

BIROn - Birkbeck Institutional Research Online

Charalampopoulos, Panagiotis and Gawrychowski, P. and Long, Y. and Mozes, S. and Pettie, S. and Weimann, O. and Wulff-Nilsen, C. (2023) Almost optimal exact distance oracles for planar graphs. *Journal of the ACM* 70 (2), pp. 1-50. ISSN 0004-5411.

Downloaded from: <https://eprints.bbk.ac.uk/id/eprint/50932/>

Usage Guidelines:

Please refer to usage guidelines at <https://eprints.bbk.ac.uk/policies.html>
contact lib-eprints@bbk.ac.uk.

or alternatively

Almost Optimal Exact Distance Oracles for Planar Graphs

PANAGIOTIS CHARALAMPOPOULOS, Birkbeck, University of London, UK

PAWEŁ GAWRYCHOWSKI, University of Wrocław, Poland

YAOWEI LONG, Tsinghua University, China

SHAY MOZES, Reichmann University, Israel

SETH PETTIE, University of Michigan, USA

OREN WEIMANN, University of Haifa, Israel

CHRISTIAN WULFF-NILSEN, University of Copenhagen, Denmark

We consider the problem of preprocessing a weighted directed planar graph in order to quickly answer exact distance queries. The main tension in this problem is between *space* S and *query time* Q , and since the mid-1990s all results had polynomial time-space tradeoffs, e.g., $Q = \tilde{\Theta}(n/\sqrt{S})$ or $Q = \tilde{\Theta}(n^{5/2}/S^{3/2})$.

In this paper we show that there is no polynomial tradeoff between time and space and that it is possible to *simultaneously* achieve almost optimal space $n^{1+o(1)}$ and almost optimal query time $n^{o(1)}$. More precisely, we achieve the following space-time tradeoffs:

$n^{1+o(1)}$ space and $\log^{2+o(1)} n$ query time,
 $n \log^{2+o(1)} n$ space and $n^{o(1)}$ query time,
 $n^{4/3+o(1)}$ space and $\log^{1+o(1)} n$ query time.

We reduce a distance query to a variety of *point location* problems in additively weighted *Voronoi diagrams*, and develop new algorithms for the point location problem itself using several partially persistent dynamic tree data structures.

CCS Concepts: • **Theory of computation** → **Data structures design and analysis; Shortest paths.**

Additional Key Words and Phrases: planar graphs, Voronoi diagrams, distance oracles, persistent data structures

ACM Reference Format:

Panagiotis Charalampopoulos, Paweł Gawrychowski, Yaowei Long, Shay Mozes, Seth Pettie, Oren Weimann, and Christian Wulff-Nilsen. 2023. Almost Optimal Exact Distance Oracles for Planar Graphs. *J. ACM* 70, 2, Article 12 (March 2023), 51 pages. <https://doi.org/https://doi.org/10.1145/3580474>

This paper is derived from extended abstracts presented at SODA'18 [32], STOC'19 [14], and SODA'21 [50]. This work was supported by NSF grants CCF-1637546 and CCF-1815316, a grant from IIS, Tsinghua University, Israel Science Foundation grants 592/17 and 810/21, and the Starting Grant 7027-00050B from the Independent Research Fund Denmark under the Sapere Aude research career programme.

Authors' addresses: Panagiotis Charalampopoulos, p.charalampopoulos@bbk.ac.uk, Birkbeck, University of London, Department of Computer Science and Information Systems, Malet Street, London, UK, WC1E 7HX; Paweł Gawrychowski, gawry@cs.uni.wroc.pl, University of Wrocław, Instytut Informatyki, Uniwersytetu Wrocławskiego, ul. Joliot-Curie 15, Wrocław, Poland, 50-383; Yaowei Long, yaowei@umich.edu, Tsinghua University, Institute for Interdisciplinary Information Sciences, Beijing, China, 100084; Shay Mozes, smozes@idc.ac.il, Reichmann University, Efi Arazi School of Computer Science, 167, Herzliya, Israel, 4610101; Seth Pettie, pettie@umich.edu, University of Michigan, Computer Science and Engineering, 2260 Hayward St., Ann Arbor, Michigan, USA, 48109; Oren Weimann, oren@cs.haifa.ac.il, University of Haifa, Department of Computer Science, Ha-Namal St. 67, Haifa, Haifa, Israel, 3303221; Christian Wulff-Nilsen, koolooz@di.ku.dk, University of Copenhagen, BARC, Department of Computer Science, Universitetsparken 1, Copenhagen, Denmark, DK-2100.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Association for Computing Machinery.

0004-5411/2023/3-ART12 \$15.00

<https://doi.org/https://doi.org/10.1145/3580474>

1 INTRODUCTION

A *distance oracle* is a data structure that gives oracle access to the pairwise distance function $\text{dist}_G(\cdot, \cdot)$ with respect to some graph G . There are two trivial solutions to this problem: store dist_G explicitly in $\Theta(n^2)$ space, or store the graph itself and answer queries in $\Omega(m)$ time. The goal is to achieve a time-space tradeoff that approaches the constant query time of the first trivial scheme and the linear space of the second.

On *general* graphs G it seems that incorporating *approximation* into the distance estimates is necessary to get an attractive space-time tradeoff. There are approximate distance oracles for undirected graphs [18, 64] that trade space $O(n^{1+1/k})$ against multiplicative approximation $2k - 1$, with optimal $O(1)$ query time. Others [1, 58] pit space against a mix of multiplicative and additive approximation, or in sparse graphs [2, 61], space against query time. Refer to Sommer [60] for a survey on distance oracles.

When G is known to come from a *structured* class of graphs we can aspire to find *exact* distance oracles with attractive space-time tradeoffs. In this paper we develop new distance oracles for weighted, directed *planar* graphs. This problem has been studied for 25 years in both the exact [3, 9, 14, 19, 20, 24, 28, 32, 45, 54, 56, 67] and the approximate [13, 34, 42–44, 63, 68] settings. Our oracles are the first exact oracles to simultaneously achieve almost optimal space $n^{1+o(1)}$ and almost optimal query time $n^{o(1)}$. Theorem 1.1 provides a fine-grained tradeoff between space and query time.

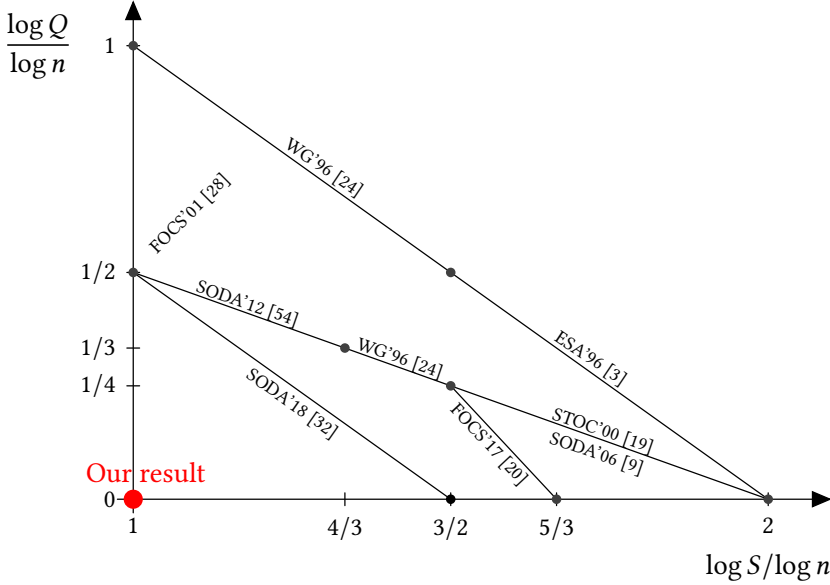


Fig. 1. Tradeoff of the space (S) vs. the query time (Q) for exact distance oracles in planar graphs on a doubly logarithmic scale, hiding subpolynomial factors. The previous tradeoffs are indicated by solid black lines and points, while our new tradeoff is indicated by the red point.

1.1 History of Planar Distance Oracles

The planar distance oracle problem was introduced in 1996 by Arikati et al. [3] and Djidjev [24]. The main technical tool used in the early planar distance oracles [3, 19, 24] is Lipton and Tarjan's planar

separator theorem [49], and its refinements by Miller [52] and Frederickson [29]. Let the query time and the space of an oracle be denoted by Q and S , respectively. The early oracles achieved a space-query tradeoff of $Q = \tilde{O}(n/\sqrt{S})$ for $S \in [n^{4/3}, n^2]$, but a weaker tradeoff of $Q = O(n^2/S)$ for $S \in [n, n^{4/3}]$.

In a very influential paper Fakcharoenphol and Rao [28] introduced *Monge* matrices to planar graph algorithms, and devised a distance oracle with $\tilde{O}(n)$ space and $\tilde{O}(\sqrt{n})$ query time, i.e., they added an additional point to the general $Q = \tilde{O}(n/\sqrt{S})$ tradeoff. Eventually, Mozes and Sommer [54] extended this tradeoff to nearly the full range $[n \log \log n, n^2]$, using [28] and ideas from Klein's [44] multiple source shortest path (MSSP) data structure.

The work of [30, 54, 56, 67] focussed on achieving *optimal* time or space, at the expense of the other measure. Wulff-Nilsen's [67] distance oracle occupies subquadratic space $O(n^2 \log^4 \log n / \log n)$ and answers queries in optimal $O(1)$ time, whereas Nussbaum [56] and Mozes and Sommer's [54] distance oracle occupies optimal $O(n)$ space and answers distance queries in $O(n^{1/2+\epsilon})$ time, for any $\epsilon > 0$. On undirected, unweighted planar graphs, the recent distance oracle of Fredslund-Hansen, Mozes, and Wulff-Nilsen [30] occupies $O(n^{5/3+\epsilon})$ space and answers queries in $O(\log(1/\epsilon))$ time for any $\epsilon > 0$.

In 2017 Cabello [10] introduced a new idea, *additively weighted planar Voronoi diagrams*,¹ and used them to solve problems concerning planar metrics (diameter, sum-of-distances) in strongly subquadratic time. Inspired by this idea, Cohen-Addad, Dahlgaard, and Wulff-Nilsen [20] realized that Voronoi diagrams can be used to obtain the first exact distance oracle for planar graphs with subquadratic space and polylogarithmic query time. The Voronoi diagram based oracle in their breakthrough paper has a space-time tradeoff of $Q = \tilde{O}(n^{5/2}/S^{3/2})$ for $S \in [n^{3/2}, n^{5/3}]$.

All of the distance oracles cited above report *exact* distances. Thorup [63] proved that on non-negatively weighted planar graphs, $(1 + \epsilon)$ -approximate distances can be reported in $O(\epsilon^{-1} + \log \log n)$ time by an oracle of space $O(n\epsilon^{-1} \log^2 n)$. Refer to [13, 34, 42–44, 48, 68] for other space-time-approximation tradeoffs on *undirected* planar graphs and to [48] for an improved tradeoff on *directed* planar graphs. See Table 1.

1.2 New Results

In this paper we show that *there is no polynomial tradeoff* between space and query time, and that near-optimality in both measures can be achieved *simultaneously*: with $n^{1+o(1)}$ space *exact* distance queries can be answered in $n^{o(1)}$ query time. Our main distance oracle (Theorem 1.1) does have a space-time tradeoff, the extremes of which let us achieve either $\tilde{O}(n)$ space or $\tilde{O}(1)$ query time but not both.

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 $O(m\kappa n^{1+1/m+1/\kappa})$ can be constructed in $\tilde{O}(n^{3/2+1/m} + n^{1+1/m+1/\kappa})$ time to answer 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.

At a high level, Theorem 1.1 reduces a distance query $\text{dist}_G(u, v)$ to a series of *point location* problems in additively weighted planar Voronoi diagrams. We compute an m -level \vec{r} -division [29, 46] where regions on level i have $O(n^{i/m})$ vertices and $O(\sqrt{n^{i/m}})$ boundary vertices (vertices shared by other regions). In the course of answering a distance query $\text{dist}_G(u, v)$ we find (u_1, u_2, \dots) , where u_i

¹Voronoi diagrams have also been used for facility location problems in planar graphs [51].

EXACT ORACLES		SPACE	QUERY TIME
Arikati, Chen, Chew Das, Smid & Zariwagis	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	$S \in [n^{4/3}, n^2]$	$O\left(\frac{n}{\sqrt{S}} \log\left(\frac{n}{\sqrt{S}}\right)\right)$
Fakcharoenphol & Rao	2006	$O(n \log n)$	$O(\sqrt{n} \log^2 n)$
Wulff-Nilsen	2010	$O(n^2 \frac{\log^4 \log n}{\log n})$	$O(1)$
Nussbaum	2011	$O(n)$	$O(n^{1/2+\epsilon})$
		$S \in [n^{4/3}, n^2]$	$O\left(\frac{n}{\sqrt{S}}\right)$
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	$S \in [n^{3/2}, n^{5/3}]$	$O\left(\frac{n^{5/2}}{S^{3/2}} \log n\right)$
Gawrychowski, Mozes Weimann & Wulff-Nilsen	2018	$O(n^{3/2})$	$O(\log n)$
		$S \in [n, n^{3/2}]$	$O\left(\frac{n^{3/2}}{S} \log^2 n\right)$
Fredslund-Hansen, Mozes & Wulff-Nilsen	2020	$O(n^{5/3+\epsilon})$	$\log(1/\epsilon)$ (Undir., Unweight.)
new		$n^{1+o(1)}$	$\log^{2+o(1)} n$ Theorem 1.1
		$n \log^{2+o(1)} n$	$n^{o(1)}$ Theorem 1.1
		$O(n^{4/3} \log^{1/3} n)$	$O(\log^2 n)$ Theorem 4.1
		$n^{4/3+o(1)}$	$\log^{1+o(1)} n$ Theorem 4.1

$(1 + \epsilon)$ -APPROX. ORACLES		SPACE	QUERY TIME
Thorup	2001	$O(n\epsilon^{-1} \log^2 n)$	$O(\log \log n + \epsilon^{-1})$
		$O(n\epsilon^{-1} \log n)$	$O(\epsilon^{-1})$ (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	$\overline{O}(n \log n)$	$\overline{O}(\epsilon^{-1})$ (Undir.)
		$\overline{O}(n)$	$\overline{O}(\epsilon^{-1})$ (Undir., Unweight.)
Gu & Xu	2015	$O(n \log n(\epsilon^{-1} \log n + 2^{O(1/\epsilon)}))$	$O(1)$ (Undir.)
Wulff-Nilsen	2016	$\overline{O}(n\epsilon^{-2})$	$\overline{O}(\epsilon^{-2})$ (Undir.)
Chan & Skrepetos	2019	$O(n \log n(\epsilon^{-1} \log n + \epsilon^{-4-\delta}))$	$\overline{O}(1)$ (Undir.)
Le & Wulff-Nilsen	2021	$O(n\epsilon^{-2})$	$O(\epsilon^{-2})$ (Undir.)
Le & Wulff-Nilsen	2021	$o(n\epsilon^{-1} \log n)$	$O(\log \log n + \epsilon^{-5.01})$

Table 1. Space-query time tradeoffs for exact and approximate planar distance oracles. All exact distance oracles apply to weighted, directed graphs without negative cycles. Approximate distance oracles apply to non-negatively weighted graphs; some are restricted to undirected and/or unweighted graphs. \overline{O} hides $\log(\epsilon^{-1} \log n)$ factors. The bounds for approximate distance oracles in directed planar graphs assume that the ratio between the largest and smallest edge weights is polynomial in n .

is the *last* vertex of the shortest u -to- v path lying on the boundary of a level- i region that contains u .

Theorem 3.2 proves that the point location problem itself is reducible to $O(\log n)$ distance-type queries² via a kind of binary search. We employ two strategies for answering these distance-type queries. The first is to store many MSSP structures for various subgraphs. This is time-efficient but requires space linear in the size of these subgraphs. The second is to use recursion. Specifically, given (u_1, \dots, u_i) , we can narrow the number of possibilities for u_{i+1} down to *two* candidates s_1, s_2 in $\tilde{O}(1)$ time via point location queries that are solved *without* recursion. We determine which of these two sites actually is u_{i+1} with two recursive calls to compute $\text{dist}(s_1, v)$ and $\text{dist}(s_2, v)$. This branching process leads to a query time $\tilde{O}(2^m)$ that depends exponentially on m , whereas the space of the data structure is about $\tilde{O}(n^{1+1/m})$. Thus, by setting m appropriately we can achieve $\tilde{O}(1)$ query time and $n^{1+o(1)}$ space or $n^{o(1)}$ query time and $\tilde{O}(n)$ space.

Using existing MSSP structures [44] the query time would be $O(2^m \log^3 n)$. We develop a new MSSP data structure based on Euler-tour trees [38] and partially persistent arrays [21] that may be of independent interest. It uses $O(\kappa n^{1+1/\kappa})$ space and answers distance-type queries in $O(\kappa \log \log n)$ time, for any parameter $\kappa \geq 1$. The first tradeoff of Theorem 1.1 (minimizing query time) is obtained by setting both κ, m to be $\omega(1)$ and $o(\log \log n)$.

In Theorem 4.1 we describe a simpler distance oracle that achieves different space-time tradeoffs, namely $\tilde{O}(n^{4/3})$ space and $O(\log^2 n)$ query time, or $n^{4/3+o(1)}$ space and $\log^{1+o(1)} n$ query time.

Finally, we complement our almost optimal distance oracle with an efficient preprocessing algorithm that runs in $n^{3/2+o(1)}$ time. In particular, we show an efficient algorithm for computing Voronoi diagrams in planar graphs, which we believe is of independent interest.

Provenance of the Paper. This paper is derived from three extended abstracts. The first [31], which appeared in SODA 2018, characterized the tree structure of the dual representation of Voronoi diagrams, and developed the point location mechanism described in Section 3. The distance oracle of [31] achieved a tradeoff of $Q = \tilde{O}(n^{3/2}/S)$ for $S \in [n, n^{3/2}]$, which is completely subsumed by Theorem 1.1. Therefore, it is not described in this paper. The second paper [14], which appeared in STOC 2019, observed that the same point location mechanism can be used in an *external* Voronoi diagram, i.e., the Voronoi diagram for the complement of a region in an r -division. Furthermore, it developed the recursive query structure using m -level \vec{r} -divisions. A query at level i of the recursion reduces to $O(\log n)$ queries at level $i + 1$. Thus, the query time in [14] is proportional to $(\log n)^m$ rather than to 2^m as in Theorem 1.1. The resulting distance oracle of [14] achieved $n^{1+o(1)}$ space and $n^{o(1)}$ query-time. The third paper [50], which appeared in SODA 2021, modified and extended the point location mechanism. It showed that, by using appropriate persistent data structures and further exploiting planarity, much of the point location work can be done efficiently without recursion, and that only *two* recursive calls at a higher level suffice. The MSSP data structure based on Euler-tour trees was also introduced in [50].

1.3 Organization

In Section 2 we review background on planar embeddings, planar separators, and multiple-source shortest paths (MSSP). In Section 3 we review additively weighted Voronoi diagrams, and prove that the point location problem is reducible to $O(\log n)$ distance-type queries. Section 4 describes a simple distance oracle with space $\Omega(n^{4/3})$ and faster query times than those of Theorem 1.1.

²Specifically, deciding whether the shortest s -to- v path is a prefix of the shortest s -to- x path for some vertex x , or whether it branches to the left or to the right of it.

Section 5 introduces the main distance oracle of Theorem 1.1 and the high-level query algorithm. The high-level algorithm relies on specialized point location routines, which are introduced in Section 6. Section 7 analyzes the space and query time of the distance oracle whereas its construction time is addressed in Section 8. Section 9 explains how to remove a simplifying assumption made throughout the paper, that each region is bounded by a single hole. We conclude with some remarks and open problems in Section 10.

The new MSSP implementation is described in Appendix A.

2 PRELIMINARIES

2.1 The Graph and its Embedding

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

- All the edge-weights are non-negative, i.e., $\ell : E \rightarrow \mathbb{R}_{\geq 0}$. This can be ensured in $O(n \frac{\log^2 n}{\log \log n})$ time [39, 55]. Furthermore, in $O(n)$ time, via randomized or deterministic perturbation [26], we can assume that there are no zero weight edges, and that shortest paths are *unique* in *any* subgraph of G . Each original distance can be recovered from the corresponding distance in the transformed graph in constant time.³
- The graph is strongly-connected and triangulated, i.e., each face is bounded by a 3-cycle. We can ensure this by introducing artificial edges with 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 $\{u, v\}$.) We stress that the graph is directed. That is, $\ell(u, v)$ and $\ell(v, u)$ need not be equal, and one of them may be ∞ .

Suppose $P = (v_0, v_1, \dots, v_k)$ is a path oriented from v_0 to v_k , and $e = (v_i, u)$ (where $i \in [1, k-1]$) is an edge not on P . 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 r -Divisions

Lipton and Tarjan [49] proved that every planar graph contains a *separator* of $O(\sqrt{n})$ vertices that, once removed, breaks the graph into components with at most $2n/3$ vertices each. Miller [52] showed that every triangulated planar graph has a $O(\sqrt{n})$ -size separator that consists of a simple cycle. Simple cycle separators can be used to recursively separate a planar graph until its components have constant size. Klein, Mozes, and Sommer [46] showed how to obtain, in $O(n)$ time, a complete recursive decomposition tree of G that is a binary tree whose nodes correspond to subgraphs of G , called *regions*, with the root being all of G , and the leaves being regions of constant size. The set of boundary vertices of a region R is denoted by ∂R : it consists of those vertices of R that belong to some separator along the recursive decomposition used to obtain R .

An r -*division*, introduced by Frederickson [29], is a set of $\Theta(n/r)$ regions, no two of which are ancestor of one another in the recursive decomposition tree, whose union is G (i.e., a maximal anti-chain), and such that each region has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices.

³Lemma 3.3 in [26] asserts that for any two paths p_1, p_2 from s to t , there exists a path p that is strictly shorter (under the perturbation) than at least one of p_1, p_2 . The proof of the lemma shows that the edges of p are contained in the union of the edges of p_1 and p_2 . Hence, shortest paths are unique in any subgraph of G . (In fact, [26, Lemma 3.3] discusses costs of flows, of which shortest paths is a special case.)

We use [46], for computing a hierarchical \vec{r} -division, where $\vec{r} = (r_m, \dots, r_1)$ and $n = r_m > \dots > r_1 = \Omega(1)$ in linear time. Each region in each r_i -division is a region in the complete recursive decomposition tree of G . Such an \vec{r} -division has the following properties:

- (Division & Hierarchy) For each i , \mathcal{R}_i is the set of regions in an r_i -division of G , where $\mathcal{R}_m = \{G\}$ consists of the graph itself. For each $i < i' \leq m$ and $R_i \in \mathcal{R}_i$, there is a unique $R_{i'} \in \mathcal{R}_{i'}$ such that $E(R_i) \subseteq E(R_{i'})$. The \vec{r} -division can therefore be represented as a rooted tree of regions.
- (Boundaries and Holes) The $O(\sqrt{r_i})$ boundary vertices of any $R_i \in \mathcal{R}_i$ lie on a constant number of faces of R_i called *holes*, each bounded by a cycle (not necessarily simple).

We modify the output of the Klein-Mozes-Sommer [46] \vec{r} -division in two ways. First, we supplement it with a zeroth level. The layer-0 $\mathcal{R}_0 = \{\{v\} \mid v \in V(G)\}$ consists of singleton sets, and each $\{v\}$ is attached as a (leaf) child of an arbitrary $R \in \mathcal{R}_1$ for which $v \in R$. Second, we modify the graph so that every hole of every region is bounded by a *simple* cycle in the graph. This involves cutting along paths of repeated edges; see Section 9 for details of this transformation.

Suppose that f is one of the $O(1)$ holes of region R and C_f is the cycle bounding f . The cycle C_f partitions $E(G) - C_f$ 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 $R^{f,\text{out}}$. The presentation of the algorithm is greatly simplified by assuming that in every region R , ∂R lies on a single hole f_R . We use R^{out} as a short form of $R^{f_R,\text{out}}$. In Section 9 we explain how to remove this assumption.

2.3 Multiple-Source Shortest Paths

Consider a weighted planar graph H with a distinguished face f on vertices S . Klein's MSSP algorithm [45] takes $O(|H| \log |H|)$ time and produces an $O(|H| \log |H|)$ -size data structure that, given $s \in S$ and $v \in V(H)$, returns $\text{dist}_H(s, v)$ in $O(\log |H|)$ time. Klein's algorithm can be viewed [11] as continuously moving the source vertex around the boundary face f , recording all changes to the single-source shortest paths (SSSP) tree in a Link-Cut tree data structure [59]. It is shown [45] that each edge in H enters and leaves the SSSP tree exactly once, and hence the number of changes is $O(|H|)$. Each change to the tree can be handled in $O(\log |H|)$ time [59], and the generic persistence method of [25] 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 ($O(\log |H|)$). We show that this structure can be augmented to also answer other useful queries. Further, we present alternative tradeoffs for the problem by implementing MSSP using Euler Tour trees [38], as opposed to the data structure of [45] that uses Link-Cut trees [59]. Since our data structure (with Euler Tour trees) does not satisfy the criteria of Driscoll et al.'s [25] persistence method for pointer-based data structures, we use the folklore implementation of persistent arrays⁴ 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 [45] and Cabello et al. [11]). *Let H be a planar graph, S be the vertices on some distinguished face f , and $\kappa \geq 1$ be a parameter. Consider the following queries.*

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

⁴Dietz [21] credits this method to an oral presentation of Dietzfelbinger et al. [22], which highlighted it as an application of dynamic perfect hashing. The idea is to maintain a van Emde Boas-type data structure [65, 66] for every array location $A[i]$ that contains every value written to $A[i]$, keyed by the time it is written. Both the values and timestamps are $O(\log n)$ -bit integers. Looking up $A[i]$ at time t involves a single predecessor search, which takes $O(\log \log n)$ time. Perfect hashing is used to reduce the space of each van Emde Boas structure to linear. If randomness is undesirable, we can afford to construct linear-space deterministic dictionaries with an $O(\log n)$ -factor overhead in construction time; see Hagerup, Miltersen, and Pagh [36].

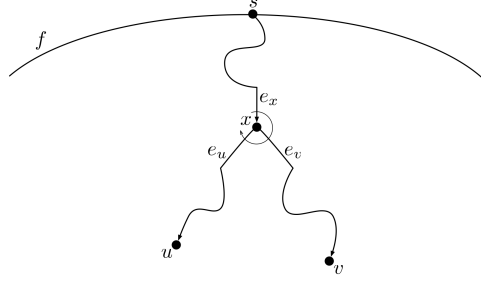


Fig. 2. The clockwise order of e_x, e_u, e_v around x tells us whether the shortest s -to- u path branches from the shortest s -to- v path to the right or left.

- Given $s \in S, u, v \in V(H)$, return (x, e_u, e_v) , where x is the lowest common ancestor (LCA) of u and v in the SSSP tree rooted at s and e_z is the edge on the path from x to z (if $x \neq z$) for $z \in \{u, v\}$.

We can achieve either of the following space-time tradeoffs:

- $O(|H| \log |H|)$ construction time, $O(|H| \log |H|)$ -space, and $O(\log |H|)$ query time, or
- $O(\kappa |H|^{1+1/\kappa})$ construction time, $O(\kappa |H|^{1+1/\kappa})$ -space, and $O(\kappa \log |H|)$ query time.

The purpose of the second query is to tell whether u lies on the shortest s -to- v path ($x = u$) or vice versa, or to tell whether the s -to- u path branches left or right from the s -to- v path. If they do branch, we also say that u is to the left/right of the s -to- v path. Once we retrieve the LCA x and edges e_u, e_v , we get the edge e_x from x to its parent. The clockwise order of e_x, e_u, e_v around x tells us⁵ whether the shortest s -to- u path branches to the left or to the right of the shortest s -to- v path. See Figure 2.

3 ADDITIVELY WEIGHTED VORONOI DIAGRAMS

Let H be a directed planar graph with real edge-lengths and no negative-length cycles. Assume that all faces of H are triangles except, perhaps, a single face h , which we regard as the *infinite face*. Let S be the set of vertices that lie on h . The vertices of S are called *sites*. Each site $s \in S$ has a weight $\omega(s) \geq 0$ associated with it. The additively weighted distance from a site $s \in S$ to a vertex $v \in V(H)$, denoted by $d^\omega(s, v)$, is defined as $\omega(s)$ plus the length of the shortest s -to- v path in H . To avoid clutter in the presentation we assume that $|S| > 2$. This is without loss of generality since when $|S| \leq 2$ (in fact, whenever $S = O(1)$) point location (Theorem 3.2) becomes trivial.

Definition 3.1. The Voronoi diagram of S within H with additive weights ω , denoted $\text{VD}[H, S, \omega]$, is a partition of $V(H)$ into pairwise disjoint sets, one set $\text{Vor}(s)$ for each site $s \in S$. The set $\text{Vor}(s)$, which is called the Voronoi cell of s , contains all vertices in $V(H)$ that are closer (w.r.t. $d^\omega(\cdot, \cdot)$) to s than to any other site in S . Ties are always broken consistently, in favor of sites with larger additive weights—formally, with respect to reverse lexicographic order on $(\omega(s), s)$.

Since every subgraph of H is strongly connected, the Voronoi cells partition $V(H)$. Due to the tie-breaking rule, for any $v \in \text{Vor}(s)$, the shortest s -to- v path is contained in $\text{Vor}(s)$. In particular, $\text{Vor}(s)$ is connected.

We say that an edge e of H *belongs* to $\text{Vor}(s)$ if both endpoints of e belong to $\text{Vor}(s)$. We say that e is a *boundary edge* of $\text{Vor}(s)$ if exactly one endpoint of e belongs to $\text{Vor}(s)$.

⁵The order can be inferred in constant time by storing with each edge its rank in a clockwise traversal of the edges incident to x , starting from an arbitrarily chosen vertex.

Next, we describe a space-efficient dual representation $VD^*[H, S, \omega]$ (or simply VD^*) of a Voronoi diagram $VD[H, S, \omega]$. Let H^* be the planar dual of H . Let VD_0^* be the subgraph of H^* consisting of the duals of edges $\{u, v\}$ of H such that u and v are in different Voronoi cells. Let VD_1^* be the graph obtained from VD_0^* by dissolving degree-2 vertices into their incident edges (or equivalently, eliminating each degree-2 vertex by contracting any one of its incident edges). The vertices of VD_1^* are called Voronoi vertices. A Voronoi vertex $f^* \neq h^*$ is dual to a triangular face f whose three vertices belong to three distinct Voronoi cells. We call such a face *trichromatic*. Each Voronoi vertex f^* stores for each vertex u incident to f the site s such that $u \in \text{Vor}(s)$. Note that h^* is a Voronoi vertex. Each face of VD_1^* corresponds to a cell $\text{Vor}(s)$. Hence there are at most $|S|$ faces in VD_1^* . Since the minimum degree in VD_1^* is 3, the total number of edges, vertices, and faces of VD_1^* is $O(|S|)$. Finally, we define VD^* to be the graph obtained from VD_1^* by splitting the node h^* into $\deg(h^*)$ copies, each one incident to an edge formerly incident to h^* . See Figure 3 for an illustration.

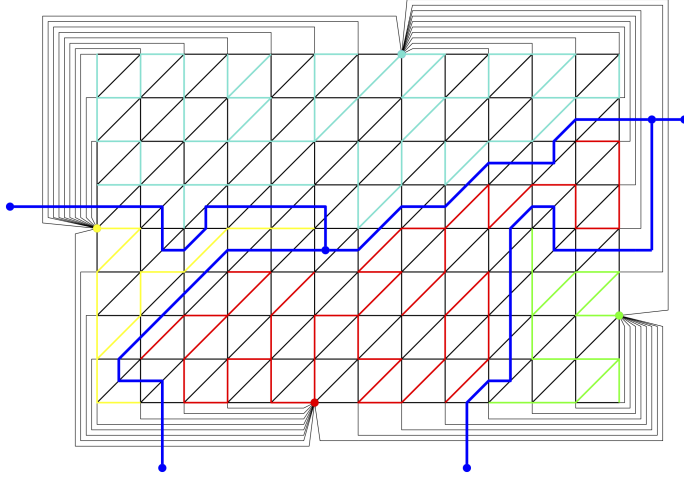


Fig. 3. A planar graph (black edges) with four sites on the infinite face together with the dual Voronoi diagram VD^* (in blue). VD^* is a tree with 6 vertices. The sites are shown together with their corresponding shortest path trees (in turquoise, red, yellow, and green).

We say that an edge e_0^* of VD_0^* is represented by an edge e^* of VD^* if e_0^* was contracted into e^* in the process defining VD^* . We say that an edge e^* of VD^* is incident to $\text{Vor}(s)$ if e^* is an edge on the face of VD_1^* that corresponds to $\text{Vor}(s)$.

Lemma 3.1. *If ω is such that every vertex of S lies in its own Voronoi cell then $VD^*[H, S, \omega]$ is a tree.*

PROOF. Suppose that VD^* contains a cycle C^* . Since the degree of each copy of h^* is one, the cycle avoids all copies of h^* . Since all the sites are on the boundary of the hole h , each of the vertices of the graph enclosed by C^* belongs in a Voronoi cell that contains no site, a contradiction.

To prove that VD^* is connected, observe that in VD_1^* , every Voronoi cell is a face bounded by a cycle that goes through h^* . Let C^* denote one such cycle. If C^* is disconnected in VD^* then, in VD_1^* , C^* must visit h^* at least twice. But this implies that the cell corresponding to C^* contains more than a single site, a contradiction. Thus, the boundary of every Voronoi cell is a connected subgraph of VD^* . Consider, for any i , the edge $\{s_i, s_{i+1}\}$. Since s_i and s_{i+1} are in the distinct Voronoi cells $\text{Vor}(s_i)$ and $\text{Vor}(s_{i+1})$, the dual of $\{s_i, s_{i+1}\}$ is represented by some edge of VD^* . Hence, for

every i , the boundaries of the Voronoi cell of s_i and of s_{i+1} share that edge, so they are in the same connected component of VD^* . It follows that the entire VD^* is connected. \square

Throughout the paper, we force the preconditions to Lemma 3.1 to hold. In particular, suppose S_0 are the vertices lying on the distinguished face h_0 in H_0 , and $S = \{s \in S_0 \mid s \in \text{Vor}(s)\}$ are those sites with non-empty Voronoi cells. Rather than construct $\text{VD}^*[H_0, S_0, \omega]$, we embed dummy edges in h_0 , so that S are the vertices of a distinguished face h in a graph H such that $\text{dist}_{H_0} = \text{dist}_H$. It follows from Lemma 3.1 that $\text{VD}^*[H, S, \omega]$ is a tree. See Figures 3 and 4 for an illustration of how dummy edges are added to a graph.

Let us also mention an alternative work-around. Consider a site s that belongs to the Voronoi cell of a different site s' . One can then substitute $\omega(s)$ by $d^\omega(s', s)$ and consider s as a proxy for s' . That is, due to our tie-breaking rules, with such updated weights, s will belong to its own Voronoi cell, and whenever we find that some vertex v belongs to the Voronoi cell of s , we know that in effect v belongs to the Voronoi cell of s' .

3.1 Point Location in Voronoi Diagrams

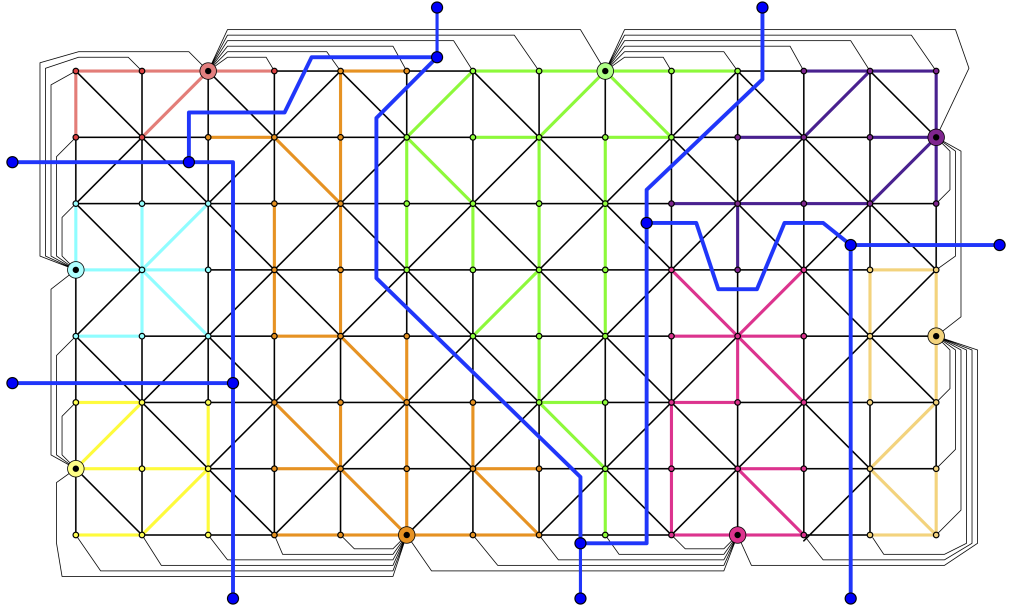
The *point location* problem is, given v and some representation of a Voronoi diagram $\text{VD}[H, S, \omega]$, to determine the site $s \in S$ for which $v \in \text{Vor}(s)$ and the distance $d^\omega(s, v)$. In this section we describe a data structure that answers point location queries efficiently, given access to an MSSP structure on the relevant graph.

THEOREM 3.2. *Let H be a planar graph and S be the vertices on a distinguished face h . Suppose we have access to an MSSP data structure for H with distinguished face h and query time t_q . After preprocessing $\text{VD}^*[H, S, \omega]$ in $O(|S|)$ time, we can answer point location queries in $O(t_q \cdot \log |S|)$ time.*

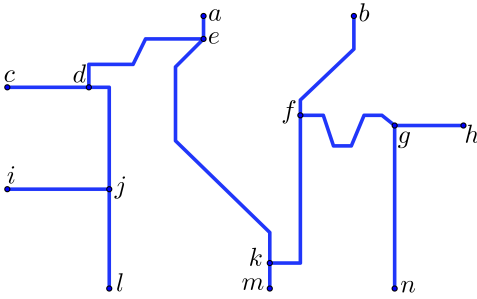
In the remainder of this section we prove Theorem 3.2. The main idea is as follows. In order to find the Voronoi cell $\text{Vor}(s)$ to which a query vertex v belongs, it suffices to identify an edge e^* of VD^* that is adjacent to $\text{Vor}(s)$. Given e^* we can simply check which of its two adjacent cells contains v by comparing the additive distances from the corresponding two sites to v using two MSSP queries. The point location data structure is based on a *centroid decomposition* of the tree VD^* into connected subtrees, and on the ability to go down this centroid decomposition, each time choosing a subtree that contains an edge adjacent to $\text{Vor}(s)$.

We assume that H is triangulated, except for the face h . In addition, for technical reasons we assume that for every face $f \neq h$ incident to $\{y_0, y_1, y_2\}$, three artificial vertices y_j^f , $j \in \{0, 1, 2\}$ have been embedded in f , each with a single zero-length incident edge (y_j, y_j^f) .⁶ This assumption does not change distances in H nor the asymptotic size of H . The preprocessing consists of just computing a centroid decomposition of VD^* . A *centroid* of an n -node tree T is a node $u \in T$ such that removing u and replacing it with copies, one for each edge incident to u , results in a set of trees, each with at most $\frac{n+1}{2}$ edges. A centroid always exists in a tree with at least one edge. The centroid decomposition of VD^* is defined recursively. In every step of the centroid decomposition we work with a connected subtree T^* of VD^* . Initially, T^* is the entire tree VD^* . Recall that there are no nodes of degree 2 in VD^* . If there are no nodes of degree 3, then T^* consists of a single edge of VD^* , and the decomposition terminates. Otherwise, we choose a centroid c^* , and partition T^* into the three subtrees T_0^*, T_1^*, T_2^* obtained by splitting c^* into three copies, one for each edge incident to c^* . Since the size of VD^* is $O(|S|)$, the depth of this recursive decomposition is $\log |S| + O(1)$.

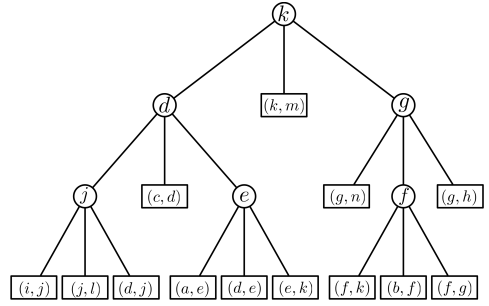
⁶The artificial vertices are leaves in any shortest path tree, while this is not true for the y_i s. Then, for every vertex $v \neq y_j^f$ that is not on the shortest s_j -to- y_j path, the shortest s_j -to- v path branches either left or right of the shortest s_j -to- y_j^f path, whereas v may be a descendant of y_j in the shortest path tree rooted at s_j . This is used, for instance, in Lemma 3.3.



(a)



(b)



(c)

Fig. 4. (a) The original H_0 is a triangulated grid, with f_0 being the exterior face. The boundary vertices S with non-empty Voronoi cells are marked with colored halos. Edges are added so that S are on the exterior face f . The vertices of VD^* are the duals of trichromatic faces, and those derived by splitting f^* into $|S|$ vertices. The edges of VD^* correspond to paths of duals of bichromatic edges. (b) The dual representation VD^* . (c) A centroid decomposition of VD^* .

Such a decomposition can be computed in $O(|S|)$ time [8, 33] and can be represented as a ternary tree which we call the *centroid decomposition tree*, in $O(|S|)$ space. Each non-leaf node of the centroid decomposition tree corresponds to a centroid vertex c^* , which is stored explicitly. We will refer to nodes of the centroid decomposition tree by their associated centroid. Each node also implicitly corresponds to the subtree of VD^* of which c^* is the centroid. The leaves of the centroid decomposition tree correspond to single edges of VD^* , which are stored explicitly. See Figure 4.

The procedure **SimpleCentroidSearch**(VD^*, v) takes as input a dual Voronoi diagram VD^* and a vertex v to be located. It returns a pair (s, d) where $v \in \text{Vor}(s)$ and $d = d^\omega(s, v)$. The procedure is recursive. It traverses a centroid decomposition for VD^* , and at intermediate invocations the procedure takes a third argument T^* , which is a subtree of the centroid decomposition. The loop invariant is that T^* contains a leaf representing some edge on the boundary of $\text{Vor}(s)$. The algorithm bottoms out in one of two base cases (Line 8 or Line 13).

The first way the recursion can end is if we reach the bottom of the centroid decomposition. If T^* is a singleton, its single node f^* corresponds to an edge in VD^* separating the Voronoi cells of two sites, say s_1 and s_2 . At this point we know that either $v \in \text{Vor}(s_1)$ or $v \in \text{Vor}(s_2)$, and determine which case is true by comparing the additive distances from each of s_1 and s_2 , which can be computed using the MSSP data structure (Lines 2–9).

We now explain how to treat the case where T^* is not a singleton. The root f^* of T^* 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 sites s_0, s_1, s_2 . Let e_0, e_1, e_2 be the edges $\{y_2, y_0\}, \{y_0, y_1\}, \{y_1, y_2\}$, respectively. For $j \in \{0, 1, 2\}$, let p_j denote the s_j -to- y_j shortest path. Further, let us denote by C_j the path obtained by concatenating path p_j , edge e_j , and the reverse of path p_{j-1} . (In notation related to a triangular face, all subscripts are naturally modulo 3, i.e., p_{j-1} is short for $p_{j-1 \pmod{3}}$.) A vertex of H either lies on one of the p_j s, or strictly to the right of exactly one of the C_j s. The second case can be equivalently restated as follows: v is enclosed by the cycle comprised of C_j and the s_{j-1} -to- s_j walk along face h that does not contain s_{j+1} . See Figure 5.

For each j , we can check whether v lies on some p_j using the MSSP data structure. If this is the case, then $v \in \text{Vor}(s_j)$, and we are done (Lines 12–13). We next show how to check whether v lies to the right of some C_j .

Lemma 3.3. *We can check whether v lies strictly to the right of C_j with a constant number of queries to an MSSP data structure for H with sources S .*

PROOF. We assume v does not lie on C_j since this was already checked. We check which of the sites s_j and s_{j-1} is closer to v with respect to the additive distances with two queries to the MSSP data structure at hand. Without loss of generality, suppose that this site is s_j . We then use a single query to the MSSP data structure to determine whether the shortest s_j -to- v path branches right from the shortest s_j -to- y_j^f path. (Recall that y_j^f is an auxiliary vertex embedded in the face f and connected to y_j with a zero length edge).

As we show next, v lies strictly to the right of C_j if and only if v lies strictly to the right of the shortest s_j -to- y_j^f path. Towards a contradiction, suppose that v lies strictly to the right of exactly one of C_j and the shortest s_j -to- y_j^f path. Then, the shortest s_j -to- v path must cross p_{j-1} . Due to planarity, it can only do so at a vertex. This yields a contradiction, as all vertices on p_{j-1} are in $\text{Vor}(s_{j-1})$, and due to the assumed uniqueness of shortest paths this would mean that s_j is not closer to v than s_{j-1} . \square

If it turns out that v is right of C_j , the algorithm recurses on the subtree of T^* rooted at the child of f^* that contains the leaf edge of VD^* representing e_j^* (Line 16).

Algorithm 1 SimpleCentroidSearch(VD^*, v, T^*)

Input: A Voronoi diagram VD^* , the vertex v to be located, and a centroid decomposition tree T^* of a subtree of VD^* . If the last argument is omitted, T^* is the decomposition tree for the entire VD^* .

Require: Some edge of the boundary of the Voronoi cell containing v in VD^* is a leaf in T^* .

Output: The site s such that $v \in \text{Vor}(s)$, and the additive distance to v .

```

1:  $f^* \leftarrow$  root of  $T^*$ 
2: if  $f^*$  is a single edge then
3:    $s_1, s_2 \leftarrow$  sites corresponding to  $f^*$ 
4:   for  $j = 1, 2$  do
5:      $d_j \leftarrow \omega(s_j) + \text{dist}_H(s_j, v)$  ▷ MSSP query
6:   end for
7:    $k \leftarrow \text{argmin}_j(d_j)$ 
8:   return  $(s_k, d_k)$ 
9: end if
10:  $s_0, s_1, s_2 \leftarrow$  sites corresponding to  $f^*$  ▷  $f$  is a face on  $\{y_0, y_1, y_2\}$ ,  $y_i \in \text{Vor}(s_i)$ 
11: for  $j = 0, 1, 2$  do
12:   if  $v$  lies on  $p_j$  then ▷ MSSP query;  $p_j$  is the shortest  $s_j$ -to- $y_j$  path in  $H$ 
13:     return  $(s_j, \omega(s_j) + \text{dist}_H(s_j, v))$ 
14:   else if  $v$  is (strictly) to the right of  $C_j$  then ▷ See Lemma 3.3;  $C_j$  is  $p_j \cup \{e_j\} \cup p_{j-1}$ 
15:      $T_j^* \leftarrow$  subtree of  $T^*$  rooted at the child of  $f^*$  containing the leaf edge of  $VD^*$  representing  $e_j^*$ 
16:     return SimpleCentroidSearch( $VD^*, v, T_j^*$ )
17:   end if
18: end for

```

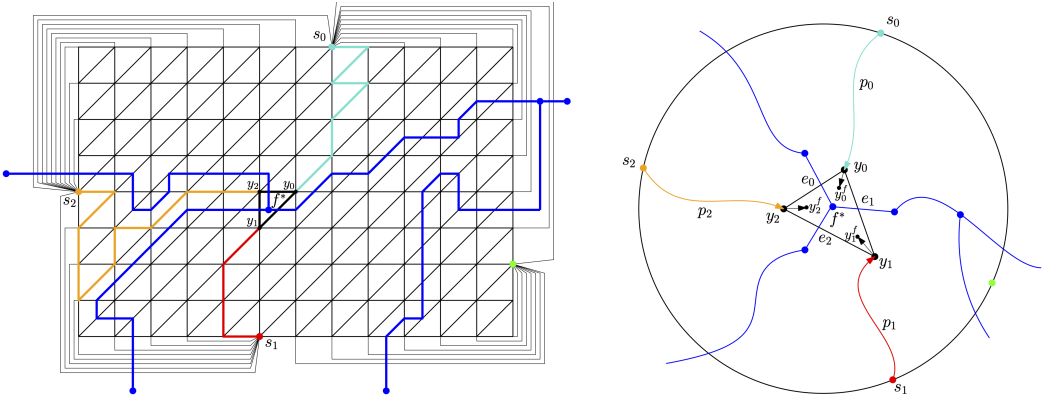


Fig. 5. Illustration of the setting and proof of Theorem 3.2. Left: A decomposition of VD^* (shown in blue) by a centroid f^* into three subtrees, and a corresponding partition of P into three regions delimited by the paths p_j (shown in red, yellow, and turquoise). Right: a schematic illustration of the same scenario.

We are now ready to complete the proof of Theorem 3.2 on the correctness and time complexity of **SimpleCentroidSearch**. Define $f, y_j, s_j, e_j^*, f^*, p_j, C_j$, for $j = \{0, 1, 2\}$ as above, and let \tilde{s} be such that $v \in \text{Vor}(\tilde{s})$. If Line 12 tells us that v is on p_j , then \tilde{s} is s_j , as returned in Line 13. The loop invariant is that T^* contains *some* leaf edge that belongs to the boundary of the cell $\text{Vor}(\tilde{s})$. This is clearly

true in the initial call, when T^* is the entire centroid decomposition of VD^* . Suppose that Line 14 tells us that v lies strictly to the right of C_j . Observe that since p_j and p_{j-1} are monochromatic, all edges of VD^* correspond to paths in H^* that are disjoint from the set of dual edges of C_j , with the exception of e_j^* . We claim that T_j^* contains at least one edge bounding $\text{Vor}(\tilde{s})$. First, this is clearly true if e_j^* is such an edge, i.e., if $\tilde{s} \in \{s_{j-1}, s_j\}$. In the complementary case, all vertices of $\text{Vor}(\tilde{s})$ are strictly to the right of C_j . Hence, none of the edges bounding $\text{Vor}(\tilde{s})$ can be in $T_{j'}^*$ for $j' \neq j$. Thus, the maintained invariant implies that there is such an edge in T_j^* .

When f^* is a single edge on the boundary of $\text{Vor}(s_1)$ and $\text{Vor}(s_2)$ (Line 2), the loop invariant guarantees that either $\tilde{s} = s_1$ or $\tilde{s} = s_2$. The additive distances d_1 and d_2 to s_1 and s_2 respectively are computed in Line 5, and \tilde{s} is the site with smaller additive distance among the two (Line 7). Hence, Line 8 returns the correct answer.

The efficiency of procedure **SimpleCentroidSearch** depends on the time required to compute distances in H (Lines 5 and 13) and whether v lies on or to the left/right of a shortest path p_j (Lines 12 and 14). By Lemma 2.1 and Lemma 3.3, each of these operations is supported by the MSSP data structure for H, S , whose query time is t_q . Hence the cost of the top-level call to **SimpleCentroidSearch** is $O(t_q \cdot \log |S|)$, $O(t_q)$ time for each of the $\log |S| + O(1)$ recursive calls.

4 A SIMPLE PLANAR DISTANCE ORACLE

In this section we present a distance oracle that is simpler than the one developed in Sections 5–8. Moreover, the time-space tradeoffs of the simpler distance oracle are actually incomparable to those of the oracle described in Sections 5–8 and would be more attractive if *query time* is prioritized over *space*. In particular, depending on the MSSP implementation (Lemma 2.1), we can achieve either $\tilde{O}(n^{4/3})$ space and $O(\log^2 n)$ query time or $n^{4/3+o(1)}$ space and $\log^{1+o(1)} n$ query time.

4.1 The Data Structure

We begin by computing an $\vec{r} = (r_3, r_2, r_1)$ division where $r_3 = n$, $r_2 \approx n^{2/3}$, and $r_1 \approx n^{1/3}$. In other words, $\mathcal{R}_3 = \{G\}$ contains one region, namely G , which is partitioned into regions \mathcal{R}_2 , each with $O(r_2)$ vertices and $O(\sqrt{r_2})$ boundary vertices, and so on. As usual, we temporarily assume that each region is bounded by a single hole and remove this assumption in Section 4.4. The data structure consists of the following three parts.

- (1) For each $R_1 \in \mathcal{R}_1$ and each pair of vertices $u, v \in R_1$, store $\text{dist}_{R_1}(u, v)$. The space for this part is $O((n/r_1) \cdot r_1^2) = O(n \cdot r_1)$.
- (2) For each $R_2 \in \mathcal{R}_2$, we store two MSSP structures, one for R_2 and one for R_2^{out} , both w.r.t. ∂R_2 . For each $R_1 \in \mathcal{R}_1$ with parent $R_2 \in \mathcal{R}_2$ we store MSSP structures for R_1 and $R_1^{\text{out}} \cap R_2$ w.r.t. ∂R_1 . The space required for these structures is, depending on the MSSP implementation, either

$$O((n/r_2) \cdot n \log n + (n/r_1) \cdot r_2 \log r_2) = \tilde{O}(n^{4/3})$$

or

$$O(\rho(n^2/r_2 + nr_2/r_1)),$$

where $\rho = \kappa n^{1/\kappa}$ is the space overhead.

- (3) Suppose vertex u is in $R_1 \in \mathcal{R}_1$, R_1 's parent is $R_2 \in \mathcal{R}_2$, and R_2 's parent is $R_3 = G \in \mathcal{R}_3$. For each vertex u we store the dual Voronoi diagrams $VD_{\text{in}}^*(u, R_1)$, $VD_{\text{in}}^*(u, R_2)$, $VD_{\text{out}}^*(u, R_1)$, and $VD_{\text{out}}^*(u, R_2)$, which are defined as follows.
 - For $i \in \{1, 2\}$, $VD_{\text{in}}^*(u, R_i)$ is $VD^*[R_i, \partial R_i, \omega]$, the dual representation of the Voronoi diagram for R_i with sites ∂R_i and additive weights given by $\omega(s) = \text{dist}_{R_{i+1}}(u, s)$.
 - For $i \in \{1, 2\}$, $VD_{\text{out}}^*(u, R_i)$ is $VD^*[R_i^{\text{out}} \cap R_{i+1}, \partial R_i, \omega]$ with $\omega(s) = \text{dist}_{R_{i+1}}(u, s)$.

The space for each dual Voronoi diagram is linear in the number of sites, i.e., over all u the total space is $O(n \cdot (\sqrt{r_2} + \sqrt{r_1})) = \tilde{O}(n^{4/3})$.

4.2 The Query Algorithm

The query algorithm **SimpleDist** (u, v, R_i) is recursive. It takes vertices $u, v \in R_i \in \mathcal{R}_i$ and reports $\text{dist}_{R_i}(u, v)$. At the top level recursive call **SimpleDist** (u, v, G) we have $i = 3$ ($G \in \mathcal{R}_3$ is the only region at level 3) and when $i = 1$ the distance can be reported immediately (Line 2), using part 1 of the data structure. Therefore the recursion depth is constant.

When $i \in \{2, 3\}$ we let R_{i-1} be a subregion of R_i containing u . There are two cases: $v \in R_{i-1}$ or $v \notin R_{i-1}$. When $v \in R_{i-1}$ the shortest u -to- v path can be contained entirely in R_{i-1} or it can cross ∂R_{i-1} . In the former case $\text{dist}_{R_i}(u, v) = \text{dist}_{R_{i-1}}(u, v)$, which is computed recursively (Line 6). In the latter case, suppose $s \in \partial R_{i-1}$ is the last boundary vertex along the shortest u -to- v path. Then

$$\text{dist}_{R_i}(u, v) = \text{dist}_{R_i}(u, s) + \text{dist}_{R_{i-1}}(s, v) = \omega(s) + \text{dist}_{R_{i-1}}(s, v),$$

where ω is the additive weight function in $\text{VD}_{\text{in}}^*(u, R_{i-1})$. In other words, computing $\text{dist}_{R_i}(u, v)$ reduces to a point location problem in $\text{VD}_{\text{in}}^*(u, R_{i-1})$ (Line 7). When $v \notin R_{i-1}$ we *know* the shortest u -to- v path crosses ∂R_{i-1} at least once; suppose that the last time it crosses is at vertex s . Then, by similar reasoning,

$$\text{dist}_{R_i}(u, v) = \text{dist}_{R_i}(u, s) + \text{dist}_{R_{i-1}^{\text{out}} \cap R_i}(s, v) = \omega(s) + \text{dist}_{R_{i-1}^{\text{out}} \cap R_i}(s, v),$$

where ω is the additive weight function from $\text{VD}_{\text{out}}^*(u, R_{i-1})$. This point location problem in $\text{VD}_{\text{out}}^*(u, R_{i-1})$ is handled in Line 10.

The query time is dominated by $O(1)$ point location queries. By Theorem 3.2, **SimpleDist** takes $O(t_q \log n)$ time, where $t_q \in \{O(\log n), O(\kappa \log \log n)\}$ depends on the MSSP implementation.

4.3 Analysis

If we use the first implementation of MSSP from Lemma 2.1, the overall space is linear in

$$nr_1 + (n^2/r_2 + nr_2/r_1) \log n + n\sqrt{r_2},$$

which is $O(n^{4/3} \log^{2/3} n)$ when $r_2 = n^{2/3} \log^{1/3} n$ and $r_1 = n^{1/3} \log^{2/3} n$. The space can be reduced to $O(n^{4/3} \log^{1/3} n)$ by using a 4-level \vec{r} -division, say, $\vec{r} = (n, n^{2/3} \log^{2/3} n, n^{(2/3)^2}, n^{(2/3)^3})$. This increases the cost of distance queries by a small constant factor.

If we use the second implementation of MSSP from Lemma 2.1, with a space overhead of $\rho = \kappa n^{1/\kappa}$, the overall space is linear in

$$nr_1 + \rho(n^2/r_2 + nr_2/r_1) + n\sqrt{r_2},$$

which is $O(n^{4/3} \rho^{2/3}) = O(\kappa^{2/3} n^{4/3+2/(3\kappa)})$ when $r_2 = n^{2/3} \rho^{1/3}$, $r_1 = n^{1/3} \rho^{2/3}$. When query time is prioritized it is best to set $\kappa = \omega(1)$ and $\log^{o(1)} n$, leading to a distance oracle with $n^{4/3+o(1)}$ space and query time $O((\kappa \log \log n) \cdot \log n) = \log^{1+o(1)} n$.

Algorithm 2 SimpleDist(u, v, R_i)**Input:** Two vertices u, v in a region $R_i \in \mathcal{R}_i, i \in \{1, 2, 3\}$.**Output:** $\text{dist}_{R_i}(u, v)$.

```

1: if  $i = 1$  then
2:   Return  $\text{dist}_{R_i}(u, v)$  ▷ Stored explicitly in Part 1.
3: end if
4:  $R_{i-1} \leftarrow$  a sub-region of  $R_i$  containing  $u$ 
5: if  $v \in R_{i-1}$  then
6:    $d_1 \leftarrow \text{SimpleDist}(u, v, R_{i-1})$ 
7:    $d_2 \leftarrow \text{SimpleCentroidSearch}(\text{VD}_{\text{in}}^*(u, R_{i-1}), v)$ 
8:   return  $\min(d_1, d_2)$ 
9: else
10:  return  $\text{SimpleCentroidSearch}(\text{VD}_{\text{out}}^*(u, R_{i-1}), v)$ 
11: end if

```

4.4 Dealing with Multiple Holes

In general the boundary vertices ∂R_i of any region R_i lie on $O(1)$ holes. We modify the data structure and query algorithm to deal with multiple holes as follows.

- (1) For each region R_i in the decomposition and each hole h of R_i we build two MSSP data structures, one for R_i and one for $R_i^{h, \text{out}}$. In both structures, the set of sources are the vertices of ∂R_i that lie on h .
- (2) Fix a vertex u that lies in $R_1 \in \mathcal{R}_1$, which is contained in $R_2 \in \mathcal{R}_2$ and $G = R_3 \in \mathcal{R}_3$. For $i \in \{1, 2\}$, for each hole h of R_i , let S be the vertices on h .
 - We store $\text{VD}_{\text{in}}^*(u, h, R_i)$, which is the dual representation $\text{VD}^*[R_i, S, \omega]$ with additive weights $\omega(s) = \text{dist}_{R_{i+1}}(u, s)$.
 - We store $\text{VD}_{\text{out}}^*(u, h, R_i)$, which is the dual representation $\text{VD}^*[R_i^{h, \text{out}} \cap R_{i+1}, S, \omega]$ with additive weights $\omega(s) = \text{dist}_{R_{i+1}}(u, s)$.

The algorithm **SimpleDist**(u, v, R_i) is modified as follows. In Line 7 we are considering u -to- v paths that cross ∂R_{i-1} , but the last ∂R_{i-1} vertex s could be on any hole of R_{i-1} . Thus, for each of the $O(1)$ holes h we execute **SimpleCentroidSearch**($\text{VD}_{\text{in}}^*(u, h, R_{i-1}), v$) and let d_2 be the minimum distance found. In Line 10, there is a unique hole h of R_{i-1} for which $v \in R_{i-1}^{h, \text{out}}$ and we know that every u -to- v path must cross h . Therefore, we still only make one call to **SimpleCentroidSearch**($\text{VD}_{\text{out}}^*(u, h, R_{i-1}), v$).

Theorem 4.1 summarizes the space-time tradeoffs achievable by our simplest distance oracle.

THEOREM 4.1. *Let G be a weighted, directed planar graph. Distance queries in G can be answered in $O(\log^2 n)$ time with an $\tilde{O}(n^{4/3})$ -size oracle, or in $\log^{1+o(1)} n$ time with an $n^{4/3+o(1)}$ -size oracle.*

4.5 Digression: Extension to Graphs of Bounded Genus

Here, we briefly describe how to generalize the oracle described in this section for graphs embeddable onto surfaces of bounded genus. As shown by Chambers, Erickson, and Nayyeri in [12], we can “planarize” an n -vertex graph G of genus g by repeating the following procedure g times: find a short non-contractible cycle in $O(gn \log n)$ time using the algorithm of Erickson and Har-Peled [27], and cut along it, duplicating its vertices and edges. This algorithm thus runs in $O(g^2 n \log n)$ time and produces an n -vertex planar graph P with exactly $2g$ holes that contain all the copies of the duplicated vertices. Each such hole, called a *boundary cycle*, is incident to $O(\sqrt{n/g} \log g)$ vertices.

To avoid clutter, we describe our oracle for $g = O(1)$. We build our $n^{4/3+2/(3\kappa)}$ -space oracle for P with $\kappa = O(1) \geq 4$ so the space is $O(n^{3/2})$. Further, for each vertex $u \in V(G)$, for each of the $O(1)$ boundary cycles, we build a Voronoi diagram for P with sites the vertices of the hole, and an additive weight function defined by distances from u in G , i.e., $\omega(s) = \text{dist}_G(u, s)$. Further, for each boundary cycle C , we build an MSSP data structure for P with sources the vertices lying on C . Setting $\kappa' = O(1) \geq 2$ the space for the MSSP structures is $O(\kappa' n^{1+1/\kappa'}) = O(n^{3/2})$ since we have $O(1)$ boundary cycles, and the space for the Voronoi diagrams is $O(n^{3/2})$ since there are $O(\sqrt{n})$ vertices on boundary cycles.

Upon a query (u, v) , we first compute $\text{dist}_P(u, v)$ in $O(\log n \log \log n)$ time ($\kappa = O(1)$) using the planar distance oracle. First, note that $\text{dist}_P(u, v) \geq \text{dist}_G(u, v)$. Further, observe that $\text{dist}_P(u, v) \neq \text{dist}_G(u, v)$ only if some vertex on the shortest u -to- v path in G has been duplicated, and thus the path has been split. Suppose that this is the case. Let s be the last vertex on the shortest u -to- v path in G that has been duplicated. Note that s is not known at query time. By choice of s , $\text{dist}_G(u, v) = \text{dist}_G(u, s) + \text{dist}_P(s, v)$. Thus, we can take care of this case by performing point location queries in each of the $O(1)$ extra Voronoi diagrams stored for u with target v , and returning the minimum distance computed by those queries. The query time is $O(\log n \log \log n)$ since $\kappa' = O(1)$.

Thus, for n -vertex graphs embeddable to surfaces of constant genus, we obtain an oracle that occupies space $O(n^{3/2})$ and answers queries in $O(\log n \log \log n)$ time.

5 THE DISTANCE ORACLE

In this section we introduce our main distance oracle referenced in Theorem 1.1. The oracle is based on an \vec{r} -division, $\vec{r} = (r_m, \dots, r_1)$, where $r_i = n^{i/m}$ and m is a parameter. Suppose that we want to compute $\text{dist}_G(u, v)$. Let $R_0 = \{u\}$ be the artificial level-0 region containing u and $R_i \in \mathcal{R}_i$ be the level- i ancestor of R_0 . (Throughout the paper, we will use “ R_i ” to refer specifically to the level- i ancestor of $R_0 = \{u\}$, as well as to a *generic* region at level- i . Surprisingly, this will cause no confusion.) Let t be the unique index for which $v \notin R_t$ but $v \in R_{t+1}$. For $0 \leq i \leq t$, define u_i to be the *last* vertex on ∂R_i encountered on the shortest path from u to v . The main task of the distance query algorithm is to compute the sequence $(u = u_0, \dots, u_t)$. Suppose that we know the identity of u_i and $t > i$. Finding u_{i+1} now amounts to a point location problem for v in $\text{VD}^*[R_{i+1}^{\text{out}}, \partial R_{i+1}, \omega]$, where $\omega(s)$ is the distance from u_i to $s \in \partial R_{i+1}$. However, we cannot afford to store an MSSP structure for every $(R_{i+1}^{\text{out}}, \partial R_{i+1})$, since $|R_{i+1}^{\text{out}}| = \Omega(|G|)$. 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 and then decides between them using two recursive distance queries, but starting one level higher 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 unveiled in Section 6.

- (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(b)) for the graph R_i^{out} , and source set ∂R_i . However, the structure only answers queries for $s \in \partial R_i$ and $u, v \in R_i^{\text{out}} \cap R_{i+1}$. 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 $R_i^{\text{out}} \cap R_{i+1}$, i.e., the parent of any vertex $v \in R_i^{\text{out}} \cap R_{i+1}$ is its nearest ancestor v' in the SSSP tree such that $v' \in R_i^{\text{out}} \cap R_{i+1}$. If (v', v) is a “shortcut” edge corresponding to a path in R_{i+1}^{out} , it has weight $\text{dist}_{R_{i+1}^{\text{out}}}(v', v)$.

We fix a κ and let the update time in the dynamic tree data structure be $O(\kappa n^{1/\kappa})$ time. Thus, the space⁷ of this structure is $O(|R_i^{\text{out}} \cap R_{i+1}| + |\partial R_i| \cdot |\partial R_{i+1}| \cdot \kappa n^{1/\kappa}) = O(r_{i+1} \cdot \kappa n^{1/\kappa})$ since

⁷This is also the construction time which will be analyzed in Section 8.

each edge in $R_i^{\text{out}} \cap R_{i+1}$ is swapped into and out of the SSSP tree once [45], and the number of shortcut edges on ∂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 $O(mkn^{1+1/m+1/\kappa})$ since $r_{i+1}/r_i = n^{1/m}$.

- (B) (**Voronoi Diagrams**) For each $i \in [0, m-2]$ 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 $\text{VD}_{\text{out}}^*(q, R_{i+1})$ to be $\text{VD}^*[R_{i+1}^{\text{out}}, \partial R_{i+1}, \omega]$, 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 $O(mn^{1+1/(2m)})$ 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 5.1. *Suppose that $q \in \partial R_i$ and $v \notin R_{i+1}$. Consider the Voronoi diagram represented by $\text{VD}_{\text{out}}^*(q, R_{i+1})$ with sites ∂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 vertex of ∂R_{i+1} that lies on the shortest path from q to v in G , and $d^\omega(s, v) = \text{dist}_G(q, v)$.*

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

5.1 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 **Dist**(u, v, R_0). In general, the function **Dist**(u_i, v, R_i) takes as arguments a region R_i , a source vertex u_i on the boundary ∂R_i , and a target vertex $v \notin R_i$. It returns a value d such that

$$\text{dist}_G(u_i, v) \leq d \leq \text{dist}_{R_i^{\text{out}}}(u_i, v). \quad (1)$$

Note that $R_0^{\text{out}} = G$, 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 $v \in R_{i+1}^{\text{out}}$ it performs point location using the function **CentroidSearch**, which culminates in up to two recursive calls to **Dist** on the level- $(i+1)$ region R_{i+1} . Thus, the correctness of **Dist** hinges on whether **CentroidSearch** correctly computes distances when $v \in R_{i+1}^{\text{out}}$.

Algorithm 3 **Dist**(u_i, v, R_i)

Input: A region R_i , source $u_i \in \partial R_i$ and $v \notin R_i$.

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 | ▷ I.e., $i = t$ |
| 2: return $d \leftarrow \text{dist}_{R_i^{\text{out}}}(u_i, v)$ | ▷ Part (A) |
| 3: end if | ▷ $v \in R_{i+1}^{\text{out}}$ |
| 4: return $d \leftarrow \text{CentroidSearch}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v)$ | |
-

The procedure **CentroidSearch** is an adaptation of **SimpleCentroidSearch**. **CentroidSearch** is given as input $u_i \in \partial R_i$, $v \in R_{i+1}^{\text{out}}$, $\text{VD}_{\text{out}}^* = \text{VD}_{\text{out}}^*(u_i, R_{i+1})$, and a subtree T^* of the centroid decomposition of VD_{out}^* . Once again, if omitted, T^* is the full centroid decomposition. It ultimately

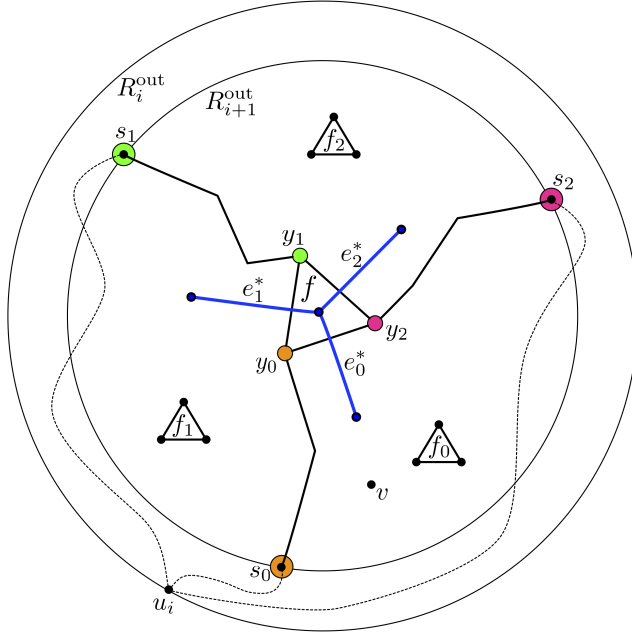


Fig. 6. Here f^* is a degree-3 vertex in $\text{VD}_{\text{out}}^*(u_i, R_{i+1})$, corresponding to a trichromatic face f on vertices y_0, y_1, y_2 , which are in the Voronoi cells of s_0, s_1, s_2 on the boundary $\partial R_{i+1}^{\text{out}}$. The shortest s_j -to- y_j paths partition $V(R_{i+1}^{\text{out}})$ into six parts: the three shortest paths and the three regions bounded by them and by f . Let e_0^*, e_1^*, e_2^* be the edges in VD_{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” $(s_j, \dots, y_j, y_{j-1}, \dots, s_{j-1})$. **CentroidSearch** locates the site whose Voronoi cell contains v via recursion. It calls each of **SitePathIndicator** and **ChordIndicator** thrice, in order to find which of the 6 parts contains v . If v lies on an s_j -to- y_j path the **CentroidSearch** recursion terminates; otherwise it recurses on the correct child f_j^* of f^* .

finds $u_{i+1} \in \partial R_{i+1}$ for which $v \in \text{Vor}(u_{i+1})$ and returns

$$\begin{aligned}
 & \omega(u_{i+1}) + \text{Dist}(u_{i+1}, v, R_{i+1}) && \text{Line 5 or 13 of } \mathbf{CentroidSearch} \\
 & \leq \text{dist}_G(u_i, u_{i+1}) + \text{dist}_{R_{i+1}^{\text{out}}}(u_{i+1}, v) && \text{Defn. of } \omega; \text{ guarantee of } \mathbf{Dist} \text{ (Eqn. (1))} \\
 & = \text{dist}_G(u_i, v). && \text{Lemma 5.1}
 \end{aligned}$$

The main difficulty in implementing **CentroidSearch** is that we cannot afford to store MSSP structures for R_{i+1}^{out} . **CentroidSearch** can be seen as an implementation of **SimpleCentroidSearch** with the following modifications.

- Distances from sites of $\text{VD}_{\text{out}}^*(u_i, R_{i+1})$ to vertices in R_{i+1}^{out} are now computed using **Dist** rather than MSSP queries. In particular, **CentroidSearch** is aware of the recursive decomposition of G .
- Line 12 of **SimpleCentroidSearch** is replaced by a call to a procedure **SitePathIndicator**, which returns a boolean indicating whether v is on the shortest s_j -to- y_j path.
- Line 14 of **SimpleCentroidSearch** is replaced by a call to a procedure **ChordIndicator**, which returns whether v lies strictly to the right of the oriented path $(s_j, \dots, y_j, y_{j-1}, \dots, s_{j-1})$. We call such a path a *chord*; these are formally defined in Section 6.2.

Algorithm 4 $\text{CentroidSearch}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, T^*)$

Input: The dual representation $\text{VD}_{\text{out}}^* = \text{VD}_{\text{out}}^*(u_i, R_{i+1})$ of a Voronoi diagram with additive weights $\omega(s) = \text{dist}_G(u_i, s)$, a vertex $v \in R_{i+1}^{\text{out}}$, and a centroid decomposition tree T^* of a subtree of VD_{out}^* . If the last argument is omitted, T^* is the decomposition tree for the entire VD_{out}^* .

Require: Some edge of the boundary of the Voronoi cell containing v in VD_{out}^* is a leaf in T^* .

Output: The distance $\text{dist}_G(u_i, v)$.

```

1:  $f^* \leftarrow$  root of  $T^*$ 
2: if  $T^*$  is a single edge then
3:    $s_1, s_2 \leftarrow$  sites corresponding to  $f^*$  ▷ Candidates for  $u_{i+1}$ 
4:   for  $j = 1, 2$  do
5:      $d_j \leftarrow \omega(s_j) + \text{Dist}(s_j, v, R_{i+1})$ 
6:   end for
7:    $k \leftarrow \text{argmin}_j(d_j)$ 
8:   return  $(s_k, d_k)$ 
9: end if
10:  $s_0, s_1, s_2 \leftarrow$  sites corresponding to  $f^*$ 
11: for  $j = 0, 1, 2$  do
12:   if  $\text{SitePathIndicator}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, f^*, j)$  returns True then
13:     return  $\omega(s_j) + \text{Dist}(s_j, v, R_{i+1})$  ▷  $s_j = u_{i+1}$ 
14:   else if  $\text{ChordIndicator}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, f^*, j)$  returns True then
15:      $T_j^* \leftarrow$  subtree of  $T^*$  rooted at the child of  $f^*$  containing the leaf edge of  $\text{VD}_{\text{out}}^*$  representing  $e_j^*$ 
16:     return  $\text{CentroidSearch}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, T_j^*)$ 
17:   end if
18: end for

```

Lemma 5.2. ***CentroidSearch** correctly computes $\text{dist}_G(u_i, v)$.*

PROOF. Let \tilde{s} be the site of VD_{out}^* for which $v \in \text{Vor}(\tilde{s})$. Apart from Lines 5, 13, **CentroidSearch** is just a different implementation of **SimpleCentroidSearch**. Thus, it follows directly from the proof of Theorem 3.2 that **CentroidSearch** either correctly identifies the site \tilde{s} in Line 12, or it identifies two candidates for \tilde{s} in Line 3. First, we have to show that the additive distance from \tilde{s} , computed in Line 5 or in Line 13 is indeed $\text{dist}_G(u_i, v)$. In either of the two cases, we have

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

Finally, if there is another candidate s' different than \tilde{s} identified in Line 3, we clearly have $\omega(s') + \text{Dist}(s', v, R_{i+1}) \geq \text{dist}_G(u_i, v)$. This completes the proof. \square

The main challenge is to efficiently implement the **SitePathIndicator** and **ChordIndicator** functions, i.e., to solve the restricted point location problem in R_{i+1}^{out} , depicted in Figure 6. We will show how to solve these two point location problems in $O(\kappa \log^{1+o(1)} n)$ time.

6 CHORDS, PIECES, AND THE INDICATOR FUNCTIONS

Recall that the main problem faced by **CentroidSearch** is to determine whether v lies on, left of, or right of the chord

$$\tilde{C} = (s_j, \dots, y_j, y_{j-1}, \dots, s_{j-1}),$$

which is a simple path joining ∂R_{i+1} -vertices in R_{i+1}^{out} . The case when $v \in \tilde{C}$ (which is detected by **SitePathIndicator**) is relatively simple, so for the purpose of this overview we shall assume $v \notin \tilde{C}$.

The index t is such that $v \in R_t^{\text{out}} \cap R_{t+1}$ so it suffices to restrict our attention to R_t^{out} . Note, however, that \tilde{C} can cross ∂R_t an unbounded number of times, meaning that the projection of \tilde{C} onto R_t^{out} consists of an unbounded number of *chords*, i.e., subpaths of \tilde{C} in R_t^{out} joining vertices of ∂R_t . These chords partition R_t^{out} into a set \mathcal{P} of *pieces*.

The strategy of **ChordIndicator** is to find any chord $C \in \mathcal{C}$ that lies on the boundary of v 's piece in \mathcal{P} . It follows that the left/right relationship between v and \tilde{C} is identical to the left/right relationship between v and C . Thus, we have reduced our problem to several structured point location problems, among them locating v in a certain set of pieces, and determining the relationship between v and a single chord C . In reality things are slightly more complicated, as we decompose \mathcal{C} (and hence \mathcal{P}) into three parts corresponding to (1) all chords in the s_j -to- y_j path that do not include y_j , (2) all chords in the s_{j-1} -to- y_{j-1} path that do not include y_{j-1} , and (3) the one chord (if any) that includes y_j and y_{j-1} .

Roadmap for Section 6. The sketch above motivates several useful subroutines. We need to be able to decide if v lies on, left of, or right of a chord C , where C is either a shortest path between ∂R_t vertices or the subpath of \tilde{C} between ∂R_t vertices that goes through y_j and y_{j-1} . These two types of queries are addressed in Lemmas 6.1 and 6.2 in Section 6.1. Section 6.1 also introduces parts (C) and (D) of the data structure, and Lemma 6.3 shows that a special case of **SimpleCentroidSearch** can be implemented efficiently. In particular, if VD^* is a Voronoi diagram for R_t^{out} and $v \in R_t^{\text{out}} \cap R_{t+1}$, **SimpleCentroidSearch**(VD^*, v) can be solved in the same time bound as in Theorem 3.2, using parts (A,D) of the data structure in lieu of a full MSSP structure.

Section 6.2 analyzes the properties of chords and pieces, and introduces part (E) of the data structure, which represents numerous chord/piece sets space-efficiently using persistent data structures. The **SitePathIndicator** and **ChordIndicator** functions are explained in Sections 6.3 and 6.4, respectively. A key subroutine of **ChordIndicator** is **PieceSearch**, which solves a certain point location problem with respect to an ensemble of chords and pieces; it is presented in Section 6.4.1.

6.1 Auxiliary Lemmas and a Special Case of SimpleCentroidSearch

We begin with the following lemma, which is used in **SitePathIndicator**, **PieceSearch**, and **ChordIndicator**.

Lemma 6.1. *Consider a region R_t , two vertices $a, b \in \partial R_t$, and a vertex $v \in R_t^{\text{out}} \cap R_{t+1}$. Let C be the shortest a -to- b path in R_t^{out} . We can test whether v lies on C and whether v lies to the right of C in $O(\kappa \log \log n)$ time, using part (A) of the data structure.*

PROOF. Let a', b' be pendant vertices attached to a, b , respectively, embedded inside the face of R_t^{out} bounded by ∂R_t . We ask the MSSP structure (part (A)) for the lowest common ancestor, w , of v and b' in the shortcutted SSSP tree rooted at a' . It follows that v lies on C if and only if $v = w$. We henceforth suppose that this is not the case. Then, the shortest a' -to- v and a' -to- b' paths branch at some point. The LCA query also returns the two tree edges $e_v, e_{b'}$ leading to v and b' , respectively. Let e_w be the edge connecting w to its parent.⁸ If the clockwise order around w is $e_w, e_{b'}, e_v$ then v lies to the right of C ; otherwise it lies to the left. Note that if the shortest a' -to- b' and a' -to- v paths in G branch at a point in R_{t+1}^{out} , then w will be the nearest ancestor of the branchpoint on

⁸The purpose of adding a', b' is to make sure all three edges $e_w, e_v, e_{b'}$ exist. The vertices a', b' are not represented in the MSSP structure. The edges (a', a) and (b, b') can be simulated by inserting them between the two boundary edges on ∂R_t adjacent to a and b , respectively.

∂R_{t+1} and one or both of $e_v, e_{b'}$ may be “shortcut” edges in the MSSP structure. See Figure 7 for an illustration. \square

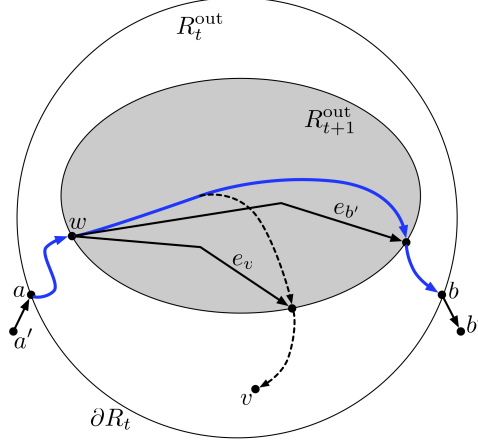


Fig. 7. The a -to- b shortest path, which may pass through R_{t+1}^{out} , in which case it is represented in the MSSP structure with shortcut edges (solid, angular edges).

Lemma 6.2. Consider a vertex $u \in R_t$ and an edge $\{y_0, y_1\}$ of R_t^{out} . For $j \in \{0, 1\}$, let x_j be the last vertex of the shortest u -to- y_j path that lies on ∂R_t , and suppose $x_0 \neq x_1$. Let C be the concatenation of the shortest x_1 -to- y_1 path in R_t^{out} , the edge $\{y_1, y_0\}$, and the reverse of the shortest x_0 -to- y_0 path in R_t^{out} . Further, for $j \in \{0, 1\}$, let \hat{x}_j be the last vertex of the shortest x_j -to- y_j path that lies on ∂R_{t+1}^{out} (if it exists).

Given $R_t, u, y_j, x_j, \text{dist}_G(u, x_j)$, and \hat{x}_j for $j \in \{0, 1\}$, and a vertex $v \in R_t^{out} \cap R_{t+1}$, we can test whether v lies on C and whether v lies to the right of C in $O(\kappa \log \log n)$ time, using part (A) of the data structure.

PROOF. Consider the following distance function \hat{d} for vertices $z \in R_t^{out}$:

$$\hat{d}(z) = \min \left\{ \text{dist}_G(u, x_0) + \text{dist}_{R_t^{out}}(x_0, z), \text{dist}_G(u, x_1) + \text{dist}_{R_t^{out}}(x_1, z) \right\}.$$

Observe that the terms involving u are given and, if $z \in R_t^{out} \cap R_{t+1}$, 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. \hat{d} has two trees, rooted at x_0 and x_1 . Using part (A) of the data structure we compute $\hat{d}(v)$, which reveals the $j^* \in \{0, 1\}$ such that v is in x_{j^*} 's tree. Let f be a face on which y_0, y_1 lie, such that the third vertex of f lies to the left of C . At this point we break into two cases, depending on whether f is in $R_t^{out} \cap R_{t+1}$ or in R_{t+1}^{out} . Without loss of generality, we assume that $j^* = 1$ and depict only this case in Figure 8(a,b).

Case a. Suppose that f is in $R_t^{out} \cap R_{t+1}$. Let y_1^f be a pendant vertex attached to y_1 embedded inside f and let x_1' be a pendant vertex attached to x_1 embedded inside the face of R_t^{out} bounded by ∂R_t . The shortest x_1' -to- y_1^f and x_1' -to- v paths share a common prefix. We query the MSSP structure (part (A)) to get the lowest common ancestor w of y_1^f and v and the three edges $e_{y_1^f}, e_v, e_w$ around w . If $v = w$ then v is on the shortest x_1 -to- y_1^f path and hence on C . If $v \neq w$ then all three edges

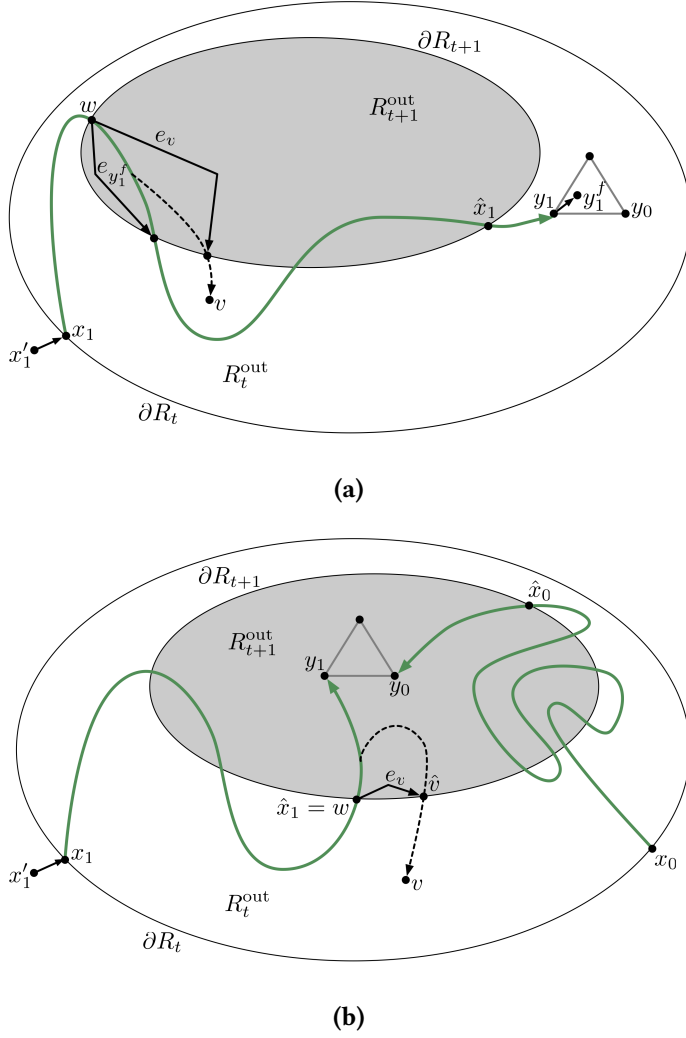


Fig. 8. An illustration of the setting in Lemma 6.2. (a) The case where f lies in $R_t^{\text{out}} \cap R_{t+1}$. (b) The case where f lies in R_{t+1}^{out} , \hat{x}_0, \hat{x}_1 are the last ∂R_{t+1} vertices on the x_0 -to- y_0 and x_1 -to- y_1 paths. If the shortest x'_1 -to- \hat{x}_1 and x'_1 -to- v paths branch, we can answer the query as in (a). If x'_1 -to- \hat{x}_1 is a prefix of x'_1 -to- v , $e_v = (\hat{x}_1, \hat{v})$, and e_v is a shortcut edge (which implies $\hat{v} \in \partial R_{t+1}$), then we can use the clockwise order of $\hat{x}_1, \hat{v}, \hat{x}_0$ around the hole on ∂R_{t+1} to determine whether v lies to the right of C . (Not depicted: the case that e_v is an original edge, where \hat{v} may not be on ∂R_{t+1} .)

$e_{y_1^f}, e_v, e_w$ are distinct and we determine whether v is to the right of C by examining the circular order of the three edges incident to w , as in the proof of Lemma 6.1. (If $j^* = 0$ then we would reverse the answer due to the reversed orientation of the x_0 -to- y_0 subpath w.r.t. C .) See Figure 8(a) for an illustration.

Case b. Now suppose f lies in R_{t+1}^{out} . We first ask the MSSP structure of part (A) for the lowest common ancestor w of \hat{x}_1 and v in the shortcutted SSSP tree rooted at x'_1 , and also get the three

incident edges $e_{\hat{x}_1}, e_v, e_w$. If $w = v$ then $v \in C$ and we are done, so we proceed under the assumption that $w \neq v$. Thus, the edges e_v and e_w exist and are different. If $w \neq \hat{x}_1$ then $e_{\hat{x}_1}$ also exists, and once again we determine whether v is to the right of C from the circular order of $e_v, e_w, e_{\hat{x}_1}$ around w . If $w = \hat{x}_1$, $e_{\hat{x}_1}$ does not exist. In this case, let \hat{v} be the endpoint of e_v that is not \hat{x}_1 . If e_v is a shortcut edge, it implies $\hat{v} \in \partial R_{t+1}$ and we can determine whether v is to the right of C from the circular order of \hat{x}_1, \hat{x}_0 and \hat{v} along ∂R_{t+1} . If e_v is an original edge, we have $e_v \in R_t^{\text{out}} \cap R_{t+1}$. By viewing (\hat{x}_1, \hat{x}_0) as a virtual shortcut edge, the left/right relationship between v and C now depends on the circular order of $e_v, e_w, (\hat{x}_1, \hat{x}_0)$ around \hat{x}_1 .⁹ See Figure 8(b) for an illustration. \square

Let us now introduce parts (C) and (D) of our data structure. The reason for storing part (C) will become clear in subsequent sections. One of the main reasons for storing the **Site Tables** of part (D) is so that we can invoke Lemma 6.2, which requires that we provide \hat{x}_0, \hat{x}_1 . The **Side Tables** of part (D) are stored so that we can handle a simple case in the **ChordIndicator** function where the chord does not interact at all with some specific part of the graph that contains v ; they store the answer for this whole part.

(C) (**More Voronoi Diagrams**) For each $i \in [1, m-1]$, 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 $\omega(s) = \text{dist}_G(q, s)$. The total space for these diagrams is $O(mn)$ and is dominated by part (B).

(D) (**Site Tables; Side Tables**) For each i and Voronoi diagram $\text{VD}_{\text{out}}^* = \text{VD}_{\text{out}}^*(u', R_i)$ from part (B) or (C), we store the following for each node f^* in the centroid decomposition of VD_{out}^* , with $y_j, s_j, j \in \{0, 1, 2\}$ defined as usual. Let $R_{i'} \in \mathcal{R}_{i'}$ be the ancestor of R_i at level $i' \geq i$. For each i' and $j \in \{0, 1, 2\}$ we store the pair (q, x) consisting of the *first* and *last* vertices on the shortest s_j -to- y_j path that lie on $\partial R_{i'}$. We also store $\text{dist}_G(u', x)$.

It may be that the shortest s_j -to- y_j path does not intersect $\partial R_{i'}$, in which case (q, x) do not exist. In this case we store a single bit indicating whether $R_{i'}^{\text{out}}$ lies to the right or left of the site-centroid-site chord $(s_j, \dots, y_j, y_{j-1}, \dots, s_{j-1})$ in R_i^{out} . The space cost for part (D) is $O(m)$ times the space cost of (B) and (C).

The following lemma is a direct consequence of Lemma 6.2, which lets us implement the non-trivial parts of **SimpleCentroidSearch** in the same time bound guaranteed by Theorem 3.2.

Lemma 6.3. *Suppose $\text{VD}^* = \text{VD}_{\text{out}}^*(u', R_t)$ is one of the Voronoi diagrams stored in part (C), and $v \in R_t^{\text{out}} \cap R_{t+1}$. Then **SimpleCentroidSearch** (VD^*, v) can be executed in $O(\kappa \log n \log \log n)$ time, using parts (A) and (D) of the data structure. (I.e., it does not require a full MSSP structure for $R_t^{\text{out}}.$)*

PROOF. Because $v \in R_t^{\text{out}} \cap R_{t+1}$, the distances in Lines 5 and 13 can be computed in $O(\kappa \log \log n)$ time using part (A). The other non-trivial steps are Lines 12 and 14, where we check whether v lies on the s_j -to- y_j path, or strictly to the right of the $(s_j, \dots, y_j, y_{j-1}, \dots, s_{j-1})$ chord. Lemma 6.2 says that these queries can also be answered in $O(\kappa \log \log n)$ time, if they are also given the boundary vertices $\hat{x}_0, \hat{x}_1, \hat{x}_2 \in \partial R_{t+1}$, which are stored in part (D). Thus, the overall time for **SimpleCentroidSearch** (including recursive calls) is $O(\kappa \log n \log \log n)$. \square

6.2 Chords and Pieces

We begin by defining the key concepts of our point location method: *chords*, *laminar chord sets*, *pieces*, and the *occludes* relation.

Definition 6.1. (Chords) Fix an R in the \vec{r} -division and two vertices $c_0, c_1 \in \partial R$. An oriented simple path $\overrightarrow{c_0 c_1}$ is a *chord* of R^{out} if it is contained in R^{out} and is internally vertex-disjoint from ∂R . When the orientation is irrelevant we write it as $\overline{c_0 c_1}$.

⁹A possible implementation is to choose an original edge e' on ∂R_{t+1} incident to \hat{x}_1 as a proxy of the virtual shortcut edge (\hat{x}_1, \hat{x}_0) , and determine the relationship by the circular order of e_v, e_w, e' around \hat{x}_1 .

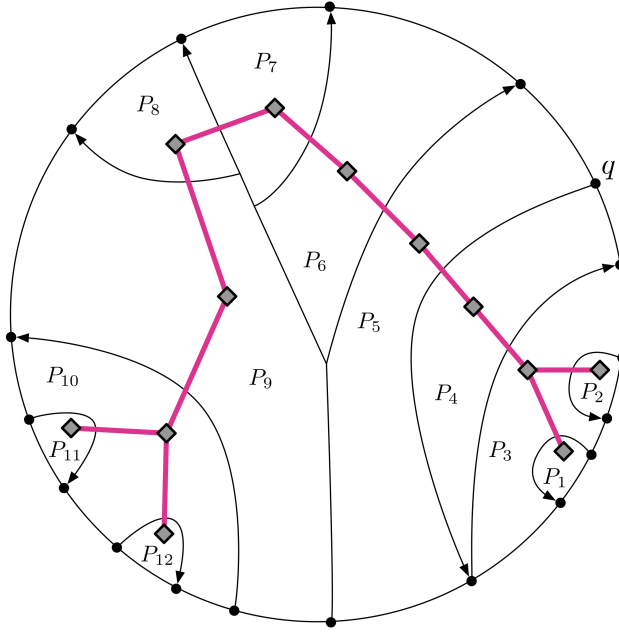


Fig. 9. A lamina set of chords partition R^{out} into pieces. Observe that the chords separating pieces P_5 – P_9 overlap in certain prefixes. The piece tree is indicated by diamond vertices and pink edges. Note that two pieces (e.g. P_5 and P_9) may share a boundary, but not be *adjacent*.

Definition 6.2. (Laminar Chord Sets) A set of chords C for R^{out} is *laminar* (non-crossing) if for any two such chords $C = \overrightarrow{c_0c_1}$, $C' = \overrightarrow{c_2c_3}$, if there exists a $v \in (C \cap C') \setminus \partial R$ then the subpaths from c_0 to v and from c_2 to v are identical; in particular $c_0 = c_2$ in this case.

The orientation of chords does not always coincide with a natural orientation of paths defined by the algorithm. For example, in Figure 6, the oriented chord $\overrightarrow{s_0s_2} = (s_0, \dots, y_0, y_2, \dots, s_2)$ is composed of three parts: a shortest s_0 -to- 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 -to- y_2 path (whose natural orientation is the reverse of its orientation in $\overrightarrow{s_0s_2}$). The orientation serves two purposes. In Definition 6.1 we can speak unambiguously about the parts of R^{out} to the *right* and *left* of $\overrightarrow{s_0s_2}$. In Definition 6.2 the role of the orientation is to ensure that the partition of R^{out} into *pieces* induced by C can be represented by a *tree*, as we show in Lemma 6.4.

Definition 6.3. (Pieces) A laminar chord set C for R^{out} partitions the faces of R^{out} into pieces, excluding the face on ∂R . Two faces f, g are in the same piece iff f^* and g^* are connected by a path in $(R^{\text{out}})^*$ that avoids duals of edges in C and of edges along the boundary cycle on ∂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 9.

Lemma 6.4. Suppose that C is a laminar chord set for R^{out} , $\mathcal{P} = \mathcal{P}(C)$ is the corresponding piece set and \mathcal{E} are the 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 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 that \mathcal{T} is a tree.

Let $C = \overrightarrow{c_0 c_1} \in C$ 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 9 the chords bounding P_1, P_2, P_{11}, P_{12} would be eligible to be C .) Let $C = (c_0 = v_0, v_1, v_2, \dots, v_k = c_1)$ and v_{j^*} be such that the edges of the suffix (v_{j^*}, \dots, v_k) are on no other chord, meaning the vertices $\{v_{j^*+1}, \dots, v_{k-1}\}$ 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 $(R^{\text{out}})^*$ that avoids the duals of all edges in C and along ∂R . All pieces adjacent to P contain some face among $\{g_{j^*}, \dots, g_{k-1}\}$, 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 ∂R . Obtain the “new” R^{out} by cutting along C and removing P , the new ∂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 that \mathcal{T} is a tree. \square

Definition 6.4. (Occluding Chords; Maximal Chords) Fix R^{out} , chord C , and two faces f, g , neither of which is the hole defined by ∂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 $C' \in C$ 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 6.4 that if C is laminar, the maximal chords with respect to f will intersect 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, which itself may be a face, a vertex, or a piece. Specifically, we can say that from some vantage, C occludes an *interval* of the boundary cycle on ∂R , say according to a clockwise traversal around the hole on ∂R in R^{out} .¹⁰ This will be used in the **ChordIndicator** procedure of Section 6.4.2.

We next present part (E) of our data structure, which will be used to implement the functions **SitePathIndicator** and **ChordIndicator**.

- (E) (**Chord Trees; Piece Trees**) For each $i \in [1, m-1]$, each $R_i \in \mathcal{R}_i$, and each source $q \in \partial R_i$, we store the SSSP tree from q with respect to G induced by ∂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 ∂R_i . Every edge of $T_q^{R_i}$ is designated a *chord* if the corresponding path is entirely contained in R_i^{out} , 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^{out} into pieces, and $\mathcal{T}_q^{R_i}$ the corresponding piece tree. Define $T_q^{R_i}[x]$ for $x \in \partial R_i$ 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 in $C_q^{R_i}[x]$ (which, if it exists, is unique) that occludes P' from vantage P .

¹⁰This is one place where we rely on the fact that each hole is bounded by a *simple* cycle.

AdjacentPiece(R_i, q, e): Here e is an edge on the boundary cycle on ∂R_i . Return the unique piece in $\mathcal{P}_q^{R_i}$ with e on its boundary.¹¹

We next describe how to compactly store Part (E) of the data structure. 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 $O(\log |\partial R_i| / \log \log |\partial R_i|)$ 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 [21]) we can answer **MaximalChord** queries relative to $C_q^{R_i}[x]$ in $O(\log |\partial R_i|)$ time.

We will make use of a data structure of Brodal et al. [7] specified in the following lemma.

Lemma 6.5. (Cf. Brodal et al. [7, Theorem 2]) *For an edge-weighted tree with k nodes, there exists a data structure that occupies $O(k)$ space and supports the following operations in $O(\log k / \log \log k)$ time.*

- **Update**(e, w): Change the weight of an edge e to w .
- **Pathmin/Pathmax**(u, v): Given two nodes u and v , return the edge with minimum/maximum weight on the path between u and v .

Lemma 6.6. *Part (E) of the data structure can be stored in $O(mn \log n / \log \log n)$ total space and answer **MaximalChord** queries in $O(\log n)$ time and **AdjacentPiece** queries in $O(1)$ time.*

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

- **Mark/Unmark**(e): Mark/unmark an edge $e \in E(\mathcal{T})$.
- **LastMarked**(P', P): Return the marked edge closest to P on the path from P' to P , or **Null** if all are unmarked.

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

The operations **Mark**, **Unmark**, and **LastMarked** are easily reducible to **Update**, **Pathmin**, and **Pathmax** from Lemma 6.5 [7]. Root the tree at an arbitrary vertex and preprocess it for LCA queries [4]. All unmarked edges carry a weight of $+\infty$ (for **Pathmin** queries) and $-\infty$ (for **Pathmax** queries). **Mark**(e) sets the weight of e to be equal to the number of edges of the path from the root to e 's farthest endpoint from the root. Consider a **LastMarked**(P', P) query and let P'' be the lowest common ancestor of P and P' . We find the edges $e_0 = \mathbf{Pathmin}(P', P'')$ and $e_1 = \mathbf{Pathmax}(P, P'')$. If e_1 exists ($P \neq P''$) and is marked (weight not $\pm\infty$) then it is the correct answer. Otherwise, if e_0 is marked then it is the correct answer. If neither case holds then there are no marked edges on the path from P' to P .

¹¹This is another place where we rely on the fact that every hole is bounded by a simple cycle.

For fixed R_i and $q \in \partial R_i$ there are $O(|\partial R_i|)$ **Mark** and **Unmark** operations, each of which takes $O(\log n / \log \log n)$ time. Over all choices of i , R_i , and q the total update time is $O(mn \log n / \log \log n)$. After applying generic persistence transformation for RAM data structures (see [21]) the space is $O(mn \log n / \log \log n)$ and the time per **LastMarked** query is $O(\log n / \log \log n \cdot \log \log n) = O(\log n)$.

Turning to **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 occupying $O(mn)$ space. \square

6.3 The SitePathIndicator Function

The **SitePathIndicator** function is relatively simple. We are given $\text{VD}_{\text{out}}^*(u_i, R_{i+1})$, $v \in R_{i+1}^{\text{out}}$, a centroid $f^* \in R_{i+1}^{\text{out}}$, 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 $v \in R_{t+1}$.

Using the lookup tables in part (D) of the data structure, we find the first and last vertices (q and x) of ∂R_t on the s_j -to- y_j path. If q, x do not exist then v is certainly not on the s_j -to- y_j path (Line 4). Using parts (A,C,D) of the data structure, we invoke **SimpleCentroidSearch** to find the last point z of ∂R_t on the shortest path (in G) from q to v . (See Lemma 6.3.) If z is not on the path from q to x in G (which corresponds to it not being on the path from q to x in $T_q^{R_t}$, stored in Part (E)), then once again v is certainly not on the s_j -to- y_j path (Line 8). So we may assume that z lies on the q -to- x path. For the case where $z = x$, we let x' be the last vertex of the shortest s_j -to- y_j path that is contained in the relevant subgraph $R_t^{\text{out}} \cap R_{t+1}$. In particular, there are three cases to consider, depending on whether the destination y_j of the path is in $R_t^{\text{out}} \cap R_{t+1}$, in R_{t+1}^{out} , or in R_t . If $y_j \in R_t^{\text{out}} \cap R_{t+1}$ we let $x' = y_j$; if $y_j \in R_{t+1}^{\text{out}}$ we let x' be the last vertex of ∂R_{t+1} encountered on the shortest s_j -to- y_j path (stored in part (D)); and if $y_j \in R_t$ we let $x' = x$. Figure 10(a,b) illustrates the first two possibilities for x' . Now, v is on the s_j -to- y_j path iff it is on the x -to- x' shortest path, which can be answered using part (A) of the data structure (Lines 19, 21). (Figure 10(b) illustrates one way for v to appear on the x -to- x' path.) In the remaining case, z is on the shortest q -to- x path but is not x , meaning that the child z' of z on $T_q^{R_t}[x]$ is well defined. If the corresponding shortest z -to- z' path lies in R_t^{out} (i.e., it is a chord $\overrightarrow{zz'}$), then v is on the shortest s_j -to- y_j path iff it is on the shortest z -to- z' path in R_t^{out} , which, once again, can be answered with part (A) of the data structure via Lemma 6.1 (Lines 25, 27). See Figure 10(a) for an illustration of this case. Finally, if the shortest z -to- z' path is internally disjoint from R_t^{out} , then v is clearly not on the shortest s_j -to- y_j path.

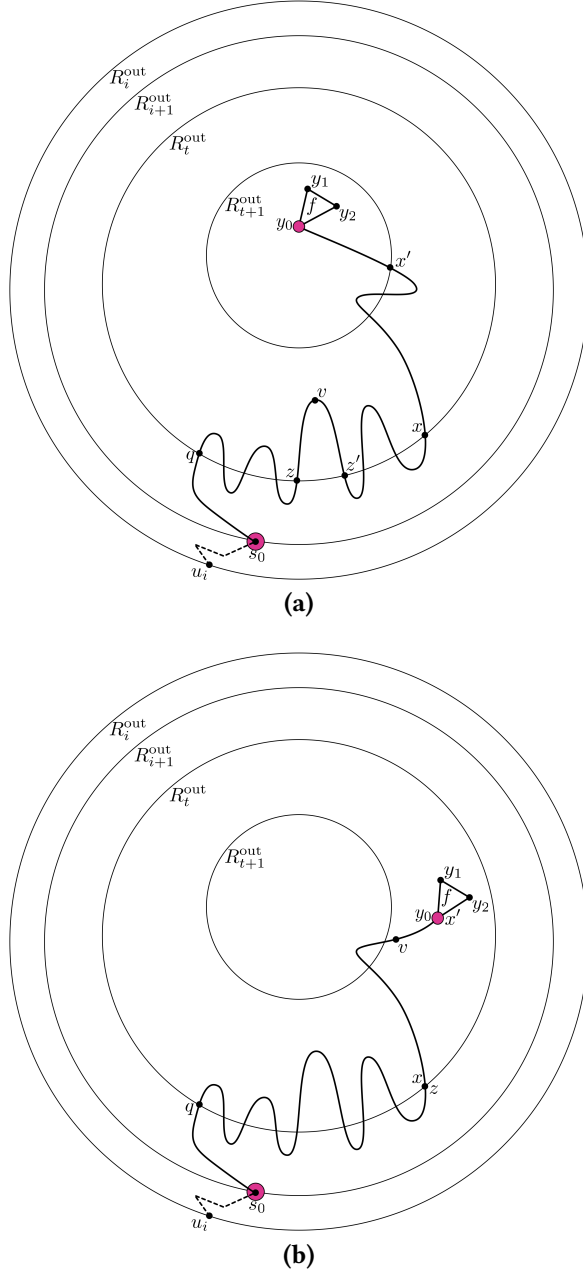


Fig. 10. **(a)** f is in R_{t+1}^{out} and x' is the last vertex on $\partial R_{t+1}^{\text{out}}$ on the s_j -to- y_j path. Since $z \neq x$ and $z \in T_q^{R_t}[x]$ the subpath from z to z' is a chord in R_t^{out} , and so we test whether v is on the chord $\overrightarrow{zz'}$. **(b)** f is in $R_t^{\text{out}} \cap R_{t+1}^{\text{out}}$ and $x' = y_j$. Since $z = x$ we test whether v is on the x -to- y_j path.

Algorithm 5 SitePathIndicator($\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, f^*, j$)

Input: The dual representation $\text{VD}^* = \text{VD}_{\text{out}}^*(u_i, R_{i+1})$ of a Voronoi diagram, a vertex $v \in R_{i+1}^{\text{out}}$, and $j \in \{0, 1, 2\}$.

Output: **True** if v is on s_j -to- y_j shortest path, where s_j, y_j are with respect to f^* in VD^* , and **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  $s_j$ -to- $y_j$  path. ▷ Part (D)
3: if  $q, x$  are Null then
4:   return False
5: end if
6:  $z \leftarrow \text{SimpleCentroidSearch}(\text{VD}_{\text{out}}^*(q, R_t), v)$  ▷ Uses parts (A,C,D); see Lemma 6.3
7: if  $z$  is not on  $T_q^{R_t}[x]$  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 the shortest  $s_j$ -to- $y_j$  path. ▷ Part (D)
15:   else
16:      $x' \leftarrow x$  ▷ I.e.,  $y_j \notin R_t^{\text{out}}$ 
17:   end if
18:   if  $v$  is on the shortest  $x$ -to- $x'$  path then ▷ Part (A)
19:     return True
20:   end if
21:   return False
22: end if
23:  $z' \leftarrow$  the child of  $z$  on  $T_q^{R_t}[x]$  ▷ Part (E)
24: if  $\overrightarrow{zz'}$  is a chord in  $C_q^{R_t}[x]$  and  $v$  is on the shortest  $z$ -to- $z'$  path in  $R_t^{\text{out}}$  then ▷ Part (A)
25:   return True
26: end if
27: return False

```

6.4 The ChordIndicator Function

The **ChordIndicator** function is given $\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v \in R_{i+1}^{\text{out}}$, 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

$$\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}},$$

which is composed of the shortest s_j -to- y_j and s_{j-1} -to- y_{j-1} paths, and the single edge $\{y_j, y_{j-1}\}$. Note that \tilde{C} is a simple path since the shortest s_j -to- y_j and s_{j-1} -to- y_{j-1} paths belong to different Voronoi cells. See Figure 6 for an illustration. It is guaranteed that v does not lie on \tilde{C} , as this case is already handled by the **SitePathIndicator** function.

Figure 11 illustrates why this point location problem is so difficult. Since we know that $v \in R_{t+1}$ and $v \notin R_t$, we can narrow our attention to $R_t^{\text{out}} \cap R_{t+1}$. However the projection of \tilde{C} onto R_t^{out} can cross the boundary ∂R_t an arbitrary number of times. Define C to be the set of oriented chords of R_t^{out} obtained by projecting \tilde{C} onto R_t^{out} .

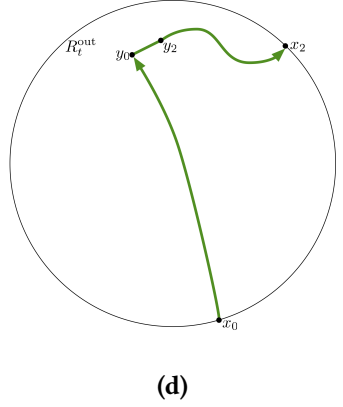
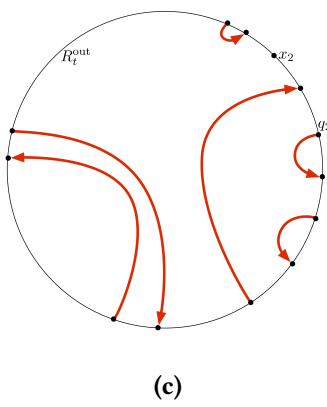
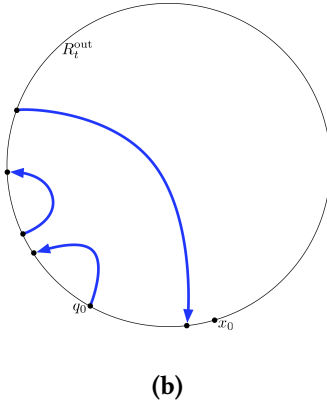
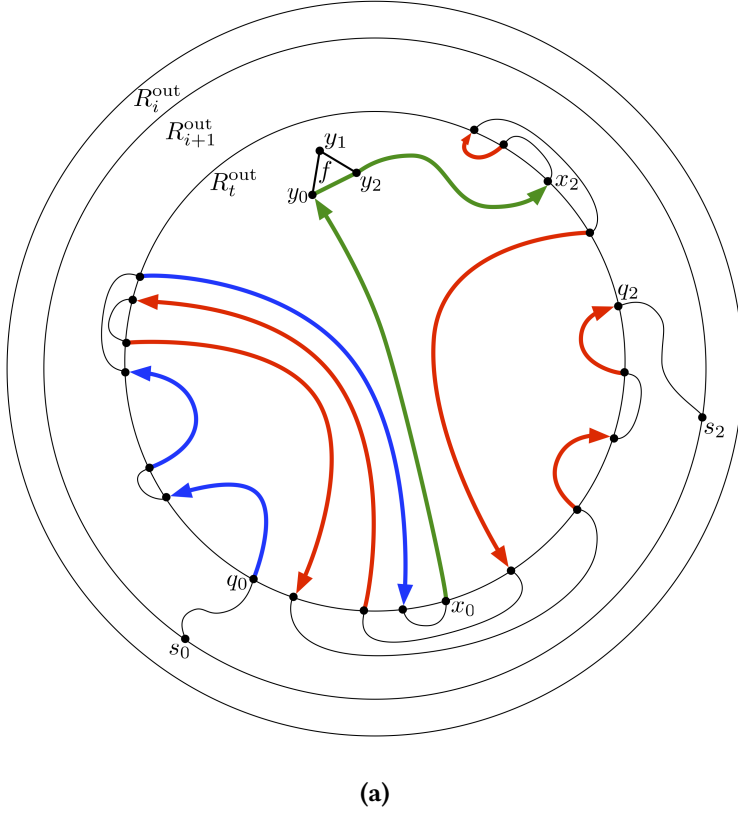


Fig. 11. **(a)** The projection of a site-centroid-site chord $\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}}$ of R_{i+1}^{out} onto R_t^{out} yields a set C of chords of R_t^{out} , partitioned into three classes. Let q_j, x_j and q_{j-1}, x_{j-1} be the first and last ∂R_t -vertices on the s_j -to- y_j and s_{j-1} -to- y_{j-1} paths. **(b)** C_1 : all chords in $T_{q_j}^{R_t}[x_j]$. **(c)** C_2 : all chords in $T_{q_{j-1}}^{R_t}[x_{j-1}]$. Their orientation is the reverse of their counterparts in \tilde{C} . **(d)** C_3 : the single chord $\overrightarrow{x_j y_j y_{j-1} x_{j-1}}$.

Luckily C has some structure. Let (q_j, x_j) and (q_{j-1}, x_{j-1}) be the first and last ∂R_t vertices on the shortest s_j -to- y_j and s_{j-1} -to- 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.

- C_1 : Define $C_1 = C_{q_j}^{R_t}[x_j]$. That is, C_1 contains all the chords on the path from q_j to x_j , stored in part (E) of the data structure. Moreover, the orientation of C_1 agrees with the orientation of \tilde{C} . The blue chords of Figure 11(a) are isolated as C_1 in Figure 11(b).
- C_2 : Define $C_2 = C_{q_{j-1}}^{R_t}[x_{j-1}]$. That is, C_2 contains all the chords on the path from q_{j-1} to x_{j-1} . The red chords of C in Figure 11(a) are *represented* by chords C_2 , but with reversed orientation. Figure 11(c) depicts C_2 .
- C_3 : This set contains the oriented chord $\overrightarrow{x_j x_{j-1}}$ (if it exists) consisting of the shortest x_j -to- y_j path, the edge $\{y_j, y_{j-1}\}$, and the reverse of the shortest x_{j-1} -to- y_{j-1} path. Figure 11(d) depicts C_3 .

The chord-set C partitions R_t^{out} into a piece-set \mathcal{P} , with one such piece $P \in \mathcal{P}$ containing v . (Remember that v is not on \tilde{C} .) We can also consider the piece-sets $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ generated by C_1, C_2, C_3 . Let $P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2, P_3 \in \mathcal{P}_3$ be the pieces containing v . Since, ignoring orientation, $C = C_1 \cup C_2 \cup C_3$, it must be that $P = P_1 \cap P_2 \cap P_3$. In order to determine whether v is to the right of \tilde{C} , it suffices to find some chord $C \in C$ bounding P and ask whether v is to the right of C . Note that such a chord C must also be on the boundary of one of P_1, P_2 , or P_3 .

The high-level strategy of **ChordIndicator** is as follows. First, we will find some piece $P'_1 \in \mathcal{P}_{q_j}^{R_t}$ that is contained in P_1 using the procedure **PieceSearch** described below. The chords of C_1 bounding P_1 are precisely the *maximal* chords in C_1 from vantage P'_1 . Using **MaximalChord** (part (E)) we will find a candidate chord $C_1 \in C_1$, and one edge e on the boundary cycle of ∂R_t occluded by C_1 from vantage P'_1 . Turning to C_2 , we use **AdjacentPiece** to find the piece $P_e \in \mathcal{P}_{q_{j-1}}^{R_t}$ adjacent to e . Then, using **PieceSearch** and **MaximalChord** again, we find a $P'_2 \in \mathcal{P}_{q_{j-1}}^{R_t}$ 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 , if any. We determine an “eligible” chord $C_\ell \in \{C_1, C_2, C_3\}$, decide whether v lies to the right of C_ℓ , and return this answer if $\ell \in \{1, 3\}$ or reverse it if $\ell = 2$. Recall that chords in C_2 have the opposite orientation as their counterparts in C .

PieceSearch is presented in Section 6.4.1 and **ChordIndicator** in Section 6.4.2.

6.4.1 PieceSearch. Given v and $q, x \in \partial R_t$, we would like to locate the piece $P \in \mathcal{P}_q^{R_t}[x]$ that contains v . Note that since $\mathcal{P}_q^{R_t}$ is a refinement of $\mathcal{P}_q^{R_t}[x]$, P is the union of some pieces in $\mathcal{P}_q^{R_t}$. Thus, it suffices to return any $P' \in \mathcal{P}_q^{R_t}$ such that $P' \subseteq P$. The procedure **PieceSearch** performs this task.

The first thing it does is find the *last* ∂R_t vertex z on the shortest path from q to v , which can be done with a call to **SimpleCentroidSearch** on $\text{VD}_{\text{out}}^*(q, R_t)$, using Lemma 6.3. (This uses parts (A,C,D) of the data structure.) The shortest path from z to v cannot cross any chord in $C_q^{R_t}[x]$, since they are part of a shortest path, but it can coincide with a prefix of some chord in $C_q^{R_t}[x]$. Thus, if no chord of $C_q^{R_t}[x]$ 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 $C_q^{R_t}$. This case is depicted in Figure 12. When $z = z_0$, we know that $v \in P_5 \cup \dots \cup P_9$ and return any such piece containing z .) In general z may be incident to up to two chords $C_1, C_2 \in C_q^{R_t}[x]$. (This occurs when the shortest q -to- x path touches ∂R_t at z without leaving R_t^{out} .) In this case we determine which side of C_1 and C_2 v is on (using Lemma 6.1) and return the appropriate piece adjacent to C_1 or C_2 . This case is depicted in Figure 12 with $z = z_1$; the three possible answers coincide with $v \in \{v_1, v_2, v_3\}$.

Algorithm 6 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: Any piece $P' \in \mathcal{P}_q^{R_t}$ that is a subpiece of the unique piece $P \in \mathcal{P}_q^{R_t}[x]$ containing v .

- 1: $z \leftarrow \text{SimpleCentroidSearch}(\text{VD}_{\text{out}}^*(q, R_t), v)$ ▷ Uses parts (A,C,D) of the data structure
- 2: **if** z is not an endpoint of any chord in $C_q^{R_t}[x]$ **then**
- 3: **return** any piece in $\mathcal{P}_q^{R_t}$ containing z .
- 4: **end if**
- 5: $C_1, C_2 \leftarrow$ two chords in $C_q^{R_t}[x]$ adjacent to z (C_2 may be **Null**)
- 6: Determine whether v is to the left or right of C_1 and C_2 . ▷ Part (A); see Lemma 6.1
- 7: **return** a piece adjacent to C_1 or C_2 that respects the queries of Line 6.

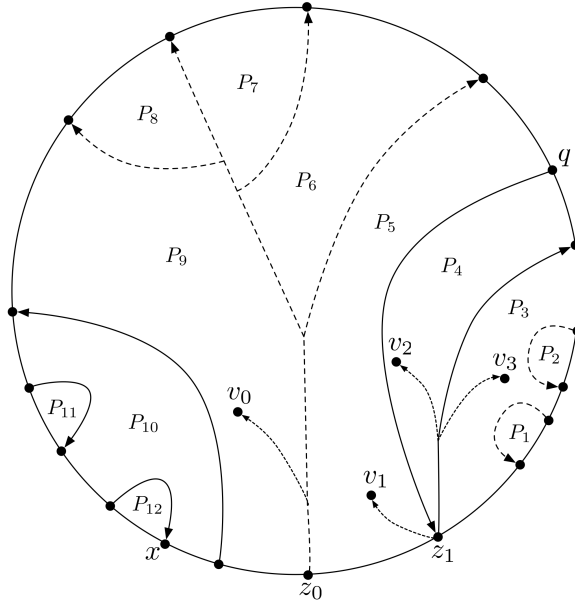


Fig. 12. Solid chords are in $C_q^{R_t}[x]$. Dashed chords are in $C_q^{R_t}$ but not $C_q^{R_t}[x]$. When $z = z_0, v = v_0$, the piece in $\mathcal{P}_q^{R_t}[x]$ containing v is the union of P_5 – P_9 . **PieceSearch** reports any piece containing z_0 . When $z = z_1, v \in \{v_1, v_2, v_3\}$, z is incident to two chords C_1, C_2 . **PieceSearch** decides which side of C_1, C_2 v is on (see Lemma 6.1), and returns the appropriate piece adjacent to C_1 or C_2 .

We remark that we could have defined **PieceSearch** to not take x as an argument, and just return a piece $P' \in \mathcal{P}_q^{R_t}$ containing v , which is, by definition, a subpiece of the piece $P \in \mathcal{P}_q^{R_t}[x]$ containing v . This would entail modifying Lines 5–6 to do a binary search on all the chords in $C_q^{R_t}$ incident to z .

6.4.2 ChordIndicator. Let us walk through the **ChordIndicator** function. If $\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}}$ does not touch the interior of R_t^{out} then the left-right relationship between \tilde{C} and $v \notin R_t$ is known, and stored in part (D) 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 $C_3 = \overrightarrow{x_j y_j y_{j-1} x_{j-1}}$. We apply Lemma 6.2 to determine whether v is to the right or left of C_3 and

return this answer (Line 8). Thus, without loss of generality we can assume that $C_1 \neq \emptyset$ and C_2 may or may not be empty.

Recall that P_1 is v 's piece in $\mathcal{P}_{q_j}^{R_t}[x_j]$. Using **PieceSearch** we find a piece $P'_1 \subseteq P_1$ in the more refined partition $\mathcal{P}_{q_j}^{R_t}$ and find a **MaximalChord** $C_1 \in C_1$ from vantage P'_1 , and hence from vantage v as well. We regard ∂R_t as circularly ordered according to a clockwise walk around the hole on ∂R_t in R_t^{out} . The chord C_1 occludes an interval I_1 of ∂R_t from vantage v . If C_1 is *not* one of the chords bounding P , then C_3 or some $C_2 \in C_2$ must occlude a superset I_2 of I_1 , 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 **AdjacentPiece** we find the unique piece $P_e \in \mathcal{P}_{q_{j-1}}^{R_t}$ with e on its boundary. Using **PieceSearch** again we find $P'_2 \in \mathcal{P}_{q_{j-1}}^{R_t}$ contained in P_e , and using **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 ∂R_t . (If C_2 is **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 $C_3 \in C_3$ exists and occludes a superset of I_1 and of I_2 on the boundary ∂R_t . We check whether v lies to the right or left of C_3 using Lemma 6.2 and let I_3 be the interval of ∂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 and C_2 . Thus, we find the chord $C_\ell \in \{C_1, C_2, C_3\}$ that covers e and maximizes $|I_\ell|$. C_ℓ must be on the boundary of P , so the left-right relationship between v and \tilde{C} is exactly the same as the left-right relationship between v and C_ℓ , 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 \tilde{C} . Figure 13 illustrates how ℓ could take on all three values.

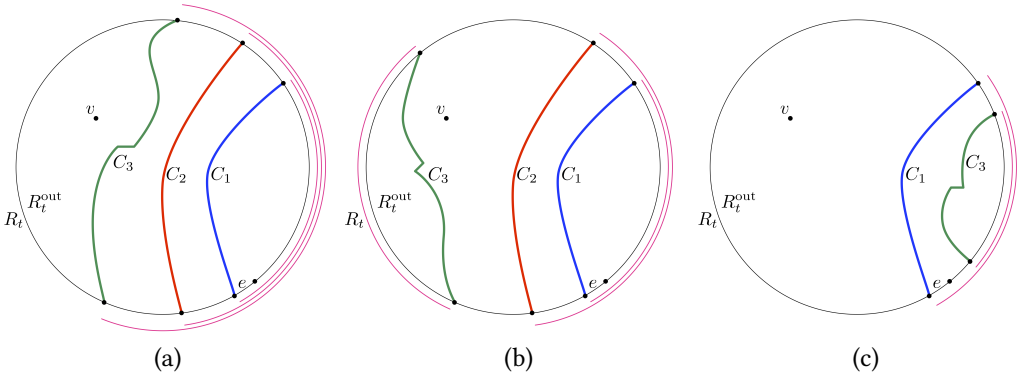


Fig. 13. The intervals I_1, I_2, I_3 are represented as pink circular arcs. The edge e is the first edge of I_1 in a clockwise walk around the hole bounded by ∂R_t in R_t^{out} . (Note that in this drawing the hole on ∂R_t is the infinite face. Thus, a clockwise walk around ∂R_t *looks like* a counter-clockwise walk in the plane.) In (a) C_2 exists and C_3 is eligible since $I_3 \supset I_2 \supset I_1$. In (b) C_2 exists, but C_3 occludes an interval I_3 that does not contain e , so C_2 is an eligible chord. In (c) C_2 is **Null**, and C_3 does not occlude e from v , so C_1 is the only eligible chord. (In the figure $I_3 \subset I_1$ but it could also be as in (b), with I_3 disjoint from I_1 .)

Algorithm 7 $\text{ChordIndicator}(\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v, f^*, j)$

Input: The dual representation $\text{VD}_{\text{out}}^* = \text{VD}_{\text{out}}^*(u_i, R_{i+1})$ of a Voronoi diagram, a centroid f^* in VD_{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 $v \in R_{i+1}^{\text{out}}$ that does not lie on the site-centroid-site chord $\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}}$.

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

- 1: $R_t \leftarrow$ the ancestor of R_i s.t. $v \notin R_t, v \in R_{t+1}$. C is the projection of \tilde{C} onto R_t^{out} .
- 2: **if** the left/right relationship between R_t^{out} and $\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}}$ is known **then**
- 3: **return** stored **True/False** answer. ▷ Part (D)
- 4: **end if** ▷ (It follows that \tilde{C} crosses ∂R_t and that $C \neq \emptyset$)
- 5: $(q_j, x_j) \leftarrow$ first and last ∂R_t -vertices on shortest s_j -to- y_j path. ▷ Part (D)
- 6: $(q_{j-1}, x_{j-1}) \leftarrow$ first and last ∂R_t -vertices on shortest s_{j-1} -to- y_{j-1} path. ▷ Part (D)
- 7: **if** $C_1 = C_2 = \emptyset$ **then**
- 8: **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. ▷ Parts (A,D)
- 9: **end if** ▷ W.l.o.g., continue under the assumption that $C_1 \neq \emptyset$.
- 10: $P'_1 \leftarrow \text{PieceSearch}(R_t, q_j, x_j, v)$ ▷ Parts (A,C,D)
- 11: $C_1 \leftarrow \text{MaximalChord}(R_t, q_j, x_j, P'_1, \perp)$ ▷ Part (E)
- 12: $I_1 \leftarrow$ the clockwise interval of hole ∂R_t occluded by C_1 from vantage v .
- 13: $e \leftarrow$ edge joining first two elements of I_1 .
- 14: $P_e \leftarrow \text{AdjacentPiece}(R_t, q_{j-1}, e)$ ▷ Part (E)
- 15: $P'_2 \leftarrow \text{PieceSearch}(R_t, q_{j-1}, x_{j-1}, v)$ ▷ Parts (A,C,D)
- 16: $C_2 \leftarrow \text{MaximalChord}(R_t, q_{j-1}, x_{j-1}, P'_2, P_e)$ ▷ Part (E); may return **Null**
- 17: $I_2 \leftarrow$ the clockwise interval of hole ∂R_t occluded by C_2 from vantage v . ▷ \emptyset if $C_2 = \text{Null}$
- 18: $C_3 \leftarrow$ single chord in C_3 , if any. ▷ May be **Null**
- 19: $I_3 \leftarrow$ the clockwise interval of hole ∂R_t occluded by C_3 from vantage v . ▷ Parts (A,D)
- 20: $\ell \leftarrow$ index such that I_ℓ covers e , and $|I_\ell|$ is maximum.
- 21: **if** v is to the right of C_ℓ and $\ell \in \{1, 3\}$ or v is to the left of C_ℓ and $\ell = 2$ **then**
- 22: **return True**
- 23: **end if**
- 24: **return False**

7 ANALYSIS

This section constitutes a proof of the claims of Theorem 1.1 concerning space complexity and query time. See Section 8 for an efficient construction and its analysis.

The cost of **PieceSearch** is dominated by the call to **SimpleCentroidSearch** in Line 1, which, by Lemma 6.3, takes $O(\kappa \log n \log \log n)$ time. **SitePathIndicator** is also dominated by one call to **SimpleCentroidSearch**. (Its other operations are handled by the MSSP structure (part (A)) and various $O(1)$ -time tree operations on $T_q^{R_i}$ and the \tilde{r} -division such as lowest common ancestors and level ancestors [4, 5, 35, 37].) It also takes $O(\kappa \log n \log \log n)$ time. The calls to **MaximalChord** and **AdjacentPiece** in **ChordIndicator** take $O(\log n)$ time by Lemma 6.6, and testing which side of a chord v lies on takes $O(\kappa \log \log n)$ time by Lemmas 6.1 and 6.2. The bottleneck in **ChordIndicator** is still **PieceSearch**; overall it takes $O(\kappa \log n \log \log n)$ time.

An initial call to **CentroidSearch** (Line 4 of **Dist**) generates at most $\log n$ recursive calls to **CentroidSearch** in total, culminating in the last recursive call making 1 or 2 calls to **Dist** with the “ i ” parameter incremented. Excluding the cost of recursive calls to **Dist**, the cost of **CentroidSearch** is dominated by calls to **SitePathIndicator** and **ChordIndicator**, i.e., an initial

call to **CentroidSearch** takes $\log n \cdot O(\kappa \log n \log \log n) = O(\kappa \log^2 n \log \log n)$ time. Let $T(i)$ be the cost of a call to **Dist**(u_i, v, R_i). We have

$$\begin{aligned} T(m-1) &= O(\kappa \log \log n) & \text{Dist returns at Line 2 with one MSSP query} \\ T(i) &= 2T(i+1) + O(\kappa \log^2 n \log \log n) \end{aligned}$$

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. (A) is $O(\kappa m n^{1+1/m+1/\kappa})$ by Lemma 2.1 and the fact that $r_{i+1}/r_i = n^{1/m}$. (B) is $O(m n^{1+1/(2m)})$ since $\sqrt{r_{i+1}/r_i} = n^{1/(2m)}$. (C) is $O(mn)$ since $\sum_i n/r_i \cdot (\sqrt{r_i})^2 = O(mn)$. (D) is $O(m)$ times the space cost of (B) and (C), namely $O(m^2 n^{1+1/(2m)})$, and (E) is $O(mn \log n / \log \log n)$ by Lemma 6.6. For the choices of m, κ considered below, the bottleneck is (A).

We now explain how m, κ can be selected to achieve the extreme space and query complexities claimed in 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

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

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

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

and the query time

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

Note that once $\kappa = \Omega(\log n)$ it is best to switch to the pointer-based MSSP implementation (see Lemma 2.1 and [25]), which saves a $\log \log n$ -factor in the query time.

7.1 Speeding Up the Query Time

Considering functions that are $\omega(1)$ and $o(\log \log n)$ is of purely theoretical nature, so in practice m and κ will just be set as constants. In this section we illustrate how the query time's dependence on m can be improved from 2^m to about $2^{m/4}$.

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

(B') (**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

$$\begin{aligned} \text{VD}_{\text{out}}^*(q, R_{i+1}) &= \text{VD}^*[R_{i+1}^{\text{out}}, \partial R_{i+1}, \omega] \\ \text{VD}_{\text{farout}}^*(q, R_{i+4}) &= \text{VD}^*[R_{i+4}^{\text{out}}, \partial R_{i+4}, \omega] \end{aligned} \quad \text{only if } i < m-4$$

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

Now the space for (A) is $\tilde{O}(n^{1+1/m+1/\kappa}) = \tilde{O}(n^{1+2/m})$ is balanced with (B') when $m = k$. 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 $v \notin R_{i+1}$ but $v \in R_{i+4}$ we proceed as usual, calling **CentroidSearch**($\text{VD}_{\text{out}}^*(u_i, R_{i+1}), v$),

and if $v \notin R_{i+4}$ we call **CentroidSearch**($\text{VD}_{\text{farout}}^*(u_i, R_{i+4}), v$). 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 $O(m2^{m/4} \log^2 n \log \log n)$ with space $\tilde{O}(n^{1+2/m})$.

8 CONSTRUCTION

In this section, we show how to construct our oracle in $n^{3/2+o(1)}$ time. We use dense distance graphs. The *dense distance graph* of a region R , denoted $\text{DDG}[R]$, is a complete directed graph on the vertices of ∂R , in which the length of edge (u, v) is $\text{dist}_R(u, v)$. We say that this kind of DDG is *internal* and, similarly, define the *external* DDG of a region R , denoted by $\text{DDG}[R^{\text{out}}]$, to be a complete directed graph on ∂R , in which the length of edge (u, v) is $\text{dist}_{R^{\text{out}}}(u, v)$.

The FR-Dijkstra algorithm [28] is an efficient implementation of Dijkstra's algorithm [23] on DDGs. In particular, it simulates the behavior of the heap in Dijkstra's algorithm without explicitly scanning every edge in the DDG. In fact, the FR-Dijkstra algorithm can run on a union of DDGs [28]. Moreover, it is shown in [6] that it is also compatible with a traditional implementation of Dijkstra's algorithm, in the following sense: 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, \dots, n_k vertices, respectively. The FR-Dijkstra algorithm can be implemented on H in $\tilde{O}(N)$ time, where $N = \sum_{i=0}^k n_i$ [28, 40, 55].

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 construction of DDGs and part (D). They are not stored in our oracle and are unrelated to the MSSP structures from part (A).

(F) (**More MSSP Structures**) For each $i \in [0, m-1]$, each $R_i \in \mathcal{R}_i$ with parent $R_{i+1} \in \mathcal{R}_{i+1}$, we build two MSSP structures for $R_i^{\text{out}} \cap R_{i+1}$ with sources on ∂R_i and ∂R_{i+1} , respectively, and an MSSP structure for R_i with sources on ∂R_i .

All these MSSP structures are constructed using Klein's MSSP algorithm [45] or the one in Appendix A.2 (with $\kappa = \log n$) in $\tilde{O}(\sum_i \frac{n}{r_i} r_{i+1}) = \tilde{O}(mn^{1+1/m})$ time.

We then compute, for each region R_i in the \tilde{r} -division, the internal DDG, the external DDG, and the DDG of $R_i^{\text{out}} \cap R_{i+1}$, denoted $\text{DDG}[R_i^{\text{out}} \cap R_{i+1}]$, defined as the complete graph with vertices $\partial R_i \cup \partial R_{i+1}$ and edge weights the distances in $R_i^{\text{out}} \cap R_{i+1}$. The internal DDG and $\text{DDG}[R_i^{\text{out}} \cap R_{i+1}]$ for each region R_i can be computed using the MSSP structures in part (F) in $\tilde{O}(r_i)$ and $\tilde{O}(r_{i+1})$ time respectively, thus in $\tilde{O}(\sum_i \frac{n}{r_i} (r_i + r_{i+1})) = \tilde{O}(mn^{1+1/m})$ time over all regions. To compute the external DDGs, we consider a top-down process on the \tilde{r} -division. The external DDG for R_i can be computed by running the FR-Dijkstra algorithm on the union of $\text{DDG}[R_{i+1}^{\text{out}}]$ and $\text{DDG}[R_i^{\text{out}} \cap R_{i+1}]$ sourced from each vertex in ∂R_i . The number of vertices in this union is $O(\sqrt{r_{i+1}})$, so computing $\text{DDG}[R_i^{\text{out}}]$ takes $\tilde{O}(\sqrt{r_i r_{i+1}})$ time, and the construction time over all external DDGs is $\tilde{O}(\sum_i \frac{n}{r_i} \sqrt{r_i r_{i+1}}) = \tilde{O}(mn^{1+1/(2m)})$. The total construction time for all DDGs is $\tilde{O}(mn^{1+1/m})$. (See [41] for a recent efficient algorithm for computing external DDGs.)

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 ∂R_i is obtained by contracting subpaths in R_{i+1}^{out} of the SSSP trees into single (shortcut) edges. In order to build the MSSP structure using dynamic trees, it suffices to compute the contracted shortest path tree for every source on ∂R_i and then compare the differences between the trees of two adjacent sources on ∂R_i .

For a single source on ∂R_i , the contracted shortest path tree can be computed with the FR-Dijkstra algorithm on the union of subgraph $R_i^{\text{out}} \cap R_{i+1}$ and $\text{DDG}[R_{i+1}^{\text{out}}]$ in time $\tilde{O}(r_{i+1})$. 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 $\tilde{O}((r_{i+1} + \sqrt{r_i r_{i+1}}) \kappa n^{1/\kappa})$ (See item (A) in the

beginning of Section 5). The total time to construct all MSSP structures is $\tilde{O}(\sum_i \frac{n}{r_i} (r_{i+1} \sqrt{r_i} + r_{i+1} \kappa n^{1/\kappa})) = \tilde{O}(n^{3/2+1/m} m + n^{1+1/\kappa+1/m} m \kappa)$.

Remark 1. Notice that in our MSSP structures for R_i^{out} , a contracted subpath should be internally disjoint from ∂R_{i+1} . However, the underlying shortest paths represented by edges in $\text{DDG}[R_{i+1}^{\text{out}}]$ may not satisfy this condition. To fix this problem, we subtract a small value from all edge weights in DDGs, so that shortest paths are not affected. With this perturbation, the path using the largest number of DDG edges will be preferred. In such a path, each edge of the DDG corresponds to a path that is internally disjoint from ∂R_{i+1} . This mechanism will also be used below.

Efficient construction of Voronoi diagrams. Explicitly computing the primal Voronoi diagram can be too expensive. We next show an efficient algorithm to compute the dual representation of a Voronoi diagram that we believe is of independent interest (see [16] for an application of this algorithm in a dynamic setting). Let us present the high-level idea of our algorithm. For conceptual simplicity, let us think of constructing $\text{VD}^*[R, \partial R, \omega]$ for a region R in the complete recursive binary decomposition tree of G , described in Section 2. Let \mathcal{P} consist of the two children of R in the recursive decomposition of G . Let u be a dummy vertex connected to ∂R with auxiliary edges (u, s) of length $\omega(s)$ for each $s \in \partial R$. We will run FR-Dijkstra from the dummy vertex u on the union of these auxiliary edges and $\text{DDG}[P]$ for $P \in \mathcal{P}$. We will show that we can then decide whether each $P \in \mathcal{P}$ contains a trichromatic face in $O(|\partial P|)$ time by looking at the restriction of the computed shortest paths tree to $\text{DDG}[P]$. We will isolate the trichromatic faces by iteratively replacing any piece containing such a face with its two sub-pieces and refining the shortest path tree accordingly.

THEOREM 8.1. *Suppose that we are given a complete recursive decomposition of a planar graph G of size n . After an $\tilde{O}(n)$ -time preprocessing, for any region R of the decomposition, we can construct $\text{VD}^*[H, \partial R, \omega]$ for $H \in \{R, R^{\text{out}}\}$ and arbitrary additive weights $\omega : \partial R \rightarrow \mathbb{R}_{\geq 0}$ in time $\tilde{O}(\sqrt{|H|} \cdot |\partial R|)$.*

PROOF. Our preprocessing of each region P in the recursive decomposition consists of computing $\text{DDG}[P]$ in $O((|P| + |\partial P|^2) \log |P|)$ time via MSSP. This requires $\tilde{O}(n)$ time in total.

For clarity, we assume that the additive weights are such that there are no empty Voronoi cells and only waive this assumption at the end of the proof.

Let K be the star with center u and leaves ∂R , such that the weight of edge (u, s) is $\omega(s)$. Consider a set \mathcal{P} of regions of the recursive decomposition that cover H , i.e., each edge in H belongs to at least one region in \mathcal{P} and no edge in $G \setminus H$ belongs to any region of \mathcal{P} . Let T be a shortest path tree rooted at u in the union of K and the DDGs of all pieces in \mathcal{P} . We shall next prove that, for each piece $P \in \mathcal{P}$, we can infer whether P contains a trichromatic face or not by inspecting the restriction of T to $\text{DDG}[P]$.

Our assumption on the additive weights guarantees that each vertex of ∂R is a child of u in T . We label each vertex v of T by its unique ancestor in T that belongs to ∂R . Note that the label of a vertex v corresponds to the Voronoi cell containing v in $\text{VD}[H, \partial R, \omega]$. For a piece $P \in \mathcal{P}$, consider the restriction of T to $\text{DDG}[P]$. We use a representation of size $O(|\partial P|)$ of the edges of T embedded as curves in P , such that each edge of T is homologous to its underlying shortest path in P . See [47, 53] for details on such a representation. We make incisions in the embedding of P along the edges of T (the endpoints of edges of T are duplicated in this process). Let \mathcal{Q} be the set of connected components of P after all incisions are made.

Claim 8.2. *P contains a trichromatic face if and only if some connected component C in \mathcal{Q} contains boundary vertices of P with at least three distinct labels.*

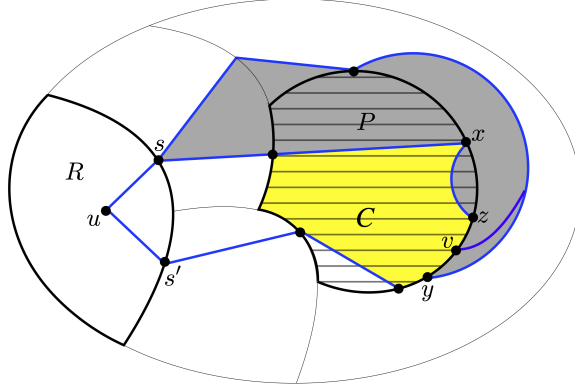


Fig. 14. Illustration for the proof of the claim in the case where $H = R^{\text{out}}$. Some pieces in a graph G are shown. Region R is shown in bold. Region P (bold boundary, horizontal stripes) lies outside R . The shortest path tree T is shown in blue, the connected component C in yellow, and the cycle D in gray. Vertices x and y have the same label s . The vertices v, z between x and y (on the cyclic walk F along the infinite face of C) must also be labeled s .

Intuitively, for each connected component C in \mathcal{Q} , each label appears as the label of boundary vertices along at most a single sequence of consecutive boundary vertices along the boundary of C . Then, since C is triangulated, apart perhaps from its infinite face, Sperner's lemma [62] directly implies that C contains a trichromatic face if and only if C has vertices with at least three distinct labels in its infinite face. Let us remark that the proof of Claim 8.2 does not rely on the single-hole assumption.

PROOF OF CLAIM 8.2. Let C be a connected component in \mathcal{Q} . First, note that each of the vertices of C belongs to the Voronoi cell of one of the sites that label the vertices in $C \cap \partial P$. Hence, if each $C \in \mathcal{Q}$ contains boundary vertices of P with at most two distinct labels, P cannot contain any trichromatic faces.

It thus suffices to show that, if some $C \in \mathcal{Q}$ contains boundary vertices of P with at least three distinct labels, then C (and P) contains a trichromatic face. Let us consider such a component C . Note that the vertices of ∂R either do not belong to C or they are incident to a single face f of C . In the former case, let f be the face of C such that ∂R is embedded in f . We think of f as the infinite face of C . Note that, because any path from ∂R to any vertex of C must intersect f , the set of labels of the vertices of f is identical to the set of labels of all of C .

We first claim that the vertices of f that have the same label are consecutive in the cyclic order of f . To see this, consider any two distinct vertices x, y of f that have the same label s . If the unique x -to- y path in T is a subpath of the boundary of f , then this is clearly the case. Otherwise, consider the (not necessarily simple) cycle D (in H) formed by the unique x -to- y path in T , and the x -to- y path F along the boundary of f , such that ∂R and C are on the same side of D . See Figure 14. By choice of D , the only vertex of ∂R that can be enclosed by D is s . Suppose, towards a contradiction, that some vertex v of F has label $s' \neq s$. Since D does not enclose s' , the s' -to- v path in T starts outside D . Further, it cannot cross the x -to- y path in T , all of whose vertices have the label s . Thus, the s' -to- v path in T must intersect C , and use an edge whose underlying shortest path is disjoint from f . But then C should have been further dissected when the incisions along T were performed, a contradiction.

The argument above established that the vertices of f that have the same label are consecutive in the cyclic order of f . Let us now recall Sperner's lemma.

Lemma 8.3 (Sperner's lemma). *Consider a planar graph J , such that each face is a triangle, apart perhaps from the infinite face g . Further, consider a vertex-coloring of J with colors $\{1, 2, 3\}$ that satisfies the following condition: there exist three vertices v_1, v_2, v_3 in g , colored 1, 2, 3, respectively, such that, for all $j \in \{1, 2, 3\}$, the vertices on the v_j -to- v_{j-1} path along g that does not contain v_{j-2} have a color in $\{j, j-1\}$ —indices here are modulo 3. Then, J contains a trichromatic face.*

Suppose that we have exactly three labels for the vertices of f . Since every face of C other than f is a triangle, and we can arbitrarily pick the v_i 's as the vertices of each label form a contiguous interval, a direct application of Sperner's lemma implies that there is a trichromatic face. If we have $k > 3$ colors, we can group $k-2$ of them that appear consecutively in f together, and apply Sperner's lemma to the new instance. This concludes the proof of Claim 8.2. \square

If $H = R^{\text{out}}$, we set \mathcal{P} to be the set of all siblings of pieces in the complete recursive decomposition tree that contain R . Else, $H = R$ and we set $\mathcal{P} = \{R\}$.

We repeat the following process ($O(\log n)$ times) until we locate all $O(|\partial R|)$ trichromatic faces.

Each iteration consists of two steps. In the first step we compute the shortest path tree T rooted at u in the union of K and the DDGs of all pieces in \mathcal{P} using FR-Dijkstra. In the second step we refine the set \mathcal{P} as follows. For each piece P , using Claim 8.2, we decide in $O(|\partial P|)$ time whether it contains any trichromatic face. If P does not contain a trichromatic face we do nothing. If it does, we remove P from \mathcal{P} and we insert to \mathcal{P} the two children of P , unless P is a leaf in the recursive decomposition, in which case we insert to \mathcal{P} the individual edges of P .

The tree structure of $\text{VD}[H, \partial R, \omega]$ is captured by the structure of the shortest path tree in the DDGs of all the pieces at the end of this process. The total time to locate all the trichromatic faces is proportional, up to polylogarithmic factors, to the total number of vertices in all of the DDGs involved in all these computations, which is bounded as follows.

Let R_H be the smallest piece in the complete recursive decomposition of G that contains H (if $H = R$ then $R_H = R$, and if $H = R^{\text{out}}$ then $R_H = G$). Note that $|R_H| = O(|H|)$. For the remainder of this proof, we use the term *decomposition tree* to refer to the subtree of the complete recursive decomposition tree rooted at R_H . Each DDG involved in the computation is either the DDG of a piece P in the decomposition tree that contains a trichromatic face, or the DDG of the sibling of such a piece P . There are $O(|\partial R|)$ trichromatic faces, and each contributes at most two DDGs at each level of the decomposition tree. It is well known (cf. [32, Lemma 3.1]) both that the sizes of pieces and the number of boundary vertices of pieces decrease geometrically as one descends down the decomposition tree. Hence, a naïve bound on the total number of boundary vertices (equivalently, DDG vertices) in all those pieces is $\tilde{O}(\sqrt{|H|} \cdot |\partial R|)$. However, this bound is not tight since it double counts the contribution of pieces containing several trichromatic faces. We follow the calculation in [17, Lemma 3.3] to avoid this double counting. Let $r = |H|/|\partial R|$. Consider an r -division of R_H in the decomposition tree. We bound separately the contribution of (a) ancestors of pieces in the r -division, and (b) descendants of pieces in the r -division.

For part (a), it is well known that the total number of vertices of all DDGs of all of the pieces in an r -division is $O(|H|/\sqrt{r}) = O(\sqrt{|H|} \cdot |\partial R|)$, and that this is also a bound on the total number of vertices in all DDGs of all the ancestors of pieces of the r -division in the decomposition tree. Hence, the contribution of part (a) is $O(\sqrt{|H|} \cdot |\partial R|)$.

For part (b), Each trichromatic face contributes at most two pieces at each level of the decomposition tree above it until reaching a piece of the r -division. Since the number of boundary vertices increases exponentially as we go up the decomposition tree, the contribution is asymptotically

dominated by the largest such ancestor, which is the piece of the r -division itself. Since each piece of the r -division has $O(\sqrt{r})$ boundary vertices, the contribution of part (b) is bounded by $O(|\partial R|\sqrt{r}) = O(\sqrt{|H|} \cdot |\partial R|)$.

Thus, the total time for finding all trichromatic faces as well as the tree structure is $\tilde{O}(\sqrt{|H|} \cdot |\partial R|)$.

We now remove the assumption that there are no empty Voronoi cells. To this end, we first run FR-Dijkstra as above on the union of star K and the DDGs of pieces in \mathcal{P} . Then, for every site s that is not a child of the root in the obtained shortest path tree T , we override its additive weight with its distance from u , and store a pointer from this site to its ancestor (site) s' in T that is a child of the root. Intuitively, s becomes responsible for the vertices v of the Voronoi cell of s' for which the shortest s' -to- v path contains s . Our tie-breaking rule ensures that with the new additive weights, $s \in \text{Vor}(s)$. This concludes the proof of Theorem 8.1. \square

(B/C) Voronoi Diagrams

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

By Theorem 8.1, the total construction time for the dual representations is

$$\tilde{O} \left(\sum_i \frac{n}{r_i} \sqrt{r_i} \sqrt{n \sqrt{r_{i+1}}} + \sum_i \frac{n}{r_i} \sqrt{r_i} \sqrt{n \sqrt{r_i}} \right) = \tilde{O} \left(\sum_i \frac{n^{3/2+1/(4m)}}{r_i^{1/4}} \right) = \tilde{O} \left(n^{3/2+1/(4m)} \right),$$

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

(D) Site Tables and Side Tables

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

$$\text{DDG}[R_{i+1}^{\text{out}} \cap R_{i+2}], \text{DDG}[R_{i+2}^{\text{out}} \cap R_{i+3}], \dots, \text{DDG}[R_{m-2}^{\text{out}} \cap R_{m-1}], \text{DDG}[R_{m-1}^{\text{out}}]$$

contains exactly all boundary vertices in R_i^{out} of ancestors $R_i, R_{i+1}, \dots, R_{m-1}$. 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 from the super-source u' using the FR-Dijkstra algorithm in $\tilde{O}(\sqrt{n})$ time.

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

In the side table, we will store the relationship (left/right/Null) between each site-centroid-site chord $\tilde{C} = \overrightarrow{s_j y_j y_{j-1} s_{j-1}}$ (using the notations in Figure 6) and each ancestor $R_{i'}^{\text{out}}$ ($i' \geq i$). With the technique used in the construction of site tables, we can extract all vertices of \tilde{C} on each $\partial R_{i'}$ from T_H , and then determine the relationship between \tilde{C} and each $R_{i'}^{\text{out}}$ with

boundary vertices on \tilde{C} . For each R_i^{out} such that \tilde{C} contains no vertices on ∂R_i , we pick an arbitrary vertex z on ∂R_i . We can retrieve from T_H the u' -to- z path and find the site s_z such that $z \in \text{Vor}(s_z)$. This can be done in $O(\sqrt{n})$ time. With T_H and the MSSP structures from part (F), we can determine the pairwise relationships among s_j -to- y_j , s_{j-1} -to- y_{j-1} , and s_z -to- z shortest paths and know whether z lies to the left or right of \tilde{C} , which immediately shows the relationship between \tilde{C} and R_i^{out} . The construction time for a side table of $\text{VD}_{\text{out}}^*(u, R_i)$ is $\tilde{O}(m\sqrt{r_i}\sqrt{n})$.

The total time for building all site tables and side tables is

$$\tilde{O}\left(\sum_i m \frac{n}{r_i} \sqrt{r_i} \sqrt{r_{i+1}} \sqrt{n}\right) = \tilde{O}\left(m^2 n^{3/2+1/(2m)}\right).$$

(E) 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 ∂R_i into single edges. Thus, it can be computed by running FR-Dijkstra on the union of $\text{DDG}[R_i]$ and $\text{DDG}[R_i^{\text{out}}]$ in $\tilde{O}(\sqrt{r_i})$ time. Regarding the construction of the piece tree $\mathcal{T}_q^{R_i}$, we first extract all the chords on $T_q^{R_i}$ in R_i^{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 the boundary cycle on ∂R_i . Observe that each piece in $\mathcal{P}_q^{R_i}$ relates to a face of Q . The piece tree $\mathcal{T}_q^{R_i}$ can be computed in $\tilde{O}(\sqrt{r_i})$ time, by taking the dual Q^* and removing the vertex corresponding to the face on ∂R_i . With the graph Q and the piece tree $\mathcal{T}_q^{R_i}$, the data structure supporting **MaximalChord** and **AdjacentPiece** in Lemma 6.6 can also be constructed in time $\tilde{O}(\sqrt{r_i})$ for the given q, R_i .

The total time to compute part (E) is $\tilde{O}(\sum_i \frac{n}{r_i} \sqrt{r_i} \sqrt{r_i}) = \tilde{O}(nm)$.

The overall construction time is $\tilde{O}(n^{3/2+1/m} + n^{1+1/m+1/\kappa})$ since m and κ should be functions of n that are $O(\log n)$.

A preprocessing-time vs. query-time tradeoff. No smooth tradeoff between the $\tilde{O}(n)$ -time preprocessing and $\tilde{O}(\sqrt{n})$ -query time oracle of Fakcharoenphol and Rao [28], and the oracle presented in this paper is known. Let us however note, that the oracle of [14] can be adapted to give the following tradeoff. For any $r = n^x$ with constant $x \in (0, 1]$, there is an oracle that can be constructed in $n^{3/2+o(1)}/r^{1/4}$ time, occupies $n^{1+o(1)}$ space, and answers queries in $r^{1/2+o(1)}$ time. We now sketch how this tradeoff can be achieved. The $n^{1+o(1)}$ -space, $n^{o(1)}$ -query time oracle presented in [14] makes use of an \vec{r} -division, and stores similar Voronoi diagrams as those presented in this paper; the main difference lies in how centroids are handled. The sole bottleneck in its construction is the construction of Voronoi diagrams, with everything else requiring time $n^{1+o(1)}$. Let k be the successor of r in \vec{r} . We obtain the tradeoff by building the Voronoi diagrams (using Theorem 8.1) only for pieces in r_i -divisions with $i \geq k$ in total time

$$\tilde{O}\left(\sum_{i \geq k} \frac{n}{r_i} \sqrt{r_i} \sqrt{n\sqrt{r_{i+1}}}\right) = n^{3/2+o(1)}/r^{1/4}.$$

Now consider a query $\text{dist}_G(u, v)$. The case where $v \in R_k$ is simple and can be handled using FR-Dijkstra in $\tilde{O}(\sqrt{r})$ time. In the complementary case, we first perform FR-Dijkstra from the source u in the union of

$$\text{DDG}[R_0^{\text{out}} \cap R_1], \text{DDG}[R_1^{\text{out}} \cap R_2], \dots, \text{DDG}[R_{k-1}^{\text{out}} \cap R_k], \text{DDG}[R_k^{\text{out}}],$$

and then issue distance queries to v from each of the boundary vertices of R_k . Finally, we return the minimum of $\text{dist}_G(u, s) + \text{dist}_{R_k^{\text{out}}}(s, v)$ over all $s \in \partial R_k$.

9 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 [46] produces a binary decomposition tree, of which our \tilde{r} -division is a coarsening. It proceeds by finding a separating cycle (as in Miller [52]), and recursively decomposes the graph inside the cycle and outside the cycle.¹² At intermediate stages the working graph contains several holes, but Miller's theorem [52] only guarantees that a small cycle separator exists if the graph is triangulated. To that end, the decomposition [46] puts an artificial vertex inside each hole and triangulates the hole. See Figure 15(a,b). If the cycle separator C (blue cycle in Figure 15(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 15(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 15(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 $R_i^{\text{par}_{R_i}(g), \text{out}}$, which we refer to as the *parent* of g in R_i . Note that all holes of R_{i+1} have the same parent in R_i , and that the ancestry of holes goes in the opposite direction of the ancestry of regions in the \tilde{r} -division. In a distance query we only deal with a series of regions $R_0 = \{u\}, R_1, \dots, R_m = G$. 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 9.1. *There is an $\tilde{O}(n)$ -space data structure that can be built in $\tilde{O}(n)$ time, and given u, v can report in $O(m)$ time the regions $R_0 = \{u\}, R_1, \dots, R_{t+1}$ and holes h_0, h_1, \dots, h_t such that $v \in R_i^{h_t, \text{out}}$, $v \notin R_t$, and $v \in R_{t+1}$.*

PROOF. Regions R_i can be reported by following parent pointers in our tree representation of the \tilde{r} -division, starting from R_0 , to which u stores a pointer.

For each region R_i , we can find the correct hole h_i as follows. We store the tree representation \mathcal{A} of the recursive decomposition computed by the algorithm of Klein et al [46]. We also store some extra information for each hole of each region in \mathcal{A} . Due to the structural organization of holes discussed above, each separator of the $O(\log n)$ ancestors of a region P in \mathcal{A} lies in $P^{h, \text{out}}$ for a unique hole h of P . For each region P , we store, for each separator of an ancestor of P in the decomposition tree, the hole h of P such that $P^{h, \text{out}}$ contains that separator. (This information can be propagated bottom-up during the construction of \mathcal{A} in $\tilde{O}(n)$ time.) In the query, by performing an LCA query for the constant size region $\{v\}$ and R_0 in \mathcal{A} , we find the separator C that separated

¹²The Klein et al. [46] 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.

¹³Given a $\text{dist}_G(u, v)$ query, we can map it to $\mathbf{Dist}(u', v', R_0)$, where u' and v' are any of the copies of u and v , respectively, and $R_0 = \{u'\}$.

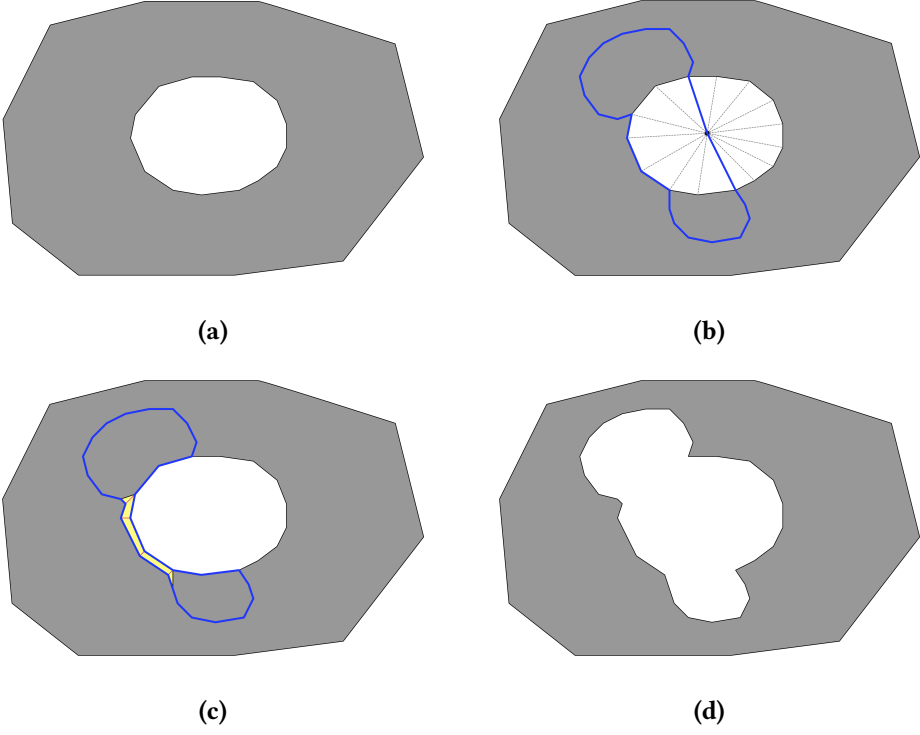


Fig. 15. (a) A subgraph with two holes. (b) We put a vertex in each hole and triangulate the hole. (The triangulation of the exterior hole is not drawn, for clarity.) A simple cycle separator (blue curve) is found in this graph. (c) The cycle is mapped to a possibly non-simple cycle in the original graph 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).

v from u . Then, for each i , we can find the appropriate hole h_i of R_i in $O(1)$ time: it is the hole h such that C is in $R_i^{h,\text{out}}$. Over all i this takes $O(m)$ time. \square

9.1 Data Structures

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

- (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 $i \in [0, m - 2]$, 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) **(Site Tables; Side Tables)** For each i and each Voronoi diagram $\text{VD}_{\text{out}}^* = \text{VD}_{\text{out}}^*(u', R_i, h_i)$ from part (B) or (C), for each node f^* in the centroid decomposition of VD_{out}^* with y_j, s_j

defined as usual, $j \in \{0, 1, 2\}$, we store the following. 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_j -to- y_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 $\overrightarrow{s_j y_j y_{j-1} s_{j-1}}$ in $R_i^{h_i, \text{out}}$, or **Null** if the relationship cannot be determined.

- (E) (**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 . $C_q^{R_i, h_i}, \mathcal{P}_q^{R_i, h_i}, \mathcal{T}_q^{R_i, h_i}$ are defined analogously, and data structures are built to answer **MaximalChord** and **AdjacentPiece** with respect to q, R_i, h_i .

9.2 Query

At the first call to **Dist**(u, v, R_0) we apply Lemma 9.1 to generate the regions R_1, \dots, R_{t+1} and holes h_1, \dots, h_t that will be accessed in all recursive calls, in $O(m)$ time.

The shortest u -to- v path in G must cross h_1, \dots, h_t . The vertex u_i is now defined to be the last vertex in h_i on the shortest u -to- v path. Given u_i , we find u_{i+1} by solving a point location problem in $\text{VD}_{\text{out}}^*(u_i, R_{i+1}, h_{i+1})$. The **SitePathIndicator** and **ChordIndicator** routines focus on the subgraph $R_t^{h_t, \text{out}}$ rather than R_t^{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 $R_t^{h_t, \text{out}}$, which does not cause further complications.

9.3 Preprocessing

The existence of multiple holes does not create any serious complications in our construction algorithm.

10 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.

In terms of the parameter m (the depth of the \tilde{r} -division), our distance oracle uses space $\tilde{O}(n^{1+1/m})$ and has query time $\tilde{O}(2^m)$. The exponential dependence on m arises from the fact that **Dist** solves one point location problem, but our point location routine narrows the number of Voronoi cells to *two* candidates, which are resolved with *two* recursive calls to **Dist** at a higher level of the \tilde{r} -division. Avoiding this exponential dependence on m may require a completely different approach to the problem.

We highlight two more open problems. The construction time of our oracle is $n^{3/2+o(1)}$. It is an important open question to compute an oracle that is optimal in space, query time, and construction time, up to $n^{o(1)}$ factors. See [15] for a recent specialized oracle with near-linear construction time. A different direction is to find efficient distance oracles for graphs embeddable on surfaces of bounded genus, as we believe that the distance oracle described in Section 4.5 can be improved.

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] Gerth Stølting Brodal, Pooya Davoodi, and S Srinivasa Rao. Path minima queries in dynamic weighted trees. In *Proceedings of the 12th Int'l Symposium on Algorithms and Data Structures (WADS)*, pages 290–301, 2011.
 - [8] Gerth Stølting Brodal, Rolf Fagerberg, Christian N. S. Pedersen, and Anna Östlin. The complexity of constructing evolutionary trees using experiments. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001*, pages 140–151, 2001.
 - [9] Sergio Cabello. Many distances in planar graphs. *Algorithmica*, 62(1-2):361–381, 2012.
 - [10] 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.
 - [11] Sergio Cabello, Erin W. Chambers, and Jeff Erickson. Multiple-source shortest paths in embedded graphs. *SIAM J. Comput.*, 42(4):1542–1571, 2013.
 - [12] Erin W. Chambers, Jeff Erickson, and Amir Nayyeri. Homology flows, cohomology cuts. *SIAM J. Comput.*, 41(6):1605–1634, 2012.
 - [13] Timothy M. Chan and Dimitrios Skrepetos. Faster approximate diameter and distance oracles in planar graphs. *Algorithmica*, 81(8):3075–3098, 2019.
 - [14] 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.
 - [15] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. An almost optimal edit distance oracle. In *Proceedings of the 48th Int'l Colloq. on Algorithms, Languages, and Programming (ICALP)*, 2021.
 - [16] Panagiotis Charalampopoulos and Adam Karczmarsz. Single-source shortest paths and strong connectivity in dynamic planar graphs. *J. Comput. Syst. Sci.*, 124:97–111, 2022.
 - [17] Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. Exact distance oracles for planar graphs with failing vertices. *ACM Trans. Algorithms*, 18(2):18:1–18:23, 2022.
 - [18] Shiri Chechik. Approximate distance oracles with improved bounds. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 2015.
 - [19] 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.
 - [20] 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.
 - [21] 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.
 - [22] 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. *SIAM J. Comput.*, 23(4):738–761, 1994.
 - [23] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
 - [24] Hristo Djidjev. Efficient algorithms for shortest path queries in 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.
 - [25] 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.
 - [26] 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.
 - [27] Jeff Erickson and Sarel Har-Peled. Optimally cutting a surface into a disk. *Discret. Comput. Geom.*, 31(1):37–59, 2004.
 - [28] 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.

- [29] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.
- [30] Viktor Fredslund-Hansen, Shay Mozes, and Christian Wulff-Nilsen. Truly subquadratic exact distance oracles with constant query time for planar graphs. *CoRR*, abs/2009.14716, 2020.
- [31] Paweł Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 495–514, 2018.
- [32] Paweł 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.
- [33] Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for centroid decomposition. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, pages 274–282, 2019.
- [34] Qian-Ping Gu and Gengchun Xu. Constant query time $(1 + \epsilon)$ -approximate distance oracle for planar graphs. *Theor. Comput. Sci.*, 761:78–88, 2019.
- [35] Torben Hagerup. Still simpler static level ancestors. *CoRR*, abs/2005.11188, 2020.
- [36] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.
- [37] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- [38] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.
- [39] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.
- [40] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017.
- [41] Adam Karczmarz and Piotr Sankowski. A deterministic parallel APSP algorithm and its applications. In *Proceedings of the 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 255–272, 2021.
- [42] 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.
- [43] 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.
- [44] Philip N. 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.
- [45] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 146–155, 2005.
- [46] Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 505–514, 2013.
- [47] Jakub Lacki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA)*, pages 155–166, 2011.
- [48] Hung Le and Christian Wulff-Nilsen. Optimal approximate distance oracle for planar graphs. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 363–374, 2021.
- [49] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.
- [50] Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *Proceedings of the 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2517–2536, 2021.
- [51] Dániel Marx and Michał Pilipeczuk. Optimal parameterized algorithms for planar facility location problems using voronoi diagrams. *ACM Trans. Algorithms*, 18(2):13:1–13:64, 2022.
- [52] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci.*, 32(3):265–279, 1986.
- [53] Shay Mozes, Kirill Nikolaev, Yahav Nussbaum, and Oren Weimann. Minimum cut of directed planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 29th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 477–494, 2018.
- [54] Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 209–222, 2012.
- [55] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*, pages 206–217, 2010.

- [56] Yahav Nussbaum. Improved distance queries in planar graphs. In *Proceedings 12th Int'l Workshop on Algorithms and Data Structures (WADS)*, pages 642–653, 2011.
- [57] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.
- [58] Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the Thorup-Zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014.
- [59] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- [60] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, 2014.
- [61] Christian Sommer, Elad Verbin, and Wei Yu. Distance oracles for sparse graphs. In *Proceedings of the 50th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 703–712, 2009.
- [62] E. Sperner. Neuer beweis für die invarianz der dimensionszahl und des gebietes. *Abh. Math. Semin. Hamburg. Univ.*, Bd. 6:265–272, 1928.
- [63] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.
- [64] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.
- [65] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.
- [66] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.
- [67] Christian Wulff-Nilsen. *Algorithms for planar graphs and graphs in metric spaces*. PhD thesis, University of Copenhagen, 2010.
- [68] Christian Wulff-Nilsen. Approximate distance oracles for planar graphs with improved query time-space tradeoff. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 2016.

A MSSP (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 [45] 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.

A.1 Adding Functionality to Link-Cut Trees MSSP

In this subsection, we explain how to augment the MSSP data structure of Klein [11, 45] to support the lowest common ancestor query of Lemma 2.1. The MSSP data structure represents the shortest path trees rooted at the vertices S of the distinguished face f using a partially persistent [25] link-cut tree [59]. The persistent representation allows us to access the desired version of the tree with a constant-time overhead. Let T_s denote the version of the shortest path tree rooted at s . The edges of the link-cut tree T_s are partitioned into solid and dashed edges. Each maximal path of solid edges is called a solid path, which is represented by a binary search tree, where the left-right order in the search tree corresponds to the top-bottom order in the solid path (the root is top).

To locate the LCA x of u and v , we list the solid paths that intersect the path from u to the root of T_s , and those that intersect the path from v to the root of T_s . Let P be the first solid path in both lists, and let u' and v' be the nearest ancestors of u and v that lie on P . The LCA x is the leftmost of u' and v' in the search tree representing P . Once we have found x , we can retrieve the edge e_z outgoing from x and leading to the subtree containing $z \in \{u, v\}$ (when $x \neq z$) in additional $O(\log n)$ time.

A.2 MSSP via Euler Tour Trees

Generally, if we maintain the SSSP tree as the source travels around f in a dynamic data structure with update time t_u and query time t_q (for distance and LCA queries), the universal persistence method for RAM data structures (see [21]) yields an MSSP data structure with space $O(|H|t_u)$ and query time $O(t_q \log \log |H|)$. Thus, to establish Lemma 2.1 it suffices to design a dynamic data structure for the following:

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

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

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

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 $O(|H|)$ **Swap** operations. Klein [45] used Sleator and Tarjan's Link-Cut trees [59], which support **Swap**, **Dist**, and **LCA** (among other operations) in $O(\log |T|)$ time. We will use a souped-up version of Henzinger and King's [38] 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_{ante} is the tree before a **Swap** operation and T_{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 concatenations, and renaming the two elements corresponding to the swapped edge. See Figure 16. We will argue that the dynamic tree operations **Swap**, **Dist**, **LCA** can be implemented using the following list operations.

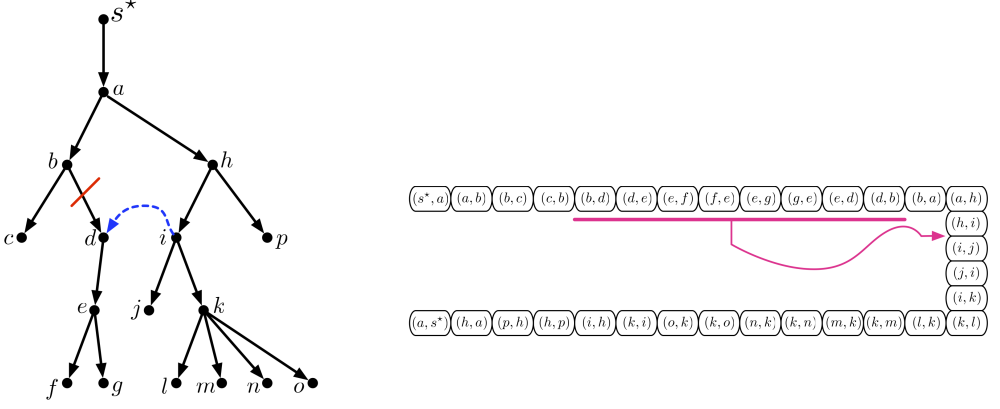


Fig. 16. The effect of **Swap**(d, i, \cdot) on the Euler Tour. The interval $((b, d), (d, e), \dots, (e, d), (d, b))$ is spliced out and inserted between (h, i) and (i, j) , and the elements (b, d) , (d, b) are renamed (i, d) , (d, i) .

InitList(L): Initialize a list L of weighted elements.

Split(e_0): Element e_0 appears in some list L . Split L immediately after element e_0 , resulting in two lists.

Concatenate(L_0, L_1): Concatenate L_0 and L_1 , resulting in one list.

Add(e_0, e_1, δ): 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.

Weight(e_0): Return the weight of e_0 .

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 **Dist** and **LCA** we will actually use the list data structure with different weight functions. For **Dist**, the weight of an edge (x, y) in $ET(T)$ is $\text{dist}_T(s^*, y)$. Thus, **Dist** is answered with a call to **Weight**. Each **Swap**(v, p, l) is effected with $O(1)$ **Split** and **Concatenate** operations, renaming the elements of the swapped edge, as well as one **Add**(e_0, e_1, δ) operation. Here (e_0, \dots, e_1) is the sub-list corresponding to the subtree rooted at v , and $\delta = \text{dist}_{T_{\text{post}}}(s^*, v) - \text{dist}_{T_{\text{ante}}}(s^*, v)$ 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 $e_0 = (p_u, u)$, $e_1 = (p_v, v)$ be the edges into u and v from their respective parents, and suppose that e_0 appears before e_1 in $ET(T)$.¹⁴ A call to **RangeMin**(e_0, e_1) returns the *first* edge $\hat{e} = (x, y)$ in the interval (e_0, \dots, e_1) minimizing the depth of y . It follows that y is the LCA of u and v . Furthermore, by the tiebreaking rule, if $\hat{e} \neq e_0$ then $\hat{e} = e_u$ is the (reversal of the) edge leading from y towards u . If $\hat{e} = e_0$ then v is a descendant of u and e_u does not exist. To find e_v , we retrieve the edge $\tilde{e} = (y, p_y)$ in $ET(T)$ from y to its parent and let \tilde{e}' be its predecessor in $ET(T)$. (Note that since s^* has degree 1, \tilde{e}, \tilde{e}' always exist.) We call **RangeMin**(e_1, \tilde{e}'). Once again, by the tiebreaking rule it returns the *first* edge $e_v = (x', y)$ incident to y in (e_1, \dots, \tilde{e}') , which is the (reversal of the) first edge on the path from y to v . See Figure 17.

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.

¹⁴As we will see, it is easy to determine which comes first.

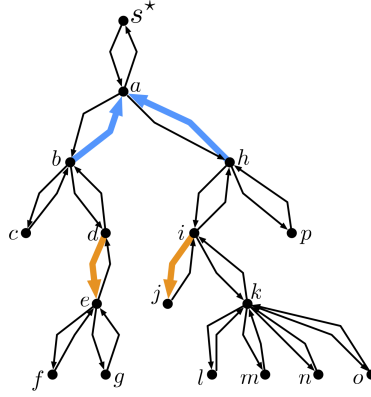


Fig. 17. An illustration of an $\text{LCA}(e, j)$ query. We do a **RangeMin** query on the interval $e_0 = (d, e), \dots, (i, j) = e_1$ and retrieve the edge $\hat{e} = e_e = (b, a)$ with weight $\text{depth}_T(a)$. We then find $\tilde{e} = (a, s^*)$ and its predecessor $\tilde{e}' = (h, a)$. Another **RangeMin** query on the interval $(i, j), \dots, (h, a)$ returns $e_j = (h, a)$.

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

- I. Each node γ of $\mathcal{T}(L)$ stores a weight offset $w(\gamma)$, a min-weight value $\min(\gamma)$ and a pointer $\text{ptr}(\gamma)$. The weight of (leaf) $e \in L$ is the sum of the $w(\cdot)$ -values of its ancestors, including e . The sum of $\min(\gamma)$ and the $w(\cdot)$ -values of all strict ancestors of γ is exactly the weight of the minimum weight descendant of γ , and $\text{ptr}(\gamma)$ points to this element.
- II. Non-root internal nodes have between $n^{1/\kappa}$ and $3n^{1/\kappa}$ children. In particular, the tree has height at most κ .
- III. Each internal node γ maintains an $O(1)$ -time range minimum structure [4] over the vector of $\min(\cdot)$ -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 \mathcal{T} . Each costs $O(n^{1/\kappa})$ time to update the information covered by Invariants I and III. The total time is therefore $O(\kappa n^{1/\kappa})$. By Invariant I, a **Weight**(e_0) query takes $O(\kappa)$ time to sum all of e_0 's ancestors' $w(\cdot)$ -values. Consider an **Add**(e_0, e_1, δ) or **RangeMin**(e_0, e_1) operation. By Invariant II, the interval (e_0, \dots, e_1) is covered by $O(\kappa n^{1/\kappa})$ \mathcal{T} -nodes, and furthermore, those nodes can be arranged into less than 2κ contiguous intervals of siblings. Thus, an **Add**(e_0, e_1) can be implemented in $O(\kappa n^{1/\kappa})$ time by adding δ to the $w(\cdot)$ -values of these nodes and rebuilding the affected range-min structures from Invariant III. A **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 **RangeMin** takes $O(\kappa)$ time.

We have shown that the dynamic tree operations necessary for an MSSP structure can be implemented with a flexible tradeoff between update time and query time. Moreover, this lower bound meets the Pătraşcu-Demaine lower bound [57]. We leave it as an open problem to implement the complete set of operations supported by Link-Cut trees, with update time $O(\kappa n^{1/\kappa})$ and query time $O(\kappa)$.

Received 24 May 2021; revised 11 June 2022; accepted 4 January 2023