e_0, e_1)$: Return the minimum-weight element between $e_0$ and $e_1$ inclusive. If there are multiple minima, return the first one.

To implement $\text{Dist}$ and $\text{LCA}$ we will actually use the list data structure with different weight functions. For $\text{Dist}$, the weight of an edge $(x, y)$ in $\text{ET}(T)$ is $\text{dist}_T(s^*, y)$. Thus, $\text{Dist}$ is answered with a call to $\text{Weight}$. Each $\text{Swap}(v, p, l)$ is effected with $O(1)$ $\text{Split}$ and $\text{Concatenate}$ operations, renaming the elements of the swapped edge, as well as one $\text{Add}(e_0, e_1, \delta)$ operation. Here $(e_0, \ldots, e_1)$ is the sub-list corresponding to the
Figure 11: An illustration of an LCA(e, j) query. We do a RangeMin query on the interval e₀ = (d, e), . . . , (i, j) = e₁ and retrieve the edge ̂e = eᵩ = (b, g) with weight depthᵩ(T). We then find ̂e′ = (g, s*) and its predecessor ̂e″ = (h, g). Another RangeMin query on the interval (i, j), . . . , (h, g) returns eⱼ = (h, g).

subtree rooted at v, and δ = distᵩₜₚᵩ,(s*, v) − distᵩₜₚᵩ,(s*, v) is the change in distance to v, and hence all descendants of v.

To handle LCA queries, we use the list data structure where the weight of (x, y) is the depth of y in T, i.e., the distance from s* to y under the unit length function. Once again, a Swap is implemented with O(1) Split and Concatenate operations, and one Add operation. Consider an LCA(u, v) query. Let e₀ = (pᵤ, u), e₁ = (pᵥ, v) be the edges into u and v from their respective parents, and suppose that e₀ appears before e₁ in ET(T).¹² A call to RangeMin(e₀, e₁) returns the first edge ̂e = (x, y) in the interval (e₀, . . . , e₁) minimizing the depth of y. It follows that y is the LCA of u and v. Furthermore, by the tiebreaking rule, if ̂e ̸= e₀ then ̂e = eₙ is the (reversal of the) edge leading from y towards u. If ̂e = e₀ then v is a descendant of u and eₙ does not exist. To find eᵥ, we retrieve the edge ̂e = (y, pᵧ) in ET(T) from y to its parent and let ̂e′ be its predecessor in ET(T). (Note that since s* has degree 1, ̂e, ̂e′ always exist.) We call RangeMin(e₁, ̂e′). Once again, by the tiebreaking rule it returns the first edge eᵥ = (x′, y) incident to y in (e₁, . . . , ̂e′), which is the (reversal of the) first edge on the path from y to v. See Figure 11.

We have reduced our dynamic tree problem to a dynamic weighted list problem. We now explain how the dynamic list problem can be solved with balanced trees.

Fix a parameter κ ≥ 1 and let n be the total number of elements in all lists. We now argue that Split, Concatenate, and Add can be implemented in O(κn¹/κ) time and Weight and RangeMin take O(κ) time. We store the elements of each list L at the leaves of a rooted tree T(L). It satisfies the following invariants.

I. Each node γ of T(L) stores a weight offset w(γ), a min-weight value min(γ) and a pointer ptr(γ). The weight of (leaf) e ∈ L is the sum of the w(·)-values of its ancestors, including e. The sum of min(γ) and the w(·)-values of all strict ancestors of γ is exactly the weight of the minimum weight descendant of γ, and ptr(γ) points to this element.

II. Non-root internal nodes have between n¹/κ and 3n¹/κ children. In particular, the tree has height at most κ.

III. Each internal node γ maintains an O(1)-time range minimum structure over the vector of min(·)-values of its children.

It is easy to show that Split and Concatenate can be implemented to satisfy Invariant II by destroying/rebuilding O(1) nodes at each level of T. Each costs O(n¹/κ) time to update the information covered by Invariants I and III. The total time is therefore O(κn¹/κ). By Invariant I, a Weight(e₀) query takes O(κ) time to sum all of e₀’s ancestors’ w(·)-values. Consider an Add(e₀, e₁, δ) or RangeMin(e₀, e₁) operation.

¹²As we will see, it is easy to determine which comes first.
By Invariant II, the interval \((e_0, \ldots, e_1)\) is covered by \(O(\kappa n^{1/\kappa})\) \(T\)-nodes, and furthermore, those nodes can be arranged into less than \(2\kappa\) contiguous intervals of siblings. Thus, an \texttt{Add}(e_0, e_1)\) can be implemented in \(O(\kappa n^{1/\kappa})\) time by adding \(\delta\) to the \(w(\cdot)\)-values of these nodes and rebuilding the affected range-min structures from Invariant III. A \texttt{RangeMin}\) is reduced to \(O(\kappa)\) range-minimum queries (from Invariant III) and adjusting the answers by the \(w(\cdot)\)-values of their ancestors (Invariant I). Each range-min query takes \(O(1)\) time and there are \(O(\kappa)\) ancestors with relevant \(w(\cdot)\)-values. Thus \texttt{RangeMin}\) takes \(O(\kappa)\) time.

We have shown that the dynamic tree operations necessary for an \texttt{MSSP}\) structure can be implemented with a flexible tradeoff between update time and query time. Moreover, this lower bound meets the Pătrașcu-Demaine lower bound \[34\]. We leave it as an open problem to implement the full complement of operations supported by Link-Cut trees, with update time \(O(\kappa n^{1/\kappa})\) and query time \(O(\kappa)\).

## B Multiple Holes and Nonsimple Cycles

We have assumed for simplicity that all regions are bounded by a simple cycle, and therefore have a single hole. We now show how these assumptions can be removed.

Let us first illustrate how a region \(R\) may get a hole with a non-simple boundary cycle. The hierarchical decomposition algorithm of Klein, Mozes, and Sommer \[29\] produces a binary decomposition tree, of which our \(\vec{r}\)-decomposition is a coarsening. It proceeds by finding a separating cycle (as in Miller \[31\]), and recursively decomposes the graph inside the cycle and outside the cycle. \[29\]. At intermediate stages the working graph contains several holes, but Miller’s theorem \[31\] only guarantees that a small cycle separator exists if the graph is triangulated. To that end, the decomposition \[29\] puts an artificial vertex inside each hole and triangulates the hole. See Figure 12(a,b). If the cycle separator \(C\) (blue cycle in Figure 12(b)) includes a hole-vertex \(v\), we splice out \(v\) and replace it with an interval of the boundary of the hole. If \(C\) also includes edges on the boundary of the hole (Figure 12(c)), the modified cycle may not be simple. If this is the case, we “cut” along non-simple parts of the cycle, replicating all such vertices and their incident cycle edges. We then join pairs of identical vertices with zero-length edges (pink edges in Figure 12(c)), and triangulate with large-length edges. This transformation clearly preserves planarity and does not change the underlying metric.\[13\]

Turning to the issue of multiple holes, we first make some observations about their structural organization. Fix any hole \(g\) of region \(R_{i+1}\) and let \(R_i\) be a child of \(R_{i+1}\). There is a unique hole \(\text{par}_{R_i}(g)\) in \(R_i\) such that \(g\) lies in \(R_{\text{par}_{R_i}(g)}\).out, which we refer to as the parent of \(g\) in \(R_i\). (Note that the ancestry of holes goes in the opposite direction of the ancestry of regions in the \(\vec{r}\)-decomposition.) In a distance query we only deal with a series of regions \(R_0 = \{u\}, R_1, \ldots, R_m = G\). The holes of these regions form a hierarchy, rooted at \(\{u\}\), which we view as a degenerate hole. For notational simplicity we use “\(g\)” to refer to the set of vertices on hole \(g\).

**Lemma B.1.** \(\text{(See } [4, \S 4.3.2]\text{)}\) There is an \(O(n)\)-space data structure that, given \(u, v\) can report in \(O(m)\) time the regions \(R_0 = \{u\}, R_1, \ldots, R_{i+1}\) and holes \(h_0, h_1, \ldots, h_i\) such that \(v \in R_{i, \text{out}}, v \notin R_i, \) and \(v \in R_{i+1}\).

The method of \[4\] simply involves doing a least common ancestor query of \(\{u\}\) and \(\{v\}\) in the “full” binary decomposition returned by the \[29\] algorithm (from which our \(\vec{r}\)-decomposition is a coarsening) in order to retrieve \(h_1\). The holes \(h_{i-1}, \ldots, h_0\) can then be found by following parent pointers in \(O(m)\) time.

### B.1 Data Structures

The following modifications are made to parts (A)–(E) of the data structure. In all cases the space usage is unchanged, asymptotically.\[14\]

\[13\]The Klein et al. \[29\] algorithm rotates between finding separators w.r.t. number of vertices, number of boundary vertices, and number of holes, but this is not relevant to the present discussion.

\[14\]Given a \texttt{dist}(\(u, v\)) query, we can map it to \texttt{Dist}(\(u', v', R_0\)), where \(u'\) and \(v'\) are any of the copies of \(u\) and \(v\), respectively, and \(R_0 = \{u'\}\).
Figure 12: (a) A subgraph with two holes. (b) We put a vertex in each hole and triangulate the hole. (The triangulation of the exterior hole is not drawn, for clarity.) A simple cycle separator (blue curve) is found in this graph. (c) The cycle is mapped to a possibly non-simple cycle in the original graph that avoids hole-vertices. We cut along non-simple parts of the cycle, duplicating the vertices and their adjacent edges on the cycle. (d) The graph remaining after removing the subgraph enclosed by the cycle from (c).
(A) (MSSP Structures) For each \( i \in [0, m - 1] \), each \( R_i \in \mathcal{R}_i \) with parent \( R_{i+1} \) and each hole \( h_i \) of \( R_i \), we build a MSSP structure for \( R_{i}^{h_i,\text{out}} \) that answers distance queries and LCA queries w.r.t. \( R_{i}^{h_i,\text{out}} \) for vertices in \( R_{i}^{h_i,\text{out}} \cap R_{i+1} \).

(B) (Voronoi Diagrams) For each \( i \in [0, m - 1] \), each \( R_i \in \mathcal{R}_i \) with parent \( R_{i+1} \in \mathcal{R}_{i+1} \), each hole \( h_{i+1} \) of \( R_{i+1} \) with parent \( h_i = \text{par}_{R_i}(h_{i+1}) \), and each \( q \in h_i \), we store the dual representation of Voronoi diagram \( \text{VD}^*_\text{out}(q, R_{i+1}, h_{i+1}) \) defined to be \( \text{VD}^*[R_{i+1}^{h_{i+1},\text{out}}, h_{i+1}, \omega] \) with \( \omega(s) = \text{dist}_G(q, s) \).

(C) (More Voronoi Diagrams) For each \( i \in [1, m - 1] \), each \( R_i \in \mathcal{R}_i \), each hole \( h_i \) of \( R_i \), and each \( q \in h_i \), we store \( \text{VD}^*_\text{out}(q, R_i, h_i) \), which is \( \text{VD}^*[R_{i}^{h_i,\text{out}}, h_i, \omega] \) with \( \omega(s) = \text{dist}_G(q, s) \).

(D) (Chord Trees; Piece Trees) For each \( i \in [1, m - 1] \), each \( R_i \in \mathcal{R}_i \), each hole \( h_i \) of \( R_i \), and source \( q \in h_i \), we store a chord tree \( T_{q}^{R_i, h_i} \) obtained by restricting the SSSP tree with source \( q \) to \( h_i \). An edge in \( T_{q}^{R_i, h_i} \) is designated a chord if the corresponding path lies in \( R_{i}^{h_i,\text{out}} \) and is internally vertex disjoint from \( h_i \). \( \mathcal{O}^{R_i, h_i, \omega}, \mathcal{P}^{R_i, h_i}, T_{q}^{R_i, h_i} \) are defined analogously, and data structures are built to answer \text{MaximalChord} \text{ and } \text{AdjacentPiece} with respect to \( q, R_i, h_i \).

(E) (Site Tables; Side Tables) Fix an \( i \) and a Voronoi diagram \( \text{VD}^*_\text{out} = \text{VD}^*_\text{out}(u', R_i, h_i) \) from part (B) or (C). Let \( f' \) be any node in the centroid decomposition of \( \text{VD}^*_\text{out} \) with \( y_j, s_j \) defined as usual, \( j \in \{0,1,2\} \). Let \( R_{i'} \in \mathcal{R}_{i'} \) be an ancestor of \( R_i \), \( i' > i \), and \( h_{i'} \) be a hole of \( R_{i'} \) lying in \( R_{i}^{h_i,\text{out}} \). We store the first and last vertices \( q, x \) on the shortest \( s_jy_j \) path that lie on \( h_{i'} \), as well as \( \text{dist}_G(u', x) \).

We also store whether \( R_{i'}^{h_i,\text{out}} \) lies to the left or right of the site-centroid-site chord \( s_jy_jy_j^{-1}s_j^{-1} \), or \text{Null} if the relationship cannot be determined.

B.2 Query
At the first call to \( \text{Dist}(u, v, R_0) \) we apply Lemma \ref{lemma:dist以外} to generate the regions \( R_1, \ldots, R_t \) and holes \( h_1, \ldots, h_t \) that will be accessed in all recursive calls, in \( O(m) \) time.

The shortest \( u-v \) path in \( G \) will eventually cross \( h_1, \ldots, h_t \). The vertex \( u_t \) is now defined to be the last vertex in \( h_t \) on the shortest \( u-v \) path. Given \( u_t \), we find \( u_{t+1} \) by solving a point location problem in \( \text{VD}^*_\text{out}(u_t, R_{t+1}, h_{t+1}) \). The \text{Navigation} routine focuses on the subgraph \( R_{i}^{h_i,\text{out}} \) rather than \( R_{i}^{\text{out}} \). The general problem is no different than the single hole case, except that there may be \( O(1) \) holes of \( R_{t+1} \) lying in \( R_{i}^{h_i,\text{out}} \), which does not cause further complications.

C Construction
As in \cite{9}, we use dense distance graphs as a tool to build our oracle. To simplify the description, we still assume that \( \partial R \) lies on a single simple cycle for every region \( R \) in the \( \mathcal{R} \)-division. Generalizing to multiple holes and nonsimple cycles is straightforward.

The \textit{dense distance graph} of a region \( R \) (denoted by \( \text{DDG}[R] \)) is a complete directed graph on the vertices of \( \partial R \), in which the length of \((u,v)\) is \( \text{dist}_R(u,v) \). We say that this kind of DDGs are \textit{internal} and, similarly, define the \textit{external} DDG of a region \( R \) (denoted by \( \text{DDG}[R^\text{out}] \)) as a complete directed graph on \( \partial R \), in which the length of \((u,v)\) is \( \text{dist}_{R^\text{out}}(u,v) \).

The FR-Dijkstra algorithm \cite{18} is an efficient implementation of Dijkstra’s algorithm \cite{14} on DDGs. In particular, it simulates the behavior of the heap in Dijkstra’s algorithm without explicitly scanning every edge in the DDGs. In fact, the FR-Dijkstra algorithm can run on a union of DDGs \cite{18}. Moreover, it is shown in \cite{6} that it can also run compatibly with a traditional Dijkstra algorithm. Suppose we have a graph \( H \) that consists of a subgraph of \( G \) on \( n_0 \) vertices, and \( k \) DDGs on \( n_1, n_2, \ldots, n_k \) vertices. The FR-Dijkstra algorithm can be implemented on \( H \) in \( O(N) \) time, where \( N = \sum_i n_i \).

Before the construction of DDGs and our oracle, we first prepare Klein’s MSSP structures (part (F) below). Note that MSSP structures in part (F) are only used in the constructions of DDGs and part (E). They are not stored in our oracle and unrelated to the MSSP structures from part (A).
(F) (More MSSP Structures) For each \( i \in [0, m - 1] \), each \( R_i \in \mathcal{R} \) with parent \( R_{i+1} \in \mathcal{R}_{i+1} \), we build two MSSP structures for \( R_i^{out} \cap R_{i+1} \) with sources on \( \partial R_i \) and \( \partial R_{i+1} \), respectively, and an MSSP structure for \( R_i \) with sources on \( \partial R_i \).

All these MSSP structures are constructed using Klein’s MSSP algorithm \[28\] or the one in Appendix A (with \( \kappa = \log n \)) in \( O(\sum_i \frac{n}{r_i} r_i + 1) = O(mn^{1+1/m}) \) time.

We then compute, for each region \( R_i \) in the \( \mathcal{R} \)-division, the internal DDG, the external DDG, and the DDG of \( R_i^{out} \cap R_{i+1} \) (denoted by DDG\([R_i^{out} \cap R_{i+1}]\)) defined as the complete graph with vertices \( \partial R_i \) and \( \partial R_{i+1} \) and edge weights the distances in \( R_i^{out} \cap R_{i+1} \). The internal DDG and DDG\([R_i^{out} \cap R_{i+1}]\) for each region \( R_i \) can be computed using MSSP structures in part (F) in \( O(r_i) \) and \( O(r_{i+1}) \) time respectively, so it takes \( \tilde{O}(\sum_i \frac{n}{r_i}(r_i + r_{i+1})) = \tilde{O}(mn^{1+1/m}) \) time over all regions. To compute the external DDGs, we consider a top-down process on the \( \mathcal{R} \)-division. The external DDG for \( R_i \) can be computed by running the FR-Dijkstra algorithm sourced from \( \partial R_i \) on the union of DDG\([R_i^{out} \cap R_{i+1}]\) and DDG\([R_i^{out} \cap R_{i+1}]\). The size of the union is \( O(\sqrt{r_i}), \) so computing DDG\([R_i^{out}]\) takes \( \tilde{O}(\sqrt{r_i} \sum \sqrt{r_{i+1}}) \) time, and the construction time over all external DDGs is \( \tilde{O}(\sum_i \frac{n}{r_i} \sqrt{r_i} \sum \sqrt{r_{i+1}}) = \tilde{O}(mn^{1+1/(2m)}) \). The total construct time for all DDGs is \( \tilde{O}(mn^{1+1/m}) \).

With dense distance graphs, all components in the oracle can be constructed as follows.

(A) MSSP Structures

Recall that our MSSP structure for \( R_i^{out} \) with sites \( \partial R_i \) is obtained by contracting subpaths in \( R_i^{out} \) of the MSSP trees into single edges. In order to build the MSSP structure using dynamic trees, it suffices to compute the contracted shortest path tree for every source on \( \partial R_i \) and then compare the differences between the trees of two adjacent sources on \( \partial R_i \).

For a single source on \( \partial R_i \), the contracted shortest path tree can be computed with FR-Dijkstra algorithm on the union of subgraph \( R_i^{out} \cap R_{i+1} \) and DDG\([R_i^{out} \cap R_{i+1}]\) in time \( \tilde{O}(r_i) \). Thus, the time for constructing and comparing the shortest path trees is \( \tilde{O}(r_{i+1} \sqrt{r_i}) \). After that, an MSSP structure for \( R_i^{out} \) can be built in time \( \tilde{O}((r_{i+1} + \sqrt{r_i} r_{i+1} \kappa n^{1/k}) \). The total time to construct all MSSP structures is \( \tilde{O}(\sum_i \frac{n}{r_i}(r_i + r_{i+1} \kappa n^{1/k})) = \tilde{O}(n^{3/2+1/m} m + n^{1+1/k+1/m} m k) \).

Remark 2. Notice that in our MSSP structures for \( R_i^{out} \), a contracted subpath should be “strictly” inside \( R_i^{out} \), which contains no vertices belonging to \( R_i^{out} \cap R_{i+1} \) except its endpoints. However, the underlying shortest paths represented by edges in DDG\([R_i^{out}]\) may not satisfy this condition. To fix this problem, we add small perturbation to all edge weights in DDGs. Note that this will not break the Monge property of a DDG’s adjacency matrix, on which the FR-Dijkstra algorithm relies. In fact, all different shortest paths from the source to \( v \) in the union represent the same unique underlying shortest path in \( R_i^{out} \), and we choose the one passing as many as possible vertices in the union, on which each edge of the DDG is “strictly” inside \( R_i^{out} \). This mechanism will also be used below.

(B/C) Voronoi Diagrams

The additive weights of all Voronoi diagrams can be computed by a FR-Dijkstra algorithm running on a union of proper DDGs. Specific to VIIF\([u_i, R_i]\) in (B), additive weights are given by considering the union of DDG\([R_i]\), DDG\([R_i^{out} \cap R_{i+1}]\), and DDG\([R_i^{out} \cap R_{i+1}]\) in time \( \tilde{O}(\sqrt{r_i}) \). For VIIF\([u_i, R_i]\) in (C), we focus on the union of DDG\([R_i]\), DDG\([R_i^{out}]\) and additive weights can be computed in time \( \tilde{O}(\sqrt{r_i}) \). The overall time to compute additive weights is \( \tilde{O}(\sum_i \frac{r_i}{\sqrt{r_i}} \sqrt{r_i} \sqrt{r_{i+1}}) = \tilde{O}(mn^{1+1/2m}) \).

An efficient algorithm to compute the dual representation of a Voronoi diagram is presented in [9], by considering the complete recursive decomposition of \( G \). Note that the complete recursive decomposition of \( G \) is a binary decomposition tree, and the \( \mathcal{R} \)-division is a coarse version of it. It can be obtained in \( \tilde{O}(n) \) time [29].

Lemma C.1. (Cf. Charalampopoulos et al. [9], Theorem 12) Given a complete recursive decomposition of \( G \), in which every region has been preprocessed for the FR-Dijkstra algorithm as in [18], the dual representation of Voronoi diagrams on the complement of a specific region \( R \) with sites \( \partial R \) and arbitrary input additive weights can be computed in time \( \tilde{O}(\sqrt{|G|} \cdot |\partial R|) \).
By Lemma [C.1], the total construction time for the dual representations is

\[ \tilde{O} \left( \sum_i \frac{n}{R_i} \sqrt{R_i} \sqrt{n \sqrt{R_{i+1}}} + \sum_i \frac{n}{R_i} \sqrt{R_i} \sqrt{n \sqrt{R_i}} \right) = \tilde{O} \left( n^{3/2+1/(4m)} \right), \]

which is also the construction time for parts (B) and (C).

(D) Chord Trees and Piece Trees

Recall that the chord tree \( T_q^{R_i} \) is obtained from the shortest path tree in \( G \) sourced from \( q \in \partial R_i \) by contracting all paths between vertices in \( \partial R_i \) into single edges. Thus, it can be computed by running FR-Dijkstra on the union of \( DDG[R_i] \) and \( DDG[R_{i+1}] \) in \( \tilde{O}(\sqrt{R_i}) \) time.

Regarding the construction of piece tree \( T_q^{R_i} \), we first extract all the chords on \( T_q^{R_i} \) in \( R_{i+1}^{out} \), i.e. the chord set \( C_q^{R_i} \). We treat each chord in \( C_q^{R_i} \) as an undirected edge and consider the undirected planar graph \( Q \) which is the union of \( C_q^{R_i} \) and \( \partial R \). Observe that each piece in \( P_q^{R_i} \) relates to a face of \( Q \).

The piece tree \( T_q^{R_i} \), can be built straightforwardly in time \( \tilde{O}(\sqrt{R_i}) \). With the graph \( Q \) and the piece tree \( T_q^{R_i} \), the data structure supporting \textbf{MaximalChord} and \textbf{AdjacentPiece} in Lemma [4.2] can also be constructed in time \( \tilde{O}(\sqrt{R_i}) \) for the given \( q, R_i \).

The total time for the whole part (D) is \( \tilde{O}(\sum_i \frac{n}{R_i} \sqrt{R_i} \sqrt{n \sqrt{R_i}}) = \tilde{O}(nm) \).

(E) Site Tables and Side Tables

We focus on the site table and side table for a specific \( VD^*_{out}(u, R_i) \), and do some preparations.

- Observe that the union of

\[ DDG[R_{i+1}^{out} \cap R_i], DDG[R_{i+1}^{out} \cap R_{i+2}], \ldots, DDG[R_{m-1}^{out}] \]

contains exactly all boundary vertices in \( R_{i+1}^{out} \) of ancestors \( R_i, R_{i+1}, \ldots, R_{m-1} \). We use \( H \) to denote this union with an artificial super-source \( u' \) connected to each site \( s \in \partial R_i \) with weight \( \omega(s) \), and construct the shortest path tree \( T_H \) in \( H \) sourced from the super-source \( u' \) by FR-Dijkstra algorithm, which costs \( \tilde{O}(\sqrt{m}) \) time.

Remember that the site table stores the first and last vertices of each site-centroid \( s-y \) path on the boundary of each ancestor \( R_{i'} \) (\( i' \geq i \)). We first find the last vertex \( x \) on the \( s-y \) path belonging to \( H \). Assume that \( y \in R_{i+1} \) but \( y \notin R_i \), where \( R_k, R_{k+1} \) are ancestors of \( R_i \). We can observe that \( x \) is the vertex in \( \partial R_k \cup \partial R_{k+1} \) with the minimal \( \text{dist}_{H}(u', x) + \text{dist}_{R_{i+1}^{out} \cup R_{k+1}}(x, v) \) (breaking ties in favor of lower \( \text{dist}_{H}(u', x) \)). The former is given by \( T_H \) and the latter can be found by querying MSSP structures in (F) for \( R_{i+1}^{out} \cap R_{k+1} \). The calculation of \( x \) needs time \( \tilde{O}(|\partial R_{k+1}|) = \tilde{O}(\sqrt{m}) \). Observe that the \( u'-x \) path on \( T_H \) includes all boundary vertices of upper regions on the \( s-y \) path. By retrieving the \( u'-x \) path on \( T_H \) in \( \tilde{O}(\sqrt{m}) \) time, we can get the required information for the site table. The construction time of a site table for \( VD^*_{out}(u, R_i) \) is \( \tilde{O}(\sqrt{m} \sqrt{R_i}) \).

In the side table, we will store the relationship (left/right/\textbf{Null}) between each site-centroid-site chord \( C = s_jy_jy_{j-1}s_{j-1} \) (using the notations in Figure [F2]) and each ancestor \( R_{i'}^{out} \) (\( i' \geq i \)). With the technique used in the construction of site tables, we can extract all vertices of \( C \) on each \( \partial R_{i'} \) from \( T_{H_i} \), and then determine the relationship between \( C \) and each \( R_{i'}^{out} \) with boundary vertices on \( C \). For each \( R_{i'}^{out} \) that \( C \) contains no vertices on \( \partial R_{i'} \), we pick an arbitrary vertex \( z \) on \( \partial R_{i'} \). We can retrieve from \( T_{H_i} \) the \( u'-z \) path and find the site \( s_z \) s.t. \( z \in \text{Vor}(s_z) \). This can be done in \( \tilde{O}(\sqrt{m}) \) time. With \( T_{H_i} \) and MSSP structures in part (F), we can determine the pairwise relationships among \( s_jy_j, s_{j-1}y_{j-1} \) and \( s_zz \) shortest paths and know whether \( z \) lies to the left or right of \( C \), which immediately shows the relationship between \( C \) and \( R_{i'}^{out} \). The construction time for a side table of \( VD^*_{out}(u, R_i) \) is \( \tilde{O}(\sqrt{m} \sqrt{R_i}) \).

The total time for building all site tables and side tables is \( \tilde{O}(\sum_i \frac{n}{R_i} \sqrt{R_i} \sqrt{R_{i+1}m \sqrt{n}}) = \tilde{O}(n^{3/2+1/(2m)}m^2) \).
The overall construction time is \( \tilde{O}(n^{3/2+1/m} + n^{1+1/m+1/\kappa}) \) since \( m \) and \( \kappa \) should be functions of \( n \) that are \( O(\log n) \).