Better Tradeoffs for Exact Distance Oracles in Planar Graphs

Paweł Gawrychowski* Shay Mozes† Oren Weimann‡ Christian Wulff-Nilsen§

Abstract

We present an $O(n^{1.5})$-space distance oracle for directed planar graphs that answers distance queries in $O(\log n)$ time. Our oracle both significantly simplifies and significantly improves the recent oracle of Cohen-Addad, Dahlgaard and Wulff-Nilsen [FOCS 2017], which uses $O(n^{5/3})$-space and answers queries in $O(\log n)$ time. We achieve this by designing an elegant and efficient point location data structure for Voronoi diagrams on planar graphs.

We further show a smooth tradeoff between space and query-time. For any $S \in [n, n^2]$, we show an oracle of size S that answers queries in $\tilde{O}(\max\{1, n^{1.5}/S\})$ time. This new tradeoff is currently the best (up to polylogarithmic factors) for the entire range of S and improves by polynomial factors over all previously known tradeoffs for the range $S \in [n, n^{5/3}]$.

1 Introduction

Computing shortest paths is a classical and fundamental algorithmic problem that has received considerable attention from the research community for decades. A natural data structure problem in this context is to compactly store information about distances in a graph in such a way that the distance between any pair of query vertices can be computed efficiently. A data structure that supports such queries is called a *distance oracle*. Naturally, there is a tradeoff between the amount of space consumed by a distance oracle and the time required by distance queries. Another quantity of interest is the preprocessing time required for constructing the oracle.

Distance oracles in planar graphs. It is natural to consider the class of planar graphs in this setting since planar graphs arise in many important applications involving distances, most notably in navigation applications on road maps. Moreover, planar graphs exhibit many structural properties that facilitate the design of very efficient algorithms. Indeed, distance oracles for planar graphs have been extensively studied. These oracles can be divided into two groups: *exact* distance oracles which always output the correct distance, and *approximate* distance oracles which allow a small stretch in the distance output. For approximate distance oracles, one can obtain near-linear space and near-constant query-time at the cost of a $(1+\epsilon)$ stretch (for any fixed ϵ) [16–18, 29, 32]. In this paper we focus on the tradeoff between space and query-time of exact distance oracles for planar graphs.

Exact distance oracles. The following results as well as ours all hold for directed planar graphs with real arc-lengths (but no negative length cycles). Djidjev [8] and Arikati et al. [1] obtained distance oracles with the following tradeoff between space and query-time. For any $S \in [n, n^2]$, they show an oracle with space S and query-time of $O(n^2/S^2)$. For $S \in [n^{4/3}, n^{1.5}]$, Djidjev's oracle achieves an improved bound of $O(n/\sqrt{S})$. This bound (up to polylogarithmic factors) was extended to the entire range $S \in [n, n^2]$ in a number of papers [3, 6, 10, 24, 27]. Wulff-Nilsen [31] showed how to achieve constant query-time with $O(n^2(\log\log n)^4/\log n)$ space, improving the above tradeoff for close to quadratic space. Very recently, Cohen-Addad, Dahlgaard, and Wulff-Nilsen [7], inspired by the ideas of Cabello [4] made significant progress by presenting an oracle with $O(n^{5/3})$ space and $O(\log n)$ query-time. This is the first oracle for planar graphs that achieves truly subquadratic space and subpolynomial query-time. They also showed that with $S \geq n^{1.5}$ space, a query-time of $O(n^{2.5}/S^{1.5} \log n)$ is possible. To summarize, prior to the results described in the current paper the best known tradeoff was $\tilde{O}(n/\sqrt{S})$ query-time for space $S \in [n, n^{1.5}]$, $\tilde{O}(n^{2.5}/S^{1.5})$ query-time for space $S \in [n^{1.5}, n^{5/3}]$, and $O(\log n)$ query-time for $S \in [n^{5/3}, n^2]$. See Figure 1 for a summary of prior results.

Our results and techniques. In this paper we show a distance oracle with $O(n^{1.5})$ space and $O(\log n)$ query-time. More generally, for any $r \leq n$ we construct a distance oracle with $O(n^{1.5}/\sqrt{r} + n \log r \log(n/r))$ space and $O(\sqrt{r} \log n \log r)$ query-time. This improves the currently best known tradeoffs for essentially the entire range of S: for space $S \in [n, n^{1.5}]$ we obtain an oracle with $\tilde{O}(n^{1.5}/S)$ query-time, while for space

*University of Haifa, Department of Computer Science, gawry@cs.uni.wroc.pl. Partially supported by the Israel Science Foundation grant 794/13.

†IDC Herzliya, Efi Arazi School of Computer Science, smozes@idc.ac.il. Partially supported by the Israel Science Foundation grants 794/13 and 592/17.

‡University of Haifa, Department of Computer Science, oren@cs.haifa.ac.il. Partially supported by the Israel Science Foundation grants 794/13 and 592/17.

§University of Copenhagen (DIKU), Department of Computer Science, koolooz@di.ku.dk.

$S \in [n^{1.5}, n^2]$, our oracle has query-time of $O(\log n)$.

To explain our techniques we need the notion of an additively weighted Voronoi diagram on a planar graph. Let $P = (V, E)$ be a directed planar graph, and let $S \subseteq V$ be a subset of the vertices, which are called the *sites* of the Voronoi diagram. Each site $u \in S$ has a weight $\omega(u) \geq 0$ associated with it. The distance between a site $u \in S$ and a vertex $v \in V$, denoted by $d(u, v)$, is defined as $\omega(u)$ plus the length of the u-to-v shortest path in P.

DEFINITION 1.1. *The* additively weighted Voronoi diagram *of (S, ω) within P, denoted* $\mathrm{VD}(S, \omega)$*, is a partition of V into pairwise disjoint sets, one set* $\mathrm{Vor}(u)$ *for each site $u \in S$. The set* $\mathrm{Vor}(u)$*, which is called the* Voronoi cell *of u, contains all vertices in V that are closer (w.r.t. $d(\cdot, \cdot)$) to u than to any other site in S (assuming that the distances are unique).*

There is a dual representation $\mathrm{VD}^*(S, \omega)$ of a Voronoi diagram $\mathrm{VD}(S, \omega)$ as a planar graph with $O(|S|)$ vertices and edges. See Section 2.

We obtain our results using point location in additively weighted Voronoi diagrams. This approach is also the one taken in [7]. However, our construction is arguably simpler and more elegant than that of [7]. Our main technical contribution is a novel point location data structure for Voronoi diagrams (see below). Given this data structure, the description of the preprocessing and query algorithms of our $O(n^{1.5})$-space oracle are extremely simple and require a few lines each. In a nutshell, the construction is recursive, using simple cycle separators. We store a Voronoi diagram for each node u of the graph. The sites of this diagram are the vertices of the separator and the weights are the distances from u to each site. To get the distance from u to v it suffices to locate the node v in the Voronoi diagram stored for u using the point location data structure. Since the cycle separator has $O(\sqrt{n})$ vertices, this yields an oracle requiring $O(n^{1.5})$ space.

The oracles for the tradeoff are built upon this simple oracle by storing Voronoi diagrams for just a subset of the nodes in a graph (the so called boundary vertices of an r-division). This requires less space, but the query-time increases. This is because a node u now typically does not have a dedicated Voronoi diagram. Therefore, to find the distance from u to v, we now we need to locate v in multiple Voronoi diagrams stored for nodes in the vicinity of u.

As we mentioned above, our main technical tool is a data structure that supports point location queries in Voronoi diagrams in $O(\log n)$ time. This is summarized in the following theorem.

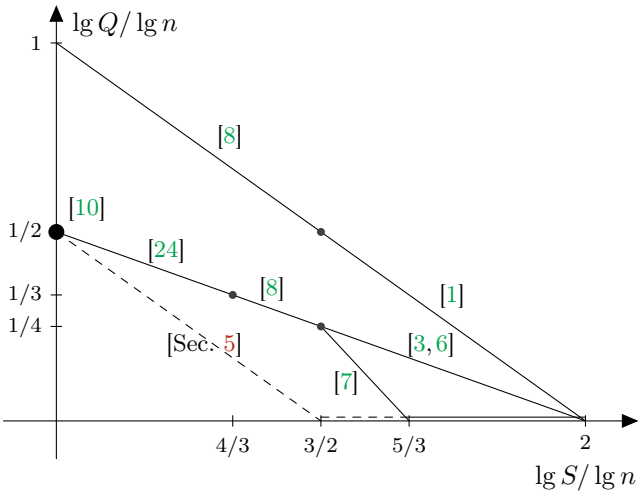

Figure 1: Tradeoff of the Space $[S]$ vs. the Query time $[Q]$ for different exact distance oracles on a doubly logarithmic scale, ignoring constant and logarithmic factors. Prior results are indicated by solid lines. The new improved tradeoff is indicated by dashed lines.

THEOREM 1.1. *Let P be a directed planar graph with real arc-lengths, and no negative length cycles. Let S be a set of sites that lie on a single face of P. We can preprocess P in $O(|S| \cdot |P|)$ time and $O(|S| \cdot |P|)$ space so that, given the dual representation of any additively weighted Voronoi diagram, $VD^*(S, \omega)$, we can extend it in $O(|S|)$ time and space to support the following queries. Given a vertex v of P, report in $O(\log |S|)$ time the site u such that v belongs to* $\mathrm{Vor}(u)$.

A data structure for the same task was described in [7]. Our data structure is both significantly simpler and more efficient. Roughly speaking, the idea is as follows. We prove that $\mathrm{VD}^*(S, \omega)$ is a ternary tree. This allows us to use a straightforward centroid decomposition of depth $O(\log |S|)$ for point location. To locate the voronoi cell $\mathrm{Vor}(u)$ containing a node v we traverse the centroid decomposition. At any given level of the decomposition we only need to known which of the three subtrees in the next level contains $\mathrm{Vor}(u)$. To this end we associate with each centroid node three shortest paths. These paths partition the plane into three parts, each containing exactly one of the three subtrees. Identifying the desired subtree then boils down to determining the position of v relative to these three shortest paths. We show that this can be easily done by examining the preorder number of v in the shortest path trees rooted at three sites. In contrast, the point location data structure of [7] relies on more

complicated and less elegant characterization of the structure of Voronoi diagrams, and hence on slower and more complicated data structures. Consequently, their distance oracle is both slower and more complicated than ours.

Roadmap. Theorem 1.1 is proved in Section 4 under a simplifying assumption that every site belongs to its Voronoi cell (i.e., that there are no empty Voronoi cells). This suffices to design our distance oracle with space $O(n^{1.5})$ and query-time $O(\log n)$, which is done in Section 3, assuming that Theorem 1.1 holds. Section 5 describes the improved space to query-time tradeoff. Finally, in Section 6 we describe how to remove the simplifying assumption. Additional details and some omitted proofs appear in the appendix.

2 Preliminaries

For completeness we review some basic relevant definitions and facts concerning planar graph. See e.g., [19] for a formal introduction of these concepts.

Reduced lengths. We assume arc lengths are non-negative. There is a standard transformation [15] that, given a graph with real arc lengths (but no negative-length cycles), transforms the lengths to be non-negative while preserving the shortest paths between any pair of vertices. This is done by computing distances from an arbitrary vertex r, and shifting the length of each arc e by the difference of distances of the endpoints of e from r. It is easy to show that the resulting lengths are non-negative, and that shortest paths are preserved. In planar graphs this transformation can be performed in $O(n \log^2 n / \log\log n)$ time [25].

Plane graphs. A plane graph G is a graph equipped with an embedding mapping vertices to points on the plane and edges to curves on the place. The faces of G are the maximal regions of the plane after removing the images of the edges of G. The dual of a plane graph G is another plane graph G^* whose vertices correspond to faces of G and vice versa. Two vertices in G^* are joined by an edge if and only if the corresponding faces of G are separated by an edge of G. Thus, every edge e in G has a corresponding dual edge e^* in G. For any face f of G, let f^* denote the corresponding vertex of G^*. Similarly, for any vertex v of G, let v^* denote the corresponding face of G^*. Each dual edge e^* is embedded so that it crosses the corresponding primal edge e exactly once, and intersects no other primal edge. For an arc (directed edge) $e = uv$, we call u the tail of e and v the head of e. An arc uv emanates left (right) of a simple path P if there exist two arcs e_1, e_2 in P, such that u is the head of e_1 and the tail of e_2, and e appears between e_1 and e_2 in the clockwise (counterclockwise) order of arcs incident to u.

We assume that shortest paths are unique. This can be ensured in linear time by a random perturbation of the edge lengths [23, 26] or deterministically in near-linear time using lexicographic comparisons [5, 13]. It will also be convenient to assume that graphs are strongly connected; if not, we can always triangulate them (i.e., add edges while respecting the embedding, so that each face has size 3) with bidirected edges of infinite length.

Separators in planar graphs. Given a planar embedded graph G, a Jordan curve separator is a simple closed curve in the plane that intersects the embedding of G only at vertices. Miller [22] showed that any n-vertex planar embedded graph has a Jordan curve separator of size $O(\sqrt{n})$ such that the number of vertices on each side of the curve is at most $2n/3$. In fact, the balance of $2/3$ can be achieved with respect to any weight function on the vertices, faces and edges of the graph rather than just the uniform weight function on the vertices. Miller also showed that the vertices of the separator ordered along the curve can be computed in $O(n)$ time.

An *r-division* [11, 21] of a planar graph G, for some $r \in (1, n)$, is a partition of the edges of G into parts. The subgraphs of G induces by the parts of the partition are called the *pieces* of the r-division. Each piece in an r-division has at most r vertices and $O(\sqrt{r})$ *boundary* vertices (vertices shared with other pieces). There is an $O(n)$ time algorithm that computes an r-division of a planar graph with the additional property that, in every piece, the number of faces of the piece that are not faces of the original graph G is constant [21, 30] (such faces are called *holes*).

Voronoi diagrams on planar graphs. Recall the definition of additively weighted Voronoi diagrams $\mathrm{VD}(S, \omega)$ from the introduction. We write just VD when the particular S and ω are not important, or when they are clear from the context.

We restrict our discussion to the case where the sites S lie on a single face, denoted by h. We work with a dual representation of $\mathrm{VD}(S, \omega)$, denoted $\mathrm{VD}^*(S, \omega)$ or simply VD^*. Let P^* be the planar dual of a planar graph P. Let VD_0^* be the subgraph of P^* consisting of the duals of edges uv of P such that u and v are in different Voronoi cells. Let VD_1^* be the graph obtained from VD_0^* by contracting edges incident to degree-2 vertices one after another until no degree-2 vertices remain. The vertices of VD_1^* are called Voronoi vertices. A Voronoi vertex f^* is dual to a face f such that the nodes incident to f belong to at least three different Voronoi cells. In particular, h^* (i.e., the dual vertex corresponding to the face h to which all the sites are incident) is a Voronoi vertex. Each face of VD_1^*

corresponds to a cell $\mathrm{Vor}(v_i)$. Hence there are at most $|S|$ faces in VD_1^*. By sparsity of planar graphs, and by the fact that the minimum degree of a vertex in VD_1^* is 3, the complexity (i.e., the number of vertices, edges and faces) of VD_1^* is $O(|S|)$. Finally, we define VD^* to be the graph obtained from VD_1^* after replacing the node h^* by multiple copies, one for each occurance of h as an endpoint of an edge in VD_1^*. The original Voronoi vertices are called *real*. See Figure 2.

Given a planar graph P with r nodes and a set S of b sites on a single face h, one can compute any additively weighted Voronoi diagram $\mathrm{VD}(S, \omega)$ naively in $O(r)$ time by adding an artificial source node, connecting it to every site s with an edge of length $\omega(s)$, and computing the shortest path tree using [14]. The dual representation $\mathrm{VD}^*(S, \omega)$ can then be obtained in additional $O(r)$ time by following the constructive description above. There are more efficient algorithms [4, 12] when one wants to construct many different additively weighted Voronoi diagrams for the same set of sites S. The basic approach is to invest superlinear time in preprocessing P, but then construct $\mathrm{VD}(S, \omega)$ for multiple choices of ω in $\tilde{O}(|S|)$ time each (instead of $O(r)$). Since the focus of this paper is on the tradeoff between space and query-time, and not on the preprocessing time, the particular algorithm used for constructing the Voronoi diagrams is less important.

3 The Oracle

In this section we describe our distance oracle assuming Theorem 1.1. Let G be a directed planar graph with non-negative arc-lengths. At a high level, our oracle is based on a recursive decomposition of G into pieces using Jordan curve separators. Each piece $R = (V, E)$ is a subgraph of G. The boundary vertices of R are vertices of R that are incident (in G) to edges not in R. The holes of R are faces of R that are not faces of G. Note that every boundary vertex of R is incident to some hole of P.

A piece $R = (V, E)$ is decomposed into two smaller pieces on the next level of the decomposition as follows. We choose a Jordan curve separator $C = (v_1, v_2, \ldots, v_k)$, where $k = O(\sqrt{|V|})$. This separates the plane into two parts and defines two smaller pieces P and Q corresponding to, respectively, the subgraphs of R inside and the outside of C. Every edge of R is assigned to either P or Q. Thus, on every level of the recursive decomposition into pieces, an edge of G appears in exactly one piece. The separators in levels congruent to 0 modulo 3 are chosen to balance the total number of nodes (i.e., finding a separator with respect to a weight function assigning uniform weight to all nodes). The separators in levels congruent to 1 modulo 3 are chosen to balance the number of boundary nodes (i.e., assigning uniform weight to all boundary nodes and zero weight to all other nodes). The separators in levels congruent to 2 modulo 3 are chosen to balance the number of holes (i.e., assigning uniform weight just to holes, and zero weight to all other faces). This guarantees that the number of holes in each piece is constant, and that the number of vertices and boundary vertices decrease exponentially along the recursion. In particular, the depth of the decomposition is logarithmic in $|V|$. These properties are summarized in the following lemma whose proof is in the appendix.

LEMMA 3.1. *Choosing the separators as described above guarantees that (i) each piece has $O(1)$ holes, (ii) the number of nodes in a piece on the ℓ-th level in the decomposition is $O(n/c_1^{\ell/3})$, for some constant $c_1 > 1$, (iii) the number of boundary nodes in a piece on the ℓ-th level in the decomposition is $O(\sqrt{n}/c_2^{\ell/3})$, for some constant $c_2 > 1$.*

Preprocessing. We compute a recursive decomposition of G using Jordan separators as described above. For each piece $R = (V_R, E_R)$ in the recursive decomposition we perform the following preprocessing. We compute and store, for each boundary node v of R, the shortest path tree T_v^R in R rooted at v. Additionally, we store for every node u of R the distance from v to u and the distance from u to v in the whole G. For a non-terminal piece R, let $P = (V_P, E_P)$ and $Q = (V_Q, E_Q)$ be the two pieces into which R is separated. For every node $u \in V_Q$ and for every hole h of P we store an additively weighted Voronoi diagram $\mathrm{VD}(S_h, \omega)$ for P, where the set of sites S_h is the set of boundary nodes of P incident to the hole h, and the additive weights ω correspond to the distances in G from u to each site in S_h. We enhance each Voronoi diagram with the point location data structure of Theorem 1.1. We also store the same information with the roles of Q and P exchanged.

Query. To compute the distance from u to v, we traverse the recursive decomposition starting from the piece that corresponds to the whole initial graph G. Suppose that the current piece is $R = (V, E)$, which is partitioned into P and Q with a Jordan curve separator C. If $v \in C$ then, because the nodes of C are boundary nodes in both P and Q, we return the additive weight $\omega(v)$ in the Voronoi diagram stored for u, which is equal to the distance from u to v in G. Similarly, if $u \in C$ then we retrieve and return the distance from u to v in the whole G. The remaining case is that both u and v belong to a unique piece P or Q. If both u and v belong to the same piece on the lower level of the decomposition, we continue to that piece. Otherwise,

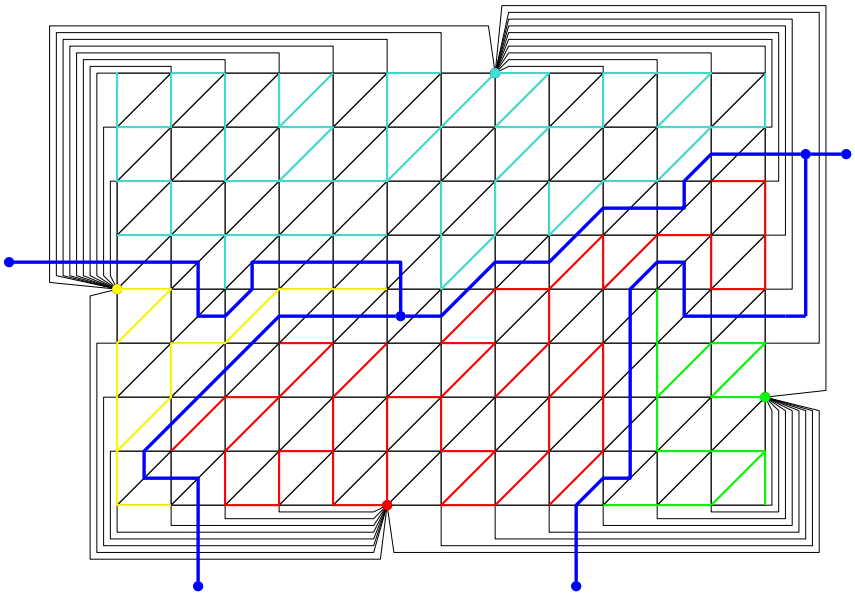

Figure 2: A planar graph (black edges) with four sites on the infinite face together with the dual Voronoi diagram VD^* (in blue). The sites are shown together with their corresponding shortest path trees (in turquoise, red, yellow, and green). Two of the Voronoi vertices (in blue) are real.

assume without loss of generality that $u \in Q$ and $v \in P$. Then, the shortest path from u to v must go through a boundary node v_i of P. We therefore perform a point location query for v in each of the Voronoi diagrams stored for u and for some hole h of P. Let $s_1, \ldots, s_g$ be the sites returned by these queries, where $g = O(1)$ is the number of holes of P. The distance in G from u to s_i is $\omega(s_i)$, and the distance in P from s_i to v is stored in $T^P_{s_i}$. We compute the sum of these two terms for each s_i, and return the minimum sum computed.

Analysis. First note that the query-time is $O(\log n)$ since, at each step of the traversal, we either descend to a smaller piece in $O(1)$ time or terminate after having found the desired distance in $O(\log n)$ time by $O(1)$ queries to a point location structure.

Next, we analyze the space. Consider a piece R with $O(1)$ holes. Let $n(R)$ and $b(R)$ denote the number of nodes and boundary nodes of R, respectively. The trees T^R_u and the stored distances in G require a total of $O(b(R) \cdot n(R))$ space. Let R be further decomposed into pieces P and Q. We bound the space used by all Voronoi diagrams created for R. Recall that every Voronoi diagram and point location structure corresponds to a node u of P and a hole of Q, or vice versa. The size of each additively weighted Voronoi diagram stored for a node of P is $O(b(Q))$, so $O(n(P) \cdot b(Q))$ for all nodes of P. The additional space required by Theorem 1.1 is also $O(n(P) \cdot b(Q))$. Finally, for every node of R we record if it belongs to the Jordan curve separator used to further divide R, and, if not, to which of the resulting two pieces it belongs. This takes only $O(n(R))$ space. The total space for each piece R is thus $O(n(R) \cdot b(R))$ plus $O(n(P) \cdot b(Q) + n(Q) \cdot b(P))$ if R is decomposed into P and Q.

We need to bound the sum of $O(n(R) \cdot b(R))$ over all the pieces R. Consider all pieces $R_1, R_2, \ldots, R_s$ on the same level ℓ in the decomposition. Because these pieces are edge-disjoint, $\sum_i n(R_i) = O(n)$. Additionally, $b(R_i) = O(\sqrt{n}/c^\ell)$ for any i, where $c > 1$, so $\sum_i O(n(R_i) \cdot b(R_i)) = O(n^{1.5}/c^\ell)$. Summing over all levels ℓ, this is $O(n^{1.5})$. The sum of $O(n(P) \cdot b(Q) + n(Q) \cdot b(P))$ over all pieces R that are decomposed into P and Q can be analysed with the same reasoning to obtain that the total size of the oracle is $O(n^{1.5})$.

Finally, we analyze the preprocessing time. For each piece R, the preprocessing of Theorem 1.1 takes $O(n(R) \cdot b(R))$. Then, we compute $O(n(R))$ different additively weighted Voronoi diagrams for R. Each diagram is built in $O(n(R))$ time, and its representation is extended in $O(b(R))$ time to support point location with Theorem 1.1. The total preprocessing time for R is hence $O((n(R))^2)$, which sums up to $O(n^2)$ overall by Lemma 3.1. We also need to compute the distances between pairs of vertices in G. This can be also done in $O(n^2)$ total time by computing the shortest path tree rooted at each vertex in $O(n)$ [14].

4 Point Location in Voronoi Diagrams

In this section we prove Theorem 1.1. Let P be a piece (i.e., a planar graph), and S be a set of sites that lie on a single face (hole) h of P.

Our goal is to preprocess P once in $O(|P||S|)$ time and space, and then, given any additively weighted Voronoi diagram $\mathrm{VD}^*(S,\omega)$ (denoted VD^* for short), preprocess it in $O(|S|)$ time and space so as to answer point location queries in $O(\log|S|)$ time.

We assume that the hole h incident to all nodes in S is the external face. We assume that all nodes of P^*, except for h^*, have degree 3. This can be achieved by triangulating P with infinite length edges. We also assume that all nodes incident to the external face belong to S and denote them $s_1, s_2, \ldots, s_b$, according to their clockwise order on h. We assume $b \geq 3$ (for any constant b point location is trivial).

Recall that for a site u and a vertex v we define $d(u,v)$ as $\omega(u)$ plus the length of the u-to-v shortest path in P. We further assume that no Voronoi cell is empty. That is, we assume that, for every pair of distinct sites $u, u' \in S$, $\omega(u) < d(u',u)$. If this assumption does not hold, let S' be the subset of the sites whose Voronoi cells are non empty. We can embed inside the hole h infinite length edges between every pair of consecutive sites in S', and then again triangulate with infinite length edges. This results in a new face h' whose vertices are the sites in S'. Replacing S with S' and h with h' enforces the assumption. Note that since this transformation changes P, it is not suitable when working with Voronoi diagrams constructed by algorithms that preprocess P, such as the ones in [4,12] mentioned in Section 2. The transformation, however, is suitable, when computing Voronoi diagrams naively, as we do in this work. In Section 6 we prove Theorem 1.1 without this assumption. As we argued above, this generalization is not required for the distance oracles in this paper. It makes the theorem slightly stronger, but also complicates the proof.

4.1 Preprocessing P

The preprocessing for P consists of computing shortest path trees T_v for every boundary node $v \in S$, decorated with some additional information which we describe next. We stress that the additional information does not depend on any weights ω (which are not available at preprocessing time).

Let T_i be the shortest path tree in P rooted at s_i. For a technical reason that will become clear soon, we add some artificial vertices to T_i. For each face f of P other than h, we add an artificial vertex v_f whose embedding coincides with the embedding of the dual vertex f^*. Let y_f be closest vertex to s_i in P that is incident to f. We add a zero length arc $y_f v_f$ to T_i. Note that v_f is a leaf of T_i. Let $p_{i,f}$ be the shortest s_i-to-v_f path in T_i. We say that a vertex v of T_i is to the right (left) of $p_{i,f}$ if the shortest s_i-to-v path emanates right (left) of $p_{i,f}$. Note that, since v_f is a leaf of T_i, v is either right of $p_{i,f}$, left of $p_{i,f}$, or a vertex of $p_{i,f}$; the goal of adding the artificial vertices v_f is to guarantee that these are the only options. The following proposition can be easily obtained using preorder numbers and a lowest common ancestor (LCA) data structure [2] for T_i.

PROPOSITION 4.1. *There is a data structure with $O(|P|)$ preprocessing time that can decide in $O(1)$ time if for a given query vertex v and query face f, v is right of $p_{i,f}$, left of $p_{i,f}$, or a vertex of $p_{i,f}$.*

Proof. We assign to each vertex u its preorder number in a depth-first-search of T_i, where the descendants of u are visited according to their clockwise order in the embedding, starting from the parent edge of u in T_i.

A vertex u is a vertex of $p_{i,f}$ if and only if $u = \mathrm{LCA}(u, v_f)$. If $u \neq \mathrm{LCA}(u, v_f)$ then, by definition of preorder, u is right (left) of $p_{i,f}$ if the preorder number of u is lower (greater) than that of v_f. □

We compute and store the shortest path trees T_i rooted at each site s_i, along with preorder numbers and LCA data structures required by Proposition 4.1. This requires preprocessing time $O(|P||S|)$ by computing each T_i in $O(|P|)$ time [14], and can be stored in $O(|P||S|)$ space.

4.2 Handling a Voronoi diagram $\mathrm{VD}^*(S,\omega)$

We now describe how to handle an additively weighted (dual) Voronoi diagram $\mathrm{VD}^* = \mathrm{VD}^*(S,\omega)$. This consists of a preprocessing stage and a query algorithm. The description in this section relies on the fact that, under the assumption that each site is in its own Voronoi cell, VD^* is a tree.

LEMMA 4.1. *VD^* is a tree.*

Proof. Suppose that VD^* contains a cycle C^*. Since the degree of each copy of h^* is one, the cycle does not contain h^*. Therefore, since all the sites are on the boundary of the hole h, the vertices of P enclosed by C^* are 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 (cycle) going through h^*. Let C^* denote this 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, contradicting our assumption. Thus, the boundary of every Voronoi cell is a connected

subgraph of VD^*. Since the boundaries of the cell of s_i and the cell of s_{i+1} both contain the dual of the edge $s_i s_{i+1}$, it follows that the entire modified VD^* is connected. □

We briefly describe the intuition behind the design of the point location data structure. To find the Voronoi cell $\mathrm{Vor}(s)$ to which a query vertex v belongs, it suffices to identify an edge e^* of VD^* that is adjacent to $\mathrm{Vor}(s)$. Given e^* we can simply check which of its two adjacent cells contains v by comparing the distances from the corresponding two sites to v. Our point location structure is based on a *centroid decomposition* of VD^* into connected subtrees, and on the ability to determine, in constant time, which of the subtrees is the one that contains the desired edge e^*.

Preprocessing. 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 more than one edge. In every step of the centroid decomposition of VD^*, we work with a connected subtree T^* of 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 f^*, and partition T^* into the three subtrees T_0^*, T_1^*, T_2^* obtained by splitting f^* into three copies, one for each edge incident to f^*. Clearly, the depth of the recursive decomposition is $O(\log |S|)$. The decomposition can computed in $O(|S|)$ time and be represented as a ternary tree, which we call the *decomposition tree*, in $O(|S|)$ space.

Point location query. We first describe the structure that gives rise to the efficient query, and only then describe the query algorithm. Consider a centroid f^* used at some step of the decomposition. Let $s_{i_0}, s_{i_1}, s_{i_2}$ denote the three sites adjacent to f^*, listed in clockwise order along h. Let f be the face of P whose dual is f^*. Let y_0, y_1, y_2 be the three vertices of f, such that y_j is the vertex of f in $\mathrm{Vor}(s_{i_j})$. Let e_j^* be the edge of VD^* incident to f^* that is on the boundary of the Voronoi cells of s_{i_j} and $s_{i_{j-1}}$ (indices are modulo 3). Let T_j^* be the subtree of T that contains e_j^*. Let p_j denote the shortest s_j-to-v_f path. Note that the vertex preceding v_f in p_j is y_j. See Figure 3 (right).

LEMMA 4.2. *Let s be the site such that $v \in \mathrm{Vor}(s)$. If T^* contains all the edges of VD^* incident to $\mathrm{Vor}(s)$, and if v is closer to site s_{i_j} than to sites $s_{i_{j-1}}, s_{i_{j+1}}$ (indices are modulo 3), then one of the following is true:*

- *$s = s_{i_j}$,*
- *v is to the right of p_j and all the boundary edges of $\mathrm{Vor}(s)$ are contained in T_j^*,*
- *v is to the left of p_j and all the boundary edges of $\mathrm{Vor}(s)$ are contained in T_{j+1}^*.*

Proof. In the following, let $rev(q)$ denote the reverse of a path q. See Figure 3 for an illustration of the proof.

Let p be the shortest path from s_{i_j} to v. If p is a subpath of p_j, then $s = s_{i_j}$. Assume that p emanates right of p_j (the other case is symmetric). First observe that the path consisting of the concatenation $p_j \circ rev(p_{j-1})$ intersects VD^* only at f^*. This is because, apart from the artificial arc $y_j v_f$, each shortest path p_j is entirely contained in the Voronoi cell of s_j. Therefore, none of the subtrees $T_{j'}^*$ contains an edge dual to $p_j \circ rev(p_{j-1})$. Since the path $p_j \circ rev(p_{j-1})$ starts on h, ends on h and contains no other vertices of h, it partitions the embedding into two subgraphs, one to the right of $p_j \circ rev(p_{j-1})$, and the other to its left. Since e_j^* is the only edge of T^* that emanates right of $p_j \circ rev(p_{j-1})$, the only edges of T^* in the right subgraph are those of T_j^*.

Next observe that p does not cross p_j (since shortest paths from the same source do not cross), and does not cross p_{j-1} (since v is closer to s_{i_j} than to $s_{i_{j-1}}$). Since we assumed p emanates right of p_j, the only edges of T^* whose duals belong to p are edges of T_j^*. Consider the last edge e^* of p that is not strictly in $\mathrm{Vor}(s)$. If e^* does not exist then p consists only of edges of $\mathrm{Vor}(s_{i_j})$, so $s = s_{i_j}$. If e^* does exist then it is an incident to $\mathrm{Vor}(s)$. By the statement of the lemma all edges of VD^* incident to $\mathrm{Vor}(s)$ are in T^*. Therefore, by the discussion above, $e^* \in T_j^*$. We have established that some edge of VD^* incident to $\mathrm{Vor}(s)$ is in T_j^*. It remains to show that all such edges are in T_j^*. The only two Voronoi cells that are partitioned by the path $p_j \circ rev(p_{j-1})$ are $\mathrm{Vor}(s_{i_j})$ and $\mathrm{Vor}(s_{i_{j-1}})$. Since v is closer to s_{i_j} than to $s_{i_{j-1}}$, $s \neq s_{i_{j-1}}$. Hence either $s = s_{i_j}$, or all the edges of VD^* incident to $\mathrm{Vor}(s)$ are in T_j^*. □

We can finally state and analyze the query algorithm. We have already argued that, to locate the Voronoi cell $\mathrm{Vor}(s)$ to which v belongs, it suffices to show how to find an edge e^* incident to $\mathrm{Vor}(s)$. We start with the tree $T^* = \mathrm{VD}^*$ which trivially contains all edges of VD^* incident to $\mathrm{Vor}(s)$. We use the notation from Lemma 4.2. Note that we can determine in constant time which of the three sites s_{i_j} is closest to v by explicitly comparing the distances stored in the shortest path trees $T_{s_{i_j}}$. We use Proposition 4.1 to determine, in constant time, whether v is right of p_j, left of p_j, or a node on p_j. In the latter case, by Lemma 4.2, we can immediately infer that v is in the Voronoi cell

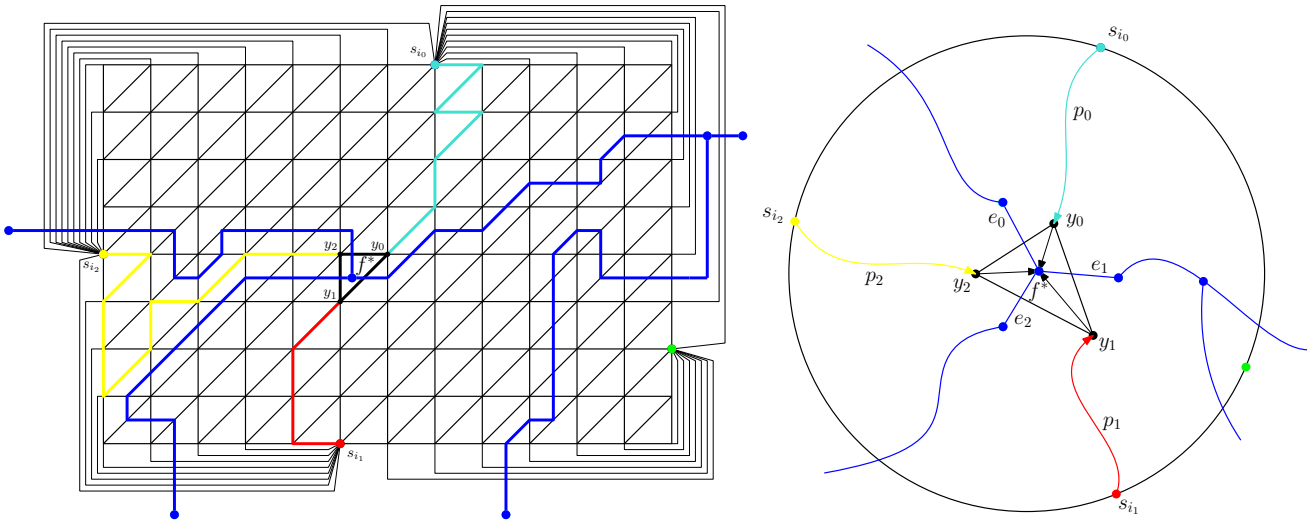

Figure 3: Illustration of the setting and proof of Lemma 4.2. Left: A decomposition of VD* (shown in blue) by a centroid f^* into three subtrees, and a corresponding partition of P into three regions delimited by the paths p_i (shown in red, yellow, and turquoise). Right: a schematic illustration of the same scenario.

of s_{i_j}. In the former two cases we recurse on the appropriate subtree containing all the edges of VD* incident to Vor(s). The total time is dominated by the depth of the centroid decomposition, which is $O(\log |S|)$.

5 The Tradeoff

In this section we generalize the construction presented in Section 3 to yield a smooth tradeoff between space and query-time. In the following, an MSSP data structure refers to Klein's multiple-source shortest paths data structure [5,20]. This data structure represents all the shortest path trees rooted at the vertices of a single face f in a planar graph using a persistent dynamic tree. It can be constructed in $O(n \log n)$ time, requires $O(n \log n)$ space, and can report any distance between a vertex of f and any other vertex in the graph in $O(\log n)$ time.

5.1 Preprocessing The data structure achieving the tradeoff is recursive using Jordan curve separators as described in Section 3; at each recursive level we have a piece $R = (V_R, E_R)$, which is decomposed by a Jordan curve separator C into $P = (V_P, E_P)$ and $Q = (V_Q, E_Q)$, where C is chosen to balance the number of nodes, the number of boundary nodes, or the number of holes, depending on the remainder modulo 3 of the recursive level. The main difference compared to the oracle of Section 3 is that we do not store an additively weighted Voronoi diagram of P for each node u in Q (and similarly we do not store a diagram of Q for each node of P). Instead, we use an r-division to decrease the number of stored Voronoi diagrams by a factor of $\sqrt{r}$. Additionally, we stop the decomposition when the number of vertices drops below r. More specifically, for every non-terminal piece R in the recursive decomposition such that $n(R) > r$ that is decomposed into P and Q with a Jordan curve separator C, we store the following:

1. For each hole h of P, an MSSP data structure capturing the distances in P from all the boundary nodes of P incident to h to all nodes of P. The MSSP data structure is augmented with predecessor and preorder information (see below).

2. An r-division for Q, denoted D_Q, with $O(1)$ MSSP data structures for each piece of D_Q, one for each hole of the piece. All the boundary nodes of Q (in particular, all nodes of C) are considered as boundary nodes of D_Q (see below).

3. For each boundary node u of D_Q, and for each boundary node v of P, the distance $d_G(u, v)$ from u to v in G, and also the distance $d_G(v, u)$ from v to u in G.

4. For each boundary node u of D_Q, and for each hole h of P, an additively weighted Voronoi diagram VD(S_h, ω) for P, where the set of sites S_h is the set of boundary nodes of P incident to the hole h, and the additive weights ω correspond to the distances in G from u to each site in S_h. We enhance

each Voronoi diagram with the point location data structure of Theorem 1.1.

We also store the same information with the roles of Q and P exchanged. For a terminal piece R, i.e. when $n(R) \leq r$, instead of further subdividing R we revert to the oracle of Fakcharoenphol and Rao [10], which needs $O(n(R)\log n(R))$ space, answers a query in $O(\sqrt{n(R)}\log^2 n(R))$ time, and can be constructed in $O(n(R)\log^2 n(R))$ time. We also construct an r-division D_R for R together with the MSSP data structures. The boundary nodes of R are considered as boundary nodes of D_R. For each boundary node $u \in \partial D_R$, and for each boundary node $v \in R$, we store the distance $d_G(u,v)$ from u to v in G and, for each hole of R, an enhanced additively weighted Voronoi diagram $\mathrm{VD}(S_h,\omega)$ for R, where the set of sites S_h is the set of boundary nodes of R incident to the hole h, and the additive weights ω correspond to the distances in G from u to each site in S_h.

The MSSP data structure in item 1 is a modification of the standard MSSP of Klein [20], where we change the interface of the persistent dynamic tree representing the shortest path tree rooted at the boundary nodes of R incident to h, as stated by the following lemma whose proof is in the appendix.

LEMMA 5.1. *Consider a directed planar embedded graph on n nodes with non-negative arc-lengths, and let $v_1, v_2, \ldots, v_s$ be the nodes on the boundary of its infinite face, in clockwise order. Then, in $O(n \log n)$ time and space, we can construct a representation of all shortest path trees T_i rooted at v_i, that allow answering the following queries in $O(\log n)$ time:*

- *for a vertex v_i and a vertex $v \in V$, return the length of the v_i-to-v path in T_i.*
- *for a vertex v_i and vertices $u, v \in V$, return whether u is an ancestor of v in T_i.*
- *for a vertex v_i and vertices $u, v \in V$, return whether u occurs before v in the preorder traversal of T_i.*

In the r-division in item 2 we extend the set of boundary nodes ∂D_Q of D_Q to also include all the boundary nodes of Q. In more detail, D_Q is obtained from Q by the same recursive decomposition process as the one used to partition G; on every level of the recursive decomposition we choose a Jordan curve separator as to balance the total number of nodes, boundary nodes, or holes, depending on the remainder of the level modulo 3, and terminate the recursion when the number of nodes in a piece is $O(r)$ and the number of boundary nodes is $O(\sqrt{r})$. Every piece of D_Q consists of $O(r)$ nodes and, because the boundary nodes of Q are incident to $O(1)$ holes, its $O(\sqrt{r})$ boundary nodes are incident to $O(1)$ holes. Because of the boundary nodes inherited from Q, the number of pieces in D_Q is not $O(n(Q)/r)$, and $|\partial D_Q|$ is not $O(n(Q)/\sqrt{r})$. We will analyze $|\partial D_Q|$ later. The MSSP data structures in item 2 stored for every piece of D_Q are the standard structures of Klein. The distances in item 3 are stored explicitly. The point location mechanism used for the Voronoi diagrams in item 4 is the one described in Section 4 with the following important modification. Instead of storing the shortest path trees rooted at every site of the Voronoi diagram explicitly to report distances, preorder numbers, and ancestry relations in $O(1)$ time, use the MSSP data structure stored in item 1. Clearly, with such queries one can implement Proposition 4.1 in $O(\log(|P|))$ time instead of $O(1)$.

5.2 Query To compute the distance from u to v, we traverse the recursive decomposition starting from the piece that corresponds to the whole initial graph G as in Section 3. Eventually, we reach a piece $R = (V_R, E_R)$ such that $u, v \in V_R$ and either $n(R) \leq r$, or $n(R) > r$ and R is decomposed into P and Q with a Jordan curve separator C such that either $u \in C$, or $v \in C$, or u and v are separated by C.

We first consider the case when $n(R) > r$. If $u \in C$ or $v \in C$ then, because the nodes of C are boundary nodes the r-divisions D_P and D_Q, the distance from u to v in G can be extracted from item 3. Otherwise, u and v are separated by C, and we assume without loss of generality that $u \in Q$ and $v \in P$. Let Q' be the piece of the r-division D_Q that contains u. Any path from u to v must visit a boundary node of Q'. Thus, we can iterate over the boundary nodes u' of Q', retrieve $d_{Q'}(u,u')$ (from the MSSP data structure in item 2), and then, for each hole h of P, use the Voronoi diagram $\mathrm{VD}(S_h,\omega)$ for P (item 4) to find the node $v' \in S_h$ that minimizes $d_G(u',v') + d_P(v',v)$ (computed from item 3 and item 1). The minimum value of $d_{Q'}(u,u') + d_G(u',v') + d_P(v',v)$ found during this computation corresponds to the shortest path from u to v.

The remaining possibility is that $n(R) \leq r$. Then the shortest path from u to v either visits some boundary node of R or not. To check the former case, we proceed similarly as above: we find the piece R' of the r-division D_R that contains u, iterate over the boundary nodes u' of R', retrieve $d_{R'}(u,u')$, and use the diagram $\mathrm{VD}(S_h,\omega)$ for R to find the node $v' \in S_h$ that minimizes $d_G(u',v') + d_R(v',v)$. To check the latter case, we query the oracle of Fakcharoenphol and Rao [10] stored for R, and return the minimum of these two distances.

5.3 Analysis For a piece R, we denote by $n(R)$ and $b(R)$ the number of nodes and boundary nodes of R, respectively. We first analyze the query-time. In $O(\log n)$ time we reach the appropriate piece R. Then, we iterate over $O(\sqrt{r})$ boundary nodes. For each of them, we first spend $O(\log r)$ time to retrieve the distance from u to u'. Then, we need $O(\log(b(P))\log(n(P)))$ time to query the Voronoi diagram. If $n(R) \leq r$, this changes into $O(\log(b(R))\log(n(R)))$ and additional $O(\sqrt{n(R)}\log^2(n(R))) = O(\sqrt{r}\log^2 r)$ time for the oracle of Fakcharoenphol and Rao [10]. Thus, the total query-time is $O(\sqrt{r}\log^2 n)$.

We bound the space required by the data structure for a piece R which is divided into pieces P and Q. Each MSSP data structure in item 1 requires $O(n(P)\log(n(P)))$ space, and there are $O(1)$ of them. Representing the r-division D_Q and the MSSP data structures for all the pieces in item 2 can be done within $O(n(Q)\log r)$ space. Then, for every boundary node of D_Q the distances in item 3 and the $O(1)$ Voronoi diagrams in item 4 can be stored in $O(b(P))$ space. Thus, we need to analyze the total number of boundary nodes of D. As we explained above, $|\partial D_Q|$ would be simply $O(n(Q)/r)$ if not for the additional boundary nodes of Q. We claim that $|\partial D_Q| = O(n(Q)/\sqrt{r}+b(Q))$.

To prove the claim we slightly modify the reasoning used by Klein, Mozes, and Sommer [21] to bound the total number of boundary nodes in an r-division without additional boundary vertices. They analyzed the same recursive decomposition process of a planar graph Q on n nodes by separating to balance the number of nodes, boundary nodes, or holes, depending on the remainder modulo 3 of the current level.[1] Let $\mathcal{T}$ be a tree representing this process, and $\hat{x}$ be the root of $\mathcal{T}$. Every node x of $\mathcal{T}$ corresponds to a piece. For example, the piece corresponding to the root $\hat{x}$ is all of Q. We denote by $n(x)$ and $b(x)$ the number of nodes and boundary nodes, respectively, of the piece corresponding to x. Define S_r to be the set of rootmost nodes y of $\mathcal{T}$ such that $n(y) \leq r$.

LEMMA 5.2. $\sum_{x\in S_r} b(x) = O(n(Q)/\sqrt{r} + b(Q))$

Proof. For a node x of $\mathcal{T}$ and a set S of descendants of x such that no node of S is an ancestor of any other, define $L(x,S) := -n(x)+\sum_{y\in S} n(y)$. Essentially, $L(x,S)$ counts the number of new boundary nodes with multiplicities created when replacing x by all pieces in S. Lemma 8 in [21] states that $L(\hat{x},S_r) = O(n/\sqrt{r})$. I.e., the number of new boundary nodes (with multiplicities) created when replacing the single piece Q by the pieces in S_r is $O(n(Q)/\sqrt{r})$. We assume that each node of Q has constant degree (this can be guaranteed with a standard transformation). Thus, each boundary vertex of Q appears in a constant number of pieces in S_r. Since the number of boundary vertices in Q is $b(Q)$, the lemma follows. □

Let $S'_r(x)$ be the set of rootmost descendants y of x such that $b(y) \leq c'\sqrt{r}$, where c' is a fixed known constant. The r-division found by the recursive decomposition process is $S'_r = \bigcup_{x\in S_r} S'_r(x)$. Indeed, each piece x in S'_r has $n(x) \leq r$, $b(x) \leq c'\sqrt{r}$, and $O(1)$ holes. This is true by definition of S'_r, even though, instead of starting with a graph with no boundary nodes, we start with a graph containing $b(Q)$ boundary nodes incident to $O(1)$ holes.

The following claim is proved in Lemma 9 of [21].

LEMMA 5.3. $|S'_r(x)| \leq \max\{1, \frac{40b(x)}{c'\sqrt{r}}\}$

COROLLARY 5.1. $|\partial D_Q| = O(n(Q)/\sqrt{r} + b(Q))$

Proof.

$$|\partial D_Q| \leq \sum_{x\in S'_r} b(x) \leq c'\sqrt{r} \sum_{x\in S_r} |S'_r(x)| \leq$$

$$c'\sqrt{r}\left(|S_r| + \frac{40}{c'\sqrt{r}} \sum_{s\in S_r} b(x)\right) = O(n(Q)/\sqrt{r} + b(Q)).$$

Here, the first inequality follows by definition of D_Q and S'_r. The second inequality follows by definition of $S'_r(x)$ and by the fact that for any $x \in S'_r$, $b(x) \leq c'\sqrt{r}$. The third inequality follows by Lemma 5.3. The last inequality follows from the fact that $|S_r| = O(n(Q)/r)$, and from Lemma 5.2. □

We have shown that, starting the recursive decomposition of Q with $b(Q)$ boundary nodes incident to $O(1)$ holes, we obtain an r-division D consisting of pieces containing $O(r)$ nodes and $O(\sqrt{r})$ boundary nodes incident to $O(1)$ holes, and $O(n(Q)/\sqrt{r} + b(Q))$ boundary nodes overall. Consequently, the space required by the data structure for a piece R with $n(R) > r$ that is separated into pieces P and Q is $O(n(P)\log(n(P) + n(Q)\log r + n(Q)/\sqrt{r} + b(Q))b(P))$, plus a symmetric term with the roles of P and Q exchanged. If $n(R) \leq r$ the space is $O(n(R)\log(n(R)))+(n(R)/\sqrt{r}+b(R))b(R))$. Overall, $O(n(R)\log(n(R))$ sums up $O(n\log n\log(n/r))$. On the ℓ-th level of the decomposition, $O(n(P)/\sqrt{r}\cdot b(Q) + n(Q)/\sqrt{r}\cdot b(P))$ sums up to $O(n^{1.5}/(\sqrt{r}\cdot c^\ell))$, so $O(n^{1.5}/\sqrt{r})$ over all levels. $O(b(P)\cdot b(Q))$ can be bounded by $O(b(Q)\cdot\sqrt{n})$, so we only need to bound the total number of boundary nodes in all pieces of the

[1] In [21] simple cycle separators are used (rather than Jordan curve separators), and thus every piece along the recursion needs to be re-triangulated. The analysis of the number of boundary nodes, however, is the same.

recursive decomposition of the whole graph. For the terminal pieces R, it directly follows from Lemma 5.2 (with Q being the entire graph G) that the total number of boundary nodes is $O(n/\sqrt{r})$, but we also need to analyse the non-terminal pieces. Because the size of a piece decreases by a constant factor after at most three steps of the recursive decomposition process, it suffices to bound only the total number of boundary nodes for pieces in the sets S_{r_i}, for $r_i = r \cdot 2^i$, $i = 0, 1, \ldots, \log(n/r)$. By applying Lemma 5.2, with $r = r_i$ and $Q = G$ we get that the total number of boundary nodes for pieces in S_{r_i} is $O(n/\sqrt{r_i})$, which sums up to $O(n/\sqrt{r})$ over all i. Thus, the sum of $O(b(P) \cdot b(Q))$ over all non-terminal pieces $O(n^{1.5}/\sqrt{r})$. For all terminal pieces R, $O(n(R)/\sqrt{r} \cdot b(R))$ adds up to $O(n^{1.5}/\sqrt{r})$. $O(b(R) \cdot b(R))$ can be bounded by $O(b(R) \cdot \sqrt{n})$, which we have already shown to be $O(n^{1.5}/\sqrt{r})$ overall. Thus, the total space is $O(n^{1.5}/\sqrt{r} + n \log n \log(n/r))$.

The preprocessing time can be analyzed similarly as in Section 3, except that now we need to compute only $O(n(P)/\sqrt{r} + b(P))$ Voronoi diagrams for P, each in $\tilde{O}(n(P))$ time. As shown above, the overall number of boundary nodes is $O(n/\sqrt{r})$, so this is $\tilde{O}(n^2/\sqrt{r})$ total time. Additionally, we need to compute the distance between pairs of vertices of P in G (item 3). One of these vertices is always a boundary node of the r-division, so overall we need $\tilde{O}(n/\sqrt{r})$ single-source shortest paths computations in G, which takes $\tilde{O}(n^2/\sqrt{r})$ total time. Additionally, we need to construct the oracles when $n(R) \leq r$ in $O(n \log^2 r)$ total time. Thus, the total construction time is $\tilde{O}(n^2/\sqrt{r})$ overall.

5.4 Improved query-time The final step in this section is to replace $\log^2 n$ with $\log n \cdot \log r$ in the query-time. This is done by observing that the augmented MSSP data structure takes linear space, but for smaller values of r we can actually afford to store more data. In the appendix we show the following lemma.

LEMMA 5.4. *For any $r \in [1, n]$, the representation from Lemma 5.1 can be modified to allow answering queries in $O(\log r)$ time in $O(s \cdot n/\sqrt{r} + n \log r)$ space after $O(s \cdot n/\sqrt{r} \log r + n \log n)$ time preprocessing.*

This decreases the query-time to $O(\sqrt{r} \log n \log r)$ at the expense of increasing the space taken by the MSSP data structures in item 1 to $O(n(P)/\sqrt{r} \cdot b(P) + n(P) \log r)$. Summing over all levels ℓ and including the space used by all other ingredients, this is $O(n^{1.5}/\sqrt{r} + n \log r \log(n/r))$.

6 Removing the Assumption on Sites

We now remove the assumption that all the vertices on the hole h are sites of the Voronoi diagram whose Voronoi cells are non-empty. Recall that $\mathrm{VD}^*(S, \omega)$ is obtained from VD_1^* by replacing h^* with multiple copies, one copy h_e^* for each edge e of $\mathrm{VD}_0^*(S, \omega)$ incident to h^*. Consider the proof of Lemma 4.1. The argument showing that $\mathrm{VD}^*(S, \omega)$ contains no cycles still holds. However, because now vertices incident to the hole are not necessarily sites, the argument showing connectivity fails, and indeed, $\mathrm{VD}^*(S, \omega)$ might be a forest. We turn the forest VD^* into a tree $\widehat{\mathrm{VD}}^*$ by identifying certain pairs of copies of h^* as follows. Consider the sequence E_h of edges of VD^* incident to h^*, ordered according to their clockwise order on the face h. Each pair of consecutive edges e, e' in E_h delimits a subpath Q of the boundary walk of h. Note that Q belongs to a single Voronoi cell of some site $s \in S$. If Q does not contain s we connect the two copies h_e^* and $h_{e'}^*$ with an artificial edge. We denote the resulting graph by $\widehat{\mathrm{VD}}^*$. See Figure 4.

LEMMA 6.1. $\widehat{\mathrm{VD}}^*$ *is a tree.*

Proof. We show that $\widehat{\mathrm{VD}}^*$ is connected and has no cycles. Consider the Voronoi cell of a site $s \in S$. In VD_1^* (i.e., before splitting h^*) the boundary of this cell is a non-self-crossing cycle C^*. Consider the restriction of C to edges incident to h^*. Consider two consecutive edges e, e' in the restriction. If e and e' are not consecutive in C^* (i.e., if e and e' do not meet at h^*), then they remain connected in VD^* (i.e., after splitting h^*). If e and e' are consecutive on C^*, then they are also consecutive in E_h, so they become disconnected in VD^*, but get connected again in $\widehat{\mathrm{VD}}^*$. This is true unless the subpath Q of the face h delimited by e and e' contains s, but this only happens for one pair of edges in C^*. Therefore, since C^* was 2-connected (a cycle) in VD_1^*, it is 1-connected in $\widehat{\mathrm{VD}}^*$. Now, since adjacent Voronoi cells share edges, the boundaries of any two adjacent Voronoi cells are connected. It follows from the fact that the dual graph of $\widehat{\mathrm{VD}}^*$ is connected that the boundaries of all cells are connected after the identification step.

Assume that $\widehat{\mathrm{VD}}^*$ contains a cycle C^*. Then C^* must also be a cycle in VD_1^*. Since every cycle in VD_1^* contains h^* and encloses at least one site, C^* contains a copy of h^* and encloses at least one site. Consider a decomposition of C^* into maximal segments between copies of h^*. By construction of $\widehat{\mathrm{VD}}^*$, whenever two segments of C^* are connected with an artificial edge, the segment of the boundary of h delimited by these two segments and enclosed by C^* does not contain a site. Since all the sites are on the boundary of h, it follows that C^* does not enclose any sites, a contradiction. □

We next describe how to extend the point location

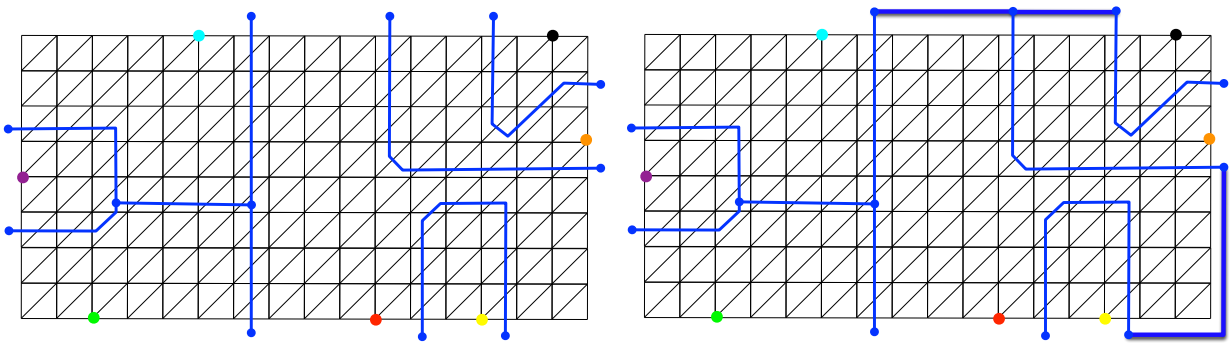

Figure 4: Left: a Voronoi diagram VD^* (blue) forms a forest. Right: the tree $\widehat{\mathrm{VD}}^*$ obtained by adding three artificial edges (thicker blue lines).

data structure. First observe that since $\widehat{\mathrm{VD}}^*$ is a tree, it has a centroid decomposition. In VD^*, the copies of h^* are all leaves. In $\widehat{\mathrm{VD}}^*$ each copy of h^* is incident to at most two artificial edges. Hence the maximum degree in $\widehat{\mathrm{VD}}^*$ is still 3. If the centroid of $\widehat{\mathrm{VD}}^*$ is not a copy of h^* then Lemma 4.2 holds. We need a version of Lemma 4.2 for the case when the centroid is a copy of h^* with degree greater than 1 (i.e., incident to one or two artificial edges). This is in fact a simpler version of Lemma 4.2. The difference between a copy of h^* and a Voronoi vertex f^* is that f^* is a triangular face incident to three specific vertices y_0, y_1, y_2, whereas h^* is incident to all vertices of the hole h. Recall that we connect two copies h^*_e and $h^*_{e'}$ if the segment Q of the boundary of h delimited by the edges e and e' belongs to the Voronoi cell of a site s but does not contain s. When we add this artificial edge, we associate with h^*_e and with $h^*_{e'}$ an arbitrary primal vertex y on Q. Thus, each copy f of h^* is associated with at most two primal vertices.

We describe the case where the centroid of T^* is a copy $\hat{h}^*$ of h^* with degree 3. The case of degree 2 and one associated vertex is similar. In the case of degree 3, $\hat{h}^*$ is incident to one edge $e_1 \in E_h$, and to two artificial edges which we denote e_0, and e_2, so that the counterclockwise order of edges around $\hat{h}^*$ is e_0, e_1, e_2. Removing $\hat{h}^*$ breaks T^* into three subtrees. Let T^*_j be the subtree of T^* rooted at the endpoint of e_j that is not $\hat{h}^*$. Recall that, since the degree of $\hat{h}^*$ is 3, it has two associated vertices, y_0, y_1, where y_j belongs to the subpath of the boundary of h delimited by e_j and e_{j+1}. Let s_{i_j} be the site such that y_j belongs to $\mathrm{Vor}(s_{i_j})$. Let p_j be the shortest path from s_{i_j} to y_j. See Figure 5.

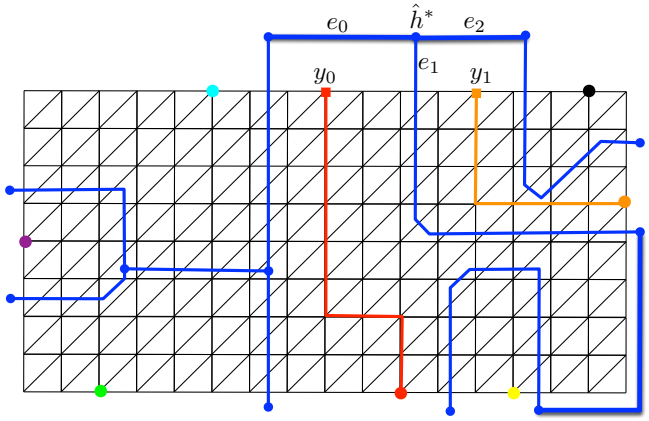

Figure 5: Illustration of the case when the centroid is a copy $\hat{h}^*$ of h^* of degree three. $\hat{h}^*$ has two incident artificial edges (e_0, e_2), and one Voronoi edge (e_1). The two vertices y_0, y_1 associated with $\hat{h}^*$ are shown, as well as the shortest paths p_0 and p_1.

LEMMA 6.2. *Let s be the site such that $v \in \mathrm{Vor}(s)$. If T^* contains all the edges of $\widehat{\mathrm{VD}}^*$ incident to $\mathrm{Vor}(s)$, and if v is closer to site s_{i_j} than to site $s_{i_{j+1}}$ (indices are modulo 2), then one of the following is true:*

- $s = s_{i_j}$,
- *v is to the left of p_j and all the edges of $\widehat{\mathrm{VD}}^*$ incident to $\mathrm{Vor}(s)$ are contained in T^*_j,*
- *v is to the right of p_j and all the edges of $\widehat{\mathrm{VD}}^*$ incident to $\mathrm{Vor}(s)$ are contained in T^*_{j+1}.*

Proof. Observe that all the edges of p_j belong to $\mathrm{Vor}(s_{i_j})$, while for every $i \in \{0, 1, 2\}$, the duals of edges of T^*_i have endpoints in two different Voronoi cells. Therefore, the paths p_j do not cross the trees T^*_i. Since p_0 and p_1 are paths that start and end on the boundary of h and do not cross each other, they partition G into three subgraphs $\{G_i\}_{i=0}^2$. Let G_0 be the subgraph to the left of p_0, G_1 the subgraph to the right of p_0 and to

the left pf p_1, and G_2 the graph to the right of p_1. It follows from the above that each subtree T_i^* belongs to the subgraph G_i.

The remainder of the proof is almost identical to that of Lemma 4.2. Let p be the shortest path from s_{i_j} to v. If p is a subpath of p_j then $s = s_{i_j}$. Otherwise, assume p emanates left of p_0 (the other cases are similar). Consider the last edge e^* of p that is not strictly in $\mathrm{Vor}(s)$. If e^* does not exist then $s = s_{i_0}$. If it does exist, then it must be en edge of T_0^*. Since the only Voronoi cell partitioned by p_0 is that of s_{i_0}, either $s = s_{i_0}$, or all edges of $\widehat{\mathrm{VD}}^*$ incident to Vor_s belong to T_0^*. □

References

[1] S. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *ESA*, pages 514–528, 1996.

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.

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

[4] S. Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. In *SODA*, pages 2143–2152, 2017.

[5] S. Cabello, E. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.

[6] D. Z. Chen and J. Xu. Shortest path queries in planar graphs. In *STOC*, pages 467–478, 2000.

[7] V. Cohen-Addad, S. Dahlgaard, and C. Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. In *FOCS*, pages 962–973, 2017.

[8] H. Djidjev. On-line algorithms for shortest path problems on planar digraphs. In *WG*, pages 151–165, 1996.

[9] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

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

[11] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[12] P. Gawrychowski, H. Kaplan, S. Mozes, M. Sharir, and O. Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time. In *SODA*, 2018. To appear.

[13] D. Hartvigsen and R. Mardon. The all-pairs min cut problem and the minimum cycle basis problem on planar graphs. *Journal of Discrete Mathematics*, 7(3):403–418, 1994.

[14] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[15] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, 1977.

[16] K. Kawarabayashi, P. N. Klein, and C. Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *ICALP*, pages 135–146, 2011.

[17] K. Kawarabayashi, C. Sommer, and M. Thorup. More compact oracles for approximate distances in undirected planar graphs. In *SODA*, pages 550–563, 2013.

[18] P. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *SODA*, pages 820–827, 2002.

[19] P. Klein and S. Mozes. Optimization algorithms for planar graphs. http://planarity.org. Book draft.

[20] P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA*, pages 146–155, 2005.

[21] P. N. Klein, S. Mozes, and C. Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *STOC*, pages 505–514, 2013. Full version at https://arxiv.org/abs/1208.2223.

[22] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci.*, 32(3):265–279, 1986.

[23] R. Motwani and P. Raghavan. *Randomized algorithms*. Press Syndicate of the University of Cambridge, 1995.

[24] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *SODA*, pages 209–222, 2012.

[25] S. Mozes and C. Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log\log n)$ time. In *Proc. 18th Eur. Symp. Algorithms (ESA)*, pages 206–217, 2010.

[26] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[27] Y. Nussbaum. Improved distance queries in planar graphs. In *WADS*, pages 642–653, 2011.

[28] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[29] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

[30] F. van Walderveen, N. Zeh, and L. Arge. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. In *SODA*, pages 901–918, 2013.

[31] C. Wulff-Nilsen. Algorithms for planar graphs and graphs in metric spaces, 2010. PhD thesis, University of Copenhagen.

[32] C. Wulff-Nilsen. Approximate distance oracles for planar graphs with improved query time-space tradeoff. In *SODA*, pages 351–362, 2016.

A Missing Proofs

LEMMA 3.1. *Choosing the separators as described above guarantees that (i) each piece has $O(1)$ holes, (ii) the number of nodes in a piece on the ℓ-th level in the decomposition is $O(n/c_1^{\ell/3})$, for some constant $c_1 > 1$, (iii) the number of boundary nodes in a piece on the ℓ-th level in the decomposition is $O(\sqrt{n}/c_2^{\ell/3})$, for some constant $c_2 > 1$.*

Proof. The number of holes increases by at most one in every recursive call, but decreases by a constant multiplicative factor every 3 recursive calls, and is initially equal to 0, so part (i) easily follows. The number of nodes never increases, and decreases by a constant multiplicative factor every 3 recursive calls, and is initially equal to n, so part (ii) follows. The situation with the number of boundary nodes is slightly more complex, because it increases by $O(\sqrt{n})$ in every recursive call, and decreases by a constant multiplicative factor every 3 recursive calls, where n is the number of nodes in the current piece. For simplicity, we analyze a different process, in which the number of boundary nodes decreases by a constant multiplicative factor and then increases by $O(\sqrt{n})$ in every recursive call. The asymptotic behavior of these two processes is identical. Thus, we want to analyze the following recurrence:

$$b(\ell+1) = b(\ell)/c + \frac{\sqrt{n}}{(c')^\ell}.$$

for some constants $c, c' > 1$. Then

$$b(\ell+1) = \sum_{i=0}^{\ell} \frac{\sqrt{n}}{c^i (c')^{\ell-i}} = \frac{\sqrt{n}}{(c')^\ell} \sum_{i=0}^{\ell} (\frac{c'}{c})^i.$$

We consider two cases:

1. $c' \leq c$, then $b(\ell+1) \leq \sqrt{n}\frac{\ell}{(c')^\ell} \leq \frac{\sqrt{n}}{(\sqrt{c'})^\ell}$ for ℓ large enough.
2. $c' > c$, then $b(\ell+1) = O(\frac{\sqrt{n}}{c^\ell})$.

In both cases, $b(\ell) = O(\frac{\sqrt{n}}{c_2^\ell})$ for some constant $c_2 > 1$ as claimed. □

LEMMA 5.1. *Consider a directed planar embedded graph on n nodes with non-negative arc-lengths, and let $v_1, v_2, \ldots, v_s$ be the nodes on the boundary of its infinite face, in clockwise order. Then, in $O(n \log n)$ time and space, we can construct a representation of all shortest path trees T_i rooted at v_i, that allow answering the following queries in $O(\log n)$ time:*

- *for a vertex v_i and a vertex $v \in V$, return the length of the v_i-to-v path in T_i.*
- *for a vertex v_i and vertices $u, v \in V$, return whether u is an ancestor of v in T_i.*
- *for a vertex v_i and vertices $u, v \in V$, return whether u occurs before v in the preorder traversal of T_i.*

Proof. We proceed as in the original implementation of MSSP, that is, we represent every T_i with a persistent link-cut tree. We discuss some of the details that are required for explaining how to implement the queries.

In MSSP, we start with constructing T_1 with Dijkstra's algorithm in $O(n \log n)$ time. Then, we iterate over $i = 2, 3, \ldots, s$. The current T_i is maintained with a persistent link-cut tree of Sleator and Tarjan [28]. The gist of MSSP is that every edge of the graph goes in and out of the shortest path tree at most once, and that we can efficiently retrieve the edges that should be removed from and added to T_{i-1} to obtain T_i (in $O(\log n)$ time per edge). Thus, if we are able to remove or insert an edge from T_{i-1} in $O(\log n)$ time, the total update time is $O(n \log n)$. With a link-cut tree, we can indeed remove or insert edge in such time (note that we prefer the worst-case version instead of the simpler implementation based on splay trees). We make our link-cut tree partially persistent with a straightforward application of the general technique of Driscoll et al. [9]. This requires that the in-degree of the underlying structure is $O(1)$, which is indeed the case if the degrees of the nodes in the graph (and hence in every T_i) are $O(1)$. This can be guaranteed by replacing a node of degree $d > 4$ by a cycle on d nodes, where every node has degree 3. We now verify that the in-degree is $O(1)$ for such a structure by presenting a high-level overview of link-cut trees.

The edges of a rooted tree are partitioned into solid and dashed. There is at most one solid edge incoming into any node, so we obtain a partition of the tree into node-disjoint solid paths. For every solid path, we maintain a balanced search tree on a set of leaves corresponding to the nodes of the path in the natural top-bottom order when read from left to right. To obtain a worst-case time bound, Sleator and Tarjan use biased binary trees. Every node stores a pointer to the leaf in the corresponding biased binary tree, and additionally the topmost node of a heavy path stores a pointer to its parent in the represented tree (together with the cost of the corresponding edge). The nodes of every biased binary tree store standard data (a pointer to the left child, the right child, and the parent) and, additionally, every inner node (that corresponds to a fragment of a solid path) stores the total cost of the corresponding fragment. An additional component of the link-cut tree is a complete binary tree on n leaves corresponding to the nodes of the tree (called $1, 2, \ldots, n$). This is required, so that we can access a

node of the tree on demand in $O(\log n)$ time, as random access is not allowed in this setting. The access pointer points to the root of the complete binary tree. One can indeed verify that the in-degree of the structure is $O(1)$.

Assuming that a representation of every T_i with a partially persistent link-cut tree is available, we can answer the queries as follows.

First, consider calculating the distance from v_i to some $v \in V$. We retrieve the access pointer of T_i and navigate the complete binary tree to reach the node v. Then, we navigate up in the link-cut representation of T_i starting from v. In every step, we traverse a biased binary tree starting from a leaf corresponding to some ancestor u of v. Conceptually, this allows us to jump to the topmost node of the solid path containing u. While doing so, we accumulate the total cost of the prefix of that solid path ending at u. Then, we follow the pointer from the topmost node of the current solid path to reach its parent in T_i, add its cost to the answer, and continue in the next biased binary tree. It is easy to see that in the end we obtain the total cost of the path from v to the root of T_i, and by the properties of biased binary trees the total number of steps is $O(\log n)$.

Second, consider checking if u is an ancestor of v in T_i. We navigate in the link-cut representation of T_i starting from u and marking the visited solid paths (in more detail, every solid path stores a timestamp of the most recent visit; the timestamps are not considered a part of the original partially persistent structure and the current time is increased after each query). Then, we navigate starting from v, but stop as soon as we reach a solid path already visited in the previous step. For u to be an ancestor of v, this must be the path containing u, and furthermore u must be on the left of v in the corresponding biased binary tree. This can be all checked in $O(\log n)$ time.

Third, consider checking if u occurs before v in the preorder traversal of T_i. By proceeding as in the previous paragraph we can identify the LCA of u and v, denoted w. Assuming that $w \neq u$ and $w \neq v$, we can also retrieve the edge outgoing from w leading to the subtree containing u, and similarly for v, in $O(\log n)$ total time. We can also retrieve the edge incoming to w from its parent in $O(1)$ additional time. Then, we check the cyclic order on the edges incident to w in the graph to determine if u comes before v in the preorder traversal of T_i (this is so that we do not need to think about an embedding of T_i while maintaining the link-cut representation). □

LEMMA 5.4. *For any $r \in [1, n]$, the representation from Lemma 5.1 can be modified to allow answering queries in $O(\log r)$ time in $O(s \cdot n/\sqrt{r} + n \log r)$ space after $O(s \cdot n/\sqrt{r} \log r + n \log n)$ time preprocessing.*

Proof. We construct an r-division R of the graph. The structure consists of parts: micro components and macro component.

Consider a shortest path tree T_i. We construct a new (smaller) tree T_i' as follows. First, mark in T_i v_i and all boundary nodes of R. Then, T_i' is the tree induced by the marked nodes in T_i (in other words: for any two marked nodes we also mark their LCA in T_i, and then construct T_i' by connecting every marked node to its first marked ancestor with an edge of length equal to the total length of the path connecting them in T_i. Then, $|T_i'| = O(n/\sqrt{r})$. Intuitively, T_i' gives us a high-level overview of the whole T_i. We augment it with the usual preorder numbers and LCA data structure. We call this the macro component.

For every piece R_j of R, consider the subgraph of T_i consisting of all edges belonging to R_j. This subgraph is a collection of trees rooted at some of the boundary nodes of R_j. We represent this forest $T_{i,j}$ with a persistent link-cut forest. While sweeping through the nodes $v_1, v_2, \ldots, v_s$, every edge of the graph goes in and out of the shortest path tree at most once. Hence, every edge of R_j goes in and out of $T_{i,j}$ at most once. Thus, the persistent link-cut representation of $T_{i,j}$ takes $O(|R| \log |R|)$ time and space. To answer a query concerning $T_{i,j}$, we first need to retrieve the corresponding version of the link-cut forest. This can be done with a predecessor search, if we store a sorted list of the values of i together with a pointer to the corresponding version, in $O(\log r)$ time, as there are at most $O(|R|)$ versions. Then, a query concerning $T_{i,j}$ can be answered in $O(\log r)$ time.

We claim that combining the micro and macro components allows us to answer any query in $O(\log r)$ time. Consider calculating the distance from v_i to some $v \in V$. We retrieve the piece R_j containing v and, by using the micro component, find the root r of the tree containing v in the forest $T_{i,j}$ together with the distance from r to v in $O(\log r)$ time. Then, r is a boundary node, so the macro component allows us to find the distance from v_i to r in $O(1)$ time. Other queries can be processed similarly by first look at the pieces containing u and v, then replacing them by appropriate boundary nodes, and finally looking at the macro component.

The total space is clearly $O(n \log r)$ to represent all the micro components, and $O(s \cdot n/\sqrt{r})$ for the macro component. To bound the preprocessing time, observe that constructing the macro component requires extracting $O(n/\sqrt{r})$ nodes from the persistent link-cut representation of T_i. If, instead of accessing them one by one, we work with all of them at the same time, can be seen to take $O(n/\sqrt{r} \log r)$ time by the convexity of log. □