

Faster Construction of a Planar Distance Oracle with $\tilde{O}(1)$ Query Time*

Itai Boneh^{1,2}, Shay Golan^{1,2}, Shay Mozes¹, Daniel Prigan¹, and Oren Weimann²

¹Reichman University, Israel

²University of Haifa, Israel

Abstract

We show how to preprocess a weighted undirected n -vertex planar graph in $\tilde{O}(n^{4/3})$ time, such that the distance between any pair of vertices can then be reported in $\tilde{O}(1)$ time. This improves the previous $\tilde{O}(n^{3/2})$ preprocessing time [JACM'23].

Our main technical contribution is a near optimal construction of *additively weighted Voronoi diagrams* in undirected planar graphs. Namely, given a planar graph G and a face f , we show that one can preprocess G in $\tilde{O}(n)$ time such that given any weight assignment to the vertices of f one can construct the additively weighted Voronoi diagram of f in near optimal $\tilde{O}(|f|)$ time. This improves the $\tilde{O}(\sqrt{n|f|})$ construction time of [JACM'23].

1 Introduction

In this paper we investigate the following question: *How fast can you preprocess a planar graph so that subsequent distance queries can be answered in near-optimal $\tilde{O}(1)$ time?* Imagine you could do the preprocessing in near-optimal $\tilde{O}(n)$ time. This would be truly remarkable, since it would allow us to efficiently convert between standard graph representations, which support quick local (e.g., adjacency) queries, into a representation that supports quick non-local distance queries. In a sense, this can be thought of as an analogue for distances of the Fast Fourier Transform (which converts in $\tilde{O}(n)$ time between the time domain and the frequency domain). Such a tool would allow us to develop algorithms for planar graphs that can access any pairwise distance in the graph essentially for free! Unfortunately, we are still far from obtaining $\tilde{O}(n)$ preprocessing time. The fastest preprocessing time is $\tilde{O}(n^{3/2})$ [9]. We make a step in this direction by improving the preprocessing time to $\tilde{O}(n^{4/3})$ in weighted undirected planar graphs.

Formally, a *distance oracle* is a data structure that can report the distance $\text{dist}(u, v)$ between any two vertices u and v in a graph G . Distance oracles for planar graphs have been studied extensively in the past three decades both in the exact [2, 4, 9, 10, 14–16, 19, 21, 23, 28, 34, 38, 40, 44] and in the approximate [8, 24–27, 31, 42, 45] settings. In the approximate setting, Thorup [42] presented a near-optimal oracle that returns $(1 + \varepsilon)$ -approximate distances (for any constant ε) in $\tilde{O}(1)$ time and requires $\tilde{O}(n)$ space and construction time (see also [8, 24–27, 31, 45] for polylogarithmic improvements). Is it possible that the same near-optimal bounds can be achieved without resorting to approximation? We next review the rich history of *exact* oracles in planar graphs.

The history of exact planar distance oracles. Let Q and S denote the query-time and the space of an oracle, respectively. The early planar distance oracles [2, 14, 16] were based solely on *planar separators* [20, 33, 35] and achieved a tradeoff of $Q = \tilde{O}(n/\sqrt{S})$ for $S \in [n^{4/3}, n^2]$, and $Q = O(n^2/S)$ for $S \in [n, n^{4/3}]$. In [19], Fakcharoenphol and Rao introduced the use of *Monge matrices* to distance oracles, and devised an oracle with $\tilde{O}(n)$ space and $\tilde{O}(\sqrt{n})$ query time. By combining their ideas with Klein's [6, 27] *multiple source shortest path* (MSSP) data structure, Mozes and Sommer [38] obtained the $Q = \tilde{O}(n/\sqrt{S})$ tradeoff for nearly the full range $[n \log \log n, n^2]$. Other works [21, 38, 40, 44] focused on achieving strictly optimal query-time or

*This research was partially funded by Israel Science Foundation grant 810/21.

strictly optimal space. Namely, Wulff-Nilsen’s work [44] gives optimal $O(1)$ queries with weakly subquadratic $O(n^2 \log^4 \log n / \log n)$ space, whereas Nussbaum [40] and Mozes and Sommer’s [38] oracles give optimal $O(n)$ space with $O(n^{1/2+\epsilon})$ query-time. Except for [44], all of the above oracles can be constructed in $\tilde{O}(n)$ time. However, none of them provides polylogarithmic $\tilde{O}(1)$ query-time using truly subquadratic $O(n^{2-\epsilon})$ space.

In FOCS 2017, Cohen-Addad, Dahlgaard, and Wulff-Nilsen [15] (inspired by Cabello’s [5] breakthrough use of *Voronoi diagrams* for computing the diameter of planar graphs) realized that Voronoi diagrams, when applied to regions of an r -division, can be used to break the barrier mentioned above. In particular, they presented the first oracle with $\tilde{O}(1)$ query-time and truly subquadratic $O(n^{5/3})$ space, or more generally, the tradeoff $Q = \tilde{O}(n^{5/2}/S^{3/2})$ for any $S \in [n^{3/2}, n^{5/3}]$. This came at the cost of increasing the preprocessing time from $\tilde{O}(n)$ to $O(n^2)$, which was subsequently improved to match the space bound [22]. In SODA 2018, Gawrychowski, Mozes, Weimann, and Wulff-Nilsen [23] improved the space and preprocessing time to $\tilde{O}(n^{3/2})$ with $\tilde{O}(1)$ query-time (and the tradeoff to $Q = \tilde{O}(n^{3/2}/S)$ for $S \in [n, n^{3/2}]$) by defining a dual representation of Voronoi diagrams and developing an efficient *point-location* mechanism on top of it. In STOC 2019, Charalampopoulos, Gawrychowski, Mozes, and Weimann [10] observed that the same point-location mechanism can be used on the Voronoi diagram for the complement of regions in the r -division. This observation alone suffices to improve the oracle size to $O(n^{4/3})$ (while maintaining $\tilde{O}(1)$ query-time). By combining this with a sophisticated recursion (where a query at recursion level i reduces to $\log n$ queries at level $i+1$) they further obtained an oracle of size $n^{1+o(1)}$ and query-time $n^{o(1)}$. Finally, in SODA 2021, Long and Pettie [34] showed how much of the point-location work can be done without recursion, and that only two (rather than $\log n$) recursive calls suffice. This led to the state of the art oracle, requiring $n^{1+o(1)}$ space and $\tilde{O}(1)$ query-time or $\tilde{O}(n)$ space and $n^{o(1)}$ query-time.¹

In terms of space and query time, these latter Voronoi-based oracles are almost optimal. However, their construction time is $\tilde{O}(n^{3/2})$, and improving it is mentioned in [9] as an important open problem. The bottleneck behind this $\tilde{O}(n^{3/2})$ bound is the time for constructing Voronoi diagrams as we next explain.

Point-location in Voronoi diagrams. Let X be a planar graph, and let f be a face of X . The vertices of f are called the *sites* of the Voronoi diagram, and each site s has a weight $\omega(s) \geq 0$ associated with it. The distance between a site s and a vertex $v \in X$, denoted by $\text{dist}^\omega(s, v)$ is defined as $\omega(s)$ plus the length of the s -to- v shortest path in X . The *additively weighted Voronoi diagram* $\text{VD}(f, \omega)$ is a partition of X ’s vertices into pairwise disjoint sets, one set $\text{Vor}(s)$ for each site s . The set $\text{Vor}(s)$, called the *Voronoi cell* of s , contains all vertices in X that are closer (w.r.t. $\text{dist}^\omega(\cdot, \cdot)$) to s than to any other site (we assume that distances are unique to avoid the need to handle ties²). A *point-location* query v asks for the site s whose Voronoi cell $\text{Vor}(s)$ contains v . There exists a dual representation $\text{VD}^*(f, \omega)$ of $\text{VD}(f, \omega)$. The size of this representation is $O(|f|)$, and together with an MSSP data structure it supports point-location queries in $\tilde{O}(1)$ time.

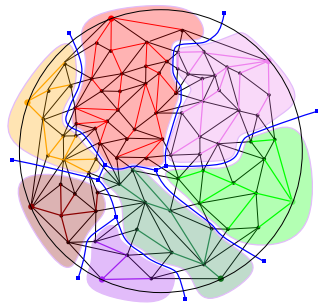


Figure 1: A graph (piece) X . The vertices of X ’s infinite face f are the sites of the Voronoi diagram VD . Each site is represented by a unique color, which is also used to shade its Voronoi cell. The dual representation VD^* is illustrated as the blue tree. The tree has 7 leaves (corresponding to 7 copies of f^*) and 5 internal nodes (corresponding to the 5 trichromatic faces of VD). The edges of the tree correspond to contracted subpaths in X^* .

Our results and techniques. In [9], it was shown that given the MSSP of a graph (piece) X , a face f of X , and additive weights $\omega(\cdot)$ to the vertices of f , the Voronoi diagram $\text{VD}^*(f, \omega)$ can be constructed in $\tilde{O}(\sqrt{n|f|})$ time. In this paper, we show (see Theorem 1) how to construct it in near-optimal $\tilde{O}(|f|)$ time when the graph is undirected. As discussed below, this leads to a static distance oracle of space $\tilde{O}(n^{4/3})$, construction

¹A journal version containing all the above Voronoi-based oracles was published in JACM in 2023 [9].

²This assumption can be achieved in $O(n)$ time. With high probability using [36, 39], or deterministically using [18].

time $\tilde{O}(n^{4/3})$, and query-time $\tilde{O}(1)$. In addition, it implies a *dynamic* distance oracle that supports $\tilde{O}(1)$ -time distance queries from a single-source s , and $\tilde{O}(n^{2/3})$ -time updates consisting of edge insertions, edge deletions, and changing the source s . The dynamic distance oracle is obtained by simply plugging our construction into the oracle of Charalampopoulos and Karczmarz [12] (thus improving its update time from $\tilde{O}(n^{4/5})$ to $\tilde{O}(n^{2/3})$). We note that the current best *all-pairs* dynamic oracle by Fakcharoenphol and Rao [19] has $\tilde{O}(n^{2/3})$ update time and $\tilde{O}(n^{2/3})$ query time. Therefore, in undirected graphs, our new oracle achieves the same bounds but has the benefit that, between source changes, the query takes only $\tilde{O}(1)$ time.

Our construction is inspired by Charalampopoulos, Gawrychowski, Mozes, and Weimann who showed in ICALP 2021 [11] that near-optimal $\tilde{O}(|f|)$ construction time is possible for the special case when the planar graph is an *alignment graph* of two strings. The alignment graph is an acyclic grid graph of constant degree. More importantly, shortest paths in this graph are monotone, in the sense that they only go right or down along the grid. This makes the Voronoi diagram of every piece X much more structured. In particular, one can find the trichromatic vertices of VD^* (i.e., the faces of X whose vertices belong to three different Voronoi cells) by recursively zooming in on them using cycle separators; The intersection of a cycle separator with each Voronoi cell consists of at most two contiguous intervals of the cycle separator. This partition induced by these intervals can be found with a binary search approach, whose complexity (up to a logarithmic factor) is linear in the number of trichromatic vertices enclosed by the cycle. From the partition it is easy to deduce whether a trichromatic vertex of VD^* is enclosed by the cycle separator, and infer whether we should recurse on each side of the separator. In this paper we extend this binary search approach to general undirected planar graphs.

At a first glance, such an extension seems impossible as the intersection of cycle separators and Voronoi cells might be very fragmented, even when there are only three sites (see Fig. 2). To overcome this, we use *shortest path separators*. Such separators have been widely used in essentially all approximate distance oracles for planar graphs, but also in exact algorithms (e.g., [3, 7, 30, 37]). These cycle separators may contain $\Theta(n)$ edges (rather than $O(\sqrt{n})$), but consist of two shortest paths P, P' , so that, because the graph is undirected, any other shortest path can intersect each of P, P' at most once. In our context, this implies a monotonicity property when we restrict our attention to shortest paths that enter the cycle separator only from the left or only from the right. If we only allow paths to enter P from one side, say the left, then the intersection of each Voronoi cell (with distances now defined under this restriction) with P is a single contiguous interval.

To better explain the difficulties with this approach and how we overcome them, let us first define some terminology. We say that vertex $v \in P$ *prefers* a site s if v belongs to $\text{Vor}(s)$ (defined without any restrictions). We say that $v \in P$ *left-prefers* a site s if v belongs to $\text{Vor}(s)$ under the restriction that paths are only allowed to enter P from the left. Repeating the above discussion using this terminology, the partition induced by the *prefers* relation may be very fragmented, but the partition induced by the *left-prefers* relation is not fragmented. There are two immediate problems with working with the partition induced by *left-preference* (or *right-preference*). First, we are interested in constructing the Voronoi diagram without any restrictions, and second, we do not know how to efficiently compute shortest paths under restrictions on the direction in which they enter P .

To address the first difficulty we show that one can infer which sides of the cycle separator contain a trichromatic vertex of the Voronoi diagram (without restrictions) by inspecting the endpoints of the intervals of the two partitions - the one with respect to *left-preference* and the one with respect to *right-preference*. This is shown in Section 4. We also show, by adapting the divide-and-conquer construction algorithm of [22], how to reduce the problem of computing a Voronoi diagram with many sites to the problem of computing the trichromatic vertex of a Voronoi diagram with only three sites. This is explained in Section 5.

Overcoming the second difficulty is the heart of our technical contribution. We know how to efficiently determine, by enhancing the standard MSSP data structure, whether the true shortest path from a site s to a vertex $v \in P$ enters P from the right or from the left. However, we do not know how to efficiently report the shortest path from s to v that enters P from the other side (because that path is not the overall shortest path from s to v). We observe that we can work with *relaxed* preferences. If v *prefers* s (without restrictions on left/right), then: (1) we might as well say that v both *right-prefers* and *left-prefers* s (because

either way this will point us to s), and (2) if the shortest path from s to v enters P from the right, we do not really care what the left preference of v is, so we might as well say that v left-prefers s' for any site s' . In Section 3 we describe how to compute a partition with respect to such a relaxed notion of left-preference and right-preference. The exact details are a bit more complicated than described here. In particular we use the term *like* rather than *prefer* to express that the preference is more loose.

Constructing a partition with relaxed preferences turns out to be more complicated than with strict preferences. One challenge comes from the fact that with relaxed preferences making progress in a binary search procedure is problematic. This is because a vertex v of P no longer has a unique left-preferred site s . In fact, when working with relaxed preferences, a vertex v may equally like all sites. Thus, we need to have some way to recognize a “winner” site with respect to a vertex that does not have a unique preference (see Fig. 5). Another challenge we encounter is what we call *swirly* paths - when the s -to- v path goes around P before entering P (see Fig. 3). Swirly paths make the realizations of many of the arguments outlined here more complicated. The structure of the shortest paths that allows us to make progress in the binary search procedure is different and more complicated in the presence of swirly paths. Moreover, we do not always know how to efficiently identify whether a shortest path is swirly or not. To simplify the presentation we first explain how to obtain a partition with respect to relaxed preferences when there are no swirly paths, and in Appendix C describe how to handle swirly paths.

We believe that our result is a promising step toward exact oracles that simultaneously have optimal space, query-time and preprocessing time. One way to achieve this would be to be able to find a way to use recursion to obtain the functionality of our enhanced MSSP data structure without actually computing it over and over in large regions of the graph, in a similar manner to the way this issue was avoided for the standard MSSP in [9]). This seems to be the only significant obstacle preventing us from pushing our new ideas all the way through, and obtaining an almost optimal $n^{1+o(1)}$ construction time for undirected planar graphs.

2 Preliminaries

Representation of Voronoi diagrams. With a standard transformation (without increasing the size of G asymptotically) we can guarantee that each vertex of G has constant degree and that G is triangulated. Consider a subgraph X (called a *piece*) of G whose boundary vertices ∂X (vertices incident to edges in $G \setminus X$) lie on a single face f of X . Note that f is the only face of X that is not a triangle. The vertices of f , assigned with weights $\omega(\cdot)$, are the sites of the Voronoi diagram $\text{VD} = \text{VD}(f, \omega)$ of X . There is a dual representation VD^* of VD as a tree with $O(|f|)$ vertices (see Fig. 1): Let X^* be the planar dual of X . Consider the subgraph of X^* consisting of the duals of edges uv of X such that u and v are in different Voronoi cells. In this subgraph, we repeatedly contract edges incident to degree-2 vertices. The remaining vertices (edges) are called *Voronoi vertices (edges)*. A Voronoi vertex is dual to a face whose three vertices belong to three different Voronoi cells. We call such a face (and its corresponding dual vertex) *trichromatic*. Finally, we define VD^* to be the tree obtained from the subgraph by replacing the node f^* by multiple copies, one for each edge incident to f^* . The complexity (i.e., the number of vertices and edges) of VD^* is $O(|f|)$. For example, in a VD of just two sites s and t , there are no trichromatic faces. In this case, VD^* is just a single edge corresponding to an st -cut or equivalently to a cycle in the dual graph. We call this cycle the st -bisector and denote it by $\beta^*(s, t)$. Note that a trichromatic vertex is a meeting point of three bisectors. Another important example is a VD of three sites (we call this a *trichromatic VD*). Such a VD has at most one trichromatic face (in addition to face f itself).

Using point-location for distance oracles. To see why point-location on Voronoi diagrams is useful for distance oracles, we describe here the $\tilde{O}(n^{4/3})$ -space $\tilde{O}(1)$ -query oracle of [10] that we will be using. It begins with an r -division of the graph G . This is a partition of G into $O(n/r)$ subgraphs (called *pieces*) such that every piece X contains $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices ∂X (vertices shared by more than one piece). An r -division can be computed in $\tilde{O}(n)$ time [29] with the additional property that the boundary vertices ∂X of every piece X lie on a constant number of faces of the piece (called *holes*). To

simplify the presentation, we assume that ∂X lies on a single hole (in general, we apply the same reasoning to each hole separately, and return the minimum distance found among the $O(1)$ holes). The oracle consists of the following for each piece X of the r -division:

1. The $O(|X|^{3/2})$ space, $O(|X|^{3/2})$ construction time, $\tilde{O}(1)$ query-time distance oracle of [23] on the graph X . In total, these require $O(n\sqrt{r})$ space and construction time.
2. Two MSSP data structures [27], one for X and one for $X^{out} = G \setminus (X \setminus \partial X)$, both with sources ∂X . The MSSP for X requires space $O(r \log r)$, and the MSSP for X^{out} requires space $O(n \log n)$. Using these MSSPs, we can then query in $\tilde{O}(1)$ time the u -to- v distance for any $u \in \partial X$ and $v \in G$. The total space and construction time of these MSSPs is $\tilde{O}(n^2/r)$, since there are $O(n/r)$ pieces.
3. For each vertex u of X , compute the Voronoi diagram $\text{VD}_{in}(u, X)$ (resp. $\text{VD}_{out}(u, X)$) for X (resp. X^{out}) with sites ∂X and additive weights the distances from u to these vertices in G (the additive weights are computed in $\tilde{O}(|\partial X|) = \tilde{O}(\sqrt{r})$ time using [19]). Each Voronoi diagram can be represented in $O(\sqrt{r})$ space [23] and can be constructed in $\tilde{O}(\sqrt{n\sqrt{r}})$ time [10]. Hence, all Voronoi diagrams require $O(n\sqrt{r})$ space and $\tilde{O}(n^{3/2}r^{1/4})$ construction time.

To query a u -to- v distance, let X be the piece that contains u . If $v \notin X$ then the u -to- v path must cross ∂X . We perform a point-location query for v in $\text{VD}_{out}(u, X)$ in time $\tilde{O}(1)$ [23]. If $v \in \text{Vor}(s)$ then we return the u -to- s distance (from the precomputed additive weight) plus the s -to- v distance (from the MSSP of X^{out}). Otherwise, $v \in X$ and we return the minimum of two options: (1) The shortest u -to- v path crosses ∂X . This is similar to the previous case except that the point-location query for v is done in $\text{VD}_{in}(u, X)$. (2) The shortest u -to- v path does not cross ∂X (i.e., the path lies entirely within X). We retrieve this distance by querying the distance oracle stored for X .

By choosing $r = n^{2/3}$ we get an oracle of space $\tilde{O}(n^{4/3})$ and query-time $\tilde{O}(1)$. The construction time however is $\tilde{O}(n^{5/3})$. Notice that the only bottleneck preventing $\tilde{O}(n^{4/3})$ construction time is the construction of the Voronoi diagrams.

Shortest path separators. A *shortest path separator* Q is a balanced cycle separator consisting of two shortest paths P and P' (emanating at the same vertex) plus a single edge. A complete recursive decomposition tree T of G using shortest path separators can be obtained in linear time [32].

An arc $e = uv$ emanates (enters) left of a simple path P if there exist two arcs e_1, e_2 in P , such that u (v) is the head of e_1 and the tail of e_2 , and e appears between e_1 and e_2 in the clockwise order of arcs incident to u . Otherwise, e emanates (enters) right of P . Notice that this is not defined for the endpoints of P , however, in the context in which we will use it, we will always extend P so that its original endpoints will become internal. Specifically, consider a shortest path separator Q composed of an x -to- y path P , an x -to- y' path P' , and an edge (y, y') . Then, we will extend P on one side with y' and on the other side with the vertex following x on P' . A symmetric extension will be done for P' .

3 Efficient Computation of a Relaxed Partition

In this section we define a relaxed partition, and introduce an algorithm that computes it in $\tilde{O}(1)$ time.

3.1 Definitions and data structures

In our settings, let F denote the face, and let $H \subseteq F$ denote a subset of vertices (sites) of F (for the most part, we will use $|H| = 3$ in order to find trichromatic faces, see Section 4). We say that vertex $v \in P$ *prefers* a site $c \in H$ if v belongs to $\text{Vor}_H(c)$. It is natural to partition the vertices of P according to the site in H that they prefer. In the case of the alignment graph of [11], each part in this partition is a contiguous interval of P . In general planar graphs however, this is not the case, and the parts can be very fragmented (see Fig. 2).

We observe that the partition would induce contiguous intervals if we only consider shortest paths that are allowed to enter P from the right (or only from the left). In that case we can say that a vertex $v \in P$

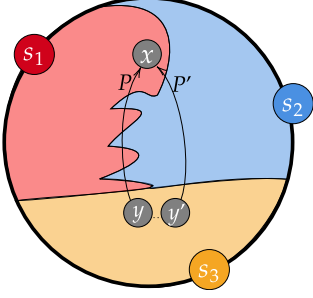


Figure 2: The partition of P into parts corresponding to the Voronoi cells of $\text{VD}(s_1, s_2, s_3)$ is very fragmented.

right-prefers (left-prefers) c . We know how to determine efficiently whether the true shortest path from a site c to a vertex $v \in P$ enters P from the right or from the left. But the problem is that then we do not know how to efficiently report the shortest path from c to v that enters P from the other side (because it is not the overall shortest path from c to v).

To overcome this issue we observe that we can work with relaxed preferences, which we call *like*. If v prefers c (without restrictions on left/right), then we might as well say that v both right-likes and left-likes c (because either way this will point us to c). Similarly, if v prefers c (again, without restrictions on left/right) then if the shortest path from c to v enters P from the right, we do not really care what the left like of v is (since in this case shortest paths entering P from the left are just irrelevant), so we may as well say that v left-likes c' for any site c' .

Recall that the endpoints of P are x and y . We denote by s_x and s_y the sites of H such that $x \in \text{Vor}_H(s_x)$ and $y \in \text{Vor}_H(s_y)$. In practice we will use the left-like (and symmetrically the right-like) relation with respect to a subset $S \subseteq H$ of sites. If the true site in $S \cup \{s_x, s_y\}$ that v prefers (without restrictions on left/right) is not one of the sites in S then we may as well say that v both left-likes and right-likes all sites in S because eventually they will have no effect on the true answer.

The following definition formalizes the above discussion.

Definition 1 ((S , left)-like). *Let P be an x -to- y shortest path, let $H = \{h_1, h_2, \dots, h_{|H|}\}$ be a cyclic subsequence of a face F , and let $S = \{s_1, s_2, \dots, s_{|S|}\}$ be a subsequence of H . Let $s_x \in H$ (resp. s_y) be the site such that $x \in \text{Vor}_H(s_x)$ (resp. $y \in \text{Vor}_H(s_y)$). A vertex $v \in P$ that belongs to the Voronoi cell $\text{Vor}_{S \cup \{s_x, s_y\}}(c)$ is said to (S , left)-like s_i if at least one of the following holds: (1) $c = s_i$, (2) the shortest path from c to v enters P from the right, or (3) $c \notin S$.*

This notion of (S , left)-like, induces a *relaxed partition* of P , in which the parts do form contiguous intervals along P . The relaxed partition has the property that for every $v \in P$ if $v \in \text{Vor}_H(c)$ and c reaches P from some *direction* (left or right) then in the $(H, \text{direction})$ -relaxed partition v is in the part of P corresponding to site c . In other words, together, the (H, left) and (H, right) -relaxed partitions assign to each vertex of P at most two sites in H , and it is guaranteed that one of these two sites is the true site of v .

Definition 2 (Relaxed Partition). *Let P be a shortest path, and let S and H be cyclic subsequences of a face F with $S \subseteq H$, denoted by $S = \{s_1, s_2, \dots, s_{|S|}\}$ and $H = \{h_1, h_2, \dots, h_{|H|}\}$.*

A relaxed (S , left)-partition of P w.r.t. S is a partition of P into $|S|$ disjoint subpaths $P_i = P[u_i, v_i]$ such that every $x \in P_i$ is a vertex that (S , left)-likes s_i .

The definitions of (S , right)-like and of an (S , right)-relaxed partition are symmetric. The main part of our algorithm is dedicated to finding a relaxed partition. First, in Section 3.3 we will show how to find a relaxed partition of P w.r.t. $S \subset H$ consisting of only two sites. Then, in Section 3.4, we explain how to obtain a relaxed partition of P w.r.t. H by combining the relaxed partitions of pairs of sites in H .

For two vertices u, v we denote the shortest path in G between u and v as $R_{u,v}$. For a site $s \in F$ we denote by $\text{Left}(P, s)$ the set of vertices $v \in P$ such that $R_{s,v}$ enters P from the left.

The following lemma describes a data structure that will later (Lemma 2) allow to find an (F , left)-partition of P . A similar proof finds an (F , right)-partition.

Lemma 1 (Enhanced MSSP data structure). *Given a planar graph G , a face f and a recursive decomposition T of G , one can construct in $\tilde{O}(n)$ time a data structure supporting the following queries, each in $\tilde{O}(1)$ time. For any source $s \in f$, shortest path $P \in T$, and vertex $v \in P$:*

1. $\text{dist}(s, v)$ - return the distance between s and v in G .
2. $\text{direction}(s, v, P)$ - return whether $R_{s,v}$ enters P from left or right.
3. $\text{count}(s, P, i, j)$ for $i, j \in P$ - return the number of vertices x on $P[i, j] \cap \text{Left}(P, s)$.
4. $\text{select}(s, P, i, j, k)$ for $i, j \in P$ - return the k 'th vertex x of $P[i, j] \cap \text{Left}(P, s)$.
5. $\text{ancestor}(s, v, d)$ - return the ancestor of v that is in depth d in the shortest paths tree of s .

The proof of Lemma 1 is deferred to Appendix A. In the rest of this section we prove the following lemma.

Lemma 2. *Given F , H , and T , together with their enhanced MSSP, and a shortest path $P \in T$, one can compute an (H, left) -partition of P , in $\tilde{O}(|H|) = \tilde{O}(1)$ time.*

3.2 Setup of the query

Recall that the endpoints of P are x and y , and that s_x and s_y are the sites of H such that $x \in \text{Vor}_H(s_x)$ and $y \in \text{Vor}_H(s_y)$. Notice that it is straightforward to find s_x and s_y in $\tilde{O}(|H|) = \tilde{O}(1)$ time by explicitly checking the distance from each of the sites in H to x and to y using the enhanced MSSP. We denote $R_x = R_{s_x, x}$ and $R_y = R_{s_y, y}$. We assume $s_x \neq s_y$, since the case $s_x = s_y$ is a degenerated case.³ We also assume that $H \cap P = \emptyset$, since otherwise we find a partition for $H \setminus P$, and then add the intervals of the sites in $H \cap P$, using a binary search along P . The sites s_x and s_y partition the face F into two intervals which we denote by F_{left} and F_{right} . In addition, let $H_{\text{left}} = F_{\text{left}} \cap H$ and $H_{\text{right}} = F_{\text{right}} \cap H$. The following lemma (see proof in Appendix B) states that when computing the (H, left) -partition, we can ignore every $s \in H_{\text{right}}$. Thus, every time we consider a partition with respect to some $S \subseteq H_{\text{left}}$, we consider the vertices of S from the closest vertex to s_x to the closest vertex to s_y (on F_{left}). In addition, the partition is always such that the order of the parts corresponding to vertices of S is from the one containing x to the one containing y .

Lemma 3. *Every $(H_{\text{left}}, \text{left})$ -partition is an (H, left) -partition.*

Remark 1. *When considering a partition of P with respect to a set $S \subseteq H_{\text{left}}$, the parts of P in our partition are chosen so that they are consistent with the order of the vertices in S . For example, the first part, corresponds to the vertex of S closest to s_x , contains x (unless it is empty) and the part of the vertex closest to s_y , contains y (unless it is empty).*

Trimming P . By definition $x \in \text{Vor}_H(s_x)$ and for every $v \in R_x$ we also have $v \in \text{Vor}_H(s_x)$. Therefore, w.l.o.g., we assume that $R_x \cap P$ contains only the vertex x , since all the vertices of $R_x \cap P$ are known to belong to $\text{Vor}_H(s_x)$. Similarly, we assume that $R_y \cap P = \{y\}$.

Swirly paths. For vertices $s \in F_{\text{left}}$ and $v \in P$, we say that $R_{s,v}$ is a *swirly* path if $v \in \text{Left}(P, s)$ and $R_{s,v}$ crosses R_x or R_y . Moreover, a path is called *x-swirly* (resp. *y-swirly*) if the first path it crosses among R_x and R_y is R_x (resp. R_y). See Fig. 3.

Swirly paths complicate life. We shall first (in Section 3.3) explain how to obtain an (S, left) -partition P when $|S| = 2$ and there are no swirly paths. Next, in Appendix C.1 we show how to obtain an $(\{s_x, s_y\}, \text{left})$ partition. Then, in Appendix C.2, we will show how to use the partition for $\{s_x, s_y\}$ to obtain the partition for any $|S| = 2$, even if there are swirly paths. Finally, in Section 3.4 we will show how to obtain an (H, left) -partition of P , removing the restriction to two vertices and proving Lemma 2.

³If $s_x = s_y$ one can think of the graph obtained by making an incision along R_x , which allows us to think of s_x as being split into two vertices, where one is the site of x and the other is the site of y , see also Remark 2.

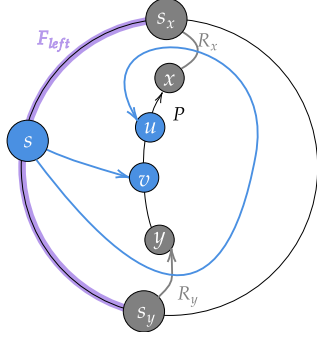


Figure 3: The path P with endpoints x and y . We think of P as being oriented from y to x . The corresponding sites s_x and s_y with $R_x = R_{s_x,x}$ and $R_y = R_{s_y,y}$. The site s is on F_{left} , the path $R_{s,v}$ is a non-swirly path, and the path $R_{s,u}$ is a y -swirly path.

3.3 Partition with respect to two sites

In this subsection we show how to compute an (S, left) -partition when $|S| = 2$. We start with a special case, where we consider a subpath \hat{P} of P that is not involved in any swirly paths. Later, in Lemma 5 we introduce the algorithm for computing an (S, left) -partition when $|S| = 2$.

Lemma 4. *Let $s_1, s_2 \in F_{\text{left}} \setminus P$ be two sites such that s_1 is closer to s_x on F_{left} than s_2 . Let $\hat{P} = P[a, b]$ be a subpath of P with a being closer to x than b , such that for every $i \in \{1, 2\}$ and $v \in \text{Left}(\hat{P}, s_i)$ the path $R_{s_i,v}$ is non-swirly. One can compute in $\tilde{O}(1)$ time an $(\{s_1, s_2\}, \text{left})$ -partition of \hat{P} into \hat{P}_1, \hat{P}_2 such that $a \in \hat{P}_1$ and $b \in \hat{P}_2$, unless the partition is trivial.*

The proof of Lemma 4 consists of several steps. We first introduce the notion of *winning*, and show (Claim 1) that it is a monotone property of vertices of \hat{P} . We will then use this monotonicity to perform binary search. For a vertex $v \in \hat{P}$ we say that s_1 *wins at v* if there exists an s_1 -to- v non-swirly path R that enters P from the left and $\text{len}(R) < \text{dist}(s_2, v)$. We define s_2 winning at v symmetrically.

To avoid clutter we assume $\hat{P} = P$, otherwise one just has to change in the following proofs P to \hat{P} and x, y to the endpoints of \hat{P} .

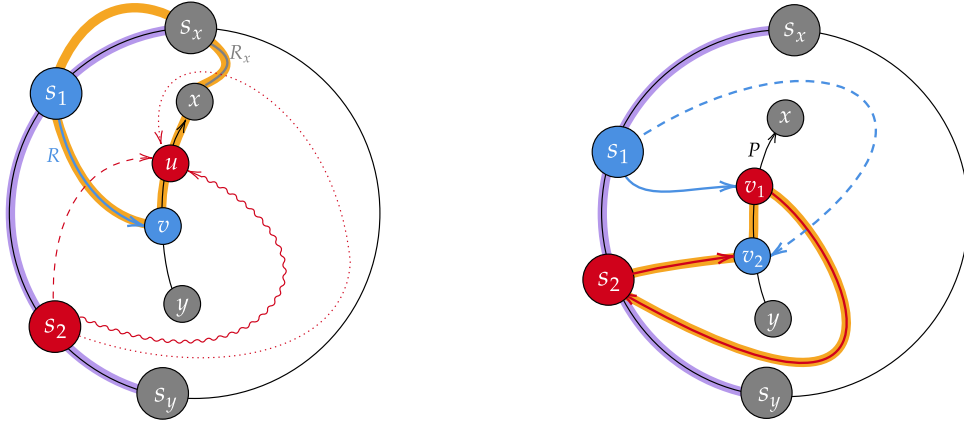


Figure 4: Left: The (orange) cycle C in the proof of Claim 1. The three options for $R_{s_2,u}$ are in red. Dashed red contradicts v being closer to s_1 than to s_2 , dotted red contradicts x being closer to s_x than to s_2 , and wavy red contradicts $R_{s_2,u}$ entering P from the left. Right: The (orange) cycle C in the proof of Claim 2. Since s_1 is (strictly) on one side of C and v_2 is on the other side of C , the dashed R_{s_1,v_2} path must cross C in either $P[v_2, v_1]$ or R_{s_2,v_2} . Both these crosses are impossible.

Claim 1. *Let $v \in P$ such that s_1 wins at v . Then, every $u \in P[x, v]$ is a vertex that $(\{s_1, s_2\}, \text{left})$ -likes s_1 . Symmetrically, if s_2 wins at v , then every $u \in P[v, y]$ is a vertex that $(\{s_1, s_2\}, \text{left})$ -likes s_2 .*

Proof. **TOPROVE 0** □

Having proved that the winning property is sufficient to perform a binary search to obtain a partition of P , we next show how to find a pair (s_w, v_t) (where $w \in \{1, 2\}$ and $t \in \{1, 2\}$) such that s_w is a winner at v_t . We distinguish two scenarios that are handled differently (Claims 2 and 3). Later we will put it all together and show how to make this into an efficient binary search procedure.

Claim 2. *Let v_1 and v_2 be vertices such that $v_1 \in \text{Left}(P, s_1)$ and $v_2 \in \text{Left}(P, s_2)$. If v_1 is closer on P to x than v_2 then there is an $\tilde{O}(1)$ -time algorithm that outputs $w \in \{1, 2\}$ and $t \in \{1, 2\}$ such that s_w wins at v_t .*

Proof. **TOPROVE 1** □

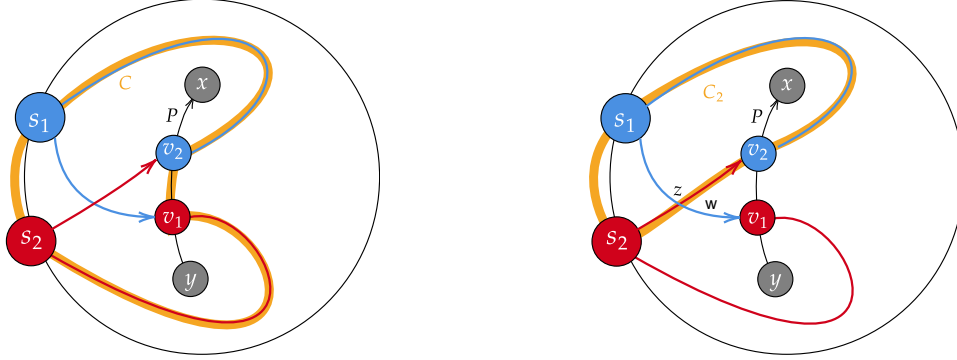


Figure 5: The (orange) cycles C (left image) and C_2 (right image) in the proof of Claim 3. The situation here is that s_1 reaches v_1 from the left and loses to s_2 that reaches v_1 from the right, and the symmetric issue for v_2 . Thus, v_1 and v_2 both $(\{s_1, s_2\}, \text{left})$ -likes both s_1 and s_2 . To recognize a winner we detect which site is closer to z .

Claim 3. *Let v_1 and v_2 be vertices such that $v_1 \in \text{Left}(P, s_1)$ and $v_2 \in \text{Left}(P, s_2)$. If v_2 is closer on P to x than v_1 then there is an $\tilde{O}(1)$ -time algorithm that outputs $w \in \{1, 2\}$ and $t \in \{1, 2\}$ such that s_w wins at v_t .*

Proof. **TOPROVE 2** □

Using Claims 1 to 3 we introduce a recursive binary search partition algorithm, completing the proof of Lemma 4. In each recursive step, the algorithm works on a subpath $\bar{P} = P[a, b]$, and returns a partition of \bar{P} into a (possibly empty) prefix \bar{P}_1 and a (possibly empty) suffix \bar{P}_2 such that every $v \in \bar{P}_i$ is a vertex that $(\{s_1, s_2\}, \text{left})$ -likes s_i (for $i \in \{1, 2\}$).

When working on $\bar{P} = P[a, b]$, the algorithm starts by finding a ‘left median’ vertex of s_1 in $P[a, b]$. Formally, a left median of s_1 in $P[a, b]$ is a vertex $v_1 \in P[a, b]$ such that the shortest path between s_1 and v_1 enters P from the left, and the number of vertices whose shortest path from s_1 enters P from the left in $P[a, v_1]$ differs by at most 1 from the number of vertices whose shortest path from s_1 enters P from the left in $P[v_1, b]$. The algorithm finds left medians v_1 and v_2 for s_1 and s_2 , respectively, in $P[a, b]$ by using a count query to the enhanced MSSP, and then applying binary search using **select** and **count** queries.

If no left-median exists for s_1 , the algorithm returns the partition $\bar{P}_1 = \emptyset$ and $\bar{P}_2 = \bar{P} = P[a, b]$. Similarly, if no left-median exists for s_2 , the algorithm returns the partition $\bar{P}_1 = \bar{P}$ and $\bar{P}_2 = \emptyset$. Otherwise, depending on the order of v_1 and v_2 on P , the algorithm either applies Claim 2 or Claim 3 to find $w \in \{1, 2\}$ and $t \in \{1, 2\}$ such that s_w wins at v_t . If $w = 1$, the algorithm recursively obtains a partition (P'_1, P'_2) of $P(v_t, b]$, and returns $\bar{P}_1 = P[a, v_t] \circ P'_1$ and $\bar{P}_2 = P'_2$. If $w = 2$, the algorithm recursively obtains a partition (P'_1, P'_2) of $P[a, v_t)$ and returns $\bar{P}_1 = P'_1$ and $\bar{P}_2 = P'_2 \circ P[v_t, b]$.

Correctness. The correctness of the halting condition follows from the definition of an $(\{s_1, s_2\}, \text{left})$ -partition. If there is no left-median for s_1 (resp. s_2), in particular there are no vertices that s_1 (resp. s_2) reaches from the left on \bar{P} . Therefore, every vertex on \bar{P} is a vertex that $(\{s_1, s_2\}, \text{left})$ -likes s_2 (resp. s_1) and therefore a partition that sets $P_2 = \bar{P}$ (resp. $P_1 = \bar{P}$) is valid. The correctness of the recursive step of the algorithm follows directly from Claim 1 and the correctness of the recursion.

Complexity. The non-recursive part of the algorithm consists of a polylogarithmic number of queries for the enhanced MSSP, and therefore takes $\tilde{O}(1)$ time. We claim that the recursion depth is bounded by $O(\log n)$. This is because in every recursive step the algorithm either reduces the number of vertices that s_1 reaches from the left or the number of vertices that s_2 reaches from the left - by half. Since initially each of these numbers is bounded by $|P| \leq n$, a halting condition must be satisfied after at most $2 \log n$ recursive calls. The overall time complexity is therefore $\tilde{O}(1)$ thus completing the proof of Lemma 4. \square

The following Lemma is the general version of Lemma 4 in the presence of swirly paths. Its proof is in Appendix C.

Lemma 5. *Let $s_1, s_2 \in F_{\text{left}} \setminus P$ be two sites such that s_1 is closer to s_x on F_{left} than s_2 . One can compute in $\tilde{O}(1)$ time an $(\{s_1, s_2\}, \text{left})$ -partition of P into P_1, P_2 such that $x \in P_1$ and $y \in P_2$, unless the partition is trivial.*

3.4 Relaxed partition for H (proof of Lemma 2)

Using Lemma 5 we are finally ready to prove Lemma 2 on the construction of a relaxed partition for H . By Lemma 3, it is enough to compute an $(H_{\text{left}}, \text{left})$ -partition. If $|H_{\text{left}}| = 1$ the partition is trivial. If $|H_{\text{left}}| = 2$ we obtain the partition from Lemma 5. When $|H_{\text{left}}| = 3$ with s_1, s_2, s_3 being the three sites according to their order on F_{left} , starting from $s_1 = s_x$, we apply Lemma 5 on three pairs (s_1, s_2) , (s_2, s_3) and (s_1, s_3) . Let $P_i^{i,j}$ denote that part corresponds to s_i in the partition computed for (s_i, s_j) . If $P_2^{1,2} \cap P_2^{2,3} \neq \emptyset$ the algorithm returns $P_1^{1,2}, P_2^{1,2} \cap P_2^{2,3}, P_2^{2,3}$. Otherwise, if $P_2^{1,2} \cap P_2^{2,3} = \emptyset$ the algorithm returns $P_1^{1,3}, P_3^{1,3}$. Clearly, the algorithm takes $\tilde{O}(1)$ time.

Correctness. We first consider the case where $P_2^{1,2} \cap P_2^{2,3} = \emptyset$. Let $v \in P_1^{1,3}$ and assume to the contrary that v does not $(\{s_1, s_2, s_3\}, \text{left})$ -like s_1 . It must be that there exists $c \in \{s_1, s_2, s_3, s_x, s_y\}$ such that $v \in \text{Vor}_{\{s_1, s_2, s_3, s_x, s_y\}}(c)$ and $v \in \text{Left}(P, c)$. Since v is a vertex that $(\{s_1, s_3\}, \text{left})$ -likes s_1 , it must be that $c = s_2$. However, this means that v does not $(\{s_1, s_2\}, \text{left})$ -like s_1 and also v does not $(\{s_2, s_3\}, \text{left})$ -like s_3 . Hence, $v \in P_2^{1,2} \cap P_2^{2,3}$, a contradiction. Thus, v indeed $(\{s_1, s_2, s_3\}, \text{left})$ -likes s_1 . The proof for vertices in $P_3^{1,3}$ is similar.

We next consider the case $P_2^{1,2} \cap P_2^{2,3} \neq \emptyset$. For this case we first prove that every $v \in P_2 = P_2^{1,2} \cap P_2^{2,3}$ must $(\{s_1, s_2, s_3\}, \text{left})$ -like s_2 . Assume to the contrary that there exists some $v \in P_2$ such that $v \in \text{Vor}_{\{s_1, s_2, s_3, s_x, s_y\}}(c)$ and $v \in \text{Left}(P, c)$ and $c \in \{s_1, s_2, s_3\}$ and $c \neq s_2$. W.l.o.g. we can assume $c = s_1$ (the case $c = s_3$ is symmetric). However, in this case v does not $(\{s_1, s_2\}, \text{left})$ -like s_2 , contradicting $v \in P_2^{1,2}$.

Finally, we prove that every $v \in P_1^{1,2}$ must $(\{s_1, s_2, s_3\}, \text{left})$ -like s_1 (the proof for $P_3^{2,3}$ is symmetric). Notice that $P_1^{1,2} \subset P_2^{2,3}$ since $P_2^{1,2} \cap P_2^{2,3} \neq \emptyset$. Assume to the contrary there exists some $v \in P_1^{1,2}$ such that $v \in \text{Vor}_{\{s_1, s_2, s_3, s_x, s_y\}}(c)$ and $v \in \text{Left}(P, c)$ and $c \in \{s_1, s_2, s_3\}$ and $c \neq s_1$. It cannot be that $c = s_2$, because $v \in P_1^{1,2}$. Assume $c = s_3$, then we have that v does not $(\{s_2, s_3\}, \text{left})$ -like s_2 , contradicting $v \in P_1^{1,2} \subset P_2^{2,3}$.

Complexity. Finally, the running time of the algorithm is $\tilde{O}(1)$ since all distance and direction queries are answered in $\tilde{O}(1)$ time per query by the enhanced MSSP data structure of Lemma 1 and the binary search of w increases the running time only by additional $\tilde{O}(1)$ time. This concludes the proof of Lemma 2.

Remark 2. *We note that the proof of Lemma 2 is written for $|H| = 3$. However, the algorithm described in the proof can be easily generalized to an algorithm that constructs a partition for a set of k sites from a partition of $k-1$ sites. A standard amortization argument shows that the total running time for constructing a partition of k sites is $\tilde{O}(k)$.*

4 From a Relaxed Partition to a Trichromatic Face

In this section we prove the following lemma, stating that one can find a trichromatic face in $\tilde{O}(1)$ time.

Lemma 6. *Given a planar graph G , and a face F , one can preprocess G in $\tilde{O}(|G|)$ time such that given a subset $H \subseteq F$ of 3 sites, and additive weights $w : H \rightarrow \mathbb{R}^+$, one can compute the trichromatic face of $\text{VD}(H, w)$ in time $\tilde{O}(1)$.*

Proof. **TOPROVE 3** □

5 Computing the Voronoi Diagram

In this section, we describe an algorithm that, given access to the mechanism for computing trichromatic faces provided by Lemma 6, computes $\text{VD}^*(F)$ in $\tilde{O}(|F|)$ time. Thus, we establish the main theorem of this paper (note that we use F to denote both the set of sites and the face to which they belong):

Theorem 1. *Given a planar graph X , and a face F , one can preprocess X in $\tilde{O}(|X|)$ time such that given additive weights $\omega : F \rightarrow \mathbb{R}^+$, one can compute $\text{VD}^*(F)$ in $\tilde{O}(|F|)$ time.*

Representing the Diagram. We use the definitions of the dual Voronoi diagram $\text{VD}^*(F)$ from [9]. As was done there, we assume that the graph we work with is triangulated, except for the single face F , whose vertices are exactly the set of sites of the diagram. We also assume that each site induces a non empty Voronoi cell (including at least the site itself). Thus, $\text{VD}^*(F)$ is a degree-3 tree with $O(|F|)$ nodes whose leaves are the copies of the dual vertex F^* (corresponding to the face F). Following [22], we use the Doubly-Connected Edge List (DCEL) data structure for representing planar maps to represent the tree $\text{VD}^*(F)$.

The Divide-and-Conquer Mechanism. Denote $|F| = k$. We describe a divide-and-conquer algorithm for constructing $\text{VD}^*(F)$ in $\tilde{O}(k)$ time. If $k < 3$ then $\text{VD}^*(F)$ contains no trichromatic vertices and its representation is trivial. If $k = 3$, then $\text{VD}^*(F)$ consists of either no trichromatic faces or just the single trichromatic face \hat{f} obtained using Section 4. When $k > 3$ we partition the set S of sites into two contiguous subsets along the face F of (roughly) $k/2$ sites each. For simplicity, we assume that each subset has size exactly $k/2$. We call the sites $G = g_1, \dots, g_{k/2}$ in one subset the green sites, listed in counterclockwise order along F . Similarly, we call the other subset $R = r_1, \dots, r_{k/2}$ the red sites, listed in clockwise order along F . Note that the ordering is such that g_1 and r_1 are neighboring sites on the face F . We recursively compute $\text{VD}^*(G)$ and $\text{VD}^*(R)$, the Voronoi diagram of G and of R , respectively. We now describe how to merge these two diagrams into $\text{VD}^*(F)$ in $\tilde{O}(k)$ time. The idea is similar to the stitching algorithm used in [22]. The main differences are that unlike [22] we have not precomputed the bisectors, but on the other hand, we utilize the point location mechanism of $\text{VD}^*(G)$ and $\text{VD}^*(R)$, which was not done in [22]. In a nutshell, consider a super green vertex G connected to all green sites with edges whose lengths correspond to the additive weight to each green site. Similarly, consider a super red vertex R . Now consider the bisector $\beta^*(G, R)$. $\text{VD}^*(F)$ is obtained by cutting both $\text{VD}^*(G)$ and $\text{VD}^*(R)$ along $\beta^*(G, R)$, "glueing" the green side (according to $\beta^*(G, R)$) of $\text{VD}^*(G)$ with the red side of $\text{VD}^*(R)$.

This intuitive process can be performed efficiently using the following procedure. To cut $\text{VD}^*(G)$ along $\beta^*(G, R)$, we go over the $O(k/2)$ nodes of $\text{VD}^*(G)$. Each of these nodes corresponds to a trichromatic face f (or to an edge of the hole F if the node is a leaf of $\text{VD}^*(G)$). For each vertex v of f (there are 3 or 2 such vertices depending on whether we handle an internal node or a leaf of $\text{VD}^*(G)$), let g_i be the green site closest (in the sense of additive distance) to v (the identity of g_i is stored explicitly in the representation of $\text{VD}^*(G)$). We perform a point location query for v in $\text{VD}^*(R)$, obtaining the red site r_j closest (in the sense of additive distance) to v . We compare the distance from r_j to v with the distance from g_i to v (these distances are available through the MSSP data structure for F). If the distance from r_j is smaller, we know that f is not a trichromatic face in $\text{VD}^*(F)$, so we delete the corresponding node from $\text{VD}^*(G)$.⁴

⁴To be precise, f might be a trichromatic face of $\text{VD}^*(F)$, but it is not an all-green trichromatic face in $\text{VD}^*(F)$, so it is not contributed to $\text{VD}^*(F)$ by $\text{VD}^*(G)$.

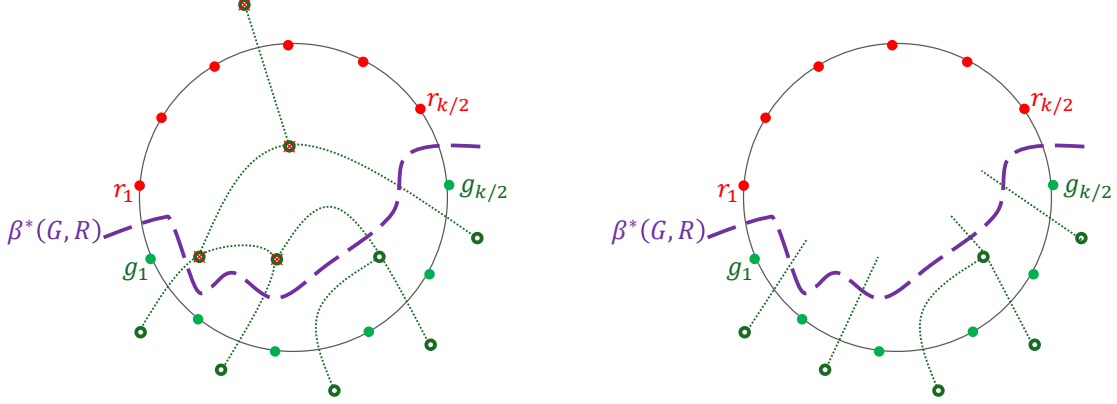


Figure 6: A schematic demonstration of the Voronoi vertex deletion process caused by the interaction of the $\beta^*(G, R)$ bisector with the green diagram $\text{VD}^*(G)$. Left: the $\text{VD}^*(G)$ diagram of the green sites, represented in dotted green lines, and the bisector $\beta^*(G, R)$ represented in a dashed purple line. The Voronoi (dual) vertices are represented in hollow green circles. The Vertices deleted in the process are crossed in red. Right: the remaining connected components and their respective dangling Voronoi edges after the deletion process.

Some Voronoi edges of $\text{VD}^*(G)$ have both their endpoints deleted by this process. These edges are entirely on the red side of $\beta^*(G, R)$, and do not participate in $\text{VD}^*(F)$. Other edges have both their endpoints not deleted. These edges are entirely on the green side of $\beta^*(G, R)$, and are part of $\text{VD}^*(F)$. We call the Voronoi edges with one endpoint deleted and the other not deleted *dangling* edges. The bisector $\beta^*(G, R)$ intersects $\text{VD}^*(G)$ at the dangling edges. We associate each dangling edge with the two green sites whose Voronoi cells (in $\text{VD}^*(G)$) are on either side of the dangling edge. The following lemma proves that there is at most a single dangling edge in each of the components of $\text{VD}^*(G)$ obtained by the above deletion process.

Lemma 7. *When the deletion process described above terminates, each surviving connected component of $\text{VD}^*(G)$ contains at most a single dangling edge.*

Proof. **TOPROVE 4**

□

Having found the components of $\text{VD}^*(G)$ (and by a similar process of $\text{VD}^*(R)$) that form $\text{VD}^*(F)$, we trace $\beta^*(G, R)$, and stitch the components of $\text{VD}^*(G)$ and $\text{VD}^*(R)$ at new trichromatic vertices that we identify along the way. The first endpoint of $\beta^*(G, R)$ (which is a leaf of $\text{VD}^*(F)$) is a copy of F^* that lies at the end of a dual of the edge of F separating between the sites $g_i = g_1$ and $r_j = r_1$. The next trichromatic vertex along $\beta^*(G, R)$ occurs when $\beta^*(G, R)$ intersects the boundary of the Voronoi cell of either g_i or of r_j .

Recall that each dangling edge with the two green sites whose Voronoi cells (in $\text{VD}^*(G)$) are on either side of the dangling edge. We shall prove in Lemma 8, that the sites g_1 and $g_{k/2}$ are associated with exactly one dangling edge, and all other sites are associated with either exactly two or exactly zero dangling edges. To identify the next trichromatic vertex of $\text{VD}^*(F)$ along $\beta^*(G, R)$, we inspect the dangling edge associated with g_i . When $g_i = g_1$, there is only one associated dangling edge. In general, there are two associated dangling edges, but only one was not yet handled by the stitching process. The dangling edge that was not yet handled represents a bisector between two green sites. One of them must be g_i (since the intersection is a trichromatic face on the boundary of the Voronoi cell of g_i). Denote the other one by $g_{i'}$. Similarly, a possible candidate for the next trichromatic vertex of $\text{VD}^*(F)$ along $\beta^*(G, R)$ may come from the yet unhandled dangling edge of a connected component of $\text{VD}^*(R)$ that is associated with r_j , which represents a bisector between r_j and some other red site $r_{j'}$.

We use Lemma 6 to find the two possible candidates: the trichromatic face f_g of $(r_j, g_i, g_{i'})$ and the trichromatic face f_r of $(r_j, r_{j'}, g_i)$. Only one of them is a true trichromatic vertex of $\text{VD}^*(F)$, which can be decided by using the same point location process we had used in the deletion process above for each of its incident primal vertices. Suppose w.l.o.g. that we identified that the next trichromatic face is the face f_g

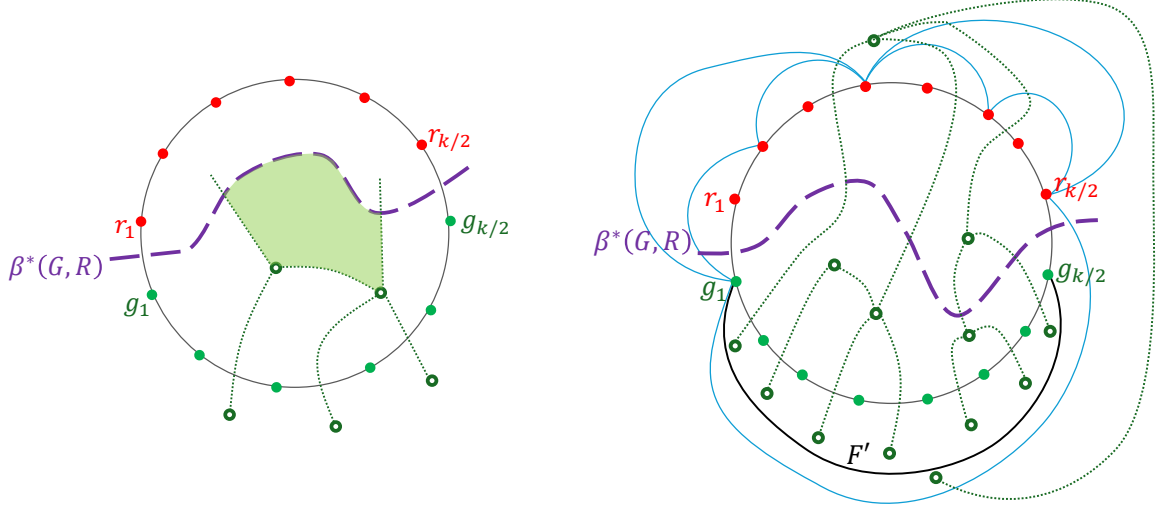


Figure 7: Left: The cycle C (enclosing the shaded green area) for the contradiction in the proof of Lemma 7. Right: Enforcing the assumptions before the recursive call that computes $\text{VD}^*(G)$. The bold black edge guarantees that the green sites are exactly the sites of a face (the face F'). The blue artificial edges guarantee triangulation.

of $(r_j, g_i, g_{i'})$ (the procedure for f_r is symmetric). We connect in $\text{VD}^*(S)$ the previous trichromatic face on $\beta^*(G, R)$ with f_g via a new Voronoi edge, and make f_g the new endpoint of the dangling edge associated with g_i .

Next, we infer the identity of the edge of the $\beta^*(G, R)$ bisector leaving f_g via which the traversal of $\beta^*(G, R)$ continues. Since f_g was the new trichromatic face, then $\beta^*(G, R)$ just crossed in $\text{VD}^*(G)$ from the cell of g_i to the cell of $g_{i'}$, so we repeat the process of finding the next trichromatic face along $\beta^*(G, R)$ with sites r_j and $g_{i'}$. The process terminates when it has handled all of the dangling edges in all the components of $\text{VD}^*(G)$ and $\text{VD}^*(R)$.

To complete the correctness argument it remains to prove the bound on the association between sites and dangling edges (Lemma 8). However, before doing that, we need to elaborate on a technical issue we had glossed over in the description of the recursive approach. We had assumed that in any VD^* in the recursion, the sites are exactly the vertices of some face F of the graph X in which VD^* is computed, and that F is the only face of X that is not a triangle. To satisfy this requirement, before making the recursive call that computes $\text{VD}^*(G)$, we add to X an artificial infinite-length edge connecting g_1 and $g_{k/2}$. This artificial edge is embedded in the face F , splitting it into two new faces F' and F'' , such that the vertices of F' are exactly the green sites. We triangulate F'' with infinite length edges. Let X' denote the resulting graph. See Fig. 7 (right). Note that X' now satisfies the assumptions so we can invoke the construction algorithm recursively on X' and obtain $\text{VD}^*(G)$. In $\text{VD}^*(G)$, the boundary of the Voronoi cell of each g_i forms a simple path between the two leaves (copies of F' corresponding to dual edges of F' between $g_{i-1}g_i$ and $g_i g_{i+1}$) in $\text{VD}^*(G)$, which we associate with g_i . Note that for sites g_i for $1 < i < k/2$, both associated leaves are real edges of P , whereas for g_1 and $g_{k/2}$ one leaf is real, but the other is dual to an artificial edge of X' .

Since $\text{VD}^*(G)$ may contain (dual) artificial vertices (i.e., faces X' that are not faces of X), at the beginning of the deletion process we delete all artificial vertices of $\text{VD}^*(G)$, and then apply the deletion process described above. Observe that exactly one of the leaves corresponding to g_1 and $g_{k/2}$ is deleted by the deletion process, and none of the two leaves associated with the other sites is deleted. This is because each G_i is in its own Voronoi cell in $\text{VD}^*(F)$.

Lemma 8. *When the deletion process of $\text{VD}^*(G)$ described above terminates, sites g_1 and $g_{k/2}$ are associated with exactly one dangling edge, and all other sites are associated with either two or zero dangling edges.*

References

- [1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.
- [2] S. R. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. H. M. Smid, and C. 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, pages 514–528, 1996.
- [3] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen. Min *st*-cut oracle for planar graphs with near-linear preprocessing time. *ACM Transactions on Algorithms*, 11(3):1–29, 2015.
- [4] S. Cabello. Many distances in planar graphs. *Algorithmica*, 62(1-2):361–381, 2012.
- [5] S. 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.
- [6] S. Cabello, E. W. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM J. Comput.*, 42(4):1542–1571, 2013.
- [7] P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In J. I. Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 828–829. SIAM, 2004.
- [8] T. M. Chan and D. Skrepetos. Faster approximate diameter and distance oracles in planar graphs. *Algorithmica*, 81(8):3075–3098, 2019.
- [9] P. Charalampopoulos, P. Gawrychowski, Y. Long, S. Mozes, S. Pettie, O. Weimann, and C. Wulff-Nilsen. Almost optimal exact distance oracles for planar graphs. *J. ACM*, 70(2):12:1–12:50, 2023.
- [10] P. Charalampopoulos, P. Gawrychowski, S. Mozes, and O. 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.
- [11] P. Charalampopoulos, P. Gawrychowski, S. Mozes, and O. Weimann. An almost optimal edit distance oracle. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 198, pages 48:1–48:20, 2021.
- [12] P. Charalampopoulos and A. Karczmarz. Single-source shortest paths and strong connectivity in dynamic planar graphs. *J. Comput. Syst. Sci.*, 124:97–111, 2022.
- [13] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.
- [14] D. Z. Chen and J. Xu. Shortest path queries in planar graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 469–478, 2000.
- [15] V. Cohen-Addad, S. Dahlgaard, and C. 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.
- [16] H. 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, pages 151–165, 1996.

- [17] 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.
- [18] J. Erickson, K. Fox, and L. 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.
- [19] 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.
- [20] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.
- [21] V. Fredslund-Hansen, S. Mozes, and C. Wulff-Nilsen. Truly subquadratic exact distance oracles with constant query time for planar graphs. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*, pages 25:1–25:12, 2021.
- [22] 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. *SIAM J. Comput.*, 50(2):509–554, 2021.
- [23] P. Gawrychowski, S. Mozes, O. Weimann, and C. 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.
- [24] Q. Gu and G. Xu. Constant query time $(1 + \epsilon)$ -approximate distance oracle for planar graphs. *Theor. Comput. Sci.*, 761:78–88, 2019.
- [25] K. Kawarabayashi, P. N. Klein, and C. 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, pages 135–146, 2011.
- [26] K. Kawarabayashi, C. Sommer, and M. Thorup. More compact oracles for approximate distances in undirected planar graphs. In *Proceedings of the 24th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 550–563, 2013.
- [27] P. 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.
- [28] P. 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.
- [29] P. N. Klein, S. Mozes, and C. 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.
- [30] J. Lacki and P. 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.
- [31] H. Le and C. 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.
- [32] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.
- [33] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

- [34] Y. Long and S. 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.
- [35] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci.*, 32(3):265–279, 1986.
- [36] R. Motwani and P. Raghavan. *Randomized algorithms*. Press Syndicate of the University of Cambridge, 1995.
- [37] S. Mozes, K. Nikolaev, Y. Nussbaum, and O. 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.
- [38] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 209–222, 2012.
- [39] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.
- [40] Y. Nussbaum. Improved distance queries in planar graphs. In *Proceedings of the 12th International Workshop on Algorithms and Data Structures (WADS)*, pages 642–653, 2011.
- [41] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- [42] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.
- [43] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14(1):232–253, 1985.
- [44] C. Wulff-Nilsen. *Algorithms for planar graphs and graphs in metric spaces*. PhD thesis, University of Copenhagen, 2010.
- [45] C. Wulff-Nilsen. Approximate distance oracles for planar graphs with improved query time-space trade-off. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 2016.

A The Enhanced MSSP Data Structure (proof of Lemma 1)

In this section we prove Lemma 1 by extending the seminal MSSP data structure[6, 27]. Let $s_1, \dots, s_{|f|}$ be the vertices of the face f in cyclic order. The MSSP is initialized with the shortest paths tree T_1 rooted at s_1 (computed in $O(n \log n)$ time using Dijkstra). Then, T_2 is computed from T_1 using *pivots*. A pivot is the process of replacing an edge of T_1 with an edge not in T_1 . Then, T_3 is computed from T_2 and so on until $T_{|f|}$ is computed. Klein showed that: (1) over the entire process of computing $T_1, \dots, T_{|f|}$ there are only $O(n)$ pivots, (2) every pivot can be found and executed in $O(\log n)$ time by maintaining the dual tree of the current T_k in a dynamic tree data structure [41]. Finally, by using the persistence technique of [17], all the T_k 's can be recorded in $(n \log n)$ space so as to permit $\text{dist}(s, v)$ queries. We slightly modify the MSSP data structure, such that the dynamic trees would be implemented with top-trees of Alstrup et al. [1]. This retains all features of the original implementation while additionally supporting $\text{ancestor}(s, v, d)$ queries in $O(\log n)$ time per query.

In order to implement $\text{direction}(s, v, P)$ queries, we additionally maintain a persistent [17] range data structure [13, 43] Left_P for every shortest path P in the decomposition tree. The data structure Left_P stores, throughout the running of MSSP, the set of vertices (ordered by their order on P) that the current source vertex s_i reaches from the left. Once Left_P is maintained properly, $\text{count}(s, P, i, j)$ queries can be

supported in $\tilde{O}(1)$ time by accessing $Left_P$ at the time in which the stored MSSP tree is the shortest path tree rooted at s . Moreover, **select** queries can be supported via binary search that utilizes **count** queries.

Initially, $Left_P$ is obtained in $O(|P|)$ time by checking the parent of every vertex $v \in P$ in the shortest paths tree of s_1 (if the parent is on P , then v inherits the direction from its parent). In order to maintain $Left_P$, we store an additional range data structure $Cross(u)$ for every vertex $u \in G$. For a vertex u , let e_1, e_2, \dots, e_d be the edges incident to u in clockwise order. For every separator path P such that $u \in P$, there is a cyclic interval of the edges of u that are on the left of P , and a cyclic interval that is on the right. The data structure $Cross(u)$ stores the left and right cyclic intervals of all paths P containing u such that given two edges e_i and e_j incoming to u , it returns the set of paths P such that e_i and e_j are on different sides of P . Since each vertex u of G has constant degree, implementing the $Cross(u)$ data structures is trivial and requires $\tilde{O}(|P|)$ time for each P . This sums up to $\tilde{O}(n)$.

To maintain $Left_P$ during the MSSP execution, while obtaining T_k from T_{k-1} , whenever a pivot is performed, we apply the following: Let u be the vertex such that the pivot changes u 's incoming edge from e_i to e_j . It is possible that the Left/Right status of u has been changed with respect to (possibly several) paths P containing u . Notice that if such a status change occurs for u , it also occurs for all descendants of u in T_k that lie on P (since P is a shortest path, and due to the uniqueness of shortest paths, those descendants all lie on a single subpath of P that contains u). We need to find the set of paths P for which the Left/Right status of u changes. We do this by querying $Cross(u)$ to obtain all paths P such that e_i and e_j are on different sides of P . If e_i or e_j are on P , the new/old type of u is determined by its first ancestor in T_k/T_{k-1} that is not on P (which can be found by binary search on P and the tree edges of T_k/T_{k-1}). If the type of u changes, we remove/insert u from $Left_P$ (depending on the new type). We also run the following procedure in order to update the Left/Right status of all descendants of u in P , if necessary. Starting from u , we traverse P twice, once in each direction of P . As long as a descendant of u is met, we update the traversed vertex with the new type of u . We halt when we reach a vertex of P that is not an ancestor of u .

Correctness: We claim that all $Left_P$ data structures are updated correctly after every pivot. Clearly, if both e_i and e_j are left or right edges, the left/right status of the vertex u does not change, so only the set of paths returned from $Cross(u)$ needs to be checked. This set is updated naively - both u and all of its decedents on P are updated one by one. Due to the uniqueness of shortest paths, the decedents of u on P form a consecutive subpath of P . Therefore, the algorithm did not miss any descendants of u on P as a result of halting when encountering a non-descendant in each direction. Finally, vertices that are not decedents of u did not have their shortest path change as a result of the pivot, and therefore their Left/Right status did not change with respect to any path P .

Running time. The algorithm queries a $Cross(u)$ data structure per pivot, which sums up to $\tilde{O}(n)$ across the run of MSSP. As for the updates to $Left_P$, we claim that every vertex u is accessed in this way $O(1)$ times per path P containing it. Let us consider the event in which a vertex u is accessed because e_j is not of the same type as e_i with respect to the path P . We distinguish between two types of vertex access events: an event in which u changes its Left/Right status with respect P , and an event in which it does not. If u does not change its status, we necessarily have that either the old edge e_i or the new edge e_j is on the path P (as otherwise one is a right edge and the other is a left edge, leading to a status change of u with respect to P). It immediately follows from the fact that every edge participates in $O(1)$ pivots that this event occurs at most once per $u \in P$, and therefore the number of occurrences of this event is proportional to the total length of all paths P , which is $\tilde{O}(n)$. If on the other hand u does change its status, then by the following claim this can happen at most two times throughout the running of the algorithm.

Claim 4. *Let P be a separator path and let u be a vertex of P . The set of source vertices s_k such that s_k reaches u from the left with respect to P form a cyclic interval on the face f .*

Proof. **TOPROVE 6** □

To summarize, it follows from Claim 4 that u changes its Left/Right status with respect to P at most twice throughout the algorithm (namely, when the boundaries of the cyclic interval indicated by Claim 4 are processed). Therefore, the number of times the algorithm processes a vertex that changes its status is

bounded by the total size of all paths in the decomposition, which is $O(n)$. Since a vertex is processed in $\tilde{O}(1)$ time, the total running time is bounded by $\tilde{O}(n)$ as required.

B Proof of Lemma 3

Assume to the contrary that an $(H_{\text{left}}, \text{left})$ -partition is not an (H, left) -partition. In particular, it must be that there exists some $v \in P$ such that v $(H_{\text{left}}, \text{left})$ -likes some site $s_{\text{left}} \in H_{\text{left}}$ but does not (H, left) -like s_{left} . This means that for some $c \in H$ we have that (1) $v \in \text{Vor}_{H \cup \{s_x, s_y\}}(c)$, (2) $c \neq s_{\text{left}}$, and (3) $v \in \text{Left}(P, c)$.

We first claim that $c \in H \setminus H_{\text{left}}$. Assume to the contrary that $c \in H_{\text{left}}$, and recall that $v \in \text{Left}(P, c)$. Therefore, it must be that $v \in P_c$ in the $(H_{\text{left}}, \text{left})$ -partition. Since $P_c \cap P_{s_{\text{left}}} = \emptyset$, we obtain a contradiction. Thus, $c \in H \setminus H_{\text{left}}$.

Consider the cycle C obtained by concatenating R_y, P, R_x and the Jordan curve connecting s_x and s_y embedded in the face F . We think of C as an oriented cycle whose orientation is consistent with that of P so as to define left and right properly. Notice that R_x and R_y do not cross each other. Therefore, C is non-self crossing. Since $c \in H \setminus H_{\text{left}}$, c is to the right of C . Therefore, $R_{c,v}$ must cross C (recall that $v \in \text{Left}(P, c)$). By uniqueness of shortest paths $R_{c,v}$ does not cross P . Therefore, it must cross either R_x or R_y . Contradicting $v \in \text{Vor}_H(c)$.

C Partition of Two Sites with Swirly Paths

In this section we prove Lemma 5. We first (in Appendix C.1) prove the lemma for the special case where $s_1 = s_x$ and $s_2 = s_y$. Then (in Appendix C.2), we utilize the partition obtained for s_x, s_y to compute a partition for any pair $s_1, s_2 \in F_{\text{left}}$.

C.1 Partition for s_x and s_y

To prove Lemma 5 we begin with the simpler case of $s_1 = s_x$ and $s_2 = s_y$. We start by proving the following structure regarding s_x and s_y .

The following lemma shows that for $s \in \{s_x, s_y\}$ we can recognize swirly and non-swirly paths to vertices in $\text{Left}(P, s)$ by comparing their position in the shortest paths tree of s compared to x or y .

Lemma 9. *Let $v \in \text{Left}(P, s_y)$ be a vertex, if v is to the right of y in the shortest paths tree of s_y , then $\text{dist}(s_y, v) > \text{dist}(s_x, v)$ and $v \in \text{Left}(P, s_x)$. Otherwise, s_y reaches v via a non-swirly path.*

Similarly, for a vertex $v \in \text{Left}(P, s_x)$, if v is to the left of x in the shortest paths tree of s_x , then $\text{dist}(s_x, v) > \text{dist}(s_y, v)$ and $v \in \text{Left}(P, s_y)$. Otherwise, s_x reaches v via a non-swirly path.

Proof. **TOPROVE 7** □

Lemma 10. *Given s_x and s_y , there is an algorithm that outputs in $\tilde{O}(1)$ -time a partition of P into two (possibly empty) parts $P = P_1^x \circ P_2^x$ such that:*

1. *Every vertex $v \in \text{Left}(P_1^x, s_x)$ is reached from s_x via a non-swirly path.*
2. *For every vertex $v \in \text{Left}(P_2^x, s_x)$ we have $\text{dist}(s_x, v) > \text{dist}(s_y, v)$.*

Similarly, the algorithm outputs a partition of P into two (possibly empty) parts $P = P_1^y \circ P_2^y$ such that:

1. *For every vertex $v \in \text{Left}(P_1^y, s_y)$ we have $\text{dist}(s_y, v) > \text{dist}(s_x, v)$.*
2. *Every vertex $v \in \text{Left}(P_2^y, s_y)$ is reached from s_y via a non-swirly path.*

Proof. **TOPROVE 8** □

Lemma 11. *There is an $\tilde{O}(1)$ algorithm that given s_x and s_y outputs an $(\{s_x, s_y\}, \text{left})$ -partition of P .*

Proof. **TOPROVE 9** □

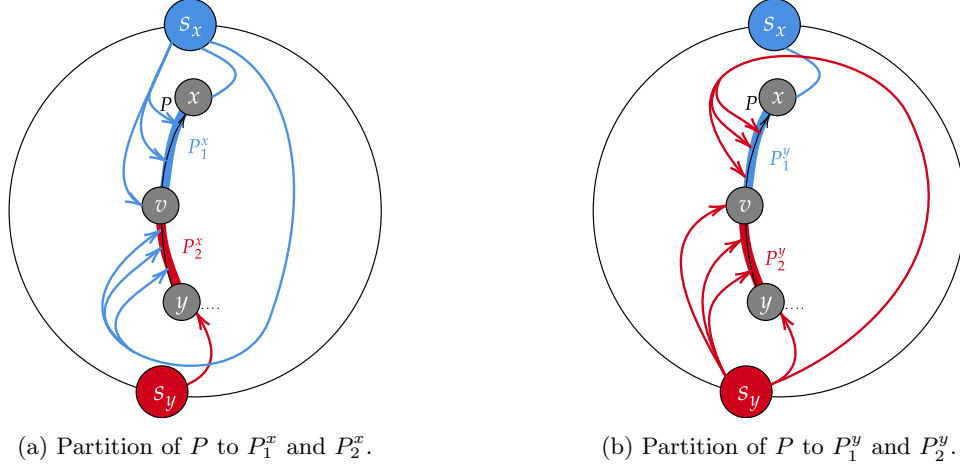


Figure 8: Figures for the statement of Lemma 10.

C.2 Partition for $s_1, s_2 \in F_{\text{left}}$

In this section we show how to compute the relaxed partition for any pair of vertices $s_1, s_2 \in F_{\text{left}}$ in $\tilde{O}(1)$ time, thus proving Lemma 5. The main idea is to recognize, separately for each $s \in \{s_1, s_2\}$, one of two cases: Either that s is not the site of any vertex in $\text{Left}(P, s)$ which means we can assign all P to the other site. Or, recognize a partition of $\text{Left}(P, s)$ into up to two intervals, where one interval contains only vertices v with $R_{s,v}$ being non-swirly, and the other interval with all vertices v with $R_{s,v}$ being swirly. To use these partition we exploit the partition of P with respect to $\{s_x, s_y\}$ (Appendix C.1). We then show that when considering both s_1 and s_2 , we can identify a prefix and a suffix of P that contains all swirly paths and are assigned to s_1 and s_2 . Between the prefix and suffix there is an interval that does not contain any swirly path from s_1 or s_2 , and we use Section 3.3 to partition it. By combining the prefix-suffix with the partition for the middle part, we obtain the desired partition for P with respect to $\{s_1, s_2\}$.

Let $P_x = P_x^{\text{left}}$ (resp. P_y) be the prefix (resp. suffix) of P computed by Lemma 11. We note that by definition of $(\{s_x, s_y\}, \text{left})$ -like, for every $v \in \text{Left}(P, s_x) \cap \text{Left}(P, s_y)$ we have $v \in P_x$ if and only if $\text{dist}(s_x, v) < \text{dist}(s_y, v)$. The following lemma shows that for s_x , all the shortest paths to vertices of $\text{Left}(P_x, s_x)$ are non-swirly.

Lemma 12. *If $v \in \text{Left}(P_x, s_x)$ then $R_{s_x, v}$ is a non-swirly path. Similarly, if $v \in \text{Left}(P_y, s_y)$ then $R_{s_y, v}$ is a non-swirly path.*

Proof. **TOPROVE 10** □

Recall that a path $R_{s,v}$ from $s \in F_{\text{left}}$ to $v \in \text{Left}(P, s)$ is swirly if it crosses either R_x or R_y . The following lemma states that a swirly path must cross R_x or R_y from left to right, and must intersect the second path (among R_x and R_y).

Lemma 13. *Let $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$, and let $v \in \text{Left}(P, s)$. If $R_{s,v}$ is an x -swirly path, then $R_{s,v}$ crosses R_x from left to right (when considering R_x as being oriented from x to s_x). Moreover, $R_{s,x}$ must intersects R_y .*

Similarly, if $R_{s,v}$ is a y -swirly path, then $R_{s,v}$ crosses R_y from left to right (when considering R_y as being oriented from s_y to y). Moreover, $R_{s,y}$ must intersects R_x .

Proof. **TOPROVE 11** □

The following lemma and corollary show that a swirly path $R_{s,v}$ is y -swirly if and only if $v \in P_x$.

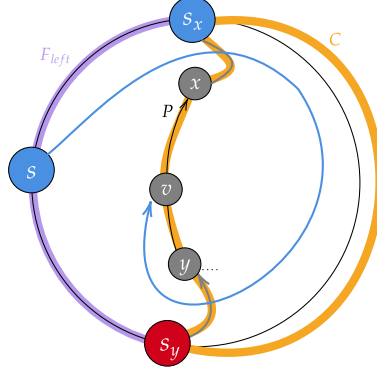


Figure 9: The cycle C in the proof of Lemma 13.

Lemma 14. *Let $s \in F_{\text{left}} \setminus \{s_x, s_y\}$ and let $v \in \text{Left}(P, s)$ such that $R_{s,v}$ is an x -swirly path. Then, v appears to the left of x in the shortest paths tree of s_x , $v \in \text{Left}(P, s_x)$, $v \in \text{Left}(P, s_y)$ and $\text{dist}(s, v) > \text{dist}(s_x, v) \geq \text{dist}(s_y, v)$.*

Similarly, if $R_{s,v}$ is a y -swirly path, then, v appears to the right of y in the shortest paths tree of s_y , $v \in \text{Left}(P, s_y)$, $v \in \text{Left}(P, s_x)$ and $\text{dist}(s, v) > \text{dist}(s_y, v) \geq \text{dist}(s_x, v)$.

Proof. **TOPROVE 12** □

The following is a direct consequence of Lemmas 9 and 14.

Corollary 1. *Let $s \in F_{\text{left}} \setminus \{s_x, s_y\}$ and let $v \in \text{Left}(P_x, s)$. If $R_{s,v}$ is a swirly path, then it is a y -swirly path. Furthermore, if $v \in \text{Left}(P_x, s_x)$ then $R_{s_x,v}$ is a non-swirly path.*

Similarly for $v \in \text{Left}(P_y, s)$, if $R_{s,v}$ is a swirly path, then it is an x -swirly path. Furthermore, if $v \in \text{Left}(P_y, s_y)$ then $R_{s_y,v}$ is a non-swirly path.

In the following Lemmas (Lemmas 15 to 19) we show that for any $s \in F_{\text{left}} \setminus \{s_x, s_y\}$ we can either deduce that s is not the site of any $v \in \text{Left}(P, s)$ or a partition of $\text{Left}(P, s)$ into an interval of non-swirly paths, and an interval of swirly paths.

Lemma 15. *Let $s \in F_{\text{left}} \setminus P$ and $v_{\text{top}}, v_{\text{bottom}} \in \text{Left}(P, s)$ such that v_{top} is closer to x than v_{bottom} and in the shortest paths tree of s , v_{bottom} appears not to the left of v_{top} . Then, for every $u \in \text{Left}(P[v_{\text{top}}, v_{\text{bottom}}], s)$ we have that u appears not to the left of v_{top} in the shortest paths tree of s .*

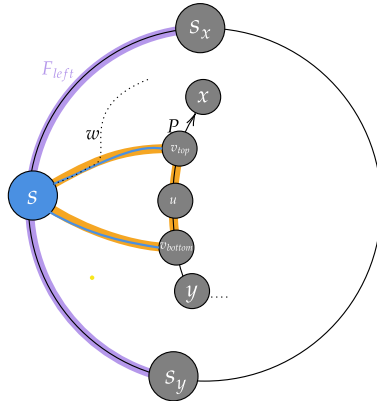


Figure 10: The cycle C for the proof of Lemma 15.

Proof. **TOPROVE 13** □

A subpath of $P[a, b]$ is called (s, left) swirly (resp. x -swirly, y -swirly, non-swirly) if for every $v \in \text{Left}(P[a, b], s)$, the path $R_{s,v}$ is a swirly (resp. x -swirly, y -swirly, non-swirly) path.

Lemma 16. *Let $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$ and $v_{\text{top}}, v_{\text{bottom}} \in \text{Left}(P, s)$ such that v_{top} is closer to x than v_{bottom} and v_{bottom} appears not to the left of v_{top} in the shortest paths tree of s , then $P[v_{\text{top}}, v_{\text{bottom}}]$ is either a (s, left) non-swirly, a (s, left) x -swirly or a (s, left) y -swirly subpath.*

Proof. **TOPROVE 14** □

Lemma 17. *There exists an algorithm that given $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$ and $v_{\text{top}}, v_{\text{bottom}} \in P$ such that v_{top} is closer to x than v_{bottom} and in the shortest paths tree of s , v_{top} appears to the left of v_{bottom} , reports in $\tilde{O}(1)$ time one of the following:*

1. Reports that $P[v_{\text{top}}, v_{\text{bottom}}]$ is an (s, left) non-swirly subpath.
2. Reports that for every $v \in \text{Left}(P[v_{\text{top}}, v_{\text{bottom}}], s)$ we have, $\text{dist}(s, v) > \min\{\text{dist}(s_x, v), \text{dist}(s_y, v)\}$.

Proof. **TOPROVE 15** □

Lemma 18. *There is an algorithm that given $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$ outputs in $\tilde{O}(1)$ time at most two subpaths $P[a_1, b_1]$ and $P[a_2, b_2]$ such that all vertices in $\text{Left}(P, s) \subseteq P[a_1, b_1] \cup P[a_2, b_2]$, and b_1 (resp. b_2) is not to the left of a_1 (resp. a_2) in the shortest paths tree of s . In addition, s reaches $a_1, b_1, a_2, b_2 \in \text{Left}(P, s)$ and a_1 is closer to x than a_2 . Moreover, if one of the two subpaths is non-swirly, then the other one is swirly.*

Proof. **TOPROVE 16** □

Lemma 19. *There exists an algorithm that given $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$ outputs in $\tilde{O}(1)$ time one of the following:*

1. At most two subpaths $P[a_1, b_1]$ and $P[a_2, b_2]$ such that $\text{Left}(P, s) \subseteq P[a_1, b_1] \cup P[a_2, b_2]$, and $a_1, b_1, a_2, b_2 \in \text{Left}(P, s)$, and flags $f_1, f_2 \in \{N', X', Y'\}$ such that:
 - (a) If $f_i = N'$, then $P[a_i, b_i]$ is an (s, left) non-swirly subpath.
 - (b) If $f_i = Y'$, then $P[a_i, b_i]$ is an (s, left) y -swirly subpath.
 - (c) If $f_i = X'$, then $P[a_i, b_i]$ is an (s, left) x -swirly subpath.
2. Reports that for every $v \in \text{Left}(P, s)$, we have $\text{dist}(s, v) > \min\{\text{dist}(s_x, v), \text{dist}(s_y, v)\}$.

Proof. **TOPROVE 17** □

In the following two lemmas, we show that we can utilize swirly paths in each of s_1 and s_2 to deduce that a prefix or suffix of P can be assigned in the relaxed partition with respect to $\{s_1, s_2\}$ with one site. More specifically, the prefix (that contains x) can be associated with s_1 and the suffix can be associated with s_2 .

Lemma 20. *Let $s \in F_{\text{left}} \setminus (P \cup \{s_x, s_y\})$ and let $v \in \text{Left}(P, s)$ such that $R_{s,v}$ is a y -swirly path. Then for every $u \in \text{Left}(P[x, v], s)$ we have that $R_{s,u}$ is also a y -swirly path.*

Similarly, if $R_{s,v}$ is an x -swirly path, then for every $u \in \text{Left}(P[v, y], s)$ we have that $R_{s,u}$ is also an x -swirly path.

Proof. **TOPROVE 18** □

Lemma 21. *Let s_1 and s_2 be two vertices on F_{left} such that s_1 is closer to s_x than s_2 on F_{left} . For every $w \in \{1, 2\}$ and $v \in P_x$, if s_w reaches v via a y -swirly path, every vertex on $P_x[a_x, v]$ ($\{s_1, s_2\}, \text{left}$)-likes s_1 .*

Similarly, for every $w \in \{1, 2\}$ and $v \in P_y$, if s_w reaches v via a x -swirly path, every vertex on $P_x[v, b_x]$ ($\{s_1, s_2\}, \text{left}$)-likes s_2 .

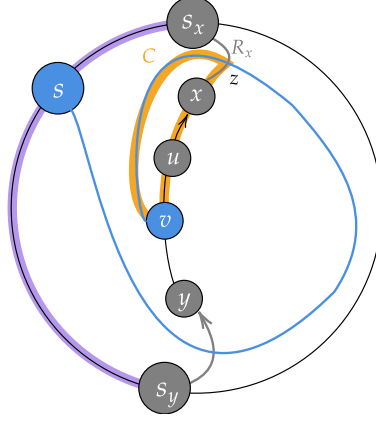


Figure 11: The cycle C for the proof of Lemma 20.

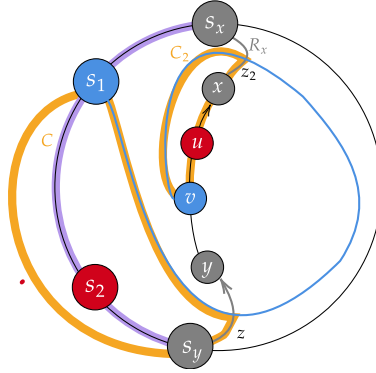


Figure 12: The cycles C and C_2 for the proof of Lemma 21.

Proof. **TOPROVE 19** □

Finally, we are ready to prove Lemma 5 by combining all the above. This way we obtain a relaxed partition for any two sites $s_1, s_2 \in F_{\text{left}}$. As described above, the idea is to assign a prefix and suffix of P to s_1 and s_2 respectively, exploiting information on swirly paths. Then the relaxed partition of P is obtained by partition the remaining interval using the non-swirly algorithm (Section 3.3).

Lemma 5. *Let $s_1, s_2 \in F_{\text{left}} \setminus P$ be two sites such that s_1 is closer to s_x on F_{left} than s_2 . One can compute in $\tilde{O}(1)$ time an $(\{s_1, s_2\}, \text{left})$ -partition of P into P_1, P_2 such that $x \in P_1$ and $y \in P_2$, unless the partition is trivial.*

Proof. **TOPROVE 20** □